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

В. І. Павловський
А. В. Петрашенко

БАЗИ ДАНИХ ТА ЗАСОБИ УПРАВЛІННЯ

ПІДРУЧНИК

Затверджено Методичною радою КПІ ім. Ігоря Сікорського
як підручник для здобувачів ступеня бакалавра
за освітньо-професійною програмою підготовки бакалаврів
спеціальності 123 «Комп'ютерна інженерія»

Київ
КПІ ім. Ігоря Сікорського
2024

УДК 004.42

Рецензенти	*Купин А.І.,* д.т.н., професор, завідувач кафедри комп'ютерних систем та мереж Криворізького національного університету
	Нікітенко Є.В., к.ф.-м.н., доцент доцент кафедри комп'ютерних систем, мереж та кібербезпеки Національного університету біоресурсів і природокористування України
Відповідальний редактор	*Заболотня Т.М.,* к.т.н, доцент, доцент кафедри програмного забезпечення комп'ютерних систем КПІ ім. Ігоря Сікорського

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № __ від ___.___._____ р.)
за поданням Вченої ради факультету прикладної математики (протокол № __ від __.__.2024 р.)

Електронне мережне навчальне видання

Павловський Володимир Ілліч, канд. техн. наук, доцент
Петрашенко Андрій Васильович, канд. техн. наук, доцент

БАЗИ ДАНИХ ТА ЗАСОБИ УПРАВЛІННЯ

ПІДРУЧНИК

Бази даних та засоби управління: підручник. [Електронний ресурс] / В. І. Павловський, А. В. Петрашенко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 2,7 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 293 с.

У даному підручнику розглядаються питання організації, проєктування та використання сучасних баз даних та систем управління ними. Особливістю підручника є систематизований виклад матеріалу, що містить необхідний теоретичний матеріал, який логічно поєднано з великою кількістю практичних прикладів. Фокусуючи увагу читача на математичному підґрунті сучасних баз даних, так званій реляційній моделі, підручник послідовно розкриває деталі реалізації загальновідомої «класичної» мови доступу до даних SQL. Таким чином, читач має змогу не тільки набути практичних навичок створення та використання баз даних, а і розуміти як вони функціонують, вміти знаходити найкращі технічні рішення в даній галузі. Окремо в підручнику розглядаються проблеми спільного доступу до даних багатьма користувачами, зокрема, використання транзакцій, забезпечення конкурентного та регламентованого доступу до даних.

Підручник розроблено для вивчення матеріалу кредитного модуля «Бази даних та засоби управління» студентів очної форми навчання за спеціальністю 123 «Комп'ютерна інженерія» факультету прикладної математики КПІ ім. Ігоря Сікорського.

УДК 004.42

© В. І. Павловський, А. В. Петрашенко, 2024
© КПІ ім. Ігоря Сікорського, 2024

ЗМІСТ

ВСТУП 12

1. ПОНЯТТЯ ТА ФУНКЦІЇ СИСТЕМ УПРАВЛІННЯ БАЗАМИ ДАНИХ 14

1.1. Коротка історична довідка 14
1.2. Поняття баз даних та систем управління базами даних 15
1.2.1. Дані 15
1.2.2. Апаратне забезпечення 16
1.2.3. Програмне забезпечення 16
1.2.4. Користувачі 16
1.3. Архітектура системи управління базами даних 18
1.4. Рівні абстракції або представлення даних в системах управління базами даних 19
1.4.1. Зовнішній рівень – зовнішнє представлення 19
1.4.2. Концептуальний рівень – концептуальне представлення 20
1.4.3. Внутрішній рівень – внутрішнє представлення 20
1.5. Функції систем управління базами даних 21
1.6. Експертні системи та бази знань 22
1.7. Контрольні запитання 22

2. МОДЕЛІ БАЗ ДАНИХ 23

2.1. Моделі даних. Дореляційні системи управління базами даних 23
2.2. Ранні чи дореляційні системи управління базами даних 23
2.3. Системи, що базуються на інвертованих списках 24
2.3.1. Структури даних 24
2.3.2. Маніпулювання даними 24
2.3.3. Обмеження цілісності 25
2.4. Ієрархічна модель даних 25
2.4.1. Ієрархічні структури даних 25
2.4.2. Маніпулювання даними 26
2.4.3. Обмеження цілісності 26
2.5. Мережева модель даних 27
2.5.1. Мережеві структури даних 27
2.5.2. Маніпулювання даними 28
2.5.3. Обмеження цілісності 29
2.6. Основні переваги та недоліки ранніх систем управління базами даних 29
2.7. Контрольні запитання 29

3. РЕЛЯЦІЙНА МОДЕЛЬ ДАНИХ ТА ЇЇ ХАРАКТЕРИСТИКИ. ЦІЛІСНІСТЬ У РЕЛЯЦІЙНІЙ МОДЕЛІ ДАНИХ 31

3.1. Реляційний підхід до представлення даних 31
3.2. Структура реляційних даних 32
3.3. Домени 32

3.4. Декартовий добуток доменів та відношень....33
3.5. Властивості відношень....34
3.6. Транзитивне замикання відношень....35
3.7. Потенційні та первинні ключі....36
3.8. Зовнішні ключі....38
3.9. Цілісність реляційних даних....39
3.9.1. Цілісність посилань та правила зовнішніх ключів....39
3.9.2. Значення NULL та підтримка цілісності посилань....40
3.10. Контрольні запитання....40

4. РЕЛЯЦІЙНА АЛГЕБРА ТА РЕЛЯЦІЙНЕ ЧИСЛЕННЯ....41

4.1. Поняття реляційної алгебри....41
4.2. Традиційні теоретико-множинні операції над відношеннями....41
4.2.1. Об'єднання....41
4.2.2. Перетин множин....42
4.2.3. Віднімання....43
4.2.4. Добуток....43
4.3. Спеціальні реляційні операції....44
4.3.1. Вибірка....44
4.3.2. Проєкція....45
4.3.3. Природне з'єднання....46
4.3.4. Θ-з'єднання....47
4.3.5. Ділення....47
4.3.6. Розширення....49
4.3.7. Підведення підсумків....49
4.3.8. Оновлення....49
4.4. Реляційне числення....50
4.5. Контрольні запитання....51

5. МОВА SQL....52

5.1. Історія створення та розвитку SQL....52
5.2. Стандарти мови SQL....53
5.3. Основні поняття мови SQL....53
5.3.1. Оператори....53
5.3.2. Назви....54
5.3.3. Типи даних....55
5.3.4. Константи....57
5.3.5. Вбудовані функції....58
5.4. Контрольні запитання....59

6. ЗАПИТИ НА ЧИТАННЯ ДАНИХ....60

6.1. Оператор SELECT....60
6.2. Мета запиту. Секція SELECT....61
6.3. Використання таблиць. Секція FROM....62

6.4. Відбір рядків. Секція WHERE 63
6.4.1. Оператори порівняння 63
6.4.2. Перевірка на належність до діапазону значень. Секція BETWEEN 64
6.4.3. Перевірка на належність до множини. Секція IN 64
6.4.4. Перевірка відповідності шаблону. Секція LIKE 65
6.4.5. Перевірка на рівність значенню. Секція IS NULL 66
6.4.6. Складові умови пошуку. Секції AND, OR і NOT 66
6.5. Контрольні запитання 67
6.6. Завдання для самопідготовки 67

7. ЗАПИТИ З ДЕКІЛЬКОМА ТАБЛИЦЯМИ 68

7.1. Природне з'єднання таблиць 68
7.2. Запити із самооб'єднання таблиць 69
7.3. Внутрішнє та зовнішнє об'єднання таблиць 72
7.4. Контрольні запитання 72
7.5. Завдання для самопідготовки 72

8. АГРЕГАТНІ ФУНКЦІЇ ТА ЗАПИТИ З ГРУПУВАННЯМ 73

8.1. Агрегатні функції 73
8.1.1. Склад агрегатних функцій 73
8.1.2. Агрегатні функції та значення NULL 74
8.2. Сортування результатів запиту. Секція ORDER BY 74
8.3. Запити з групуванням рядків. Секція GROUP BY 75
8.3.1. Підсумкові результати на основі груп рядків 75
8.3.2. Групування записів за одним стовпцем 76
8.3.3. Групування записів за кількома стовпцями 77
8.3.4. Групування записів за наборами групування. GROUPING SETS, ROLLUP, CUBE 77
8.3.5. Обмеження на запити з групуванням рядків 80
8.3.6. Значення NULL у стовпцях групування 80
8.3.7. Умови вибору груп рядків. Секція HAVING 81
8.3.8. Обмеження на умови пошуку груп рядків 81
8.3.9. Секція HAVING без GROUP BY 82
8.4. Контрольні запитання 82
8.5. Завдання для самопідготовки 82

9. ВКЛАДЕНІ ЗАПИТИ. ПІДЗАПИТИ 83

9.1. Структура та типи вкладених запитів 83
9.2. Підзапити, що утворюють стовпець 84
9.2.1. Структура підзапиту, що утворює стовпець 84
9.2.2. Порівняння з єдиним результатом вкладеного запиту 85
9.3. Запити на приналежність до множини та запити з кількісними обмеженнями 86
9.4. Перевірка належності IN множині значень вкладеного запиту 86

9.5. Перевірка факту існування. Ключове слово EXISTS 87
9.6. Багаторазове порівняння. Ключові слова ANY і ALL 91
9.6.1. Ключове слово ANY 91
9.6.2. Ключове слово ALL 93
9.6.3. Вкладені запити у секції HAVING 94
9.7. Оператори UNION, EXCEPT та INTERSECT 95
9.7.1. Оператор UNION 95
9.7.2. Оператор EXCEPT 96
9.7.3. Оператор INTERSECT 97
9.8. Вкладені підзапити, що повертають таблиці або списки 98
9.8.1. Вкладені підзапити, що повертають таблиці 98
9.8.2. Вкладені підзапити, що повертають списки 99
9.9. Директива WITH 100
9.10. Рекурсивні запити 101
9.11. Контрольні запитання 104
9.12. Завдання для самопідготовки 104

10. ВНУТРІШНЄ І ЗОВНІШНЄ З'ЄДНАННЯ ТАБЛИЦЬ. СТАНДАРТ SQL2 ..106

10.1. Природні, внутрішні та зовнішні з'єднання таблиць 106
10.2. Внутрішнє з'єднання таблиць в SQL2 107
10.3. Синтаксис оператора JOIN 108
10.4. Внутрішнє з'єднання таблиць 109
10.5. Зовнішні з'єднання таблиць 110
10.6. Перехресні з'єднання та запити на об'єднання 112
10.7. Багатотабличні об'єднання 113
10.8. Вкладені запити та об'єднання 113
10.9. Реляційна повнота SQL 113
10.10. Секція SELECT INTO 114
10.11. Мова запитів за зразком QBE 115
10.12. Контрольні запитання 116
10.13. Завдання для самопідготовки 116

11. МОВА SQL. ВНЕСЕННЯ ЗМІН ДО БАЗИ ДАНИХ 117

11.1. Внесення змін до бази даних 117
11.2. Додавання нових даних. Оператор INSERT 117
11.3. Видалення існуючих даних. Оператор DELETE 119
11.4. Оновлення існуючих даних. Оператор UPDATE 119
11.5. Контрольні запитання 120
11.6. Завдання для самопідготовки 121

12. ГЕНЕРАЦІЯ ВЕЛИКОЇ КІЛЬКОСТІ ДАНИХ У ТАБЛИЦЯХ 122

12.1. Функції генерації випадкових чисел 122
12.1.1. Випадкове десяткове число 123
12.1.2. Випадкове ціле число 123

12.1.3. Випадкове значення дати, часу, дати_часу 124
12.2. Функції, які повертають рядки значень 124
12.3. Генерація серії значень 125
12.3.1. Генерація серії чисел 125
12.3.2. Генерація серії дат 126
12.3.3. Генерація серії часу 126
12.3.4. Генерація серії дат і часу 127
12.3.5. Генерація серії псевдовипадкових чисел 128
12.3.6. Генерація псевдовипадкових рядків 128
12.4. Функції, що генерують індекси масивів 129
12.5. Криптографічна хеш-функція MD5 129
12.6. Конструктор таблиці константних значень 130
12.7. Створення та заповнення таблиці випадковими значеннями 130
12.8. Контрольні запитання 132
12.9. Завдання для самопідготовки 133

13. ПОВНОТЕКСТОВИЙ ПОШУК 134

13.1. Засоби повнотекстового пошуку PostgreSQL 134
13.2. Пошук простої відповідності 135
13.3. Конфігурування пошуку відповідності 136
13.4. Пошук у таблиці 136
13.5. Ранжування результатів пошуку 136
13.6. Виділення результатів пошуку 137
13.7. Індекси і прискорення повнотекстового пошуку 138
13.7.1. Створення та використання індексів текстових полів 138
13.7.2. Порівняння способів індексації GiST та GIN 140
13.8. Контрольні запитання 141
13.9. Завдання для самопідготовки 141

14. ВИЗНАЧЕННЯ БАЗИ ДАНИХ 142

14.1. Визначення структури даних в SQL 142
14.2. Створення бази даних. Оператор CREATE DATABASE 142
14.3. Видалення БД. Оператор DROP DATABASE 144
14.4. Зміна опису бази даних. Оператор ALTER DATABASE 144
14.5. Контрольні запитання 145
14.6. Завдання для самопідготовки 145

15. ВИЗНАЧЕННЯ ТАБЛИЦЬ 146

15.1. Створення таблиці. Оператор CREATE TABLE 146
15.2. Створення стовпців таблиці 147
15.3. Зміна таблиці. Оператор ALTER TABLE 150
15.4. Видалення таблиці. Оператор DROP TABLE 151
15.5. Визначення доменів. Оператори CREATE, ALTER та DROP DOMAIN 151
15.6. Визначення індексів. Оператори CREATE та DROP INDEX 152

15.6.1. Поняття індексів 152
15.6.2. Створення та видалення індексів 154
15.7. Контрольні запитання....155
15.8. Завдання для самопідготовки 155

16. ПРЕДСТАВЛЕННЯ 156

16.1. Поняття представлення 156
16.2. Створення представлень. Оператор CREATE VIEW 157
16.3. Оновлення представлень....157
16.4. Видалення представлень. Оператор DROP VIEW....158
16.5. Контрольні запитання....159
16.6. Завдання для самопідготовки 159

17. КУРСОРИ 160

17.1. Поняття курсора....160
17.2. Створення та використання курсора....162
17.3. Контрольні запитання....164
17.4. Завдання для самопідготовки 164

18. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ....165

18.1. Призначення та переваги збережених процедур 165
18.2. Можливості програмування....166
18.3. Безпека 166
18.4. Засоби реалізації збережених процедур у промислових СУБД 166
18.5. Контрольні запитання....167

19. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ ТА ФУНКЦІЇ В POSTGRESQL....168

19.1. Оператор Create Function мови PL/pgSQL....168
19.2. Виклик збережених функцій мови PL/pgSQL у прикладній програмі 170
19.3. Приклади збережених функцій та їх виклики....170
19.3.1. Функція без вхідних параметрів, яка повертає одне значення....170
19.3.2. Приклад функції з двома вхідними параметрами, яка не повертає значень 171
19.3.3. Приклад функції з одним вхідним параметром, яка повертає одне значення....171
19.3.4. Приклад функції з одним вхідним і двома вихідними параметрами, через які повертаються результати 172
19.3.5. Повернення таблиці. Тип TABLE 173
19.3.6. Повернення множини рядків таблиці. Тип SETOF 174
19.3.7. Приклад функції з двома вхідними параметрами, яка повертає множину елементів (записів) таблиці....175
19.3.8. Змінні типу даних стовпця таблиці. Конструкція %TYPE 175
19.3.9. Приклад функції з одним вхідним параметром, яка повертає стовпець значень таблиці....176

19.3.10. Змінні типу кортеж. Конструкція %ROWTYPE 176
19.3.11. Приклад функції з одним вхідним параметром повертає кортеж значень таблиці 177
19.3.12. Змінні типу record 178
19.3.13. Приклад функції з одним вхідним параметром, яка повертає кортеж значень таблиці 179
19.4. Видалення збереженої функції. DROP FUNCTION 180
19.5. Контрольні запитання 181
19.6. Завдання для самопідготовки 182

20. ТРИГЕРИ 183

20.1. Поняття тригера 183
20.2. Програмування тригера в PostgreSQL 183
20.3. Видалення тригера 190
20.4. Контрольні запитання 190
20.5. Завдання для самопідготовки 191

21. ІНДЕКСУВАННЯ ТАБЛИЦЬ БАЗ ДАНИХ В POSTGRESQL 192

21.1. Індекси 192
21.2. Створення та видалення індексів 192
21.2.1. Створення індексу 192
21.2.2. Видалення індексу 194
21.2.3. Поведінка та використання індексів 195
21.3. Типи індексів 196
21.3.1. B-дерева 196
21.3.2. Хеш-індекси 196
21.3.3. Індекси GiST, SP-GiST і GiN 196
21.3.4. Індекси GiST 197
21.3.5. Індекси SP-GiST 197
21.3.6. Індекси GIN 198
21.3.7. Індекси BRIN 199
21.4. Дослідження результатів пошуку за індексованим стовпчиком 199
21.5. Складені індекси 200
21.6. Контрольні запитання 200
21.7. Завдання для самопідготовки 200

22. РОЛІ. ПРИВІЛЕЇ. ДИРЕКТИВИ GRANT І REVOKE 202

22.1. Ролі 202
22.2. Привілеї та права доступу 203
22.3. Команда GRANT 203
22.4. Команда REVOKE 204
22.5. Контрольні запитання 206
22.6. Завдання для самопідготовки 206

23. СИСТЕМНИЙ КАТАЛОГ АБО СЛОВНИК ДАНИХ 207

23.1. Поняття системного каталогу/словника даних 207
23.2. Системний каталог та стандарт ANSI/ISO 207
23.3. Вміст системного каталогу 208
23.4. Інформаційна схема в стандарті SQL2 208
23.5. Офіційні стандарти мови SQL 214
23.6. Контрольні запитання 214
23.7. Завдання для самопідготовки 215

24. КЛІЄНТ/СЕРВЕРНІ ТЕХНОЛОГІЇ СУБД 216

24.1. Системи типу клієнт/сервер 216
24.2. Сервери баз даних 218
24.3. Концепція відкритих систем 219
24.4. Відкритий зв'язок із базою даних. ODBC 219
24.5. Особливості технології доступу до даних ADO 221
24.6. Особливості технології доступу до даних ADO.Net 223
24.6.1. Приєднаний та від'єднаний доступ до даних 223
24.6.2. Концепція доступу до даних в ADO.Net 224
24.6.3. Об'єктна модель ADO.Net 225
24.6.4. Класи від'єднаних об'єктів 226
24.7. Доступ до баз даних у мовах програмування 228
24.8. Доступ до бази даних в Java. JDBC 229
24.9. Доступ до бази даних в Python 231
24.10. Доступ до бази даних в C# 232
24.11. SQL і мови програмування 234
24.11.1. Взаємодія з базою даних в Java 234
24.11.2. Взаємодія з базою даних у Python 236
24.11.3. Взаємодія з базою даних в C# 237
24.11.4. Виклик збережених функцій в Java 238
24.11.5. Виклик збережених функцій у Python 240
24.11.6. Виклик збережених функцій в C# 241
24.12. Контрольні запитання 246
24.13. Завдання для самопідготовки 247

25. ТРАНЗАКЦІЇ. ПРИЗНАЧЕННЯ І ВИКОРИСТАННЯ 248

25.1. Транзакції 248
25.2. Схема управління транзакціями в SQL 249
25.3. Адміністрування транзакцій 251
25.4. Виконання транзакції 252
25.5. Журналювання транзакцій 253
25.6. Журналювання транзакцій та буферизація сторінок БД 257
25.7. ACID властивості транзакцій 258
25.8. Контрольні запитання 259

26. ПАРАЛЕЛІЗМ ТРАНЗАКЦІЙ. БЛОКУВАННЯ. ВІДНОВЛЕННЯ БАЗИ ДАНИХ 260

26.1. Проблеми паралелізму транзакцій 260
26.1.1. Неповторюване читання 260
26.1.2. Брудне читання або незафіксовані залежності 261
26.1.3. Втрачене оновлення 262
26.1.4. Фантомна вставка або проблема несумісного аналізу 263
26.2. Блокування транзакцій 264
26.3. Вирішення проблем паралелізму транзакцій 266
26.3.1. Протокол доступу до даних – протокол блокування 266
26.3.2. Брудне читання 267
26.3.3. Втрачене оновлення 269
26.4. Тупикові ситуації 269
26.5. Рівні ізоляції транзакцій 271
26.6. Визначення параметрів транзакції в SQL 272
26.7. Управління транзакціями в мовах програмування 273
26.7.1. Управління транзакціями в Java 273
26.7.2. Управління транзакціями в Python 275
26.7.3. Управління транзакціями в C# 276
26.8. Відновлення БД 278
26.8.1. Індивідуальний відкат транзакції 278
26.8.2. Відновлення після відмов системи 279
26.8.3. Відновлення після м'якого збою 280
26.8.4. Фізична узгодженість БД 281
26.8.5. Відновлення після жорсткого збою 284
26.9. Контрольні запитання 285

СПИСОК ЛІТЕРАТУРИ 286

ПРЕДМЕТНИЙ ПОКАЖЧИК 287

ДОДАТОК А. СТРУКТУРА НАВЧАЛЬНОЇ БД "ДЕКАНАТ ВНЗ" 290

ДОДАТОК Б. СТРУКТУРА БД "ПОСТАЧАННЯ" 293

ВСТУП

У цьому підручнику з дисципліни «Бази даних та засоби управління» розглядається організація, проєктування та використання баз даних (БД). Це одна з найважливіших тем, без ґрунтовного ознайомлення з якою в наш час неможливо стати кваліфікованим фахівцем в області інформаційних технологій.

Основною метою підручника є систематичне ознайомлення з ідеями та методами, що використовуються в сучасних системах управління базами даних (СУБД). Курс не охоплює якусь одну популярну СУБД, викладений матеріал однаково застосовний до будь-якої сучасної реляційної системи.

Насамперед, поставлене в підручнику завдання полягає в систематичному ознайомленні з основами реляційної моделі даних і принципами функціонування реляційних баз даних.

Розвиток автоматизованих систем обробки даних в даний час характеризується зміщенням акценту з процедурної обробки на структуру і зберігання даних, що передбачає використання їх у сфері декларативного опису та зберігання даних у базах даних. Бази даних стали невід'ємною частиною сучасних інформаційних систем. Їх основне призначення – забезпечення зберігання і обслуговування інтегрованої бази даних в системі, що є динамічною інформаційною моделлю предметної області (реального світу), в межах якої функціонує СУБД.

Необхідність ефективного доступу користувачів до бази даних, вимоги до мобільності, безпеки, цілісності інформаційних систем з урахуванням організації взаємодії з базами даних і багато інших факторів призвели до виникнення проблеми забезпечення незалежності даних від структур зберігання даних і специфіки програмного забезпечення, що використовується. Дослідження даної проблеми показує, що теорія реляційних баз даних на сьогодні є найбільш перспективною для її практичного розв’язання.

Реляційна модель заснована на теорії множин і математичній логіці. Такий фундамент забезпечує математичну строгість реляційної моделі даних.

Реляційна модель описує, які дані можуть зберігатися в реляційних базах даних і як цими даними можна маніпулювати. Вона вимагає, щоб дані організовувалися та зберігалися в так званих плоских таблицях. Точніше, користувачі та програми, які отримують доступ до даних, повинні взаємодіяти з даними так, ніби вони розміщені в таких таблицях. У спрощеному вигляді плоска таблиця – це таблиця, в якій кожну клітинку можна однозначно ідентифікувати, вказавши рядок і стовпець таблиці. Крім того, всі клітинки в одному стовпці

повинні містити дані одного простого типу. Точне визначення плоскої таблиці дається в описі реляційної моделі даних.

У свою чергу, на основі реляційної моделі були розроблені різні мови доступу до реляційних даних, такі як SEQUEL, SQL, QUEL та інші. Де-факто галузевим стандартом зараз є SQL (Structured Query Language).

Різні виробники СУБД пропонують власні реалізації мови SQL. Ці реалізації відрізняються як одна від одної, так і від стандартизованої мови SQL. Це і добре, і погано. Позитивним є те, що певна реалізація мови може включати більше можливостей, ніж стандартизований SQL, наприклад, більше типів даних, більше команд, більше додаткових опцій доступних команд. Такі особливості роблять роботу з конкретною СУБД значно ефективнішою. Крім того, такі нестандартні особливості мови перевіряються на практиці і з часом можуть бути включені в стандарт. Негативний наслідок полягає в тому, що відмінності в синтаксисі реалізацій SQL ускладнюють переміщення додатків з однієї системи в іншу. Наприклад, якщо додаток було написано для бази даних MS SQL Server з використанням власного діалекту SQL Transact-SQL, коли система передається в базу даних ORACLE, не всі мовні конструкції будуть зрозумілі відповідному діалекту SQL PL/SQL.

Взаємозв'язок між реляційною моделлю даних, стандартом SQL і різними його реалізаціями можна приблизно зобразити на рисунку 1 у вигляді наступної піраміди:

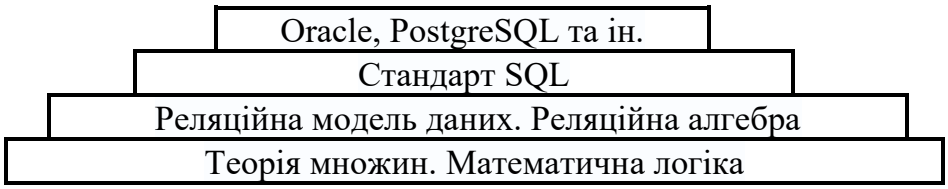


Рис. 1. Взаємозв'язок реляційної моделі даних та її реалізацій

Даний підручник, насамперед, зорієнтовано на опис та роз’яснення таких понять сучасних баз даних як реляційна модель даних, у тому числі цілісність, ключі, мова доступу до даних, клієнт-серверна архітектура програмних додатків, орієнтованих на використання реляційних баз даних, а також поняття транзакцій та спільного доступу до даних у подібних системах.

1. ПОНЯТТЯ ТА ФУНКЦІЇ СИСТЕМ УПРАВЛІННЯ БАЗАМИ ДАНИХ

1.1. Коротка історична довідка

Історично склалося так, що розвиток обчислювальних систем викликав необхідність зберігання все більшої кількості інформації в електронному вигляді. Одночасно з удосконаленням і подальшим розвитком обчислювальних систем зростав обсяг інформації, що підлягає обробці і зберіганню.

Труднощі, що виникали при вирішенні завдань структурованого зберігання та ефективної обробки все більших обсягів інформації на практиці, стимулювали дослідження у відповідних напрямках. Водночас було формалізовано задачі зберігання та обробки даних. Була створена теоретична база для вирішення завдань цього класу, реалізація яких на практиці вилилась в системи, призначені для організації обробки, зберігання і забезпечення доступу до інформації. Пізніше такі системи отримали назву *системи баз даних*.

Одночасно з розвитком систем баз даних відбувався інтенсивний розвиток комп'ютерної техніки, що використовується для роботи з великими обсягами інформації. Обчислювальні потужності і, особливо, запам'ятовуючі потужності перших обчислювальних систем були недостатніми для зберігання і обробки інформації в необхідних на практиці обсягах.

З розвитком систем баз даних змінилися принципи організації даних в них. Спочатку дані були представлені на основі ієрархічної, а пізніше мережевої моделі даних.

В кінці 1970-х – початку 1980-х років почали з'являтися перші продукти, засновані на реляційній моделі даних. В той час системи баз даних, що створювались на реляційній моделі, займали провідні позиції, незважаючи на заяви багатьох дослідників про швидкий перехід до об'єктно-орієнтованих систем. І, хоча деякі перспективні дослідження щодо об’єктно-орієнтованих моделей і систем проводились протягом декількох десятиліть тому, основною моделлю і напрямком розвитку в галузі баз даних залишається реляційна. Виходячи зі статистики, що надається відомим інтернет-ресурсом DB-Engines [1], обсяг ринку реляційних баз даних залишається на рівні 72%, що перевищує 2/3 усіх сучасних баз даних. Решта систем (приблизно 1/3 ринку) відноситься здебільшого до так званого класу нереляційних або NoSQL систем. Їх виникнення та активний розвиток протягом останнього десятиліття зумовлений експоненційним зростанням обсягів інформації, що генерується в мережі

інтернет як її користувачами, так і програмним забезпеченням. Проте, як відзначалось вище, реляційні СУБД продовжують домінувати через їхню стабільність, ефективність та можливість забезпечення надійної структури для зберігання та обробки даних.

1.2. Поняття баз даних та систем управління базами даних

Дані (Data) – це зареєстровані певним чином відомості про факти, поняття, явища тощо у формі, прийнятній для спілкування, інтерпретації, передачі або обробки людиною або за допомогою автоматизованих засобів.

База даних (*БД, DB, Database*) – це сукупність організованих та збережених певним чином даних, доступ та маніпулювання якими виконуються відповідно до визначених правил організації даних. Система управління базами даних (СУБД) – це комп'ютеризована система, основне завдання якої – зберігання інформації та забезпечення доступу до неї за вимогою.

Система управління базою даних включає в себе (рис. 1.1):

1. Дані, які безпосередньо зберігаються в базі даних.
2. Апаратне забезпечення.
3. Програмне забезпечення.
4. Користувачі:
   4.1. Прикладні програмісти.
   4.2. Кінцеві користувачі.
   4.3. Адміністратори бази даних.

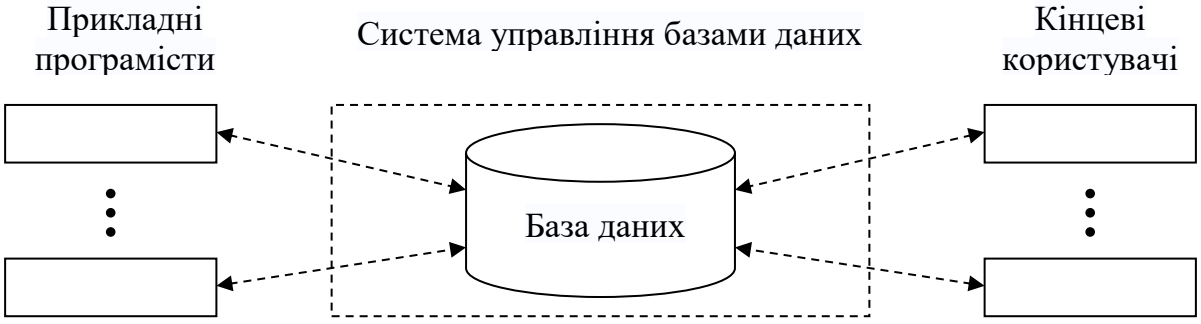


Рис. 1.1. Система БД

1.2.1. Дані

Дані в базі даних є інтегрованими і, як правило, спільними. Поняття *інтегрованих даних* фокусує увагу на можливості представлення бази даних у вигляді комбінації декількох окремих файлів даних, які не перетинаються повністю або частково.

Під *спільними даними* розуміють можливість використання окремих областей даних в базі даних декількома різними користувачами.

1.2.2. Апаратне забезпечення

Апаратне забезпечення щодо підтримки надійної та продуктивної роботи баз даних включає в себе сервери баз даних, які виконують обробку та зберігання даних, а також накопичувачі (наприклад, твердотільні та жорсткі диски) для ефективного зберігання та отримання інформації. Також важливою частиною апаратного забезпечення є високоефективні процесори та оперативна пам'ять для оптимальної роботи бази даних.

1.2.3. Програмне забезпечення

Програмне забезпечення баз даних включає власне СУБД, яка не лише забезпечує ефективне зберігання, організацію та доступ до даних, але також взаємодіє з користувачем, надаючи інструменти адміністрування, мови запитів (наприклад, SQL) та інші функціональні можливості, спрямовані на спрощення роботи з базою даних.

1.2.4. Користувачі

Прикладні програмісти відповідають за розробку прикладних програм, які використовують бази даних.

Кінцеві користувачі взаємодіють з базою даних безпосередньо через робочі станції, термінали або браузери, отримуючи доступ за допомогою відповідного прикладного програмного забезпечення.

Адміністратори СУБД – це технічні фахівці, які відповідають за створення баз даних, технічний контроль за роботою СУБД та інші операції.

Адміністратор бази даних (*Адміністратор даних*) визначає, які дані слід зберігати в базі даних, забезпечує утримання та використання цих даних. СУБД несе відповідальність за реалізацію рішень щодо управління даними.

Функції адміністратора бази даних:

1. Визначення концептуальної та логічної схем даних. Адміністратор БД визначає, які саме дані потрібно зберігати в БД, проводячи концептуальне та логічне проєктування БД за допомогою мови опису даних (МОД).
2. Визначення фізичної (внутрішньої) схеми. Адміністратор БД вирішує, як дані мають бути представлені в БД, здійснюючи фізичне

проєктування та створюючи структуру зберігання, а також визначаючи відображення між внутрішньою та концептуальною схемою.

3. Взаємодія з користувачами – визначення зовнішніх схем. Адміністратор БД надає користувачам необхідні дані, описуючи або надаючи допомогу в написанні необхідних зовнішніх схем, а також визначає відображення між зовнішньою та фізичною схемами.
4. Визначення правил безпеки та цілісності даних. Це включає в себе встановлення правил видалення чи оновлення даних, а також обмежень на допустимі значення даних.
5. Визначення процедур резервного копіювання та відновлення. Адміністратор БД вирішує, як відновлювати дані в разі пошкодження, використовуючи, наприклад, періодичне створення резервних копій. У разі пошкодження будь-якої частини БД внаслідок помилки людини, відмови обладнання або збою допоміжної операційної системи критично важливо мати можливість відновити дані з мінімальною затримкою і з найменшим впливом на решту системи. В ідеальному випадку дані, які не були пошкоджені, мають зберегтись у повному обсязі. Адміністратор БД повинен визначити і реалізувати відповідну схему відновлення, що використовує, наприклад, періодичне вивантаження («дампінг») БД на пристрій резервного копіювання і процедури, що дозволяють за необхідності завантажити БД з останньої резервної копії.
6. Управління продуктивністю та реагування на мінливі вимоги користувачів чи прикладних задач підприємства. Адміністратор БД відповідає за таку організацію системи, за якої можна отримати найбільшу для всієї організації продуктивність, а також за переналаштування системи відповідно до мінливих вимог користувачів чи програмних додатків. Наприклад, може виникнути необхідність поетапної реорганізації БД, щоб рівень продуктивності залишався прийнятним. Будь-які зміни на рівні фізичної пам'яті системи (внутрішньому рівні) повинні супроводжуватися відповідними змінами визначення відображення з концептуального рівня, тому концептуальна схема може залишатися незмінною.

База даних складається з деякого набору постійних даних, які використовуються прикладними системами підприємства. *Постійні дані* – це дані, які відрізняються від інших, більш мінливих даних, таких як проміжні дані та всі тимчасові дані в цілому. Постійні дані можуть залишатися такими недовго,

оскільки дані в базі повинні відображати інформацію про мінливі об'єкти реального світу і взаємозв'язки між ними.

Використання БД для зберігання інформації дозволяє організувати централізоване управління даними, що забезпечує наступні переваги:

1. Скорочення надмірності даних.
2. Усунення (до певної міри) суперечливості даних.
3. Загальний доступ до даних.
4. Відповідність стандартам.
5. Уведення обмежень для забезпечення безпеки даних.
6. Забезпечення цілісності даних.
7. Балансування суперечливих вимог.
8. Забезпечення незалежності даних. Оскільки програмне забезпечення відокремлюється від даних, що зберігаються БД, зміни, що вносяться в структуру БД, в більшості випадків не повинні призводити до необхідності внесення радикальних змін до програмного забезпечення.

1.3. Архітектура системи управління базами даних

Архітектура СУБД забезпечує потреби різних користувачів, виконання їх запитів, а також внутрішні потреби, пов'язані з представлення даних у зовнішній пам'яті комп'ютера та доступом до них. Загальноприйнятим на даний час є підхід, що забезпечує чотирирівневе представлення даних:

1. *Зовнішній рівень* – найбільш близький до користувачів. Він пов'язаний зі способами представлення даних відповідно до запитів окремих користувачів.
2. *Концептуальний рівень* або *змістовний рівень* даних – є проміжним між зовнішнім та внутрішнім рівнями. Цей рівень пов'язаний з узагальненими представленнями користувачів про предметну галузь на відміну від зовнішнього рівня, пов'язаного з індивідуальними представленнями користувачів. Він відповідає інтегральному погляду на дані всіх категорій користувачів БД.
3. *Внутрішній рівень* даних – найбільш близький до фізичного зберігання. Він пов'язаний із способами організації інформації на фізичних пристроях та відповідає погляду на дані прикладних програмістів та операторів БД.
4. *Фізичний рівень* даних чи власне база даних. Він пов'язаний зі способами зберігання та доступу до даних на фізичних носіях і відповідає погляду на дані адміністратора БД.

СУБД підтримує різні описи даних на всіх чотирьох рівнях представленнях даних та їх перетворення з одних видів представлення в інші.

1.4. Рівні абстракції або представлення даних в системах управління базами даних

Відповідно 4-м рівням архітектури СУБД виділяють три рівні абстракції даних в СУБД. Вони реалізуються через відповідні *представлення даних*. Іноді замість поняття представлення даних використовується поняття *модель даних* (рис. 1.2)

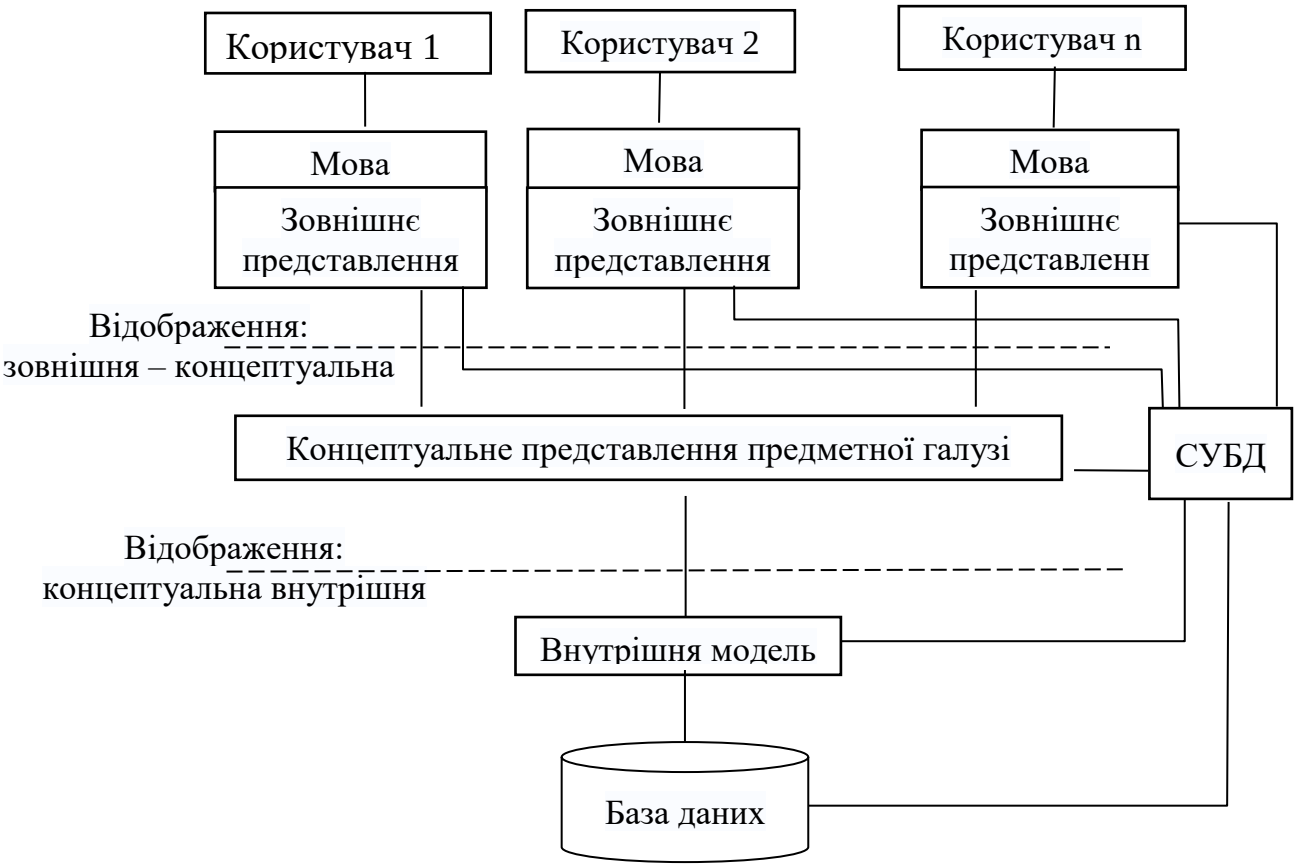


Рис. 1.2. Архітектура систем з БД

1.4.1. Зовнішній рівень – зовнішнє представлення

Зовнішній рівень – індивідуальний рівень користувача. Користувач може бути як прикладним програмістом, так і кінцевим користувачем з будь-яким рівнем професійної підготовки. Кожен користувач має свою мову спілкування з СУБД. Для програміста – це будь-яка мова програмування, для користувача – мова запитів або мова, заснована на формах та меню. Будь-яка з цих мов включає підмову даних, тобто множину операторів всієї мови, пов'язану лише з об'єктами

та операціями БД. Таким чином, підмову даних вбудовано в базову мову користувача, яка також забезпечує пов'язані з БД можливості.

Представлення окремого користувача БД на зовнішньому рівні архітектури називають зовнішнім представленням. Таким чином, зовнішнє представлення даних – це вміст БД, яким його бачить окремий користувач. Наприклад, співробітник відділу кадрів бачить БД як набір записів про співробітників плюс набір записів про підрозділи. У загальному випадку зовнішнє представлення складається з множини екземплярів кожного типу зовнішнього запису, які не обов'язково збігаються із записами, що зберігаються. Підмову даних користувача визначено у термінах зовнішніх записів. Кожне зовнішнє представлення визначається засобами зовнішньої схеми, яка в основному складається з визначень кожного типу записів у зовнішньому уявленні.

1.4.2. Концептуальний рівень – концептуальне представлення

Концептуальне представлення – це представлення всієї інформації БД в термінах і поняттях моделі предметної області (ПО), наприклад в таких змістовних поняттях як студент, викладач, дисципліна, тощо. Воно є абстрактною формою порівняно з фізичним способом зберігання даних.

Концептуальне представлення дані якими вони є насправді і в термінах і поняттях, притаманних конкретній проблемній області, а не такими, якими їх повинен бачити користувач на зовнішньому рівні у межах певної мови програмування. Концептуальне представлення складається з множини екземплярів кожного типу концептуального запису, хоча в деяких системах способи концептуального представлення даних можуть бути іншими – наприклад, у вигляді об'єктів та зв'язків між ними.

Концептуальне представлення визначається засобами концептуальної схеми даних, яка описується за допомогою концептуальної мови визначення даних. Концептуальна схема складається з визначень кожного типу концептуальних записів. Ці визначення стосуються лише змісту інформації, а не способу її зберігання. Таким чином, концептуальне представлення забезпечує незалежність даних від способу їх зберігання.

1.4.3. Внутрішній рівень – внутрішнє представлення

Внутрішнє представлення – це представлення нижнього рівня представлень всієї БД, що відображає погляд всіх категорій користувачів на організацію даних. Внутрішнє представлення складається з множини

екземплярів кожного типу внутрішнього запису, наприклад таблиці із записами одного типу, та способи доступу до них – рядками та стовпцями. Внутрішній запис відповідає запису, що зберігається.

Внутрішнє представлення не пов'язане з фізичним рівнем представлення й у ньому не розглядаються фізичні записи. Внутрішнє представлення передбачає існування нескінченного лінійного адресного простору, в якому розміщуються дані та виконується їх організація. Деталі відображення цього простору на фізичні пристрої зберігання не включено в загальну архітектуру СУБД через сильну залежність від конкретних рішень в конкретних СУБД.

Внутрішнє представлення описується за допомогою внутрішньої схеми, яка описується за допомогою внутрішньої мови визначення даних (МВД).

Між трьома рівнями представлень є два рівня відображень. Відображення концептуального рівня на внутрішній, а також відображення зовнішнього рівня – на концептуальний. Відображення між різними рівнями зберігають незалежність даних у разі внесення змін до структури БД на якомусь окремому рівні.

1.5. Функції систем управління базами даних

1. *Визначення даних*. СУБД повинна допускати визначення даних – зовнішні визначення, концептуальну і внутрішню схеми, а також відповідні відображення. Для цього СУБД включає мовний процесор для різних мов визначення даних (МВД).
2. *Обробка даних.* СУБД повинна обробляти запити користувача на вибірку та оновлення даних. Для цього СУБД включає компоненти процесора МВД.
3. *Безпека та цілісність даних*. СУБД повинна контролювати запити і припиняти спроби порушення правил безпеки та цілісності даних.
4. *Архівація даних та деархівація даних*. СУБД повинна надавати можливість по стислому зберіганню попередніх версій даних та їх відновлення в разі необхідності.
5. *Відновлення даних та дублювання даних*. СУБД має забезпечити відновлення даних після збоїв.
6. *Підтримка транзакцій*. Транзакції забезпечують можливість виконувати окрему групу операцій як одне ціле або не виконувати її взагалі.
7. *Словник даних*. СУБД має забезпечити функцію словника даних або системного каталогу. Сам словник можна вважати системною БД, що

містить дані про дані користувацької БД, тобто містить визначення інших об'єктів системи. Словник інтегровано в обумовлену ним БД і тому містить опис самого себе. Він ведеться безпосередньо СУБД.

8. *Продуктивність.* СУБД має виконувати свої функції з максимальною продуктивністю, яка задовольняє виробничим вимогам конкретних користувачів.

1.6. Експертні системи та бази знань

Останнім часом з'явилась необхідність зберігання та використання слабкоструктурованих даних, якими є, наприклад, людські знання в експертних системах.

Експертна система – система штучного інтелекту, що включає знання про певну слабо структуровану і важко формалізовану вузьку ПО і здатна пропонувати і пояснювати користувачеві розумні рішення. Експертна система складається з бази знань, механізму логічного висновку та підсистеми пояснень.

База знань – семантична (змістовна) модель, що описує ПО і дозволяє відповідати на такі питання з цієї ПО, відповіді які у явному вигляді немає у базі. База знань є основним компонентом інтелектуальних та експертних систем.

Для зберігання баз знань у сучасних експертних системах використовуються або промислові СУБД і проміжне спеціалізоване програмне забезпечення, або повністю спеціалізоване програмне забезпечення.

У цьому підручнику основна увага приділяється проєктуванню БД та організації зберігання у них структурованої інформації.

1.7. Контрольні запитання

1. Дайте визначення поняття БД.
2. Дайте визначення поняття СУБД.
3. Перелічіть складові та зв'язки архітектури СУБД.
4. Охарактеризуйте рівні абстракції або представлення даних до СУБД.
5. Зовнішній, концептуальний та внутрішній рівні представлення даних.
6. Перелічіть основні функції СУБД.

2. МОДЕЛІ БАЗ ДАНИХ

2.1. Моделі даних. Дореляційні системи управління базами даних

2.2. Ранні чи дореляційні системи управління базами даних

Розглянемо системи, що історично передували реляційним, це необхідно для правильного розуміння причин масового переходу до реляційних систем управління даними. Крім того, внутрішня організація реляційних систем багато в чому ґрунтується на використанні методів ранніх систем. І, нарешті, деяке знання в галузі ранніх систем буде корисним для розуміння шляхів розвитку *постреляційних СУБД.*

Обмежимося розглядом лише загальних підходів до організації трьох типів ранніх систем, а саме систем, заснованих на *інвертованих списках, ієрархічних та мережевих СУБД.*

Розглянемо деякі найбільш загальні характеристики ранніх систем:

1. Ці системи активно використовувалися протягом багатьох років, довше, ніж використовується деякі з сучасних реляційних СУБД. Насправді, деякі з ранніх систем використовуються навіть у наш час, оскільки в них накопичено величезні обсяги даних і однією з актуальних проблем інформаційних систем є використання цих систем спільно з сучасними.
2. Усі ранні системи не ґрунтувалися на будь-яких абстрактних моделях. Поняття моделі даних фактично узвичаїлося фахівцями у галузі БД лише разом із реляційним підходом до моделювання даних. Абстрактні представлення ранніх систем з'явилися пізніше на основі аналізу та виявлення загальних ознак у конкретних системах.
3. У ранніх системах доступ до БД здійснювався лише на рівні фізичних записів. Користувачі цих систем здійснювали явну навігацію у БД, використовуючи мови програмування, розширені функціями СУБД. Інтерактивний доступ до БД підтримувався лише шляхом створення відповідних прикладних програм із власним інтерфейсом.
4. Навігаційна природа ранніх систем та доступ до даних на рівні записів змушували користувача власноруч виконувати оптимізацію доступу до БД без підтримки системи.
5. Після появи реляційних систем в більшість ранніх систем почали додавати реляційні інтерфейси. Однак у більшості випадків це не зробило їх по-справжньому реляційними системами, оскільки

залишалась можливість маніпулювати даними в природному для них режимі.

2.3. Системи, що базуються на інвертованих списках

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

Організація доступу до даних на основі інвертованих списків індексів використовується практично у всіх сучасних реляційних СУБД, але в цих системах користувачі не мають безпосереднього доступу до інвертованих списків.

2.3.1. Структури даних

У БД, організованої з допомогою інвертованих списків, таблиці та шляхи доступу до них видимі для користувачів. При цьому:

1. Рядки таблиць упорядковані системою у певній фізичній послідовності.
2. Для кожної таблиці можна визначити довільну кількість ключів пошуку, за якими будуються індекси. Ці індекси автоматично підтримуються системою, але явно видимі користувачам.

2.3.2. Маніпулювання даними

Підтримуються два класи операторів:

1. Оператори, що встановлюють адресу запису, серед яких:
   1.1. Прямі пошукові оператори (наприклад, знайти перший запис таблиці за деяким шляхом доступу).
   1.2. Оператори, що знаходять запис у термінах відносної позиції від попереднього запису за деяким шляхом доступу.
2. Оператори над записами, що адресуються. Типовий набір операторів:
   2.1. Знайти перший запис таблиці T у фізичному порядку.
   2.2. Знайти перший запис таблиці T із заданим значенням ключа пошуку K.
   2.3. Знайти запис, який слідує за записом із заданим адресом по заданому шляху доступу.
   2.4. Знайти наступний запис таблиці T у порядку шляху пошуку із заданим значенням ключа K.

2.5. Знайти перший запис таблиці T у порядку ключа пошуку K зі значенням ключового поля, більшим за задане значення K.
2.6. Вибрати запис із зазначеним адресом.
2.7. Оновити запис із зазначеним адресом.
2.8. Видалити запис із зазначеним адресом.
2.9. Включити запис до зазначеної таблиці; операція генерує адрес запису.

2.3.3. Обмеження цілісності

Загальні правила визначення цілісності БД відсутні. У деяких системах підтримуються обмеження унікальності значень деяких полів, але переважно все покладається на прикладну програму.

2.4. Ієрархічна модель даних

Типовим представником є Information Management System (IMS) фірми IBM. Перша версія з'явилась у 1968 р. Досі підтримується багато БД з цією моделлю, що створює суттєві проблеми з переходом на нову технологію БД.

2.4.1. Ієрархічні структури даних

Ієрархічна БД складається з впорядкованого набору дерев – точніше, з впорядкованого набору кількох екземплярів одного типу дерева.

Тип дерева (рис. 2.1) складається з одного кореневого типу запису та впорядкованого набору з нуля або більше типів піддерев, кожне з яких є деяким типом дерева. Тип дерева загалом є ієрархічно організованим набором типів записів.

Рис. 2.1. Приклад типу дерева – схеми ієрархічної БД

Тут вузол Група є батьківським вузлом для вузлів Куратор та Студенти, а вузли Куратор та Студенти є нащадками вузла Група. Між типами записів підтримуються позиційні зв'язки.

Запис БД з такою схемою міг би виглядати наступним чином (рис. 2.2):

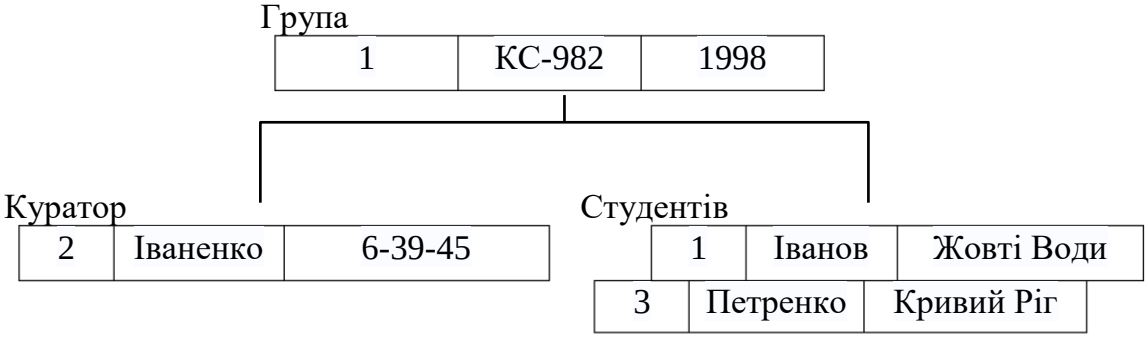


Рис. 2.2. Один екземпляр дерева

Усі екземпляри даного типу нащадку із загальним екземпляром типу батьківського елементу називаються *близнюками*. Для БД визначено повний порядок обходу дерева – зверху-вниз і зліва-направо.

2.4.2. Маніпулювання даними

Прикладами типових операторів маніпулювання ієрархічно організованими даними можуть бути:

1. Знайти вказане дерево БД (наприклад, відділ 310).
2. Перейти від одного дерева до іншого.
3. Перейти від одного запису до іншого всередині дерева, наприклад від відділу – до першого співробітника.
4. Перейти від одного запису до іншого у порядку обходу ієрархії.
5. Вставити новий запис у вказану позицію.
6. Видалити поточний запис.

2.4.3. Обмеження цілісності

У ієрархічних БД автоматично підтримується цілісність посилань між батьківськими вузлами та нащадками. Основне правило – жоден нащадок не може існувати без свого батька. Зауважимо, що аналогічна підтримка цілісності за посиланнями між записами, що не входять до однієї ієрархії, не підтримується.

Прикладом такого "зовнішнього" посилання може бути вміст поля К_номер в екземплярі типу запису Куратор.

У ієрархічних системах підтримувалась деяка форма представлень БД з урахуванням обмеження ієрархії. Прикладом представлення БД з прикладу вище може бути ієрархія, зображена на рис.2.3.

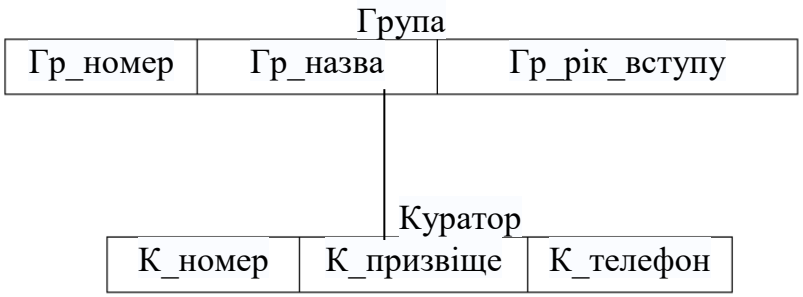


Рис. 2.3. Приклад представлення ієрархічної БД

2.5. Мережева модель даних

Типовим представником є Integrated Database Management System (IDMS) компанії Cullinet Software, Inc., призначена для використання на машинах основного класу IBM під управлінням більшості операційних систем.

Архітектуру системи засновано на пропозиціях DataBase Task Group (DBTG) та Комітету з мов програмування систем даних Conference on Data Systems Languages (CODASYL), організації, що відповідала за створення мови програмування Кобол. Звіт DBTG було опубліковано 1971 р., а у 70-х роках з'явилось декілька систем, серед яких була і IDMS.

2.5.1. Мережеві структури даних

Мережевий підхід до організації даних є розширенням ієрархічного. У ієрархічних структурах запис-нащадок повинен мати точно одного предка; у мережевий структурі даних нащадок може мати будь-яку кількість предків.

Мережева БД складається з набору екземплярів кожного типу запису та набору екземплярів кожного типу зв'язку (рис. 2.4)

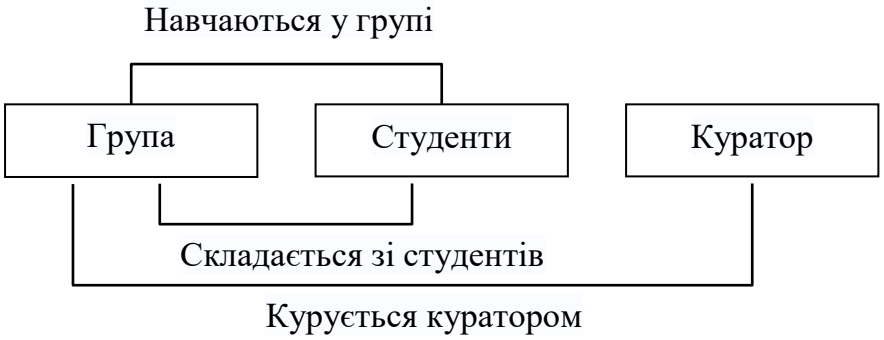


Рис. 2.4. Простий приклад мережної схеми БД

Тип зв'язку визначається для двох типів запису – предку та нащадку. Примірник типу зв'язку складається з одного екземпляра типу запису предка та впорядкованого набору екземплярів типу запису нащадка. Для цього типу зв'язку L з типом запису предка P і типом запису нащадка C повинні виконуватися такі дві умови:

1. Кожен екземпляр типу P є предком лише одного екземпляру L.
2. Кожен екземпляр C є нащадком більше, ніж одного екземпляру L.

На формування типів зв'язку не накладаються особливих обмежень. Можливі, наприклад, такі ситуації:

1. Тип запису нащадка в одному типі зв'язку L1 може бути типом запису предка в іншому типі зв'язку L2 (як ієрархії).
2. Даний тип запису P може бути типом запису предка у будь-якій кількості типів зв'язку.
3. Даний тип запису P може бути типом запису нащадка у будь-якій кількості типів зв'язку.
4. Може існувати будь-яка кількість типів зв'язку з одним і тим самим типом запису предка і тим самим типом запису нащадка; і якщо L1 і L2 – два типи зв'язку з одним і тим самим типом запису предка P і тим самим типом запису нащадка C, то правила, якими утворюється успадкування, у різних зв'язках можуть відрізнятися.
5. Типи запису X і Y можуть бути предком і нащадком в одному зв'язку і нащадком і предком – в іншому.
6. Предок і нащадок можуть бути одного типу запису.

2.5.2. Маніпулювання даними

Типовий набір операцій з маніпулювання даними може бути наступним:

1. Знайти конкретний запис у наборі однотипних записів (інженера Петренка).
2. Перейти від предка до першого нащадка за деяким зв'язком (до першого співробітника відділу 310).
3. Перейти до наступного нащадка у зв'язку (від Петренка до Іваненка).
4. Перейти від нащадка до предка за деяким зв'язком (знайти відділ, де працює Петренко).
5. Створити новий запис.
6. Вилучити запис.
7. Модифікувати запис.
8. Включити у зв'язок.
9. Виключити зі зв'язку.
10. Переставити в інший зв'язок тощо.

2.5.3. Обмеження цілісності

В загальному випадку підтримка цілісності не є обов’язковою, але іноді потрібна цілісність за посиланнями (як в ієрархічній моделі).

2.6. Основні переваги та недоліки ранніх систем управління базами даних

Ключовими аспектами ранніх СУБД є наступні:

1. Розвинені засоби управління даними у зовнішній пам'яті на низькому рівні.
2. Можливість побудови вручну ефективних прикладних систем.
3. Можливість економії пам'яті з допомогою поділу підоб'єктів (в мережевих системах).

Основними недоліками ранніх СУБД є:

1. Занадто складність користування.
2. Фактично необхідні знання про фізичну організацію даних.
3. Прикладні системи залежать від конкретної організації даних і перехід до систем з іншою організацією даних вимагає суттєвих переробок програми.
4. Логіка таких СУБД перевантажена деталями організації доступу до БД.

2.7. Контрольні запитання

1. Дайте визначення поняття моделі даних.

2. Дайте визначення та перелічіть основні характеристики систем, що базуються на інвертованих списках.
3. Дайте визначення та перелічіть основні характеристики ієрархічної моделі даних.
4. Дайте визначення та перелічіть основні характеристики мережевої моделі даних.
5. Перелічіть основні операції з маніпулювання даними в ієрархічній моделі даних.
6. Перелічіть основні операції маніпулювання даними в мережевій моделі даних.
7. Перелічіть основні операції маніпулювання даними в системах, які ґрунтуються на інвертованих списках.

3. РЕЛЯЦІЙНА МОДЕЛЬ ДАНИХ ТА ЇЇ ХАРАКТЕРИСТИКИ. ЦІЛІСНІСТЬ У РЕЛЯЦІЙНІЙ МОДЕЛІ ДАНИХ

3.1. Реляційний підхід до представлення даних

У 1980-х років реляційні системи практично витіснили зі світового ринку ранні СУБД і сьогодні є найпоширенішими. Першою експериментальною реляційною СУБД, яка підтвердила переваги реляційного підходу, була система System-R компанії IBM.

До переваг реляційного підходу до організації БД можна віднести:

1. Наявність невеликого набору інтуїтивно зрозумілих абстракцій, які дозволяють порівняно просто моделювати більшу частину поширених предметних областей (ПО) та допускають точні формальні визначення.
2. Наявність простого і потужного математичного апарату, що спирається головним чином на теорію множин і математичну логіку.
3. Можливість ненавігаційного маніпулювання даними без необхідності знання конкретної фізичної організації БД у зовнішній пам'яті.
4. Єдиний спосіб представлення даних – об'єктів реального світу, та зв'язків між ними.

Нині основним предметом критики реляційних СУБД є:

1. Притаманна цим системам певна обмеженість (прямий наслідок простоти) під час використання у нетрадиційних областях (наприклад, системи автоматизації проєктування), у яких потрібні складніші структури даних.
2. Відображення семантики ПО. Інакше кажучи, можливості представлення семантичної специфіки ПО у реляційних системах значно обмежені. Сучасні дослідження у сфері постреляційних систем переважно присвячені саме усуненню цих недоліків.

У реляційній моделі розглядаються 3 аспекти даних [2]:

1. Структура даних.
2. Цілісність даних.
3. Обробка даних.

3.2. Структура реляційних даних

Структура реляційних даних представлена на рис. 3.1.

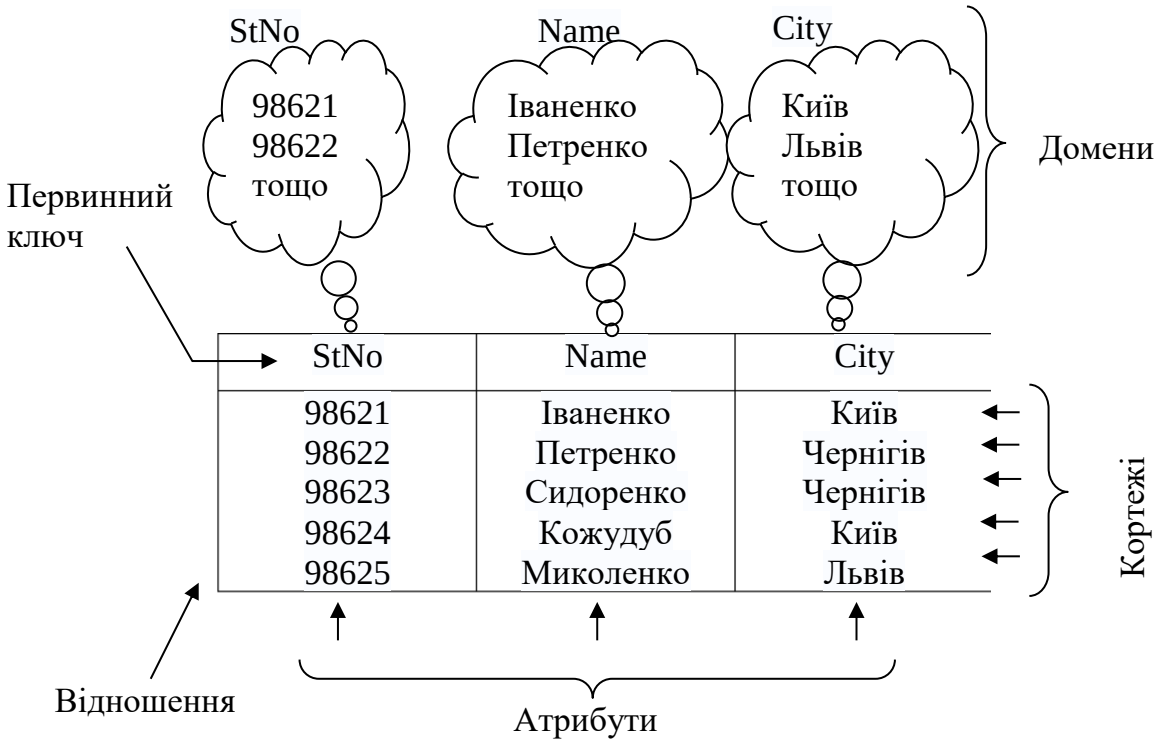


Рис. 3.1. Відношення

Як елементи цієї структури можна виділити домени, атрибути, кортежі та первинні ключі. Кількість кортежів у відношенні називається його *кардинальним числом*.

Нижче розглядається відповідна термінологія, що застосовується у рамках аспекту *структура реляційних даних*.

3.3. Домени

Домен (Domain) – іменована множина скалярних значень одного типу. Наприклад, домен «місто» – це множина усіх можливих назв міст. Те, що значення домену в конкретному випадку розглядається як скалярне (атомарне, просте), насправді не означає, що воно не має внутрішньої структури. Наприклад, при проєктуванні БД в одному випадку може бути прийнято рішення розглядати поєднання *Прізвище – Ім'я – По-батькові* як скалярне значення домену, а в іншому – як три окремі домени *Прізвище, Ім'я, По-батькові*.

Домени є загальними сукупностями значень, з яких беруться *реальні* значення атрибутів. Зазвичай, у будь-який момент часу в домені є значення, які не відповідають значенням жодного з атрибутів з цього домену.

Основне призначення доменів полягає у обмеженні порівняння їх значень. Порівняння матиме сенс для атрибутів, що належать одному домену. Наприклад, не можна порівнювати числовий код студента та стипендію студента. Хоча й те й інше цілі числа, проте таке порівняння не має сенсу.

Домени, насамперед, мають концептуальне призначення. Вони можуть бути або не бути явно збережені в БД як реальні набори значень, але вони повинні бути специфіковані в рамках визначень БД, завдяки чому буде відомо які атрибути можна порівнювати, а які – ні.

3.4. Декартовий добуток доменів та відношень

Прямий або *декартовий* добуток двох доменів (множин) – це множина, елементами якої є усі впорядковані пари елементів вихідних доменів.

Нехай задано дві множини X і Y. Декартовий добуток множини X і множини Y є така множина X*Y, елементами якої є впорядковані пари <x, y> для будь-яких елементів $x \in X$ та $y \in Y$.

Приклад. Нехай у межах університету задано 2 домени – *Студенти* та *Дисципліни*. Тоді декартовим добутком цих доменів буде множина пар – кортежів виду <студент дисципліна>

<Іваненко математика>

< Іваненко програмування>

…

<Петренко математика>

<Петренко програмування>

тощо.

Декартовий добуток n множин X, Y, …, Z є така множина X*Y…*Z, елементами якої є впорядковані n-ки – кортежі виду <x, y, …,z> для будь-яких $x \in X$, $y \in Y$, …, $z \in Z$. Число n називається *ступенем декартової множини*.

Декартовий добуток є засобом конструювання нових об'єктів. Але з прикладу видно, що отримані з його допомогою потенційні об'єкти не завжди відповідають реаліям навколишнього світу. Так, студенти напряму підготовки «Комп'ютерна інженерія» не вивчають дисципліни напряму підготовки «Економіка».

Тому, крім декартового добутку, вводиться поняття відношення.

Відношення R, визначене на декартовому добутку DM, є підмножиною істинних кортежів множини кортежів DM.

Таким чином, в рамках наведеного прикладу відношення утворюють тільки ті кортежі, які для кожного студента відповідають дисциплінам, що ним реально вивчаються.

Примітка. Слід підкреслити, що відношення є математичним аналогом таблиць, а кортежі – аналогом рядків цих таблиць. Таблиці ж є основним елементом сучасних реляційних баз даних.

3.5. Властивості відношень

Відношення R, визначене на множині не обов'язково різних доменів D_1, D_2, ..., D_n, містить дві частини – *заголовок* і *тіло*:

1. Заголовок відношення містить фіксовану множину атрибутів, а точніше, пар

<назва_атрибута: назва_домену>: {<A_1:D_1>, <A_2:D_2>, …, <A_n:D_n>},

Усі назви атрибутів A_1, A_2, …, A_n відношення різні. Кожен атрибут A_j відповідає одному і лише одному, з лежачих в його основі з доменів D_j (j=1,2, …, n).

2. Тіло відношення складається з множини кортежів. Кожен кортеж, у свою чергу, містить множину пар

<назва атрибута: значення атрибута>: {<A_1:V_{i1}>, <A_2:V_{i2}>, …, <A_n:V_{in}>},

де i = 1, 2, …, m, де m – кількість кортежів у цій множині. Для будь-якої пари <A_j:V_{ij}> елемент V_{ij} є значенням з унікального домену D_j, який пов'язаний з атрибутом A_j.

Значення m і n називаються відповідно *кардинальним числом* та *ступенем* відношення R.

До основних властивостей відношень належить:

1. У відношенні відсутні однакові кортежі. Ця властивість випливає з того факту, що тіло відношення сприймається як математична множина кортежів. Наведена властивість служить ілюстрацією різниці між відношенням і таблицею оскільки таблиця, як правило, може містити однакові рядки.

Важливим наслідком того, що у відношення немає однакових рядків, є те, що у відношення завжди існує *первинний ключ*.

2. Кортежі не впорядковуються зверху вниз. Ця властивість також випливає з того, що тіло відношення розглядається як математична множина. Таким чином, відношення і таблиця – це не одне й те саме, тому що рядки таблиці впорядковані зверху вниз, а кортежі відношення – ні.

3. Атрибути не впорядковані зліва направо. Ця властивість випливає з того, що заголовок відношення визначається як множина атрибутів. Отже, відношення і таблиця – це не одне й те саме, оскільки стовпці в таблиці впорядковані зліва направо.
4. Усі значення атрибутів атомарні (скалярні). Ця властивість є наслідком того, що всі домени, що лежать в основі атрибутів, містять тільки атомарні значення.

Відношення, в якому всі значення атрибутів атомарні, називається нормалізованим або представленим у першій нормальній формі (1НФ).

Примітка. З властивостей відношення випливає, що не кожна таблиця може задавати відношення. Для того, щоб деяка таблиця задавала відношення, необхідно, щоб таблиця мала просту структуру (містила б тільки рядки та стовпці, при чому, у кожному рядку була б однакова кількість полів), у таблиці не повинно бути однакових рядків, будь-який стовпець таблиці повинен містити дані тільки одного типу, а всі типи даних, що використовуються, повинні бути атомарними.

3.6. Транзитивне замикання відношень

Введемо поняття транзитивного замикання, яке пов'язане з бінарними відношеннями.

Нехай відношення R задано на декартовому добутку деякої множини A. Транзитивним замиканням відношення R називається нове відношення RR, що складається з кортежів <x, y>, для яких виконуються наступні умови:

1. або кортеж <x, y>$\in$R,
2. або існує кінцева послідовність елементів z_1, z_2, … , z_n, така, що всі кортежі <x, z_1>, <z_1, z_2>, … , <z_n, y> належать відношенню RR.

Очевидно, що всі кортежі відношення R є підмножиною кортежів відношень RR, тобто R$\subseteq$RR.

Пояснимо поняття транзитивного замикання у вигляді прикладу. Нехай множина A являє собою таку множину деталей та конструкцій:

A = {Болт, Гайка, Двигун, Автомобіль, Колесо, Вісь}

при чому деякі з деталей та конструкцій можуть використовуватися при складанні інших конструкцій, наприклад, гайка використовується при складанні колеса. Взаємозв'язок деталей описується відношенням "Безпосередньо використовується в" і складається з кортежів, наведених у таблиці 3.1.

Таблиця 3.1. Відношення R

Конструкція	Де використовується
Болт	Двигун
Болт	Колесо
Гайка	Двигун
Гайка	Колесо
Двигун	Автомобіль
Колесо	Автомобіль
Вісь	Колесо

Транзитивне замикання RR складається з наступних кортежів наведених у таблиці 3.2:

Таблиця 3.2. Відношення RR

Конструкція	Де використовується
Болт	Двигун
Болт	Колесо
Гайка	Двигун
Гайка	Колесо
Двигун	Автомобіль
Колесо	Автомобіль
Вісь	Колесо
Болт	Автомобіль
Гайка	Автомобіль
Вісь	Автомобіль

Очевидний сенс замикання RR полягає в описі включення деталей одна в одну не тільки безпосередньо, але і через їх використання в проміжних деталях, наприклад, болт використовується в автомобілі, оскільки він використовується у двигуні, а двигун використовується в автомобілі.

3.7. Потенційні та первинні ключі

Нехай R – деяке відношення. Тоді *потенційний ключ* K для відношення R – це підмножина множини атрибутів цього відношення R, що має наступні властивості:

1. Властивість унікальності – немає двох різних кортежів у відношенні R з однаковим значенням K.
2. Властивість не надмірності – жодна з підмножин ключа K не має властивості унікальності.

Розглядаючи потенційні ключі, слід звернути увагу на наступне:

1. На практиці більшість відношень мають лише один потенційний ключ, хоча в цілому їх може бути декілька.
2. Потенційні ключі визначені як множини атрибутів. Потенційний ключ що складається з одного атрибута, називається простим, а з кількох атрибутів – складовим.
3. Потенційний ключ не повинен бути надмірним. Якщо визначити потенційний ключ надмірним, системі не буде відомо про це і вона не зможе належним чином забезпечити відповідне обмеження цілісності. Наприклад, щодо відношення Students потенційним ключем є унікальний код студента StdNmb. Однак можна визначити надмірний потенційний ключ, що складається з коду студента StdNmb та назви міста, в якому він проживає, City. У такому разі система не зможе дотримуватись обмеження, що забезпечує унікальність коду студента в глобальному сенсі, натомість буде виконуватися більш слабке обмеження, що забезпечує унікальність коду студента в межах міста.

Причина важливості потенційних ключів у тому, що вони забезпечують основний механізм адресації на рівні кортежу в реляційній системі. Єдиний спосіб точно вказати на будь-який кортеж – це вказати точне значення потенційного ключа.

В принципі, відношення може мати більше одного потенційного ключа. У цьому випадку в реляційній моделі один з них вибирається як первинний ключ (primary key) в базовому відношенні, а інші потенційні ключі, якщо вони є, будуть називатися альтернативними ключами. Наприклад, альтернативними ключами для студента може бути його ідентифікаційний код платника податків StdIDCode або серія та номер паспорта StdPaspNo.

Якщо множина потенційних ключів містить більше одного елемента, вибір первинного ключа, у загальному випадку, може вимагати глибшого аналізу. Так, якщо як первинний ключ обрано унікальний код студента StdNmb, що відповідає номеру студентського квитка, то у разі втрати студентського квитка або зміні прізвища студента йому видається дублікат студентського квитка з тим же номером. А це не потребує жодних коригувань у БД. Крім того, код студента StdNmb може нести додаткову інформацію, корисну в рамках навчального закладу, наприклад, рік вступу.

Якщо в якості первинного ключа вибрано серію та номер паспорта StdPaspNo, то є ризик не врахувати ситуацію, коли студент з якихось причин (втратив паспорт або змінив прізвище) змінює паспорт, і, отже, змінюється первинний ключ – номер паспорта. А це при неправильному проєктуванні БД

може призвести до порушення цілісності посилання, оскільки первинний ключ може змінитися, а відповідні зовнішні ключі можуть залишитися незмінними. Щоправда, в сучасних СУБД можна попередити подібне порушення цілісності, ввівши при проєктуванні бази даних каскадну заміну зовнішніх ключів, проте номер паспорта не містить специфічної інформації для навчального закладу.

3.8. Зовнішні ключі

Нехай R_1 – базове відношення. Тоді зовнішній ключ (foreign key) – FK щодо R_2 – це підмножина множини атрибутів R_2 така, що:

1. Існує базове відношення R_1 з потенційним ключем PK.
2. Кожне значення FK у поточному значенні R_2 завжди збігається зі значенням PK деякого кортежу у поточному значенні R_1.

Варто зазначити, що:

1. Зовнішні ключі, як і потенційні, визначені як множина атрибутів.
2. Даний зовнішній ключ буде складовим тоді і тільки тоді, коли відповідний потенційний ключ також буде складним. Він буде простим тоді і тільки тоді, коли відповідний потенційний ключ також буде простим.
3. Кожен атрибут, що входить до цього зовнішнього ключа, повинен бути визначений на тому ж домені, що і відповідний атрибут відповідного потенційного ключа.
4. Значення зовнішнього ключа представлено посиланням на кортеж, що містить відповідне значення потенційного ключа (посилальний кортеж або цільовий кортеж). Проблема забезпечення того, що БД не включає жодних неправильних значень зовнішніх ключів, відома як проблема *цілісності посилань*. Обмеження, згідно з яким значення даного зовнішнього ключа повинні бути адекватними значенням відповідного потенційного ключа, називають *обмеженням посилань*.
5. Відношення R_1 та R_2 не обов'язково різні. Тобто відношення може посилатися на самого себе. Такі відношення іноді називають тими, що самопосилаються (самореференціями).
6. Відношення самопосилань є окремим випадком, коли можуть виникнути посилальні цикли, коли останнє відношення самопосилання в ланцюжку, що посилається, посилається на перше.

3.9. Цілісність реляційних даних

У будь-який момент часу будь-яка база даних містить певну конфігурацію значень даних, і вважається, що це конфігурація відображає реальність – тобто є знаковою моделлю частини реального світу чи моделлю предметної області. Довільна конфігурація значень даних немає сенсу, якщо значення у цій конфігурації не представляють певного стану реального світу.

Реально поняття цілісності даних пов'язані з правилами оновлення даних – вставки, видалення та заміни даних. Іноді до цілісності даних відносять набори або діапазони допустимих значень, хоча в багатьох випадках для цього використовують термін несуперечності даних.

Виходячи зі сказаного вище, визначення БД потребує розширення, що включає правила цілісності, призначення яких у тому, щоб інформувати СУБД про різні обмеження реального світу.

Основним компонентом тієї частини реляційної моделі, що стосується цілісності, виступають два особливі правила цілісності. Ці правила відносяться до потенційних та первинних ключів, а також до зовнішніх ключів.

3.9.1. Цілісність посилань та правила зовнішніх ключів

БД не повинна містити неузгоджених значень зовнішніх ключів. Неузгоджене значення зовнішнього ключа – це таке значення зовнішнього ключа, для якого не існує відповідного йому значення відповідного потенційного ключа у відповідному базовому відношенні.

Для того, щоб уникнути некоректних станів для кожного зовнішнього ключа, необхідно встановити правила, на підставі яких СУБД діятиме при спробі:

1. Видалити об'єкт (кортеж), на який посилається зовнішній ключ;
2. Оновити потенційний ключ, на який посилається зовнішній ключ.

Для кожного з цих випадків можна передбачити щонайменше дві можливості:

1. При спробі видалити об'єкт, на який посилається зовнішній ключ:
    1.1. Обмежити – призупинити операцію видалення, до моменту, коли не будуть видалені всі об'єкти, що посилаються на нього, в інших відношеннях.
    1.2. Каскадувати – виконати каскадне видалення, видаливши відповідні об'єкти, що посилаються на нього.

2. При спробі оновити потенційний ключ, на який посилається зовнішній ключ:
    2.1. Обмежити – призупинити операцію оновлення, до моменту, коли не будуть існувати всі об'єкти, що посилаються на нього, в інших відношеннях.
    2.2. Каскадувати – виконати каскадне оновлення, замінивши значення зовнішнього ключа у відповідних об'єктах, що посилаються на нього, в інших відношеннях.

При каскадному видаленні та оновленні слід врахувати можливість наявності циклів посилань, які можуть призвести до ускладнення процедури модифікації БД.

3.9.2. Значення NULL та підтримка цілісності посилань

Значення NULL використовують для позначення факту відсутності інформації. Наприклад, дата народження людини може бути невідомою. Для даного атрибуту може бути дозволено або заборонено містити значення NULL.

Можливість наявності значень NULL призводить до необхідності формування правила цілісності об'єктів.

Цілісність об'єктів полягає у тому, що жоден елемент первинного ключа не може містити значення NULL.

Правило цілісності посилань також має бути розширене з урахуванням можливості наявності значень NULL.

Можливість присутності значень NULL призводить до виникнення низки труднощів і критикується деякими дослідниками.

3.10. Контрольні запитання

1. У чому полягає реляційний підхід до представлення даних?
2. Дайте визначення поняття домену та наведіть приклад.
3. Дайте визначення поняття декартового добутку доменів і наведіть приклад.
4. Дайте визначення поняття відношення та наведіть приклад.
5. Дайте визначення поняття потенційний та первинний ключ.
6. Дайте визначення поняття «зовнішні ключі».
7. Дайте визначення поняття цілісність реляційних даних.
8. Поясніть термін цілісності посилань.

4. РЕЛЯЦІЙНА АЛГЕБРА ТА РЕЛЯЦІЙНЕ ЧИСЛЕННЯ

4.1. Поняття реляційної алгебри

Основним компонентом тієї частини реляційної моделі, яка стосується обробки даних, є так звана *реляційна алгебра*, яка в основному складається з набору операторів, що використовують відношення як операнди і повертають відношення як результат.

Реляційна алгебра, визначена Е. Коддом, складається з 8 операторів, які складають 2 групи по 4 оператори в кожній:

1. Традиційні операції над множинами: об'єднання, перетин, віднімання та декартовий добуток.
2. Спеціальні реляційні операції: вибірка, проєкція, з'єднання та ділення.

Результатом кожної реляційної операції над відношенням (або реляційною операцією) також є відношення. Результат однієї операції може бути використано як вхідні дані для іншої операції.

4.2. Традиційні теоретико-множинні операції над відношеннями

Будемо говорити, що два відношення сумісні за типом, якщо вони мають ідентичні заголовки, а саме:

1. Кожне з відношень має однаковий набір назв атрибутів;
2. Відповідні атрибути визначені на одному й тому ж домені.

Операції об'єднання, перетину та віднімання вимагають від операндів сумісності за типом.

4.2.1. Об'єднання

Об'єднанням двох сумісних за типом відношень A і B (A UNION B) називається відношення з тим самим заголовком, як у відношеннях A і B, і тілом, що складається з множини всіх кортежів, що належать відношенню A або відношенню B, або обом з них.

Приклад операції об'єднання відношень показано на рис. 4.1.

A

CityNo	CityName	RgNo
1	Київ	1
2	Харків	3
3	Чернігів	4

B

CityNo	CityName	RgNo
2	Харків	3
3	Чернігів	4
4	Львів	2

A UNION B

CityNo	CityName	RgNo
1	Київ	1
2	Харків	3
3	Чернігів	4
4	Львів	2

Рис. 4.1. Результат об'єднання відношень A і B

Примітка. Аналог операції об'єднання в мові SQL очевидний і реалізується за допомогою конструкції - запит на вибірку

```
SELECT CityNo, …
FROM A
  UNION
     SELECT CityNo, …
     FROM B
```

4.2.2. Перетин множин

Перетином двох сумісних за типом відношень A і B (A INTERSECT B) називається відношення з таким самим заголовком, як у відношеннях A і B, і тілом, яке складається з множини всіх кортежів, які одночасно належать обом відношенням A і B.

Результат операції перетину відношень наведених на рис. 4.1, відображено на рис. 4.2.

A INTERSECT B

CityNo	CityName	RgNo
2	Харків	3
3	Чернігів	4

Рис. 4.2. Результат операції перетину відношень A і B

Примітка. Аналогом операції перетину в SQL є запит на вибірку

```
SELECT CityNo, …
FROM A
WHERE CityNo IN (SELECT CityNo
                 FROM B)
```

або конструкція

```
SELECT CityNo, …
FROM A
   INTERSECT
      SELECT CityNo, …
      FROM B
```

4.2.3. Віднімання

Відніманням двох сумісних відношень A і B (A MINUS B) називається відношення з тією ж назвою, що і у відношеннях A і B, і з тілом, що складається з множини всіх кортежів, що належать відношенню A і не належать відношенню B.

Результат операції віднімання відношень, наведених на рис. 4.1, показано на рис. 4.3.

A MINUS B

CityNo	CityName	RgNo
1	Київ	1

B MINUS A

CityNo	CityName	RgNo
4	Львів	2

Рис. 4.3. Результат операції віднімання відношень A - B і B - A

Примітка. Аналогом операції віднімання в SQL є запит на вибірку

```
SELECT CityNo, …
FROM A
WHERE CityNo NOT IN (SELECT CityNo
                     FROM B)
```

або запит

```
SELECT CityNo, …
FROM A
   EXCEPT
      SELECT CityNo, …
      FROM B
```

4.2.4. Добуток

Декартовий добуток двох відношень A і B, де A і B не мають спільних назв атрибутів (A x B), визначається як відношення з заголовком, який є об'єднанням двох заголовків вихідних відношень A і B, і тілом, що складається з множини всіх кортежів t, таких, що t є об'єднанням кортежу $a \in A$ і кортежу $b \in B$. Кардинальне число результату добутку дорівнює добутку кардинальних чисел

вихідних відношень A і B, а ступінь добутку дорівнює сумі їх ступенів. Приклад операції з декартовим добутком наведено на рис. 4.4.

A

CityNo	CityName	A_RgNo
1	Київ	1
2	Харків	3
3	Чернігів	4

B

B_RgNo	RgName
1	Київ
2	Львів

A * B

CityNo	CityName	A_RgNo	B_RgNo	RgName
1	Київ	1	1	Київ
1	Київ	1	2	Львів
2	Харків	3	1	Київ
2	Харків	3	2	Львів
3	Чернігів	4	1	Київ
3	Чернігів	4	2	Львів

Рис. 4.4. Результат операції декартового добутку відношень A і B

Примітка 1. Ця операція включена в реляційну алгебру головним чином з концептуальних міркувань.

Примітка 2. Аналогом роботи добутку в SQL є запит на вибірку з FROM, але без WHERE

SELECT *
FROM A, B

або запит на кшталт

SELECT *
FROM A CROSS JOIN B

4.3. Спеціальні реляційні операції

4.3.1. Вибірка

Вибірка – це скорочена назва від операції Θ-вибірки, де Θ позначає будь-який скалярний оператор порівняння (=, ≠, >, ≥, ≤, <).

Θ-вибіркою з відношення A за атрибутами X та Y

A **WHERE** X Θ Y

називається відношення, яке має ту саму назву, що й відношення A, і тіло, що містить множину всіх кортежів відношення A, для яких перевірка умови X Θ Y дає значення true. Атрибути X і Y повинні бути визначені на одному і тому ж домені, а оператор Θ повинен мати сенс для цього домену. На рис. 4.5 наведено приклад операції вибірки.

A

CityNo	CityName	RgNo
1	Київ	1
2	Харків	3
3	Чернігів	4

A WHERE RgNo = 1

CityNo	CityName	RgNo
1	Київ	1

Рис. 4.5. Вихідне відношення A та результат операції вибірки кортежів з відношення A за умовою RGNo = 1

Примітка. Аналогом операції вибірки в SQL є запит на вибірку

SELECT CityNo**, …**
FROM A
WHERE RgNo = 1

4.3.2. Проєкція

Проєкцією відношення A за атрибутами X, Y,..., Z, де кожен з атрибутів належить відношенню A

A [X, Y, ..., Z]

називається відношення із заголовком {X, Y ,..., Z} та тілом, що містить множину всіх кортежів {X:x, Y:y, ..., Z:z}, для яких у відношенні A значення атрибута X дорівнює x, атрибута Y дорівнює y, ..., атрибута Z дорівнює z.

За допомогою оператора проєкції виходить підмножина цього відношення, в якій виключаються всі атрибути, не зазначені у списку атрибутів, з подальшим виключенням дублюючих кортежів, як зображено на рис.4.6.

A

CityNo	CityName	RgNo
1	Київ	1
2	Харків	3
3	Чернігів	4
4	Львів	2

A [CityName]

CityName
Київ
Харків
Чернігів
Львів

Рис. 4.6. Початкове відношення A та результат операції проєкції відношення A за атрибутом CityName

Жоден атрибут не може бути вказаний більше одного разу у списку атрибутів.

Примітка. Аналогом операції проєкції в SQL є запит на вибірку

SELECT CityName

FROM A

4.3.3. Природне з'єднання

Операція з'єднання має декілька різновидів. Проте найважливішим є природне з'єднання.

Нехай відношення A і B мають заголовки

$\{X_1, X_2, ..., X_m, Y_1, Y_2, ..., Y_n\}$

і

$\{Y_1, Y_2, ..., Y_n, Z_1, Z_2, ..., Z_p\}$

Тобто атрибути $Y_1, Y_2, ..., Y_n$ є загальними для двох відношень. Припустимо також, що атрибути $Y_1, Y_2, ..., Y_n$ відношень A і B визначені на одних і тих же доменах відповідно.

Для компактності запису розглянемо вирази

$\{X_1, X_2, ..., X_m\}$, $\{Y_1, Y_2, ..., Y_n\}$ та $\{Z_1, Z_2, ..., Z_p\}$

як три складені атрибути X, Y та Z відповідно.

Тоді природнім з'єднанням відношень A і B (A JOIN B) називається відношення з заголовком {X, Y, Z} і тілом, що містить множину всіх кортежів {X:x, Y:y, Z:z}, для яких у відношенні A значення атрибута X дорівнює x, а атрибута Y дорівнює y, і в відношенні B значення атрибута Y дорівнює y, а атрибута Z дорівнює z.

Приклад операції природного з'єднання наведено на рис. 4.7.

A

CityNo	CityName	RgNo
5	Ніжин	3
6	Бахмач	3
7	Бобровиця	3

B

RgNo	RgName
3	Чернігівщина
4	Львів

A JOIN B

CityNo	CityName	A.PrHo	RgName
5	Ніжин	3	Чернігівщина
6	Бахмач	3	Чернігівщина
7	Бобровиця	3	Чернігівщина

Рис. 4.7. Початкові відношення A і B та результат операції природного з'єднання A JOIN B

Примітка. Аналогом операції природного об'єднання в SQL1 є запит на вибірку

SELECT CityNo, CityName, A.RgNo, RgName
FROM A, B
WHERE A.RgNo = B.RgNo

а в SQL2 запит на вибірку

SELECT CityNo, CityName, A.RgNo, RgName
FROM A INNER JOIN B USING (RgNo)

4.3.4. Θ-з'єднання

Операція Θ-з'єднання призначена для об'єднання двох відношень разом на основі якоїсь умови, відмінної від еквівалентності.

Θ-з'єднанням відношення A за атрибутом X з відношенням B за атрибутом Y називається результат обчислення виразу

(A **TIMES** B) **WHERE** X Θ Y

Атрибути X і Y повинні бути визначені на одному і тому ж домені, а операція повинна мати зміст для цього домену.

Таким чином, Θ-з'єднання – це відношення з тим самим заголовком, що і при декартовому добутку відношень A і B, і з тілом, що містить множину кортежів, які належать цьому декартовому добутку і задовольняють умові XΘY.

Примітка. Аналог операції Θ-з'єднання в SQL очевидний. Він аналогічний прикладу 1-го варіанту запиту для природного з'єднання, але зі злиттям за будь-яким однотипним стовпцем.

4.3.5. Ділення

Нехай відношення A – ділене, а відношення B – дільник, мають заголовки відповідно:

$\{X_1, X_2,..., X_m, Y_1, Y_2, ..., Y_n\}$ и $\{Y_1, Y_2, ..., Y_n\}$

де атрибути $Y_1, Y_2,..., Y_n$ є спільними для обох відношень і визначені на одних і тих же доменах відповідно, а відношення B не має додаткових атрибутів.

Нехай для компактності запису

$\{X_1, X_2, ..., X_m\}$ и $\{Y_1, Y_2, ..., Y_n\}$

позначають два складових атрибути X і Y відповідно.

Діленням відношення A на відношення B (A DIVIDE BY B) називається відношення з заголовком {X} і тілом, що містить множину всіх кортежів {X:x},

таких, що існує кортеж {X:x, Y:y}, який належить відношенню A для всіх кортежів {Y:y}, що належать відношенню B.

В довільній формі це можна сформулювати так: результат містить такі значення X з відношення A, для яких відповідні значення Y з відношення A включають всі значення Y з відношення B.

Примітка. Типові SQL запити, реалізовані операцією ділення, зазвичай містять у своєму формулюванні слово "all".

Приклад: Знайти коди студентів, які вивчають усі дисципліни.

Результат операції ділення наведено на рис. 4.8. Відношення M є проєкцією відношення Marks, а відношення S – проєкцією відношення Subjects. Результат операції ділення M DIVIDE BY S фактично містить номери студентів, які вивчали предмети з номерами 1 і 5.

M

StdNmb	SubjNo
1	1
1	5
2	1
2	5
3	1
4	5

S

SubjNo
1
5

M DIVIDE BY S

StdNmb
1
2

Рис. 4.8. Приклад операції ділення

Примітка. Аналог операції ділення в SQL має вигляд

```
SELECT DISTINCT M.StdNmb
FROM M
WHERE NOT EXIST
    (SELECT *
     FROM S
     WHERE NOT EXIST
        (SELECT *
         FROM M PM
         WHERE PM.StdNmb = M.StdNmb AND PM.SubjNo = S.SubjNo))
```

Примітка. Пояснення для реалізації запиту дивитися у розділі, присвяченому оператору EXISTS мови SQL.

4.3.6. Розширення

Багато авторів запропонували нові алгебраїчні оператори після того, як Е. Кодд визначив початкові 8. Деякі з них в стислій формі розглянуті нижче.

За допомогою операції розширення з певного відношення створюється нове відношення, яке схоже на початкове, але містить додатковий атрибут, значення якого отримані за допомогою деяких скалярних обчислень. На рис. 4.9 показано вихідне відношення та результат операції розширення.

EXTEND GROUPS ADD (2000-EnterYear) AS COURSE

GROUPS

GrpNo	EnterYear	GrpName
1	1998	КС-981
2	1999	КС-982
3	1998	СП-981

Результат операції розширення

GrpNo	EnterYear	GrpName	Course
1	1998	КС-981	2
2	1999	КС-982	1
3	1998	СП-981	2

Рис. 4.9. Приклад операції розширення

4.3.7. Підведення підсумків

Нехай A_1, A_2,..., A_n – окремі атрибути відношення A. Результатом операції підведення підсумків

SUMMARIZE A **BY** (A_1, A_2, … A_n) **ADD** exp **AS** Z

(яка є виразом, а не командою або оператором) буде відношення із заголовком {A_1, A_2, ..., A_n, Z} і з тілом, що містить всі кортежі, які є кортежами проєкції відношення A за атрибутами A_1, A_2, ..., A_n, розширеними значенням для нового атрибута Z. Нове значення Z обчислюється шляхом обчислення підсумкового виразу exp для всіх кортежів відношення A, які мають ті ж самі значення для атрибутів A_1, A_2, ..., A_n, що й кортеж t. Список атрибутів A_1, A_2, ..., A_n не повинен включати атрибут з назвою Z, а вираз exp не повинен посилатися на атрибут Z.

Кардинальне число результату дорівнює кардинальному числу проєкції відношення A за атрибутами A_1, A_2, ..., A_n, а ступінь результату дорівнює ступеню такої проєкції плюс одиниця.

4.3.8. Оновлення

Крім реляційної алгебри, реляційна модель (точніше, та її частина, що пов'язана з обробкою даних) може включати також операції реляційного присвоєння. Такі операції мають наступний синтаксис:

TARGET := SOURCE

де source і target – це реляційні вирази, які представляють собою сумісні за типом відношення. Обчислене значення SOURCE присвоюється відношенню TARGET, замінюючи його старе значення.

У реляційних системах також існують операції вставки INSERT, видалення DELETE і модифікації UPDATE.

Оператор вставки INSERT має наступний вигляд:

INSERT source **INTO** target

де SOURCE і TARGET – реляційні вирази, які представляють собою сумісні за типом відношення (на практиці відношення TARGET часто є просто іменованим відношенням). Значення відношення SOURCE обчислюється, і всі кортежі результату вставляються в відношення TARGET.

Оператор оновлення UPDATE має наступний вигляд:

UPDATE target attribute1 := scalar_expression,

attribute2 := scalar_expression, …, attributeN := scalar_expression

де TARGET – реляційний вираз, а кожен атрибут ATTRIBUTE належить відношенню, яке є результатом обчислення вказаного виразу. Усі кортежі у результуючому відношенні оновлюються відповідно до зазначених операторів

attribute2 := scalar_expression

На практиці вираз TARGET часто буде просто обмежувальною умовою для деякого іменованого відношення.

Оператор видалення DELETE має наступний вигляд:

DELETE target

де TARGET – реляційний вираз, який визначає кортежі, які підлягають видаленню.

Як і в випадку з оператором оновлення, вираз TARGET часто буде просто обмежувальною умовою для деякого іменованого відношення.

4.4. Реляційне числення

Другим, історично наступним, базовим механізмом маніпулювання даними в реляційній моделі є реляційне числення, засноване на математичній логіці – численні предикатів.

Так само, як і вирази реляційної алгебри, формули реляційного числення визначаються над відношеннями реляційних баз даних, і результатом обчислення також є відношення.

Однак, якщо запит, представлений мовою реляційної алгебри, може бути обчислений на основі обчислення елементарних алгебраїчних операцій з урахуванням їх пріоритету та можливих дужок, то формула реляційного числення лише встановлює умови, яким повинні задовольняти кортежі результуючого відношення. Тому мови реляційного числення є непроцедурними або декларативними.

Першим прикладом мов реляційного числення є мова DSL-α. Однак на практиці такі мови не знайшли широкого поширення і незабаром були витіснені мовою SQL.

4.5. Контрольні запитання

1. Надайте визначення реляційної алгебри.
2. Вкажіть традиційні теоретико-множинні операції над відношеннями та наведіть приклади.
3. Перелічіть спеціальні реляційні операції та поясніть їх призначення.
4. Дайте визначення і наведіть приклад реляційної операції під назвою "Вибірка".
5. Дайте визначення і наведіть приклад реляційної операції "Проєкція".
6. Дайте визначення і наведіть приклад реляційної операції "Природне з'єднання".
7. Дайте визначення та наведіть приклад реляційної операції "Θ- з'єднання".
8. Дайте визначення і наведіть приклад реляційної операції "Ділення".
9. Дайте визначення і наведіть приклад реляційної операції "Розширення".
10. Дайте визначення і наведіть приклад реляційної операції "Підведення підсумків".
11. Дайте визначення і наведіть приклад реляційної операції "Оновлення"".
12. Яке визначення реляційного числення?

5. МОВА SQL

5.1. Історія створення та розвитку SQL

Мова для взаємодії з базами даних SQL (Structured Query Language або у перекладі «мова для структурованих запитів») з'явилась в середині 70-х років і була розроблена в рамках проєкту експериментальної реляційної СУБД System R. Оригінальна назва мови SEQUEL (Structured English Query Language) лише частково відображає сутність цієї мови. Звичайно, мова була орієнтована в основному на зручне і зрозуміле користувачам формулювання запитів до реляційної БД, але насправді вже була повноцінною мовою баз даних, що містить:

1. Засоби формулювання запитів та маніпулювання базою даних.
2. Засоби визначення та маніпулювання схемою бази даних.
3. Засоби визначення обмежень цілісності та тригерів.
4. Засоби створення представлень бази даних.
5. Можливість визначати структури фізичного рівня, які підтримують ефективне виконання запитів.
6. Можливість авторизації доступу до відношень та їх полів.
7. Можливості визначення точок збереження транзакції та відкатів.

Таким чином, SQL стала досить потужною мовою для взаємодії з СУБД. На сьогоднішній день SQL є єдиною стандартною мовою запитів. SQL має такі переваги:

1. Незалежність від конкретних СУБД. Якщо при створенні бази даних не використовувалися нестандартні можливості мови SQL, передбачені певною СУБД, то таку базу даних можна без змін перенести на СУБД іншого виробника. На жаль, більшість баз даних використовують специфічні властивості СУБД, що ускладнює їх перенесення на іншу СУБД без змін.
2. Реляційна основа. Реляційна модель має міцну теоретичну основу. SQL базується на реляційній моделі і є єдиною мовою для реляційних баз даних.
3. Синтаксис мови SQL має високорівневу структуру, наближену до англійської мови.
4. SQL дозволяє створювати різноманітні представлення даних для різних користувачів.
5. SQL – це повноцінна мова для роботи з базами даних. Вона є реляційно повною, тому що за виразною силою не поступається реляційній алгебрі

(або, що те ж саме, реляційному численню), тобто будь-який оператор реляційної алгебри може бути виражений засобами мови SQL.

5.2. Стандарти мови SQL

Офіційний стандарт для мови SQL – SQL1 був опублікований ANSI та ISO у 1989 році.

У 1992 році він був значно розширений за допомогою SQL2.

У 1999 році був опублікований стандарт SQL3.

В даний час діє стандарт SQL3 з подальшими уточненнями і розширеннями в 2003, 2006, 2008 і 2016 роках. Останнє повідомлення про уточнення і розширення стандарту SQL3 датовано 2023 р.

Слід зазначити, що різні СУБД можуть мати відмінності в реалізації SQL, які розширюють можливості стандарту, але всі вони підтримують стандарт.

5.3. Основні поняття мови SQL

5.3.1. Оператори

У SQL використовується понад 30 операторів, кожен з яких формує команду СУБД виконати певну дію, наприклад, прочитати дані, створити таблицю, додати нові дані в таблицю тощо [3].

Кожен оператор SQL має строгу структуру і починається з директиви, тобто ключового слова, що описує дію, яку виконує оператор. Типовими дієсловами є:

1. SELECT (вибрати).
2. CREATE (створити).
3. INSERT (додати).
4. DELETE (видалити).
5. UPDATE (оновити).
6. DROP (видалити).
7. ALTER (змінити).
8. COMMIT (завершити та зафіксувати внесені зміни).
9. ROLLBACK (скасувати внесені зміни).

Після директиви йде одне або декілька секцій. Секція описує дані, з якими працює оператор, або містить уточнюючу інформацію про дію, яку виконує оператор. Кожна секція також починається з ключового слова, наприклад:

1. FROM (звідки).
2. WHERE (де).

3. GROUP BY (групувати за).
4. HAVING (маючи).
5. ORDER BY (впорядкувати за).
6. INTO (куди).

Деякі секції в операторі є обов'язковими, а інші – ні. Конкретна структура і зміст секції можуть змінюватися. Багато секцій містять назви таблиць або стовпців; деякі з них можуть містити додаткові ключові слова, константи та вирази.

У стандарті ANSI/ISO визначені ключові слова, які використовуються як дієслова в секціях операторів. Відповідно до стандарту, ці ключові слова не можна використовувати для найменування об'єктів бази даних, таких як таблиці, стовпці та користувачі.

5.3.2. Назви

Кожен об'єкт у базі даних має унікальну назву. Ці назви використовуються в SQL-операторах і вказують, над яким об'єктом бази даних потрібно виконати дію. Стандарт ANSI/ISO визначає що, назви існують у таблиць, стовпців та користувачів. В багатьох реалізаціях SQL також підтримуються додаткові іменовані об'єкти, такі як збережені процедури, іменовані відношення "первинний ключ – зовнішній ключ" та форми для введення даних.

Відповідно до стандарту ANSI/ISO, назви SQL2 повинні мати довжину від 1 до 128 символів, починатися з літери та не містити пробілів чи спеціальних символів пунктуації.

Повна назва таблиці складається з імені власника таблиці та назви самої таблиці, розділених крапкою (.). Наприклад, повна назва таблиці Students, власником якої є користувач під назвою Admin, має наступний вигляд:

Admin.Students

Якщо в операторі вказується назва стовпця, SQL автоматично визначає, яка з таблиць, указаних у тому ж операторі, містить цей стовпець. Однак якщо в операторі потрібно включити 2 стовпці з різних таблиць, але з однаковими назвами, необхідно вказати повні назви стовпців, які однозначно визначають їх розташування. Повна назва стовпця складається з назви таблиці, що містить стовпець, і назви стовпця (простої назви), розділених крапкою (.). Наприклад, повна назва стовпця StdName з таблиці Students матиме наступний вигляд:

Students.StdName

5.3.3. Типи даних

В стандарті ANSI/ISO визначені типи даних, які можна використовувати для представлення інформації в реляційній базі даних. Типи даних, які існують у стандарті SQL1, складають лише мінімальний набір і підтримуються у всіх комерційних СУБД. У таблиці 5.1 перелічено типи даних, визначені в стандартах SQL1 та SQL2.

Таблиця 5.1. Типи даних у SQL

Тип даних	Опис
CHAR (довжина)	Рядки символів постійної довжини
CHARACTER (довжина)	
VARCHAR (довжина)	Рядки символів змінної довжини
CHAR VARYING (довжина)	
CHARACTER VARYING (довжина)	
NCHAP(довжина)	Локалізовані рядки символів постійної довжини
NATIONAL CHAR (довжина)	
NATIONAL CHARACTER (довжина)	
NCHAR VARYING (довжина)	Локалізовані рядки символів змінної довжини
NATIONAL CHAR VARYING (довжина)	
NATIONAL CHARACTER VARYING (довжина)	
INTEGER	Цілі числа
INT	
SMALLINT	Малі цілі числа
BIT(довжина)	Бітові рядки постійної довжини
BIT VARYNG (довжина)	Бітові рядки змінної довжини
NUMERIC (точність, ступінь)	Цілі (десяткові) числа
DECIMAL (точність, ступінь)	
DEC (точність, ступінь)	
FLOAT (точність)	Числа з плаваючою комою
REAL	Низькоточні числа з плаваючою комою
DOUBLE PRECISION	Високоточні числа з плаваючою комою
DATE	Календарна дата
TIME (точність)	Час
TIME STAMP (точність)	Дата і час
INTERVAL	Часовий інтервал

У SQL1 використовуються наступні типи даних:

1. Рядки символів постійної довжини. У стовпцях, що мають цей тип даних, зазвичай зберігаються імена людей та назви компаній, адреси, описи тощо.
2. Цілі числа. У стовпцях, що мають цей тип даних, зазвичай зберігаються дані про облікові записи, кількості, вік тощо. Цілочисельні стовпці часто використовуються також для зберігання ідентифікаторів, таких як ідентифікатор клієнта, працівника або замовлення.
3. Масштабовані цілі числа. У стовпцях цього типу зберігаються числа з дробовою частиною, які потрібно обчислювати точно, наприклад курси валют та відсотки. Крім того, у таких стовпцях часто зберігаються грошові величини.
4. Числа з плаваючою комою. Стовпці цього типу використовуються для зберігання величин, які можна обчислювати приблизно, наприклад ваги та відстані. Числа з плаваючою комою можуть представляти більший діапазон значень, ніж десяткові числа, але під час обчислень можуть виникати помилки округлення.

У більшості комерційних СУБД, крім типів даних, визначених у стандарті SQL1, існує множина додаткових типів даних, більшість з яких увійшли до стандарту SQL2. Нижче наведено найважливіші з них:

1. Рядки символів змінної довжини. Майже в усіх СУБД підтримується тип даних VARCHAR, який дозволяє зберігати рядки символів, довжина яких змінюється в певному діапазоні. У стандарті SQL1 були визначені рядки постійної довжини, які доповнюються пробілами праворуч.
2. Грошові величини. У багатьох СУБД підтримується тип даних MONEY або CURRENCY, який зазвичай зберігається у вигляді десяткового числа або числа з плаваючою комою. Наявність окремого типу даних для представлення грошових величин дозволяє правильно форматувати їх при відображенні на екрані та виконанні арифметичних операцій.
3. Дата та час. Тип даних дати/часу також розповсюджено в різних СУБД, хоча способи її реалізації досить суттєво відрізняються один від одного. Зазвичай над значеннями цього типу даних можна виконувати різні операції. У стандарт SQL2 входить визначення типів даних DATE, TIME, TIMESTAMP та INTERVAL, включаючи підтримку часових поясів та можливість задавати точність, представлення часу (наприклад, десяті або соті частини секунди).

4. Логічні дані. Деякі СУБД явно підтримують логічні значення (TRUE або FALSE).

5.3.4. Константи

У стандарті ANSI/ISO визначено формати числових і рядкових констант, або літералів, які представляють конкретні значення даних. Цей формат використовується в більшості реалізацій SQL.

Числові константи. Цілі та десяткові константи (відомі також як точні числові літерали) в операторах SQL представляються у формі звичайних десяткових чисел з необов'язковим знаком плюс (+) або мінус (-) перед ними:

2100 -3200 -3752000,00 +497500,8778

Константи з плаваючою комою або крапкою (відомі також як приблизні числові літерали) визначаються за допомогою символу E і мають такий же формат, як і в більшості мов програмування. Нижче наведено приклади констант з плаваючою комою:

1.5E3 -3.14159E1 2.5E-7 0,783926E21

Символ E читається як "помножити на десять у ступені", тому перша константа представляє число "1,5 помножити на десять у ступені 3", або 1500.

Рядкові константи. Згідно зі стандартом ANSI/ISO, рядкові константи в SQL повинні бути обмежені одинарними лапками, як показано в наступних прикладах:

'Jones, John J.' 'New York' 'Western'

Якщо необхідно включити в рядкову константу одинарну лапку, замість неї слід написати дві одинарні лапки ('').

Константи дати та часу. У реляційних СУБД календарні дати, час і інтервали часу представляються у вигляді рядкових констант. Формати цих констант в різних СУБД відрізняються один від одного. Крім того, способи запису часу та дати змінюються в залежності від країни.

Символьні константи. Крім визначених користувачем констант, в SQL існують спеціальні символьні константи, що повертають значення, які зберігаються в самій СУБД.

У стандарті SQL2 увійшли найбільш корисні символьні константи з різних реалізацій SQL, зокрема константи CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, а також USER, SESSION_USER і SYSTEM_USER.

Відповідно до стандарту ANSI/ISO у виразах можна використовувати 4 арифметичні операції: додавання (+), віднімання (-), множення (*) і ділення (/), а також ряд агрегатних функцій SQL – COUNT(), AVG(), SUM(), MIN() і MAX(). Слід підкреслити, що арифметичні операції виконуються для кожного рядка таблиці окремо, тоді як агрегатні функції застосовуються до всіх значень вказаного стовпця. Для формування складних виразів можна використовувати дужки. Наприклад

(SALES / TARGET) * 100

або

SUM(SALES) / COUNT(SALES)

Відсутні дані (значення NULL). Оскільки БД представляє собою модель реального світу, певні елементи даних в ній неминуче будуть відсутні або підходити не для всіх сутностей. Для позначення таких даних використовується спеціальна константа, яка називається NULL.

5.3.5. Вбудовані функції

У стандарт SQL2 увійшли найбільш корисні функції з різних реалізацій SQL. Ці функції перелічені у таблиці 5.2.

Таблиця 5.2. Вбудовані функції SQL

Функція	Повертає
BIT LENGTH(рядок)	Кількість бітів у рядку
CAST(значення AS тип даних)	Значення, перетворене на тип даних (наприклад, дата, перетворена на рядок)
CHAR_LENGTH(рядок)	Довжина рядка символів
CONVERT(рядок USING функція)	Рядок, перетворений відповідно до вказаної функції
CURRENT_DATE	Поточна дата
CURRENT_TIME(точність)	Поточний час із заданою точністю
CURRENT_TIMESTAMP (точність)	Поточна дата й час із заданою точністю
EXTRACT(частина FROM значення)	Отримати вказану частину (DAY, HOUR тощо) зі значенням типу DATETIME
LOWER(рядок)	Рядок, перетворений до нижнього регістру
OCTETLENGTH (рядок)	Кількість байтів у рядку символів
POSITION(перший рядок IN другий рядок)	Позиція, з якої починається входження першого рядка в другий рядок
SUBSTRING(рядок FROM n FOR довжина)	Частина рядка, яка починається з n-го символу і має вказану довжину
TRANSLATE(рядок USING функція)	Рядок, перетворений за допомогою вказаної функції
TRIM(BOTH символ FROM рядок)	Рядок, у якому вилучено перший і останній вказані символи

Функція	Повертає
TRIM(LEADING символ FROM рядок)	Рядок, у якому було вилучено перші вказані символи
TRIM(TRAILING символ FROM рядок)	Рядок, у якому було вилучено останні вказані символи
UPPER(рядок)	Рядок, перетворений до верхнього регістру

5.4. Контрольні запитання

1. Яке призначення та основні поняття SQL?
2. Перелічіть основні засоби SQL.
3. Перелічіть основні можливості SQL.
4. Перелічіть основні переваги SQL.
5. Перелічіть основні директиви SQL та охарактеризуйте їх призначення.
6. Якими назвами оперує SQL?
7. Перелічіть основні типи даних SQL.
8. Перелічіть основні формати числових і рядкових констант або літералів, які представляють конкретні значення даних.

6. ЗАПИТИ НА ЧИТАННЯ ДАНИХ

Запити на зчитування або отримання даних, як правило, повинні визначати:

1. Мету вибірки.
2. Умови вибірки.
3. Джерело вибірки.

Крім того, бажано вказати спосіб упорядкування вибраних даних. Ці завдання розв'язує оператор вибірки SELECT.

6.1. Оператор SELECT

Оператор вибірки SELECT разом з операторами вставки даних INSERT, їх видалення DELETE та оновлення UPDATE утворюють підмножину SQL, яка отримала назву мова маніпулювання даними (Data Manipulation Language – DML).

Оператор SELECT використовується для побудови вибірки даних і має наступний синтаксис:

SELECT [ALL | DISTINCT] список_стовпців | *
FROM список_назв_таблиць
WHERE умова_пошуку
GROUP BY список_назв_стовпців
HAVING умова_пошуку
ORDER BY назва стовпця **[ASC | DESC]**, …

Примітка. Слід звернути увагу на різницю між поняттями оператор SELECT та виразом SELECT в складі оператора SELECT.

У типовому варіанті оператора вибірки даних SELECT, в якому задано стовпці вибірки, умови вибірки та джерело вибірки, має структуру, яка показана на рис. 6.1.

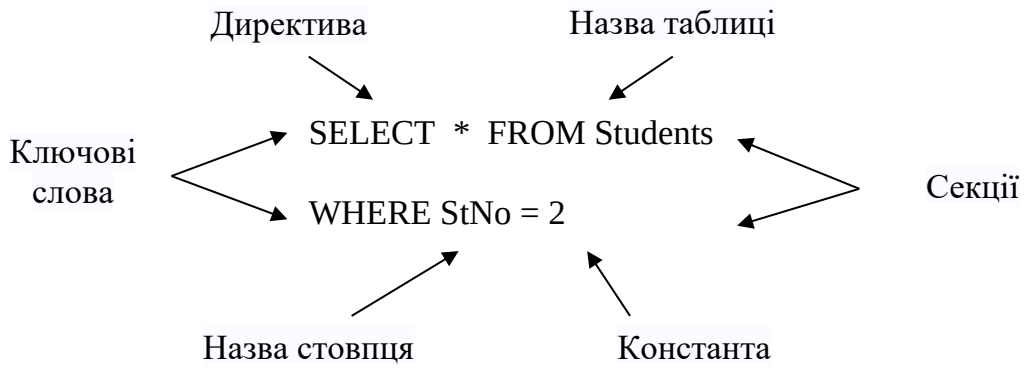


Рис. 6.1. Структура оператора SQL

6.2. Мета запиту. Секція SELECT

У секції SELECT, що є початковою для всіх операторів SELECT, необхідно вказати елементи даних, які будуть повернуті у результаті виконання запиту. Ці елементи вказуються як список стовпців, які потрібно повернути, відокремлених комами. Для кожного елемента з цього списку у таблиці результатів запиту буде створено один стовпець. Стовпці в таблиці результатів буде впорядковано в тому ж порядку, що й елементи у списку стовпців, які повертаються. Отриманий стовпець може являти собою:

1. Назву стовпця, що ідентифікує один із стовпців, які містяться у таблицях, перелічених у секції FROM. Коли повернутий стовпець є назвою стовпця в таблиці бази даних, SQL бере значення цього стовпця для кожного з рядків таблиці та розміщує його у відповідному рядку таблиці результатів запиту.
2. Константа, яка показує, що у кожному рядку таблиці результатів запиту повинно міститися одне й те саме значення.
3. Вираз, який вказує, що для кожного рядка таблиці результату SQL повинен обчислити значення за формулою, визначеною у виразі, та розміщувати його у результаті запиту.

Результатом SQL-запиту на читання завжди є таблиця, що містить дані і нічим не відрізняється від таблиць бази даних, наприклад

```
SELECT StdName, StdBrthDate, StdStipend
FROM Students
```

Слід зауважити, що крім стовпців, значення яких зчитуються безпосередньо з БД, SQL-запит на читання може містити обчислювані стовпці, значення яких відсутні в базі даних, але визначаються на основі виразів, що базуються на значеннях, які зберігаються в БД та/або окремих константах. Щоб отримати обчислюваний стовпець, у списку повернених стовпців необхідно вказати вираз функціонального типу, наприклад

```
SELECT StdName, StdBrthDate, StdStipend, StdStipend / 8,
        SUM(StdStipend) / COUNT(StdNmb)
FROM Students
```

Для кожного рядка таблиці результатів у відповідних стовпцях перелічуються результати виконання відповідних функцій. Проте у такому випадку назвою стовпця стає вираз, що не допускається в більшості програмних систем. З цієї причини SQL використовує псевдоніми для результуючих

стовпців. Псевдоніми стовпців вказуються за допомогою ключового слова AS, наприклад:

SELECT StdName, StdStipend, StdStipend/8 **AS** STD,
SUM(StdStipend) / COUNT(StdNmb) **AS** SC
FROM Students

Примітка. У якості стовпця може використовуватися і константа, наприклад, рядкова константа "Підвищена стипендія".

Приклад. Вивести список студентів з їх кодами та сформувати стовпець PS, що складається з констант "Підвищена стипендія".

SELECT StdName, StdStipend, "Підвищена стипендія" **AS** PS
FROM Students

В результаті кожне поле стовпця PS у результуючій таблиці буде містити значення "Підвищена стипендія".

Іноді потрібно отримати вміст усіх стовпців таблиці. На практиці така ситуація може виникнути, коли ви вперше стикаєтеся з новою базою даних і вам потрібно швидко розібратися в її структурі та даних, що зберігаються в ній. З огляду на це, SQL дозволяє використовувати зірочку (*) замість списку стовпців для повернення, що означає, що користувач очікує отримати всі стовпці:

Приклад. Показати всі дані, що містяться у таблиці Students.

SELECT *
FROM Students

За замовчуванням або якщо в операторі SELECT перед списком повернених стовпців зазначено ключове слово ALL, у таблиці результатів відображаються всі рядки, які відповідають критеріям пошуку. Серед них можуть бути повторювані рядки, що не завжди потрібно. Такі рядки з таблиці результатів запиту можна видалити, якщо в операторі SELECT перед списком повернених стовпців зазначити ключове слово DISTINCT, наприклад

SELECT DISTINCT StdName, StdBrthDate, StdStipend
FROM Students

6.3. Використання таблиць. Секція FROM

Секція FROM складається з ключового слова FROM, за яким слідує список назв таблиць, розділених комами. Кожна назва таблиці ідентифікує таблицю, яка містить дані, що читає запит. Такі таблиці називаються вихідними таблицями запитів (і оператора SELECT), оскільки всі дані, що містяться у таблиці результатів запиту, беруться з них.

FROM Students, Marks

Псевдоніми таблиць. Іноді зручно використовувати псевдонім таблиці замість її назви в запиті, наприклад, якщо запит стосується таблиці іншого користувача, або якщо назва таблиці дуже довга. Крім того, псевдоніми таблиць потрібні в запитах, які передбачають самооб'єднання таблиць (див. нижче).

Псевдоніми таблиць задаються праворуч від назви реальної таблиці, наприклад:

FROM Students Stdd, Marks Mk

Ці псевдоніми можна використовувати скрізь, де використовуються назви відповідних таблиць.

6.4. Відбір рядків. Секція WHERE

SQL-запити, які зчитують усі рядки з таблиці, корисні для перегляду бази даних і створення звітів, але на практиці рідко використовуються для прикладних задач. Зазвичай потрібно вибрати лише декілька рядків з таблиці і включити їх у результати запиту. Щоб вказати умову, за якою потрібно відібрати рядки, використовується оператор WHERE.

Секція WHERE складається з ключового слова WHERE, за яким слідує умова пошуку, що визначає, які саме рядки потрібно прочитати.

Якщо для рядка умова пошуку є TRUE, рядок буде включений до результатів запиту.

Якщо для рядка умова пошуку є FALSE або NULL, то рядок виключається із результатів запиту.

У SQL існує багато умов пошуку, що дозволяють ефективно і природно створювати різні типи запитів. Нижче розглядаються 5 основних умов пошуку (в стандарті ANSI/ISO вони називаються предикатами).

6.4.1. Оператори порівняння

Найпоширенішою умовою пошуку в SQL є порівняння =, <, <=, >, >=, <>. При порівнянні SQL обчислює і порівнює значення двох виразів для кожного рядка даних. Вирази можуть бути як дуже простими, наприклад, містити одну назву стовпця або константу, так і складнішими – арифметичними та іншими виразами.

Нижче наведено синтаксис оператора порівняння.

Вираз1 = | <> | < | <= | > | >= Вираз2

Приклад. Отримати список студентів, які мають стипендію > 650 грн.

SELECT StdNmb, StdName

FROM Students
WHERE StdStipend >650

6.4.2. Перевірка на належність до діапазону значень. Секція BETWEEN

Іншою формою умови пошуку є перевірка на належність до діапазону значень за допомогою ключового слова BETWEEN. При цьому перевіряється, чи знаходиться значення даних між двома визначеними значеннями. Формат команди має наступний вигляд:

Вираз_що_перевіряється [**NOT**] **BETWEEN** нижній_вираз **AND** верхній_вираз

При перевірці приналежності до діапазону верхня і нижня межі вважаються частиною діапазону, тому вони впливають на результати запиту.

Якщо вираз, що визначає нижню меу діапазону, дорівнює NULL, то перевірка BETWEEN повертає значення FALSE, якщо значення, яке перевіряється, більше верхньої межі діапазону, і значення NULL в іншому випадку.

Якщо вираз, що визначає верхню межу діапазону, має значення NULL, то перевірка BETWEEN повертає значення FALSE, якщо значення, яке перевіряється, менше нижньої меі, і значення NULL у іншому випадку.

Приклад. Отримати список студентів, у яких стипендія в діапазоні від 650 до 1100 грн.

SELECT StdNmb, StdName
FROM Students
WHERE StdStipend **BETWEEN** 650 **AND** 1100

6.4.3. Перевірка на належність до множини. Секція IN

Ще однією поширеною умовою пошуку є перевірка на належність до множини – секція IN. У цьому випадку перевіряється, чи відповідає значення даних якомусь значенню із визначеного списку.

вираз_що_перевіряється **[NOT] IN** (набір_констант)

Константи в множині констант відокремлюються одна від одної комами.

Приклад. Отримати список студентів, у яких стипендія складає 650, 700 або 900 грн.

SELECT StdName

FROM Students
WHERE StdStipend **IN** (650, 700, 900)

6.4.4. Перевірка відповідності шаблону. Секція LIKE

Для читання рядків, у яких вміст певного текстового стовпця співпадає з заданим текстом, можна використовувати просте порівняння, але іноді виникає потреба у порівнянні на основі фрагмента рядка.

Перевірка на відповідність шаблону – секція LIKE, дає змогу визначити, чи відповідає значення даних у стовпці певному шаблону.

Шаблон – це рядок, який може містити один або більше підстановних знаків. Ці знаки інтерпретуються особливим чином.

Назва_Стовпця **[NOT] LIKE** шаблон **[ESCAPE** Символ_Пропуску**]**

Підстановні знаки. Підстановний знак % відповідає будь-якій послідовності з нуля або більшої кількості символів. Наприклад, наступний запит виведе інформацію про студентів, прізвище яких починається з "Іван":

Приклад. Отримати дані про студентів, чиє прізвище починається з "Іван".

SELECT *
FROM Students
WHERE StdName **LIKE** 'Іван%'

Підстановний знак "_" (символ підкреслення) збігається з будь-яким окремим символом.

Приклад. Отримати дані про студентів з ім'ям «Наталія» або «Наталья».

SELECT *
FROM Students
WHERE StdName **LIKE** '%Натал_я'

Підстановні знаки можна розмістити в будь-якому місці рядка шаблону. В одному рядку може бути декілька підстановних знаків.

Символи пропуску. При перевірці рядків на відповідність шаблону може виявитися, що підстановні знаки входять до рядка символів як літерали. Наприклад, неможливо перевірити, чи міститься знак відсотка "%" у рядку, просто включивши його в шаблон, оскільки SQL вважатиме цей знак підстановним. Зазвичай це не викликає серйозних проблем, оскільки підстановні знаки досить рідко зустрічаються у іменах, назвах товарів та інших текстових даних, які зазвичай зберігаються в БД.

У стандарті ANSI/ISO визначено спосіб перевірки наявності у рядках літералів, що використовуються як підстановні знаки. Для цього

використовуються символи пропуску. Коли в шаблоні зустрічається такий символ, то символ, який йде безпосередньо після нього, вважається не підстановним знаком, а літералом. (Тобто пропускається символ). Безпосередньо за символом пропуска може слідувати або один із двох підстановних знаків, або сам символ пропуску, оскільки він також набуває особливого значення у шаблоні.

Символ пропуску визначається як рядок, що складається з одного символу, та секції ESCAPE. Нижче наведено приклад використання знаку долара "$" як символу пропуску:

Приклад. Отримати дані з таблиці "Data", де у полі результат міститься фрагмент тексту "менше 50%".

```
SELECT *
FROM Data
WHERE Result LIKE '%менше 50$% %' ESCAPE $
```

6.4.5. Перевірка на рівність значенню. Секція IS NULL

Значення NULL забезпечують можливість застосування тризначної логіки в умовах пошуку. Для будь-якого заданого рядка результат застосування умови пошуку може бути TRUE, FALSE або NULL, у випадку, коли один зі стовпців містить значення NULL.

Іноді виникає необхідність явно перевіряти значення стовпців на рівність NULL та обробляти їх безпосередньо. Для цього в SQL існує спеціальна перевірка на рівність значенню NULL – секція IS NULL.

назва_стовпця **IS [NOT] NULL**

Приклад. Вибрати студентів, які не отримують стипендії.

```
SELECT *
FROM Students
WHERE StdStip IS NULL
```

6.4.6. Складові умови пошуку. Секції AND, OR і NOT

Прості умови пошуку, які описано в попередніх пунктах, після застосування до певного рядка повертають значення TRUE, FALSE або NULL.

За допомогою правил логіки ці прості умови можна об'єднувати у складніші. Умови пошуку, об'єднані за допомогою ключових слів AND, OR або NOT, можуть бути складовими.

WHERE [NOT] умова_пошуку **[AND | OR] [NOT]** умова_пошуку …

Приклад. Вибрати студентів, які навчаються у групі КС-981 та не отримують стипендії.

SELECT *
FROM Students
WHERE StdGrp = StdNmb > 981621 **AND** StdStip IS NULL

6.5. Контрольні запитання

1. Опишіть призначення оператора SELECT.
2. Опишіть структуру оператора SELECT.
3. Опишіть структуру та призначення секції SELECT.
4. Опишіть структуру та призначення секції FROM.
5. Опишіть структуру та призначення секції WHERE.
6. Охарактеризуйте основні конструкції формування умов відбору рядків у секції WHERE.
7. Поясніть призначення умови BETWEEN.
8. Поясніть призначення умови IN.
9. Поясніть призначення умови LIKE.

6.6. Завдання для самопідготовки

Для бази даних Деканат складіть наступні запити:

Завдання 6.1. Виберіть з таблиці Students значення всіх стовпців.

Завдання 6.2. Виберіть з таблиці Students значення стовпців з іменами студентів та їх стипендією.

Завдання 6.3. Виберіть з таблиці Students значення стовпця з іменами студентів, у яких стипендія більше 1000 грн.

Завдання 6.4. Виберіть з таблиці Students значення стовпця з іменами студентів, у яких номер студентського квитка більше 981618 і стипендія менше 1000 грн.

7. ЗАПИТИ З ДЕКІЛЬКОМА ТАБЛИЦЯМИ

На практиці багато запитів зчитують дані одразу з декількох таблиць бази даних. У результаті виникає необхідність зв'язування таблиць між собою. Таке зв'язування можна задати декількома способами. Першим, традиційним, є зв'язування таблиць у секції WHERE. Другий спосіб зв'язування таблиць був введений у SQL2 і розглядається далі.

7.1. Природне з'єднання таблиць

Серед багатотабличних запитів найпоширенішими є запити з *природним зв'язуванням таблиць*. У таких запитах пов'язані стовпці представляють собою пару "зовнішній ключ – первинний ключ". Зв'язування таблиць виконується за рівністю значень цих ключів. Зв'язана таблиця утворюється парами рядків з різних таблиць, які мають однакові значення у пов'язаних стовпцях.

Прикладом такого запиту є запит щодо студентів, що навчаються в групі з певною назвою. У кожного студента (нащадка) є відповідна йому група (предок), і кожна група (предок) може мати багато студентів (нащадків). Таким чином, пари рядків, з яких формуються результати запиту, пов'язані відношенням предок/нащадок.

Зв'язати таблиці за деякими стовпцями (іноді їх називають полями) можна в секції WHERE.

Приклад. Отримати список прізвищ студентів і кураторів їх груп.

```
SELECT StdName, TchName AS Counsel
FROM Students, Groups, Teachers
WHERE Students.GrpNo = Groups.GrpNo
      AND Groups.GrpCns = Teachers.TNo
```

Приклад. Знайти студентів і перелік предметів, за якими студентами була отримана оцінка менше 4.

```
SELECT StdName, SubjName
FROM Students, Marks, Subjects
WHERE Marks.Mark < 4
      AND Students.StdNmb = Marks.StdNmb
      AND Marks.SubjNo = Subjects.SubjNo
```

Приклад. Знайти студентів, які склали іспит з дисципліни «Програмування» з балом 5

```
SELECT StdName
FROM Students, Marks, Subjects
```

WHERE Students.StdNmb = Marks.StdNmb
AND Marks.SubjNo = Subjects.SubjNo
AND Subjects.SubjName = 'Програмування'
AND Marks.Mark = 5

Хоча більшість багатотабличних запитів ґрунтуються на природному зв'язуванні, загалом, у SQL будь-які 2 стовпці з 2 таблиць можуть бути зв'язуючими стовпцями, якщо тільки вони мають типи даних, сумісні за порівнянням. За змістовність такого зв'язування відповідає автор запиту.

Найпоширенішими є зв'язування таблиць, створених на основі рівності зв'язуючих стовпців (об'єднання за рівністю). Однак SQL дозволяє об'єднувати таблиці за допомогою інших операцій порівняння, зокрема, об'єднання за нерівністю.

Приклад. Отримати всі коди дисциплін, за якими студентом з кодом 1 була отримана оцінка вище, ніж за дисципліною з кодом 1.

SELECT Marks1.SubjNo
FROM Marks AS Marks1, Marks AS Marks2
WHERE Marks1.Mark > Marks2.Mark
AND Marks1.StdNmb = 981621
AND Marks2.SubjNo = 123

Примітка. Текст цього запиту передбачає самооб'єднання таблиць, яке розглядається в наступному розділі.

Цей приклад має штучний характер і ілюструє, чому об'єднання за нерівністю зустрічаються так рідко. Однак вони можуть бути корисними в додатках, призначених для підтримки прийняття рішень, та в інших додатках, що досліджують більш складні взаємозв'язки в базі даних.

7.2. Запити із самооб'єднання таблиць

Деякі багатотабличні запити використовують відношення (зв'язки), які фактично існують всередині однієї й тієї ж таблиці. Наприклад, у таблиці містяться не лише відомості про самих викладачів, а й про їх службове підпорядкування. Фактично потрібно мати дві таблиці – про самих викладачів і про їх службове підпорядкування, і для отримання інформації про викладачів і їх підпорядкування потрібно склеювати ці таблиці між собою. Проте існують ситуації, коли використання двох таблиць є зайвим. Фактично можна обмежитись використанням однієї таблиці "Teachers" з додаванням стовпця про підпорядкованість викладачів. Проте питання про склеювання таблиць

залишається. Таким чином, виникає задача, коли необхідно, щоб окрім вихідної таблиці, мати її копію. Після чого ці таблиці можна зв'язати.

Замість того, щоб фактично робити реальну копію таблиці в SQL для об'єднання таблиці з самою собою (самооб'єднання таблиць) застосовується підхід, що полягає в використанні "віртуальної копії" таблиці, на яку можна посилатися, використовуючи псевдонім (аліас) таблиці.

При цьому кожен рядок таблиці об'єднується з копією цього ж рядка і копіями інших рядків цієї ж таблиці. Далі кожна така комбінація рядків оцінюється за допомогою заданої умови (критерію) пошуку, і залежно від істинності її значення включається або не включається в результуючу таблицю.

Приклад. Отримати назви предметів, що мають однакову кількість навчальних годин.

SELECT A.SubjName, B.SubjHours
FROM Subjects A, Subjects B
WHERE A.SubjHours = B.SubjHours

Варто підкреслити, що, в загальному випадку, результат буде містити зайві рядки, наприклад, *Програмування*, оскільки може бути декілька дисциплін з такою ж кількістю годин. Дублювання можна усунути, використовуючи ключове слово DISTINCT у секції SELECT. Наприклад:

SELECT DISTINCT A.SubjName, B.SubjHours
FROM Subjects A, Subjects B
WHERE A.SubjHours = B.SubjHours

Приклад. Отримати назви пар предметів, що мають однакову кількість навчальних годин.

SELECT A.SubjName, B.SubjName, B.SubjHours
FROM Subjects A, Subjects B
WHERE A.SubjHours = B.SubjHours

Оскільки предикат A.SubjHours = B.SubjHours є симетричним і призводить до з'єднання рядків з однаковою кількістю годин, будуть отримані пари виду:

Програмування Фізика
Фізика Програмування

а також пари наступних видів:

Програмування Програмування
Фізика Фізика

що абсолютно марно.

Для виключення рядків типу «Фізика» треба задати умову вибірки, яка не допускає склеювання кожного рядка з самим собою. У цьому випадку достатньо додати умову A.SubjName <> B.SubjName. Тоді отримаємо:

SELECT A.SubjName, B.SubjHours
FROM Subjects A, Subjects B
WHERE A.SubjHours = B.SubjHours
AND A.SubjName <> B.SubjName

Розглянемо ще один цікавий приклад самооб'єднання таблиць. Припустимо, що потрібно вивести список імен усіх викладачів і співробітників кафедри та їх безпосередніх керівників. Кожному члену кафедри відповідає один рядок у таблиці Teachers, а поле стовпця TchChief містить ідентифікатор керівника для відповідного співробітника. Стовпець TchChief повинен бути зовнішнім ключем для віртуальної таблиці Chiefs, в якій зберігаються дані про керівників. І він ним, фактично, є – це зовнішній ключ для самої таблиці Teachers.

Щоб відповісти на поставлений запит слід скористатися самооб'єднанням таблиць, і крім таблиці Teachers створити віртуальну таблицю Chiefs, після чого виконати склеювання цих таблиць за зовнішнім ключем таблиці Teachers і первинним ключем таблиці ChiefNo.

Приклад. Вивести список всіх викладачів і їх безпосередніх керівників.

SELECT Teachers.TchName, Chiefs.TchName
FROM Teachers, Teachers Chiefs
WHERE Teachers.TchChief = Chiefs.TchID

В результаті для кожного викладача буде виведено ім'я його безпосереднього керівника.

Приклад. Вивести керівника викладача Петренка.

SELECT Teachers.TName, Chiefs.TName
FROM Teachers, Teachers Chiefs
WHERE Teachers.TchName = ‘Петренко’ **AND**
Teachers.TchChief = Chiefs.TchID

Тут слід підкреслити, що таблиця Teachers містить ієрархічні дані, що включають в себе відомості не лише про безпосередніх, а й про прямих керівників. Проте розглянуті засоби не дозволяють отримати відомості про прямих керівників. Для цього потрібна рекурсія. Її реалізація можлива з використанням директиви WITH, яка буде розглянута нижче.

7.3. Внутрішнє та зовнішнє об'єднання таблиць

Операція природного об'єднання таблиць в SQL з'єднує інформацію з двох таблиць, утворюючи пари пов'язаних рядків з цих двох таблиць.

В загальному випадку об'єднану таблицю утворюють пари тих рядків з різних таблиць, в яких у пов'язаних стовпцях містяться однакові значення.

Зазвичай, якщо рядок однієї з таблиць не має пари в іншій таблиці, він не розглядається, і такий вид об'єднання таблиць називається внутрішнім об'єднанням (INNER JOIN). У деяких випадках це може призвести до втрати деяких даних у результатах запиту.

Іноді потрібно створювати об'єднання таблиць, у яких неоднакові значення у пов'язаних стовпцях, на основі яких здійснюється зв'язок. У цьому випадку застосовується зовнішнє об'єднання таблиць (OUTER JOIN), найбільш повно представлене в стандарті SQL2.

7.4. Контрольні запитання

1. Як формуються запити з багатьма таблицями?
2. Що таке природне з'єднання таблиць?
3. В чому полягає відмінність між внутрішнім та зовнішнім об'єднанням таблиць?
4. Поясніть, що таке самооб'єднання таблиць.
5. Опишіть, як реалізується самооб'єднання таблиць.

7.5. Завдання для самопідготовки

Для бази даних Деканат складіть наступні запити:

Завдання 7.1. Виберіть студентів та регіони, з яких вони прибули.

Завдання 7.2. Виберіть студентів групи КС-981 та міста, з яких вони прибули.

Завдання 7.3. Виберіть студентів групи КС-981, які склали іспит з дисципліни "Програмування" на відмінно.

Завдання 7.4. Знайдіть студентів, які на іспиті у викладача Петренка отримали оцінки "добре".

Завдання 7.5. Отримайте назви дисциплін, з яких студенти групи КС 981 склали іспит на "відмінно".

Завдання 7.6. Отримайте список викладачів та їх посади, які мають однакову посаду.

8. АГРЕГАТНІ ФУНКЦІЇ ТА ЗАПИТИ З ГРУПУВАННЯМ

8.1. Агрегатні функції

Для узагальнення інформації, що міститься в базі даних, в SQL передбачені агрегатні (статистичні) функції.

Агрегатна функція приймає в якості аргументу стовпець або частину стовпця даних і повертає одне значення, яке певним чином підсумовує цей стовпець або частину стовпця. Наприклад, агрегатна функція AVG() приймає в якості аргументу стовпець "Стипендія" і обчислює його середнє значення.

8.1.1. Склад агрегатних функцій

У SQL1 існує 6 агрегатних функцій, які дозволяють отримувати різні види підсумкової інформації. Нижче наведено синтаксис цих функцій:

1. Функція SUM() обчислює суму всіх значень, що містяться в стовпці

   SUM(вираз **| [DISTINCT]** назва_стовпця**)**
2. Функція AVG() обчислює середнє всіх значень, що містяться в стовпці

   AVG(вираз **| [DISTINCT]** назва_стовпця**)**
3. Функція MIN() знаходить найменше серед усіх значень, що містяться в стовпці

   MIN(вираз **|** назва_стовпця**)**
4. Функція MAX() знаходить найбільше серед усіх значень, що містяться в стовпці

   MAX(вираз **|** назва_стовпця**)**
5. Функція COUNT() підраховує кількість значень, що містяться в стовпці

   COUNT([DISTINCT] назва_стовпця**)**

Примітка. COUNT(DISTINCT) працює в ORACLE та MS SQL Server, але не працює в MS Access.

6. Функція COUNT(*) підраховує кількість рядків у таблиці результатів запиту

   COUNT(*)

У наступних версіях SQL було додано ряд інших агрегатних функцій, які тут лише перелічуються: FIRSRT(), LAST(), MAX(), MIN(), MID(), LEN(), ROUND(), NOW(), FORMAT(), UCASE(), LCASE().

Примітка. Агрегатні функції не можна застосовувати в секції WHERE, вони можуть використовуватися лише в секціях SELECT і HAVING (див. нижче).

8.1.2. Агрегатні функції та значення NULL

У стандарті ANSI/ISO визначені наступні точні правила обробки значень NULL у агрегатних функціях:

1. Якщо деякі значення, що містяться в стовпці, дорівнюють NULL, вони виключаються при обчисленні результату функції;
2. Якщо всі значення в стовпці дорівнюють NULL, то функції SUM(), AVG(), MIN() та MAX() повертають значення NULL; функція COUNT() повертає нуль;
3. Якщо в стовпці немає значень (тобто стовпець порожній), то функції SUM(), AVG(), MIN() та MAX() повертають значення NULL; функція COUNT() повертає нуль;
4. Функція COUNT(*) підраховує кількість рядків і не залежить від наявності або відсутності значень NULL у стовпці; якщо рядків у таблиці немає, ця функція повертає нуль.

Приклад. Знайти сумарне, середнє, мінімальне та максимальне значення стипендії студентів.

```
SELECT SUM(StStipend) AS Sm, AVG(StStipend) AS Av,
       MIN(StStipend) AS Mn, MAX(StStipend) AS Mx
FROM Students
```

Приклад. Знайти кількість студентів, які отримують стипендію.

```
SELECT COUNT(*) AS Cnt
FROM Students
WHERE StdStipend > 0
```

8.2. Сортування результатів запиту. Секція ORDER BY

Рядки результатів запиту, так само як і рядки таблиці бази даних, не мають визначеного порядку. Результати запиту можна відсортувати, включивши в оператор SELECT секцію ORDER BY. Ця секція складається з ключових слів ORDER BY, за якими слідує список назв стовпців, розділених комами. Таким чином, сортування може виконуватися за декількома стовпцями.

Синтаксис оператора ORDER BY має вигляд

ORDER BY назва_стовпця **[ASC | DESC]**, …

У секції ORDER BY можна вибрати порядок сортування за зростанням (ASC) або спаданням (DESC). За замовчуванням дані сортуються у порядку зростання. Щоб відсортувати їх у порядку спадання, слід включити до конструкції сортування ключове слово DESC.

Приклад. Вивести список прізвищ студентів у зворотному алфавітному порядку, які навчаються у групі з кодом 1.

```
SELECT StdName
FROM Students
WHERE GrpNo = 1
ORDER BY DESC StdName
```

Приклад. Вивести список груп у алфавітному порядку, а в межах груп список прізвищ студентів.

```
SELECT GrpName, StdName
FROM Groups, Students
WHERE StdGrp = GrpNo
ORDER BY GrpName, StdName
```

8.3. Запити з групуванням рядків. Секція GROUP BY

8.3.1. Підсумкові результати на основі груп рядків

Підсумкові запити, про які до цього йшла мова, обчислюють підсумкові результати за стовпцем на основі всіх записів у таблиці. Проте частіше, особливо при аналітичній обробці даних і складанні комплексних звітів, доводиться отримувати проміжні підсумки на основі окремих груп записів у таблиці, наприклад, по окремим навчальним групам. Ця обробка даних є надзвичайно актуальною в роботі промислових, торгових та інших підприємств. Цю можливість надає секція GROUP BY оператора вибору SELECT.

Запит, який включає секцію GROUP BY, називається *запитом з групуванням рядків* або просто *запитом з групуванням*, оскільки він об'єднує рядки вихідних таблиць в групи та для кожної групи рядків генерує один рядок таблиці результатів запиту. Стовпці, зазначені в секції GROUP BY, називаються стовпцями групування, оскільки саме вони визначають, за якою ознакою рядки таблиці розділяються на групи.

Примітка. В список вибирання полів оператора SELECT, що містить розділ GROUP BY, можна включати агрегатні функції та лише ті поля, які входять до умови групування.

Секція GROUP BY має загальну форму

GROUP BY [ALL | DISTINCT] елемент_групування **[, ...]**

Конструкція елемент_групування має декілька варіантів реалізації. Вони послідовно розглядаються нижче.

Якщо в елементі_групування задається GROUPING SETS, ROLLUP або CUBE, секція GROUP BY в цілому визначає певну кількість незалежних наборів групування. Це дає той же ефект, що й об'єднання підзапитів (з UNION ALL) з окремими наборами групування в їх секціях GROUP BY.

Необов'язкова секція DISTINCT виключає дублікати даних перед їх обробкою, але вона не перетворює UNION ALL в UNION DISTINCT. Детальніше використання наборів групування описується в підрозділі 8.3.4.

8.3.2. Групування записів за одним стовпцем

У цьому випадку елементом групування виступає назва відповідного стовпця таблиці.

Приклад. Отримати список студентів та їх середній бал.

SELECT StdName, **AVG**(Mark) **AS** AvgMark
FROM Students, Marks
WHERE Students.StdID = Marks.StdID
GROUP BY StdName

У цьому запиті елементом групування є назва стовпця StName. В результаті, наприклад, в об'єднаній таблиці 8.1 для кожного студента утворюється власна група, і для цієї групи обчислюється середня оцінка.

Таблиця 8.1. Об'єднана таблиця з елементом групування StName

StdName	Subject	Semestr	Marks		
Іваненко	Фізика	1	5	…	…
Іваненко	Хімія	2	4		
Іваненко	Математика	1	5		
Петренко	Фізика	1	4		
Петренко	Хімія	2	5		
Петренко	Математика	1	4		
Василенко	Математика	2	3		

У результаті отримуємо таблицю 8.2.

Таблиця 8.2. Результуюча таблиця з обчисленою середньою оцінкою

StdName	AvgMark
Іваненко	4,6
Петренко	4,3

З прикладу, зокрема, видно, що у список вибраних полів оператора SELECT, що містить секцію GROUP BY, включено поле StdName, за яким проведено групування, та агрегатна функція AVG(Mark). Інші поля в такому запиті включати не можна.

8.3.3. Групування записів за кількома стовпцями

У цьому випадку елементом групування виступає перелік назв відповідних стовпців таблиці.

Приклад. Отримати список студентів та їх середній бал за кожен семестр.

SELECT StdName, Semester, **AVG**(Mark) **AS** AvgMark
FROM Students, Marks
WHERE Students.StdNmb = Marks.StdNmb
GROUP BY StdName, Semester

У цьому запиті елементом групування є назва стовпця StdName і назва стовпця Semester. В результаті в об'єднаній таблиці для кожного студента утворюється власна група, а в всередині цієї групи формуються підгрупи за семестрами. Для кожної підгрупи кожного студента обчислюється середня оцінка.

Таким чином, відповідь буде виглядати як наведено у таблиці 8.3.

Таблиця 8.3. Результуюча таблиця студентів їх середнім балом за кожен семестр

StdName	Семестр	AvgMark
Іваненко	1	5
Іваненко	2	4
Петренко	1	4
Петренко	2	5

8.3.4. Групування записів за наборами групування. GROUPING SETS, ROLLUP, CUBE

Важливість, перш за все для економічного аналізу, різних варіантів групування призвела до появи у більш пізніх версіях SQL додаткових засобів для групування даних. Вони базуються на концепції *наборів групування*.

Дані, вибрані запитами SELECT, FROM і WHERE, групуються окремо для кожного заданого набору групування, а потім для кожної групи обчислюються агрегатні функції так, як це показано вище для простих запитів GROUP BY, і в кінці повертаються результати. Зміст результуючої таблиці при цьому виглядає не зовсім звично.

SQL пропонує 3 варіанти задання наборів групування – GROUPING SETS, ROLLUP і CUBE. Розглянемо їх послідовно.

8.3.4.1. Набір GROUPING SETS

Наприклад, маємо запит, в якому групування виконується за 3 групами – (StdName), (Semester) і (). На практиці у групу може входити декілька стовпців, але для прикладу достатньо і одного стовпця.

SELECT StdName, Semester, **AVG**(Mark) **AS** AvgMark
FROM Students, Marks
WHERE Students.StdNmb = Marks.StdNmb
GROUP BY GROUPING SETS ((StdName), (Semester), ())

Відповідь буде виглядати як наведено у таблиці 8.4.

Таблиця 8.4. Результат групування за трьома групами

StddName	Семестр	AvgMark
Іваненко		4,6
Петренко		4,3
	1	4,5
	2	4,5

З отриманого результату видно, що для кожної групи створюється свій набір рядків.

У кожному внутрішньому списку GROUPING SETS можуть задаватися нуль або більше стовпців або виразів, які розглядаються так само, як і якщо б вони були безпосередньо записані в запиті GROUP BY для групування за кількома стовпцями. Наприклад (StdName, Semester) і (Semester, Subject).

Порожній набір групування означає, що всі рядки зводяться до однієї групи, як це було у випадку з агрегатними функціями без секції GROUP BY.

8.3.4.2. Набір ROLLUP

Набір ROLLUP використовується для більш компактної форми запису поширеного варіанту набору GROUPING SETS, а саме замість запису

GROUPING SETS (
(a, b, c, ...),
...
(a, b,),
(a),
( )
)

можна використовувати запис

ROLLUP (a, b, c, ...)

Така форма групування часто використовується для аналізу ієрархічних даних, наприклад, для визначення середнього балу за факультетами, кафедрами та університетом в цілому.

Елементами запиту ROLLUP можуть бути як окремі вирази, так і вкладені списки елементів у дужках. Вкладені списки обробляються як атомарні одиниці, з якими формуються окремі набори групування, наприклад:

ROLLUP (a, (b, c), d))

еквівалентно

GROUPING SETS (
(a, b, c, d),
(a, b, c),
(a),
( )
)

8.3.4.3. Набір CUBE

Набір CUBE використовується для більш компактної форми запису іншого поширеного варіанту набору GROUPING SETS, а саме групування за усім списком та всіма його підмножинам. Отже, замість того, щоб писати

GROUPING SETS (
(a, b, c),
(a, b),
(a, c),
(a),
(b, c),
(b),
(c),
( )
)

можна використовувати запис

CUBE (a, b, c)

Елементами секції CUBE можуть бути або окремі вирази, або вкладені списки елементів у дужках. Вкладені списки розглядаються як атомарні одиниці, з якими формуються окремі набори групування, наприклад

CUBE ((a, b), (c, d))

еквівалентно

```
GROUPING SETS (
        (a, b, c, d),
        (a, b),
        (c, d),
        ( )
)
```

8.3.5. Обмеження на запити з групуванням рядків

Через свою специфіку запити з групуванням рядків мають деякі обмеження.

Стовпці, за якими групуються рядки, повинні бути реальними стовпцями таблиць, перелічених у секції FROM.

Не можна групувати рядки на основі значення обчислюваного виразу.

Крім того, існують обмеження на елементи списку повернених стовпців. Усі елементи цього списку повинні мати одне значення для кожної групи рядків. Це означає, що поверненим стовпцем може бути:

1. Константа.
2. Агрегатна функція, яка повертає одне значення для всіх рядків, що входять до групи.
3. Стовпець групування, який, за визначенням, має одне і те саме значення в усіх рядках групи.
4. Вираз, що включає в себе перелічені вище елементи.

На практиці в список повернених стовпців запиту з групуванням завжди включаються самі стовпці групування та агрегатні функції. Якщо агрегатна функція не вказана, це означає, що запит можна більш просто виразити за допомогою ключового слова DISTINCT без використання секції GROUP BY. І навпаки, якщо не включити стовпець групування в результати запиту, не можна буде визначити, до якої групи належить кожен рядок результатів.

8.3.6. Значення NULL у стовпцях групування

У стандарті ANSI визначено, що два значення NULL у секції GROUP BY рівні.

Рядки, які мають значення NULL у однакових стовпцях групування і однакові значення у всіх інших стовпцях групування, розміщуються в одну групу.

8.3.7. Умови вибору груп рядків. Секція HAVING

Так само, як секція WHERE використовується для вибору окремих рядків, які є частиною запиту, секція HAVING використовується для відбору груп рядків. Її формат такий самий, як і у секції WHERE.

Секція HAVING складається з ключового слова HAVING, за яким слідує умова пошуку. Ця умова пошуку дозволяє відбирати допустимі записи, що беруть участь у пошуку (аналогічно оператору WHERE), і відповідні групи записів. Відбір груп записів здійснюється за допомогою обмежень на значення агрегатних функцій для кожної групи.

8.3.8. Обмеження на умови пошуку груп рядків

Секція HAVING використовується для включення та виключення груп рядків із результатів запиту, тому на умови відбору, які він використовує, поширюються ті самі обмеження, що й на елементи списку повернених стовпців.

Примітка 1. Важливою особливістю HAVING, яка відрізняє його від WHERE, є те, що в HAVING можна використовувати агрегатні функції. Це дозволяє відбирати групи на основі обмежень на результати агрегатних функцій, що застосовуються до членів групи.

Примітка 2. В одному запиті можуть зустрітися як умови відбору рядків у розділі WHERE, так і умови відбору груп у розділі HAVING. Умови відбору груп не можна перемістити з розділу HAVING до розділу WHERE. Аналогічно, умови відбору рядків не можна перемістити з розділу WHERE до розділу HAVING, за винятком умов, що включають поля у списку групування GROUP BY.

Приклад. Отримати список студентів та їх середній бал для студентів, у яких середній бал > 4.5.

SELECT StdName, **AVG**(Mark) **AS** AvgMark
FROM Students, Marks
WHERE Students.StdNmb = Marks.StdNmb
GROUP BY StdName
HAVING AVG(Mark) > 4.5

Приклад. Отримати список студентів та їх середній бал для студентів групи з кодом 2, у яких середній бал > 4.5.

SELECT StdName, **AVG**(Mark) **AS** AvgMark
FROM Students, Marks
WHERE Students.StdNmb = Marks.StdNmb
GROUP BY StdName, GrpNo

HAVING GrpNo = 2 **AND AVG**(Mark) > 4.5

8.3.9. Секція HAVING без GROUP BY

Секція HAVING майже завжди використовується в поєднанні з секцією GROUP BY, але синтаксис оператора SELECT цього не вимагає. Якщо речення HAVING використовується без речення GROUP BY, SQL розглядає повні результати запиту як єдину групу, що аналогічно реченню WHERE. У цьому випадку агрегатні функції, що містяться в об'єкті HAVING, застосовуються до однієї і тільки однієї групи, і ця група складається з усіх рядків. На практиці речення HAVING дуже рідко використовується без відповідного речення GROUP BY.

8.4. Контрольні запитання

1. Що таке агрегатні функції і який їх склад?
2. Як відбувається сортування результатів запиту?
3. Що таке запити з групуванням рядків і для чого вони призначені?
4. Як формуються запити з групуванням рядків?
5. Як формуються запити з групуванням рядків за одним стовпцем?
6. Як формуються запити з групуванням рядків за кількома стовпцями?
7. Як формуються обмеження на умови відбору груп рядків?

8.5. Завдання для самопідготовки

Для бази даних Деканат складіть такі запити:

Завдання 8.1. Отримайте список студентів групи КС-981 та їх максимальний бал.

Завдання 8.2. Знайдіть кількість студентів групи КС-981, які не отримують стипендію.

Завдання 8.3. Отримайте список студентів групи КС-981 та їх середній бал за кожен семестр. Стовпець "Семестр" повинен бути включений до результату запиту.

Завдання 8.4. Отримайте список студентів та їх середній бал, для студентів, у яких максимальний бал > 4.5.

Завдання 8.5. Отримайте список студентів та їх середній бал, для студентів групи з кодом 2, у яких максимальний бал > 4.5.

9. ВКЛАДЕНІ ЗАПИТИ. ПІДЗАПИТИ

9.1. Структура та типи вкладених запитів

Вкладеними запитами називаються запити, що містяться в секції WHERE або HAVING поточного запиту. Вкладені запити дозволяють природним чином обробляти запити, виражені через результати інших запитів. У цьому контексті їх іноді називають асоціативними підзапитами.

Використання вкладених запитів є достатньо поширеним, зокрема, при пошуку прихованих залежностей між даними за схемою ЩО – ЯКЩО.

У SQL допускається будь-яка вкладеність запитів, але конкретні реалізації обмежують допустимий рівень вкладеності.

Найчастіше вкладені запити зазначаються в секції WHERE оператора SQL. Коли вкладений запит міститься в даному реченні, він бере участь у процесі відбору рядків.

Примітка. У вкладеному запиті можна явно не вказувати, але оперувати таблицями зовнішнього запиту.

У загальному випадку вкладений запит може повертати результати 3-х типів:

1. Стовпець.
2. Група стовпців.
3. Таблиця.

Якщо вкладений запит повертає один стовпець, то можуть бути випадки, коли цей стовпець складається лише з одного значення. У цьому випадку часто кажуть, що запит повернув одне значення, хоча насправді він повернув стовпець з одним значенням.

Якщо вкладений запит повертає таблицю, то це означає, що він повертає групу стовпців у вигляді таблиці, і їй задається назва. Цю таблицю можна використовувати далі в тексті запиту.

На практиці найчастіше використовуються вкладені запити, які повертають один стовпець.

При обробці всього запиту система спочатку обробляє найбільш вкладений запит, а потім послідовно за рівнями вкладеності до найзовнішого запиту.

Вкладений запит завжди береться в круглі дужки, але, як і раніше, зберігає структуру оператора SELECT, що містить секцію FROM та необов'язкові секції WHERE, GROUP BY та HAVING. Структура цих секцій у вкладеному запиті ідентична їх структурі в операторі SELECT. У вкладеному запиті ці речення

виконують свої звичайні функції. Однак між вкладеним запитом та оператором SELECT є декілька відмінностей:

1. Вкладений запит не може містити секцію ORDER BY. Результати вкладеного запиту використовуються тільки всередині головного запиту і для користувача залишаються невидимими, тому сортувати їх немає сенсу.
2. Вкладений запит не може бути запитом на об'єднання (UNION) декількох різних операторів SELECT (див. нижче). Дозволяється використання лише одного оператора SELECT.
3. Запити можуть бути корельованими, тобто назви стовпців у вкладеному запиті можуть бути посиланнями на стовпці таблиць головного запиту. Це так звані зовнішні посилання. Такі посилання необов'язково повинні бути присутніми у вкладеному запиті.

Вкладений запит завжди є частиною умови пошуку в секції WHERE або HAVING. Залежно від результату вкладеного запиту – один стовпець або таблиця (група стовпців) – в умовах пошуку можуть застосовуватися різні типи операторів порівняння.

9.2. Підзапити, що утворюють стовпець

9.2.1. Структура підзапиту, що утворює стовпець

Таблиця результатів підзапиту, що формує стовпець, складається лише з одного стовпця. Це означає, що в реченні SELECT підзапиту завжди вказується один повертаємий стовпець, який може містити один або декілька елементів.

Приклад. Отримати список студентів, у яких староста групи є "Ткаченко".

Щоб відповісти на цей запит, необхідно спочатку отримати код групи, у якій староста є "Ткаченко".

```
SELECT StdName
FROM Students
WHERE GrpNo = (SELECT GrpNo
               FROM Groups
               WHERE Students.StdNmb = Groups.GrpHd AND
               Students.StdName = ‘Ткаченко’)
```

Спочатку знаходиться код групи, у якій староста "Ткаченко", а потім за цим кодом знаходяться всі студенти групи. Результатом вкладеного запиту є стовпець з одним значенням.

Приклад. Вивести список студентів, середній бал яких вище за 4,5.

SELECT StdName
FROM Students
WHERE (SELECT AVG(Mark)
FROM Marks
WHERE Marks.StdNmb = Students.StdNmb**)** > 4.5

Результатом підзапиту є стовпець з одним значенням – результатом роботи функції.

Тут у підзапиті використовується вказівка на стовпець StdNmb таблиці Students з зовнішнього запиту. Таким чином, підзапит містить параметр – зовнішнє посилання, передане з основного запиту – номер студента Students.StdNmb. Такі підзапити називаються *корельованими (Correlated)*. Зовнішнє посилання приймає значення для кожного рядка-кандидата, що оцінюється за допомогою підзапиту, тому підзапит повинен виконуватися знову для кожного рядка, що відбирається в основному запиті.

9.2.2. Порівняння з єдиним результатом вкладеного запиту

Порівняння з єдиним результатом вкладеного запиту (=, <>, <, <=, >, >=) порівнює значення виразу з єдиним значенням, яке повертає вкладений запит.

Примітка. Фактично повертається стовпець з одним значенням.

Ця перевірка схожа на просте порівняння.

Вираз_що_перевіряється | = | <> | < | <= | > | >= | вкладений_запит

Порівняння з результатом вкладеного запиту є модифікованою формою простого порівняння. Значення виразу порівнюється зі значенням, яке повертає підзапит, і якщо умова порівняння виконується, то перевірка повертає значення TRUE. Ця перевірка використовується для порівняння значення з рядка, що перевіряється, з одним значенням, яке повертає підзапит, як показано в першому прикладі.

Приклад. Отримати список студентів, у яких староста групи є "Ткаченко".

SELECT StdName
FROM Students
WHERE GrpNo = **(SELECT** GrpNo
FROM Groups
WHERE Students.StdNmb = Groups.GrpHd **AND**
Students.StdName = ‘Ткаченко’**)**

Спочатку знаходять код групи зі старостою групи "Ткаченко", а потім за цим кодом знаходять всіх студентів групи.

9.3. Запити на приналежність до множини та запити з кількісними обмеженнями

Перевірка належності до множини реалізується за допомогою оператора IN.

Запити з кількісними обмеженнями – кванторами, реалізуються за допомогою операторів – EXISTS, ANY (SOME) та ALL.

Запити з перевіркою самого факту наявності реалізуються за допомогою оператора EXISTS.

Запити з багаторазовим порівнянням реалізуються за допомогою операторів ANY (SOME) та ALL.

Використання кожного типу запитів розглядається нижче окремо.

9.4. Перевірка належності IN множині значень вкладеного запиту

Перевірка належності результатам вкладеного запиту – ключове слово IN, є видозміненою формою простої перевірки на членство у множині:

вираз_що_перевіряється [**NOT**] **IN** вкладений_запит

Одне значення порівнюється зі стовпцем даних, поверненим вкладеним запитом, і якщо це значення дорівнює одному зі значень у стовпці, перевірка повертає TRUE. Ця перевірка використовується, коли потрібно порівняти значення з рядка, що перевіряється, із множиною значень, повернених вкладеним запитом.

Приклад. Вивести всю інформацію про студентів, які навчаються в групах з назвами, що починаються з літер "КС".

```
SELECT *
FROM Students
WHERE GrpNo IN (SELECT GrpNo
                FROM Groups
                WHERE GrpName LIKE 'КС%')
```

В результаті створюється множина кодів груп, у яких назва починається з "КС", потім ці значення будуть використані як умова для вибору рядків з таблиці Students.

Приклад. Отримати перелік предметів, за якими студенти отримали оцінку < 4.

```
SELECT SubjName
FROM Subjects
```

WHERE SubjNo **IN (SELECT** SubjNo
FROM Marks
WHERE Mark < 4**)**

В результаті створюється множина кодів предметів, за якими оцінка Mark < 4, потім ці значення служать умовою для вибору рядків з таблиці Subjects.

У підзапиті та в зовнішньому запиті може використовуватися одна й та ж таблиця.

Приклад. Отримати коди студентів, які складали ті ж самі іспити, що й студент з кодом 981624.

SELECT DISTINCT StdNmb
FROM Marks
WHERE SubjNo **IN (SELECT** SubjNo
FROM Marks
WHERE StdNmb = '981624'**)**

9.5. Перевірка факту існування. Ключове слово EXISTS

За допомогою перевірки на існування EXISTS можна визначити, чи міститься в результаті вкладеного запиту хоча б один рядок значень групи стовпців. Аналогічної простої перевірки не існує. Перевірка на існування EXISTS використовується тільки з вкладеними запитами.

[**NOT**] **EXISTS** вкладений_запит

Такий вираз вважається істинним, якщо результат обчислення вкладеного запиту являє собою не порожню множину, тобто існує хоча б один запис, що повертається вкладеним запитом.

На практиці підзапит для EXISTS завжди буде корельованим підзапитом, в якому у критеріях пошуку використовується посилання на елемент зовнішнього запиту. Так, у наведеному нижче прикладі використовується порівняння Marks.StdNmb = Students.StdNmb, яке звертається до елементів таблиці Students з зовнішнього запиту.

Приклад. Знайти студентів, які в 1-му семестрі здали іспит хоча б з однієї дисципліни.

SELECT StdName
FROM Students
WHERE EXISTS (SELECT *
FROM Marks
WHERE Marks.StdNmb = Students.StdNmb **AND**

```
                    Semester = 1)
                )
```

Приклад. Знайти студентів, які склали іспит з дисципліни з кодом 002.

```
SELECT StdName
FROM Students
WHERE EXISTS (SELECT *
                FROM Marks
                WHERE StdNmb = Students.StdNmb AND
                      SbjNo = '002'
              )
```

Результат виконання запиту буде еквівалентним результату наступної послідовності дій:

1. Прочитати таблицю Students, щоразу виконуючи підзапит з новим значенням номера студента;
2. Включати в результат запиту тільки ті рядки з таблиці Students, для яких підзапит повернув не пусту множину рядків.

Приклад. Знайти студентів, які не склали іспит з дисципліни з кодом 002.

```
SELECT StdName
FROM Students
WHERE NOT EXISTS (SELECT *
                    FROM Marks
                    WHERE StdNmb = Students.StdNmb AND
                          SbjNo = ‘002’
                  )
```

У запиті будуть відібрані ті рядки з таблиці Students, для яких підзапит не повертає жодного рядка.

Примітка. Фактично будь-який запит, що використовує оператор належності IN, можна записати за допомогою EXISTS (але не навпаки).

Приклад. Знайти студентів, які склали іспит з дисципліни з кодом 002.

```
SELECT StdName
FROM Students
WHERE StdNmb IN (SELECT StdNmb
                   FROM Marks
                   WHERE StdNmb = Students.StdNmb AND
                         SbjNo = ‘002’
                 )
```

Примітка. Характерною ілюстрацією використання оператора EXISTS є реалізація ним операції ділення (DIVIDE BY) в реляційній алгебрі. Варто підкреслити, що типові запити, які реалізуються за допомогою операції ділення, зазвичай містять у своєму формулюванні слово "всі"!

Розглянемо приклад для бази даних "Постачання".

Приклад. Знайти постачальників, які постачають всі деталі:

```
SELECT DISTINCT CmpName
FROM Companies
WHERE NOT EXIST
   (SELECT *
   FROM Details
   WHERE NOT EXIST
      (SELECT *
      FROM Supplies
      WHERE
         Supplies.DtlNo = Details.DtlNo AND
         Supplies.SplNo = Companies.CmpNo
      )
   )
```

Даний запит містить 2 вкладені підзапити і реалізує реляційну операцію ділення відношень.

Найвнутрішній підзапит параметризований двома значеннями (Detail.DtlNo, Companies.CmpNo) і має наступний зміст: вибрати всі рядки, що містять дані про постачання постачальника з номером CmpNo деталі з номером DtlNo. Заперечення NOT EXISTS вказує на те, що даний постачальник не постачає дану деталь. Зовнішній до нього підзапит, який також є вкладеним і параметризований параметром Companies.CmpNo, має наступний зміст: вибрати перелік деталей, які не постачаються постачальником SplNo. Заперечення NOT EXISTS вказує на те, що для постачальника з номером SNo не повинно бути деталей, які не постачалися б цим постачальником. Це в точності означає, що в зовнішньому запиті відбираються тільки постачальники, які постачають всі деталі.

Розглянемо аналогічний приклад для бази даних «Деканат ВНЗ».

Приклад. Знайти студентів, які склали іспити з усіх дисциплін.

Виявляється, що зовні простий запит має досить складне втілення. Причина в тому, що для відбору конкретного студента необхідно порівняти множину дисциплін, що вивчаються ним, з множиною всіх дисциплін, що вивчаються.

Однак в SQL немає операції порівняння 2-х множин. Тому необхідно переформулювати запит так, щоб обійти це обмеження. Щоб пояснити втілення запиту пройдемося по ряду його еквівалентних трансформацій:

1. Знайти студентів, які склали іспит з усіх дисциплін;
2. Знайти студентів з таблиці Marks, для яких не існує нескладених ними дисциплін з таблиці Subjects;
3. Знайти студентів з таблиці Marks, для яких не існує кодів дисциплін з таблиці Subjects, які вони не склали;
4. Знайти студентів з таблиці Marks, для яких не існує тих кодів дисциплін з таблиці Subjects, для яких не існує рядків про складені дисципліни в таблиці Marks для даного студента і даної дисципліни.

Останній вираз дослівно перекладається на SQL. При перекладі виразу на мову SQL потрібно врахувати, що у внутрішньому підзапиті таблиця Marks повинна бути перейменована, щоб відрізняти її від цієї ж таблиці, яка використовується в зовнішньому запиті. В результаті отримаємо запит у вигляді:

```
SELECT DISTINCT StdName
FROM Student
WHERE NOT EXIST
(SELECT *
FROM Subjects
WHERE NOT EXIST
        (SELECT *
        FROM Marks
        WHERE Marks.StdNmb = Student.StdNmb AND
        Marks.SbjNo = Subjects.SbjNo
        )
    )
```

Цей запит містить два вкладені підзапити і реалізує реляційну операцію ділення відношень.

Найвнутрішній підзапит параметризований двома параметрами (Student.StdNmb, Subjects.SubjNo) і має наступний зміст: вибрати всі рядки, що містять дані про оцінки студента з номером StdNmb за дисципліну з номером SubjNo. Заперечення NOT EXISTS вказує на те, що даний студент не склав дану дисципліну. Зовнішній до нього підзапит, який також є вкладеним і параметризованим параметром Student.StdNmb, має наступний зміст: вибрати список дисциплін, за якими немає оцінок у студента з номером StdNmb. Заперечення NOT EXISTS означає, що для студента з номером StdNmb не

повинно бути дисциплін, за якими не було складено іспит. Це в точності означає, що в зовнішньому запиті відбираються тільки студенти, які склали всі іспити.

Мова SQL має велику гнучкість. Так, наприклад, цей же запит можна реалізувати замінивши роботу з множинами на роботу з кількістю елементів у цих множинах.

```
SELECT StdNmb
FROM Marks
GROUP BY StdNmb
HAVING COUNT (Mark) = (SELECT COUNT (*)
                       FROM Subjects
                      )
```

У цьому запиті спочатку підраховується кількість кодів всіх дисциплін з Subjects. Потім виконується порівняння кількості оцінок, отриманих окремим студентом, з кількістю всіх вивчених дисциплін і вибираються ті студенти, для яких це порівняння істинне.

9.6. Багаторазове порівняння. Ключові слова ANY і ALL

Раніше, коли розглядалася перевірка входження в множину IN, було показано, що така перевірка визначає, чи не дорівнює деяке значення одному зі значень, що містяться в стовпці результату вкладеного запиту.

У SQL існують два різновиди багаторазового порівняння – ANY (SOME) і ALL, які розширюють попередню перевірку IN до порівняння якогось значення зі стовпцем даних, що повертається вкладеним запитом. У обох перевірках має місце модифікація значення операторів порівняння, коли порівнюються не дві величини, а якась величина і значення стовпця. Синтаксис такого порівняння має наступний вигляд:

перевірка_виразу | = | <> | < | <= | > | >= | ANY | ALL вкладений_запит

9.6.1. Ключове слово ANY

Під час перевірки ANY (SOME) використовується один із шести операторів порівняння (=, <>, <, <=, >, >=) для того, щоб порівняти одне значення зі стовпцем даних, що повертається вкладеним запитом.

Значення, яке потрібно перевірити, за чергою порівнюється з кожним значенням, що міститься в стовпці. Якщо хоча б одне з цих порівнянь дає результат TRUE, то перевірка ANY повертає значення TRUE.

Приклад. Знайти студентів, які отримали в першому семестрі принаймні одну оцінку відмінно.

```
SELECT StdName
FROM Students
WHERE 5 = ANY (SELECT Mark
                    FROM Marks
                    WHERE Marks.StdNmb = Students.StdNmb
                          AND Semester = 1)
```

Приклад. Знайти студентів університету, день народження яких співпадає з днем народження принаймні одного зі студентів групи КС-982.

```
SELECT *
FROM Students, Groups
WHERE StdBrthDate = ANY(SELECT Students.StdBrthDate
                        FROM Groups
                        WHERE Students.GrpNo = Groups.GrpNo
                              AND Groups.GrpName = 'КС-982')
                    AND Students.GrpNo = Groups.GrpNo
                    AND Groups.GrName <> 'КС-982'
```

Увага! З метою виключення інформації про студентів самої групи КС-982 з результату запиту, умову в зовнішньому запиті доповнено умовою.

```
Students.GrpNo = Groups.GrpNo
AND Groups.GrName <> 'КС-982'
```

Цікаво, що той самий результат можна отримати, використовуючи EXISTS.

Приклад. Знайти студентів університету, день народження яких співпадає з днем народження принаймні одного зі студентів КС-982 (з використанням EXISTS).

```
SELECT *
FROM Students A
WHERE EXISTS (SELECT *
                    FROM Students B, Groups
                    WHERE Students.GrpNo = Groups.GrpNo
                          AND Groups.GrpName = ‘КС-982’
                          AND A.StdBrthDate = B.StdBrthDate)
```

Якщо вкладений запит у перевірці ANY не створює жодного рядка результату або якщо результати містять значення NULL, то перевірка ANY може виконуватися специфічно в залежності від СУБД. У стандарті ANSI/ISO для мови SQL містяться докладні правила, що визначають результати перевірки

ANY, коли перевіряється значення порівняння зі стовпцем результатів вкладеного запиту:

1. Якщо вкладений запит повертає порожній стовпець результатів, то перевірка ANY має значення FALSE (в результаті виконання вкладеного запиту не отримано жодного значення, для якого виконувалась б умова порівняння).
2. Якщо операція порівняння має значення TRUE хоча б для одного значення у стовпці, то перевірка ANY повертає значення TRUE (є яке-небудь значення, отримане вкладеним запитом, для якого умова порівняння виконується).
3. Якщо операція порівняння має значення FALSE для всіх значень у стовпці, то перевірка ANY повертає значення FALSE (можна стверджувати, що ні для одного значення, отриманого вкладеним запитом, умова порівняння не виконується).
4. Якщо операція порівняння не має значення TRUE хоча б для одного значення у стовпці, але у ньому є одне або декілька значень NULL, то перевірка ANY повертає результат NULL. (У цій ситуації неможливо з впевненістю стверджувати, чи існує отримане вкладеним запитом значення, для якого виконується умова порівняння; воно може існувати, а може і ні – все залежить від "реальних" значень невідомих даних).

9.6.2. Ключове слово ALL

У перевірці ALL, як і в перевірці ANY, використовується один із шести модифікованих операторів (=, <>, <, <=, >, >=) для порівняння одного значення, яке перевіряється, зі стовпцем даних, що повертається вкладеним запитом. Значення, яке потрібно перевірити, за чергою порівнюється з кожним значенням, що міститься у поверненому стовпці. Якщо всі порівняння дають результат TRUE, то перевірка ALL повертає значення TRUE.

Приклад. Знайти студентів, які отримали у першому семестрі лише задовільні оцінки.

```
SELECT StdName
FROM Students
WHERE 3 = ALL (SELECT Mark
                FROM Marks
                WHERE Marks.StdNmb = Students.StdNmb
                      AND Semester = 1)
```

Приклад. Знайти студентів, які отримують стипендію більшу, ніж будь-який зі студентів групи КС-982.

```
SELECT *
FROM Students
WHERE StdStipend > ALL (SELECT StdStipend
                        FROM Students, Groups
                        WHERE Students.GrpNo = Groups.GrpNo
                              AND Groups.GrpName = 'КС-982')
```

Якщо вкладений запит у перевірці ALL не повертає жодного рядка або якщо результати запиту містять значення NULL, то в різних СУБД перевірка ALL може виконуватися по різному. У стандарті ANSI/ISO для мови SQL містяться докладні правила, що визначають результати перевірки ALL, коли перевіряється значення порівняння зі стовпцем результатів вкладеного запиту:

1. Якщо вкладений запит повертає порожній стовпець результатів, то перевірка ALL має значення TRUE. Вважається, що умова порівняння виконується, навіть якщо результати вкладеного запиту відсутні.
2. Якщо операція порівняння дає результат TRUE для кожного значення у стовпці, то перевірка ALL повертає значення TRUE. Умова порівняння виконується для кожного значення, отриманого вкладеним запитом.
3. Якщо операція порівняння дає результат FALSE для будь-якого значення у стовпці, то перевірка ALL повертає значення FALSE. У цьому випадку можна стверджувати, що умова пошуку не виконується для кожного значення, отриманого вкладеним запитом.
4. Якщо операція порівняння не дає результат FALSE для жодного значення у стовпці, але для одного або декількох значень дає результат NULL, то перевірка ALL повертає значення NULL. У цій ситуації неможливо з впевненістю стверджувати, чи справедлива умова порівняння для всіх значень, отриманих вкладеним запитом; може бути, справедлива, а може й ні – все залежить від "реальних" значень невідомих даних.

9.6.3. Вкладені запити у секції HAVING

Хоча вкладені запити найчастіше використовуються у секції WHERE, їх також можна використовувати і у секції HAVING головного запиту. Коли вкладений запит міститься у секції HAVING, він бере участь у відборі групи рядків.

9.7. Оператори UNION, EXCEPT та INTERSECT

Оператор UNION є частиною стандарту мови SQL, а оператори EXCEPT та INTERSECT входять до розширення SQL в таких СУБД, як MS SQL Server. Ці оператори повністю відповідають операторам реляційної алгебри UNION (об'єднання), MINUS (віднімання) та INTERSECT (перетин) двох відношень.

9.7.1. Оператор UNION

Оператор UNION об'єднує виведення двох або більше SQL запитів в єдиний набір рядків і стовпців.

Цей оператор корисний, коли потрібно переглянути схожі дані з різних таблиць.

Синтаксис оператора виглядає наступним чином:

Оператор **SELECT**
UNION [ALL] | EXCEPT | INTERSECT
Оператор **SELECT**
UNION [ALL] | EXCEPT | INTERSECT
Оператор **SELECT**
[, … n];

При об'єднанні двох відношень вони повинні бути сумісними за об'єднанням, тобто:

1. Таблиці мають однакову кількість стовпців;
2. і-й стовпець 1-ї таблиці і і-й стовпець 2-ї таблиці мають однаковий тип даних.

Як заголовки стовпців використовуються назви стовпців (або псевдоніми) першого запиту. Зазвичай в результат включають ще один додатковий стовпець, який фактично не існує в базі даних. Найчастіше його задають константою.

Кожен з операторів SELECT може мати свою власну умовну секцію WHERE, але весь запит може використовувати лише одну умовну секцію ORDER BY. Вона застосовується до всього результату і має знаходитися в останньому операторі SELECT.

Приклад. Отримати коди та імена студентів, які отримали стипендію більше 600 грн. або склали всі іспити в першому семестрі на 4.

SELECT StdNmb, StdName, ‘Підвищена стипендія’ **AS** What
FROM Students
WHERE StdStipend > 600
UNION

```
SELECT StdNmb, StdName, 'Гарна успішність' AS What
FROM Students
WHERE 4 = ALL (SELECT Mark
               FROM Marks
               WHERE Marks.StdNmb = Students.StdNmb AND
                     Semester = 1)
   ORDER BY 1
```

Результат запиту буде виглядати як у таблиці 9.1.

Таблиця 9.1. Результат запиту з оператором Union

StdNmb	StdName	What
981621	Іваненко	Підвищена стипендія
981622	Петренко	Гарна успішність
981623	Сидоренко	Підвищена стипендія

Основні особливості операції UNION:

1. За замовчуванням оператор UNION автоматично видаляє з результату рядки, що повторюються. Щоб уникнути цього, потрібно після оператора UNION додати ключове слово ALL.
2. За допомогою UNION можна об'єднати будь-яку кількість умовних секцій SELECT.
3. Для розрізнення рядків у результуючій таблиці корисно включати символьну константу, що показує її приналежність.
4. Якщо в запиті потрібне сортування, фраза ORDER BY повинна міститися тільки в останній секції SELECT, при чому у ній необхідно вказувати порядкові номери стовпців, а не їх назви, наприклад, ORDER BY 1.
5. Якщо в стовпцях, за якими здійснюється об'єднання, є значення NULL, ці значення будуть пропущені, оскільки немає підстав вважати, що одне невідоме значення збігається з іншим невідомим значенням.

9.7.2. Оператор EXCEPT

Оператор EXCEPT (Крім) є одним з основних операторів для фільтрації записів, коли для вибору записів використовуються 2 оператори SELECT. Він працює аналогічно оператору мінус в математиці.

Оператор EXCEPT повертає ті записи з лівого запиту SELECT, яких немає у результатах, що повертаються запитом SELECT у правій частині оператора EXCEPT.

Записи, що повертаються, представлені в єдиному екземплярі.

Назви та порядок стовпців у обох операторах повинні бути однаковими.

Приклад. Нехай є 2 таблиці книг – Усі Книги та Пригодницькі Книги. Отримати назви, категорії та ціни книг з таблиці Усі Книги за винятком книг з таблиці Пригодницькі Книги.

```
SELECT name, category, price FROM AllBooks
EXCEPT
SELECT name, category, price FROM AdventureBooks
```

Крім використання оператора EXCEPT для фільтрації записів з 2 різних таблиць, оператор EXCEPT також можна використовувати для фільтрації записів з однієї й тієї ж таблиці.

Приклад. Отримати коди та імена студентів, які отримують стипендію більше 600 грн. без тих студентів, хто склав всі іспити в першому семестрі на 4.

```
SELECT StdNmb, StdName, 'Отримують стипендію' AS What
FROM Students
WHERE StdStipend > 600
EXCEPT
SELECT StdNmb, StdName, 'Хороша успішність' AS What
FROM Students
WHERE 4 = ALL (SELECT Mark
               FROM Marks
               WHERE Marks.StdNmb = Students.StdNmb AND
                     Semester = 1)
ORDER BY 1
```

9.7.3. Оператор INTERSECT

Оператор INTERSECT (Перетин) повертає ті записи з лівого запиту SELECT, які присутні у результатах, що повертаються запитом SELECT у правій частині оператора INTERSECT. Він працює аналогічно оператору перетину в математичній логіці.

Записи, що повертаються оператором INTERSECT, представлені в єдиному екземплярі.

Назви та порядок стовпців у обох операторах повинні бути однаковими.

Приклад. Нехай є 2 таблиці книг – Книги про Подорожі та Пригодницькі Книги. Отримати назви, категорії та ціни книг, які присутні у обох таблицях з таблиці Усі Книги про Подорожі за винятком книг з таблиці Пригодницькі Книги.

```
SELECT name, category, price FROM TravelBooks
```

INTERSECT

SELECT name, category, price **FROM** AdventureBooks

Основні особливості операторів EXCEPT та INTERSECT:

1. Якщо порівнюються рядки з метою визначення різних значень, то два значення NULL вважаються однаковими.
2. Назви стовпців у результуючому наборі, що повертається оператором EXCEPT або INTERSECT, збігаються з назвами, що повертає запит, який зазначений зліва від вказаного оператора.
3. Назви стовпців або псевдоніми в реченнях ORDER BY повинні посилатися на назви стовпців, що повертає запит, зазначений ліворуч від оператора EXCEPT або INTERSECT.
4. Можливість зберігання пустих значень в якомусь стовпці результуючого набору, що повертається оператором EXCEPT або INTERSECT, залежить від того, чи підтримує це відповідний стовпець, що повертає запит, зазначений зліва від оператора.

9.8. Вкладені підзапити, що повертають таблиці або списки

Підзапити, що повертають таблиці або списки, використовуються рідко і вимагають певної кваліфікації.

9.8.1. Вкладені підзапити, що повертають таблиці

Слід відзначити, що підзапити, що повертають таблиці, повертають тимчасові таблиці, які не мають відповідності в схемі даних бази даних. Вони існують лише протягом виконання основного запиту і можуть бути використані для формування додаткових умов цього запиту. Проте в цьому випадку потрібно мати можливість звертатися до цих тимчасових таблиць як до звичайних таблиць. І для цього їм потрібно присвоювати назви – псевдоніми.

Підзапити, що створюють таблиці, повинні бути взяті у круглих дужках і обов'язково повинні мати псевдоніми. Наприклад:

…. FROM (SELECT * FROM Students …**) AS** *std_alias*

або, що теж саме

…. FROM *std_alias* **AS (SELECT * FROM** Students …**)**

Тепер тимчасову таблицю std_alias, отриману в результаті підзапиту, в інших частинах запиту можна розглядати як звичайну таблицю з назвою, заданою псевдонімом std_alias.

Ця конструкція еквівалентна розглянутим вище конструкціям у зовнішніх запитах, коли для зручності таблиці отримували псевдоніми.

… **FROM** Students **AS** std_alias …

Так вище був розглянутий приклад, в якому на різних рівнях вкладеності таблиця Students отримувала псевдоніми відповідно A і B.

Приклад. Знайти студентів університету, день народження яких співпадає з днем народження хоча б одного зі студентів КС-982 (з використанням EXISTS).

```
SELECT *
FROM Students AS A
WHERE EXISTS (SELECT *
        FROM Students AS B, Groups
        WHERE Students.GrpID = Groups.GrpID
              AND Groups.GrName = 'КС-982'
              AND A.StdBrthDate = B.StdBrthDate)
```

Приклад. Для демонстрації вкладеного підзапиту, що повертає таблицю, додаємо деяку штучну зміну в цьому прикладі і в якості поверненої таблиці з псевдонімом B використаємо підтаблицю студентів, у яких розмір стипендії більше 600 грн.

```
SELECT *
FROM Students AS A
WHERE EXISTS (SELECT *
                    FROM B AS (SELECT *
                                FROM Students
                                WHERE StddStip > 600), Groups
                    WHERE Students.GrpNo = Groups.GrpNo
                                AND Groups.GrpName = ‘КС-982’
                                AND A.StdBrthDate = B.StdBrthDate)
```

9.8.2. Вкладені підзапити, що повертають списки

Підзапити, що створюють списки VALUES, повинні бути взяті у круглі дужки і обов'язково повинні мати псевдоніми. Наприклад:

```
… FROM (VALUES ('Іван', 'Іваненко'), ('Петро', 'Петренко'),
                ('Михайло', 'Михайленко'))  AS  names(first, last)
```

Призначення псевдонімів колонкам списку VALUES не є обов'язковим, але це рекомендована практика. Наприклад, у цьому випадку стовпці списку з псевдонімом names мають псевдоніми first та last відповідно.

В окремому випадку підзапиту з VALUES може бути задана окрема константа, наприклад

… FROM *symbols(characters)* **AS** (**VALUES** ('qwertyuiopasdfghjklzxc'))

Змістовні приклади підзапитів, що утворюють таблиці або списки, будуть наведені пізніше при вивченні інших типів SQL-виразів.

9.9. Директива WITH

Для забезпечення більшої гнучкості в створенні та використанні підзапитів і, відповідно, тимчасових таблиць у більш пізніх версіях SQL було додано директиву WITH. Вона дозволяє задати один або декілька самостійних підзапитів – тимчасових наборів даних, до яких потім можна звертатися за назвою у основному SQL-запиті.

Самостійні підзапити, визначені у директиві WITH, отримали назву *загальні табличні вирази (CTE, Common Table Expressions)*. На них, точніше, на створені з їх допомогою тимчасові набори даних, можна посилатися в іншому операторі SQL, включаючи SELECT, INSERT, UPDATE або DELETE.

Директива WITH може містити один або список підзапитів, які передують основному запиту. У разі списку виконуються всі перелічені підзапити. По суті, вони формують тимчасові таблиці або представлення, до яких потім можна звертатися у списку FROM основного запиту.

Ключовою властивістю підзапитів WITH є те, що вони обчислюються один раз для всього основного запиту, навіть якщо у списку FROM основного запиту містяться декілька посилань на них. Зокрема, гарантується, що оператори, які змінюють дані, будуть виконуватися рівно один раз, незалежно від того, буде чи не буде їх результат прочитаний основним запитом і в якому обсязі.

Кожен підзапит у списку WITH може являти собою оператор SELECT, VALUES, INSERT, UPDATE або DELETE.

У випадку використання оператора SELECT як основного запиту синтаксис виглядає наступним чином

[WITH [RECURSIVE] запит_WITH **[, ...]**]
SELECT …

Підзапит WITH виглядає так:

назва_підзапиту_WITH **[(** назва_стовпця **[, ...])] AS** (вибірка)

Назву підзапиту необхідно вказувати для кожного запиту WITH. Можна задати необов'язковий список з іменами стовпців. Якщо список з іменами стовпців відсутній, імена стовпців формуються за результатом підзапиту.

Використаємо наведений вище приклад з вкладеним підзапитом, що повертає таблицю, і представимо його за допомогою директиви WITH.

Початковий запит

```
SELECT *
FROM Students as A
WHERE EXISTS (SELECT *
                    FROM B AS (SELECT *
                                      FROM Students
                                      WHERE StdStip > 600), Groups
                    WHERE Students.StdGrp = Groups.GrpID
                              AND Groups.GrpName = ‘КС-982’
                              AND A.StdBrthDate = B.StdBrthDate)
```

Аналогічний запит з WITH матиме вигляд:

```
WITH B AS (SELECT * FROM Students WHERE StdStip > 600)
SELECT *
FROM Students AS A
WHERE EXISTS (SELECT *
                    FROM B, Groups
                    WHERE Students.GrpNo = Groups.GrpNo
                              AND Groups.GrName = 'КС-982'
                              AND A.StdBrthDate = B.StdBrthDate)
```

Докладніше про директиву WITH представлено в документації за посиланням [4].

9.10. Рекурсивні запити

Рекурсивні запити зазвичай використовуються для роботи з ієрархічними або деревоподібними структурами даних.

Використання директиви WITH і створених за її допомогою підзапитів, що повертають тимчасові таблиці, надає можливість створення рекурсивних запитів.

Якщо в конструкції WITH вказано RECURSIVE:

WITH RECURSIVE запит_WITH **[, ...]**]

підзапит може посилатися сам на себе за назвою. У цьому випадку рекурсивний підзапит повинен мати вигляд:

WITH RECURSIVE Нерекурсивна_частина
UNION [ALL | DISTINCT]
Рекурсивна_частина

Це типова структура всіх рекурсивних запитів. Рекурсивне посилання на сам запит може знаходитися лише в рекурсивній частині праворуч від директиви UNION. Для одного запиту допускається лише одне рекурсивне посилання на нього ж.

Ще одна особливість використання директиви RECURSIVE полягає в тому, що підзапити WITH можуть бути невпорядкованими. Підзапит може посилатися на інший підзапит, що міститься в списку після нього. Проте циклічні посилання або взаємна рекурсія не підтримуються. Без RECURSIVE підзапит у WITH може посилатися лише на підзапити, що передують йому в списку WITH.

Для прикладу введемо таблицю 9.2 ParentOf (Батько) з двома стовпцями Parent (Батьки) і Child (Дитина).

Таблиця 9.2. Таблиця ParentOf

Parent	Child
Андрій	Іван
Надія	Іван
Анна	Василь
Іван	Василь
Василь	Петро
...	...

Приклад. Вивести всіх батьків Петра.

```
WITH RECURSIVE Ancestor (SELECT Parent AS Pr
                         FROM ParentOf
                         WHERE Child = "Петро"
    UNION ALL
        SELECT ParentOf.Parent
            FROM Ancestor, ParentOf
            WHERE Ancestor.Pr = ParentOf.Child)
SELECT * FROM Ancestor
```

У цьому прикладі вводиться рекурсивний підзапит Ancestor.

В нерекурсивній частині підзапиту (ліворуч від UNION ALL) знаходиться безпосередній батько Петра.

У рекурсивній частині підзапиту (праворуч від UNION ALL) відбувається рекурсивне звернення до підзапиту (тимчасової таблиці). Завдяки оператору UNION ALL відбувається об'єднання результатів нерекурсивної і рекурсивної частини підзапиту. Цей процес триває, поки можна знайти хоча б одного з батьків. Після завершення рекурсії витягується вміст тимчасової таблиці Ancestor.

Наведений вище приклад ілюструє рекурсію від нижнього до верхнього рівня вкладеності.

Розглянемо приклад рекурсії від верхнього до нижнього рівня вкладеності.

У наступному прикладі знаходяться всі нащадки Андрія і підраховується глибина їхнього споріднення.

```
WITH RECURSIVE Heirs (Distance, Child, Parent) AS (
  SELECT 1, Child, Parent
  FROM ParentOf
  WHERE Parent = 'Андрій'
 UNION ALL
  SELECT Heirs.Distance + 1, ParentOf.Child, ParentOf.Parent
  FROM Heirs, ParentOf
  WHERE Heirs.Child = ParentOf.Parent
 )
SELECT distance, Child FROM Heirs;
```

У наведеному вище прикладі важливо звернути увагу на те, як рекурсивно підраховується глибина споріднення Distance.

Розглянемо ще один приклад, який має велике практичне значення. Припустимо, що задано таблицю 9.3 – Devices (Пристрої), що складається з трьох стовпців: Unit (Модуль), Part (Частина) і Quantity (Кількість).

Таблиця 9.3. Devices (Пристрої)

Unit	Part	Quantity
Автомобіль	Тіло	1
Тіло	Двері	4
Двері	Замок	1
...	...	...

Приклад. Вивести всі складові частини автомобіля.

```
WITH RECURSIVE PartUnit (Part, Unit, Quantity) AS (
                    SELECT Part, Unit, Quantity
                    FROM Devices
```

```
                    WHERE Unit = 'Автомобіль'
        UNION ALL
            SELECT Dv.Part, Dv.Unit, Dv.Quantity
            FROM PartUnit Pu, Devices Dv
            WHERE Dv.Unit = Pu.Part)
    SELECT Part, SUM(Quantity) AS TotalQuantity
    FROM PartUnit
    GROUP BY Part
```

9.11. Контрольні запитання

1. Що таке вкладені або асоціативні запити?
2. Перелічіть типи результатів вкладених запитів і вкажіть їх відмінності.
3. Що таке корельовані запити?
4. Як формуються вкладені запити на отримання одного стовпця?
5. Як виконується порівняння з єдиним результатом вкладеного запиту?
6. Як формуються запити з групуванням рядків за декількома стовпцями?
7. Як формуються запити на перевірку належності множині значень вкладеного запиту IN?
8. Як формуються запити на перевірку факту наявності асоціації EXIST?
9. Як формуються запити на багаторазове порівняння з квантором ANY?
10. Як формуються запити на багаторазове порівняння з квантором ALL?
11. Як формуються запити на об'єднання результатів 2-х або більше SQL-запитів в єдиний набір рядків і стовпців?
12. Як формуються запити, що повертають рядки, наявні в одній таблиці і відсутні в іншій таблиці?
13. Як формуються запити, що повертають рядки, наявні в обох таблицях?
14. Як формуються запити, що повертають таблиці або списки?

9.12. Завдання для самопідготовки

Для бази даних Деканат зробіть наступні запити:

Завдання 9.1. Отримати список студентів, у яких куратор групи "Ткаченко".

Завдання 9.2. Отримати відомості про студентів, які навчаються у групах набору 83-го року.

Завдання 9.3. Знайти викладачів, які в 1-му семестрі приймали екзамени хоча б по одному предмету.

Завдання 9.4. Знайти викладачів, які в 1-му семестрі виставили хоча б одну оцінку відмінно.

Завдання 9.5. Знайти викладачів, які в 1-му семестрі виставили лише хороші оцінки.

Завдання 9.6. Отримати імена студентів, які проживають у Києві або здали всі екзамени в 1-му семестрі на 4.

Завдання 9.7. Нехай є дві таблиці підприємств: Постачальники і Споживачі. Отримати назви та адреси підприємств з таблиці Постачальники за винятком підприємств з таблиці Споживачі.

Завдання 9.8. Нехай є дві таблиці підприємств: Постачальники і Споживачі. Отримати назви та адреси підприємств, які наявні в обох таблицях Постачальники і Споживачі.

10. ВНУТРІШНЄ І ЗОВНІШНЄ З'ЄДНАННЯ ТАБЛИЦЬ. СТАНДАРТ SQL2

10.1. Природні, внутрішні та зовнішні з'єднання таблиць

Як зазначалося раніше, на практиці багато запитів читають дані одразу з декількох таблиць бази даних.

Операція *природного з'єднання (Natural Join)* таблиць у SQL з'єднує інформацію з двох таблиць на основі пов'язаних стовпців – зазвичай, це первинний та зовнішній ключі цих таблиць, які утворюють пари пов'язаних рядків даних, що містяться в них.

Операція *внутрішнього об'єднання (Inner Join)* таблиць з'єднує інформацію з двох таблиць, утворюючи пари пов'язаних рядків, але так, що якщо рядок однієї таблиці не має пари в іншій таблиці, він не розглядається і не входить в таблицю результату.

Варто зауважити, що використання у запиті секції WHERE передбачає виконання саме природного внутрішнього з'єднання таблиць.

У наведених вище прикладах розглядалися саме запити з природним внутрішнім з'єднанням та самоз'єднанням таблиць за рівністю (нерівністю) ключів.

Приклад. Отримати список прізвищ студентів та кураторів їх груп.

```
SELECT StdName, TchName AS Counsel
FROM Students, Groups, Teachers
WHERE Students.GrpNo = Groups.GrpNo
        AND Groups.GrpCns = Teachers.TNo
```

Тут природне внутрішнє з'єднання реалізується для таблиць Students і Groups, а також Groups і Teachers.

Практика показала, що використання внутрішнього з'єднання у деяких задачах може призвести до втрати в результатах запитів окремих даних.

Іноді потрібно створити з'єднання таблиць, які не мають пари пов'язаних рядків за стовпцями, на основі яких здійснюється зв'язування. Покажемо це на змістовному прикладі.

Припустимо, що за деяким регіоном є перелік компаній C і перелік видів виробів P, що виготовляються. Окрема компанія може виготовляти декілька видів виробів, а один вид виробу можуть виготовляти декілька компаній. У зв'язку з особливостями виробництва в певний період часу можливо, що є компанія, яка не виробляє жодного виду виробу, і є види виробів, які не виробляє

жодна компанія. У такому разі при внутрішньому з'єднанні цих переліків С та Р буде втрачено дані про компанії, які не виробляють жодного виду виробу, і про види виробів, які не виробляє жодна компанія.

Однак на практиці можуть знадобитися відповіді на наступні 3 типи запитів на з'єднання переліків С та Р:

1. Вибрати всі існуючі пари підприємство–виріб, а також всі підприємства, у яких на даний момент немає відповідних видів виробів.
2. Вибрати всі існуючі пари підприємство–виріб, а також всі види виробів, у яких на даний момент немає відповідних підприємств.
3. Вибрати всі існуючі пари підприємство–виріб, а також всі підприємства, у яких на даний момент немає відповідних видів виробів, і всі види виробів, у яких на даний момент немає відповідних підприємств.

У цьому випадку потрібно використовувати так зване зовнішнє з'єднання таблиць (Outer Join). При цьому розрізняють відповідно ліве (Left Outer Join), праве (Right Outer Join) та повне (Full Outer Join) зовнішнє з'єднання таблиць.

Підтримка зовнішнього з'єднання таблиць в SQL1 виявилась досить специфічною. Тому в стандарті SQL2 був визначений новий метод підтримки зовнішнього з'єднання, який не ґрунтувався на жодній популярній СУБД.

У специфікації стандарту SQL2 підтримка зовнішніх з'єднань здійснюється в секції FROM з ретельно розробленим синтаксисом, що дозволяє користувачу точно визначити, як саме вихідні таблиці повинні бути з'єднані в запиті.

Розширена секція FROM також підтримує операцію UNION над таблицями та дозволяє виконувати складні комбінації запитів на з'єднання операторів SELECT та з'єднання – злиття таблиць UNION.

10.2. Внутрішнє з'єднання таблиць в SQL2

Внутрішнє з'єднання двох таблиць відбувається явно за допомогою операції JOIN, а умова пошуку, що описує цю операцію, тепер знаходиться в секції ON всередині секції FROM. Умова пошуку, що розташовано після ключового слова ON, може містити будь-які критерії порівняння рядків двох таблиць, які потрібно з'єднати.

Приклад. Вивести список студентів та назви груп, в яких вони навчаються.

Його реалізація з використанням секції WHERE має вигляд

```
SELECT StdName, GrName
FROM Students, Groups
        WHERE Students.GrpNo = Groups.GrpNo
```

Або з використанням INNER JOIN

SELECT StdName, GrpName
FROM Students **INNER JOIN** Groups
ON Students.GrpNo = Groups.GrpNo

Стандарт SQL2 допускає ще один варіант запиту на просте внутрішнє з'єднання таблиць, наприклад Students і Groups. Якщо пов'язані стовпці цих таблиць мають однакові назви і порівнюються на рівність (що робиться досить часто), то можна використовувати альтернативну форму секції ON, в якій задається список назв пов'язаних стовпців:

SELECT StdName, GrpName
FROM Students **INNER JOIN** Groups **USING** (GrpNo)

10.3. Синтаксис оператора JOIN

Нижче наведено синтаксис оператора JOIN. Можна виділити 3 форми використання оператора JOIN:

1. Природне (NATURAL) об'єднання двох таблиць.

FROM специфікація_таблиць,…,
Таблиця_1
NATURAL {INNER | [OUTER] FULL | LEFT [OUTER] | [OUTER] RIGHT} JOIN
Таблиця_2 …

2. Об'єднання двох таблиць з використанням виразу ON або USING.

FROM специфікація_таблиць,…,
Таблиця_1
{INNER | [OUTER] FULL | [OUTER] LEFT | [OUTER] RIGHT} JOIN
Таблиця_2
ON умова | **USING**(список_стовпців),…

3. Об'єднання або декартовий добуток двох таблиць.

FROM специфікація_таблиць,…,
Таблиця_1 **{UNION | CROSS JOIN}** Таблиця_2,…

Перша форма використання оператора JOIN ґрунтується на природному об'єднанні двох таблиць, коли пов'язані стовпці – первинний та зовнішній ключі, мають ідентичні назви.

Приклад. Вивести прізвища студентів та назви груп, в яких вони навчаються, використовуючи внутрішнє природне об'єднання:

SELECT StdName, GrpName

FROM Students **NATURAL INNER JOIN** Groups

або

SELECT StdName, GrpName

FROM Students **NATURAL JOIN** Groups

Друга форма використання оператора JOIN надає можливість зв'язувати дві таблиці за будь-якими стовпцями. Тут можливі 2 варіанти:

1. Якщо склеювання таблиць відбувається із виконанням деяких умов, накладених на значення зв'язуючих стовпців, використовується секція ON умова.
2. Якщо склеювання таблиць відбувається за рівністю значень зв'язуючих стовпців і вони мають ідентичні назви, використовується секція USING(список_стовпців),….

Третя форма використання оператора JOIN за змістом відрізняється від двох попередніх і надає можливість з'єднання та перехресного з'єднання двох таблиць. Вона більш детально розглядається нижче.

10.4. Внутрішнє з'єднання таблиць

Внутрішні з'єднання таблиць розглядаються за замовчуванням, тобто оператор INNER можна опустити.

Внутрішні з'єднання таблиць можуть бути реалізовані за допомогою першої і другої форм використання оператора JOIN.

Наведемо приклад внутрішнього з'єднання, заснованого на природному з'єднанні двох таблиць, коли пов'язані стовпці – первинний і зовнішній ключі, мають ідентичні назви.

Приклад. Вивести прізвища студентів та назви груп, в яких вони навчаються, використовуючи внутрішнє природне з'єднання:

SELECT StdName, GrpName

FROM Students **NATURAL INNER JOIN** Groups

або

SELECT StdName, GrpName

FROM Students **NATURAL JOIN** Groups

Тепер наведемо той самий приклад внутрішнього з'єднання, що ґрунтується на з'єднанні двох таблиць за будь-якими стовпцями.

Приклад. Вивести прізвища студентів та назви груп, в яких вони навчаються, використовуючи внутрішнє з'єднання:

SELECT StdName, GrpName

FROM Students **INNER JOIN** Groups **ON** StdGrpID = GrpID

або

```
SELECT StdName, GrpName
FROM Students JOIN Groups ON StdGrpID = GrpID
```

Тепер розглянемо на прикладі склеювання 3 таблиць. За аналогією можна з'єднати довільну кількість таблиць.

Приклад. Вивести прізвища студентів, назви груп, в яких вони навчаються і прізвища кураторів цих груп, використовуючи внутрішнє з'єднання:

```
SELECT StdName, GrpName, TchName
FROM Students
   JOIN Groups ON StdGrpID = GrpID
      JOIN Teachers ON TchID = GrpCns
```

10.5. Зовнішні з'єднання таблиць

Стандарт SQL2 забезпечує повну підтримку зовнішніх з'єднань таблиць, шляхом розширення мовних конструкцій, що використовуються для внутрішніх з'єднань.

Приклад. Отримати таблицю підпорядкування викладачів.

Нагадаємо, що це приклад з самостійним з'єднанням двох таблиць. Тут використано повне зовнішнє з'єднання двох таблиць – FULL OUTER JOIN

```
SELECT Chief.TName, SubOrdinate.TName
FROM Teachers AS Chief FULL OUTER JOIN Teachers AS SubOrdinate
        ON Chief.TNo = SubOrdinate.TChiefNo
```

Результат такого запиту (дані з додатка А) наведено у таблиці 10.1.

Таблиця 10.1. Результат запиту на зовнішнє з'єднання TeachersFull

Chief.TName	SubOrdinate.TName
NULL	Іваненко
Іваненко	Петренко
Петренко	Лучниченко
Петренко	Сидоренко
Сидоренко	NULL
Лучниченко	NULL

Таблиця результатів запиту буде містити один рядок для кожної пов'язаної пари начальника/підлеглого, а також один рядок для кожного незв'язаного запису для начальника або підлеглого, розширеного значеннями NULL у стовпцях іншої таблиці.

Примітка. Слід зазначити, що згідно з умовами задачі в стовпці Chief.TName не може бути значень NULL, оскільки не може бути підлеглого, у якого немає керівника.

Ключове слово OUTER, так само як і ключове слово INNER, в стандарті SQL2 не є обов'язковим. Тому попередній запит можна переписати наступним чином:

```
SELECT Chief.TName, SubOrdinate.TName
FROM Teachers AS Chief FULL JOIN Teachers AS SubOrdinate
        ON Chief.TNo = SubOrdinate.TChiefNo
```

Зустрівши ключове слово FULL, СУБД автоматично визначає, що необхідно виконати зовнішнє з'єднання.

Природньо, що в стандарті SQL2 ліві і праві зовнішні з'єднання позначаються словами LEFT і RIGHT замість слова FULL.

Наведемо варіант того ж запиту, який визначає ліве зовнішнє об'єднання:

```
SELECT Chief.TName, SubOrdinate.TName
FROM Teachers AS Chief LEFT JOIN Teachers AS SubOrdinate
        ON Chief.TNo = SubOrdinate.TChiefNo
```

У результаті виконання такого запиту (дані з додатка А) буде отримано наступне співвідношення (таблиця 10.2).

Таблиця 10.2. Результат запиту на зовнішнє з'єднання TeachersLeft

Chief.TName	SubOrdinate.TName
Іваненко	Петренко
Петренко	Лучниченко
Петренко	Сидоренко
Сидоренко	NULL
Лучниченко	NULL

А це варіант того ж запиту, який визначає праве зовнішнє з'єднання:

```
SELECT Chief.TName, SubOrdinate.TName
FROM Teachers AS Chief RIGHT JOIN Teachers AS SubOrdinate
        ON Chief.TNo = SubOrdinate.TChiefNo
```

У результаті виконання такого запиту (дані з додатка А) буде отримано наступне співвідношення (таблиця 10.3). Варто порівняти отримане відношення з відношенням, наведеним у таблиці 10.2.

Таблиця 10.3. Результат запиту на зовнішнє з'єднання TeachersRight

Chief.TName	SubOrdinate.TName
Іваненко	Петренко
Петренко	Лучниченко
Петренко	Сидоренко

Розглянемо складніший приклад, де з'єднуються 3 таблиці і вибираються студенти та їх куратори.

Приклад. Отримати список прізвищ студентів та їх кураторів груп, використовуючи повне зовнішнє з'єднання.

```
SELECT StdName, TchName AS Counsel
FROM Students FULL JOIN Groups
    ON Students.GrpNo = Groups.GrpNo
        FULL JOIN Teachers
            ON Groups.GrpCns = Teachers.TNo
```

У цьому випадку значення NULL можуть бути у обох стовпцях.

Приклад. Отримати список прізвищ студентів та їх кураторів груп, використовуючи ліве зовнішнє з'єднання.

```
SELECT StdName, TchName AS Counsel
FROM Students LEFT JOIN Groups
        ON Students.GrpNo = Groups.GrpNo
            LEFT JOIN Teachers
                ON Groups.GrpCns = Teachers.TNo
```

У цьому випадку значення NULL можуть зустрічатись у стовпці кураторів.

Приклад. Отримати список імен студентів та імен їх кураторів груп, використовуючи праве зовнішнє з'єднання.

```
SELECT StdName, TchName AS Counsel
FROM Students RIGHT OUTER JOIN Groups
        ON Students.GrpNo = Groups.GrpNo
            RIGHT OUTER JOIN Teachers
                ON Groups.GrpCns = Teachers.TNo
```

У цьому випадку значення NULL можуть бути у стовпці студентів.

10.6. Перехресні з'єднання та запити на об'єднання

Розширена секція FROM в стандарті SQL2 також підтримує ще 2 способи з'єднання даних з двох таблиць – декартовий добуток і запити на з'єднання – злиття таблиць.

Строго кажучи, жоден з цих способів не є операцією "з'єднання", але вони підтримуються в стандарті SQL2 за допомогою тих же самих секцій, що і внутрішні та зовнішні об'єднання.

Приклад. Запит, який створює декартовий добуток таблиць Students і Subjects.

```
SELECT *
```

FROM Students **CROSS JOIN** Subjects

У результаті отримаємо таблицю, в якій кожному студенту буде відповідати перелік всіх предметів, і навпаки, кожному предмету буде відповідати перелік всіх студентів.

Раніше був розглянутий оператор UNION, який об'єднує дві або більше результатів виконання SQL запитів SELECT в єдиний набір рядків і стовпців.

В рамках SQL2 в розширеному операторі FROM існує директива UNION на об'єднання – злиття двох таблиць.

Приклад. Запит, який об'єднує дві таблиці AdventureBooks і TravelBooks.

SELECT *

FROM AdventureBooks **UNION** TravelBooks

У результаті буде отримана таблиця, яка об'єднує рядки таблиць AdventureBooks і TravelBooks.

10.7. Багатотабличні об'єднання

Однією з головних переваг розширеної секції FROM є те, що воно надає єдиний стандарт для визначення як внутрішніх, так і зовнішніх з'єднань, а також добутків і об'єднань таблиць.

Іншою, ще більш важливою перевагою цієї секції є те, що воно забезпечує зрозумілу і чітку специфікацію з'єднань 3-х, 4-х і більше таблиць, а також їх добутків і злиття.

Для побудови складних об'єднань таблиць будь-які вирази, описані раніше, можуть бути вкладені в круглі дужки. Результуючий вираз, в свою чергу, можна використовувати для створення інших виразів з'єднання, так само, як якщо б він був простою таблицею. Так само, як SQL дозволяє за допомогою круглих дужок комбінувати різні арифметичні операції (+, –, *, /) і побудову складних виразів, стандарт SQL2 дає можливість створювати складні вирази для з'єднань.

10.8. Вкладені запити та об'єднання

Багато запитів, написаних з використанням вкладених запитів, також можна записати у вигляді багатотабличних запитів. Таке трапляється досить часто, і SQL дозволяє створити запит будь-яким способом.

10.9. Реляційна повнота SQL

SQL є повноцінною мовою для роботи з БД. Вона є реляційно повною, оскільки за виразною силою не поступається реляційній алгебрі або, що те ж

саме, реляційному численню, тобто будь-який оператор реляційної алгебри може бути виражений засобами мови SQL. Відповідні можливості SQL були показані при розгляді реляційних операцій.

10.10. Секція SELECT INTO

У багатьох SQL-серверах (серверних СУБД) реалізовано можливість використання секції SELECT INTO.

Секція SELECT INTO найчастіше використовується для створення дубліката або резервної копії (Backup) таблиці.

Ця секція вибирає дані з однієї таблиці і вставляє їх в іншу таблицю.

Синтаксис секції SELECT INTO має такий вигляд:

SELECT * | назва_стовпця **[, …]**
INTO назва_нової_таблиці **[IN** зовнішня_БД**]**
FROM назва_старої_таблиці

Таким чином, секція SELECT INTO дозволяє скопіювати всю таблицю або певний набір її стовпців.

Вставка може бути виконана в іншу таблицю тієї ж бази даних або в таблицю зовнішньої бази даних, якщо буде вказаний шлях до зовнішньої бази даних і користувач має права доступу до цієї бази даних.

Приклад. Створити дублікат таблиці Students у таблиці StudentsBackup тієї самої БД.

SELECT *
INTO StudentsBackup
FROM Students

Можна використовувати секцію IN для копіювання вихідної таблиці до іншої бази даних.

Приклад. Створити дублікат таблиці Students у таблиці StudentsBackup бази даних Backup.mdb на тому ж комп'ютері.

SELECT *
INTO StudentsBackup **IN** 'Backup.mdb'
FROM Students

Можна створювати в новій таблиці копії лише кількох стовпців вихідної таблиці.

Приклад. Створити дублікат стовпців StdName і StdStipend таблиці Students у таблиці StudentsBackup на тому ж комп'ютері.

SELECT StdName, StdStipend

INTO StudentsBackup

FROM Students

Можна використовувати секцію WHERE для відбору даних, які потрібно скопіювати.

Приклад. Створити дублікат стовпців StdName і StdStipend для студентів з таблиці Students, у яких розмір стипендії більше 600 грн., у таблицю StudentsBackup на тому ж комп'ютері.

SELECT StdName, StdStipend

INTO StudentsBackup

FROM Students

WHERE StdStipend > 600

Можна використовувати секцію JOIN для вибору вихідних даних більше, ніж з однієї таблиці.

Приклад. Створити дублікат стовпців StdName і GrpName для студентів з таблиць Students і Groups у таблиці StudentsBackup на тому ж комп'ютері.

SELECT StdName, GrpName

INTO StudentsGroupsBackup

FROM Students **INNER JOIN** Groups

ON Students.GrpNo = Groups.GrpNo

10.11. Мова запитів за зразком QBE

Мова запитів за зразком QBE (Query by Example) була розроблена в 1975 р. для забезпечення інтерактивної взаємодії користувача з базою даних. Зараз вона використовується у таких СУБД як DB2, Paradox, MS Access.

Основна ідея мови полягає в тому, що користувач формує запит, вводячи у таблицю на екрані приклад можливої відповіді у відповідне місце спочатку порожньої таблиці. За можливостями QBE наближається до мов рівня SQL1, але має більш "дружній" інтерфейс для користувача-нефахівця.

Зрозуміло, що перед виконанням запиту, СУБД перетворює заповнений користувачем опис на відповідну мову (DB2 – на SQL, Paradox – PAL).

Якщо для відповіді на запит потрібно об'єднання таблиць, то вони послідовно викликаються на екран для заповнення.

Мова QBE реалізована в MS Access у вигляді Конструктора запитів. Для цього користувач викликає відповідну таблицю, вказує критерій вибірки у відповідному полі, список полів, що відображаються, та задає формат виводу.

10.12. Контрольні запитання

1. Що таке природне об'єднання таблиць?
2. Що таке внутрішнє об'єднання таблиць?
3. Що таке зовнішнє об'єднання таблиць?
4. Які обмеження несе внутрішнє об'єднання таблиць?
5. Навіщо потрібне зовнішнє об'єднання таблиць?
6. Які форми має оператор JOIN?
7. Яке призначення і можливості першої форми оператора JOIN?
8. Яке призначення і можливості другої форми оператора JOIN?
9. Яке призначення і можливості третьої форми оператора JOIN?
10. Що таке перехресне з'єднання таблиць і як воно реалізується?
11. Що таке злиття таблиць і як воно реалізується?
12. Яке призначення оператора SELECT INTO?
13. Як можна створити резервну копію таблиці?
14. Що таке запити за зразком?

10.13. Завдання для самопідготовки

Для бази даних Деканат складіть наступні запити:

Завдання 10.1. Отримати список викладачів і дисциплін, які вони викладають, за допомогою природного внутрішнього з'єднання.

Завдання 10.2. Отримати список викладачів і дисциплін, які вони викладають, за допомогою внутрішнього з'єднання.

Завдання 10.3. Отримати список викладачів і дисциплін, які вони викладають, за допомогою зовнішнього повного з'єднання.

Завдання 10.4. Отримати список викладачів і дисциплін, які вони викладають, за допомогою зовнішнього лівого з'єднання.

Завдання 10.5. Отримати список викладачів і дисциплін, які вони викладають, за допомогою зовнішнього правого з'єднання.

Завдання 10.6. Отримати перетин таблиць викладачів і дисциплін.

Завдання 10.7. Зберегти копію таблиці із відомостями про викладачів і дисциплін, які вони викладають.

11. МОВА SQL. ВНЕСЕННЯ ЗМІН ДО БАЗИ ДАНИХ

11.1. Внесення змін до бази даних

SQL є повноцінною мовою, призначеною для роботи з даними і дозволяє не лише вибирати інформацію з бази даних за допомогою запитів на читання, а й змінювати її вміст за допомогою запитів на додавання, видалення і оновлення.

Порівняно з оператором SELECT, який використовується для запитів на читання, SQL-оператори, які змінюють вміст бази даних, є більш простими. Однак при зміні вмісту бази даних до СУБД висуваються деякі додаткові вимоги. При внесенні змін СУБД повинна підтримувати цілісність даних і дозволяти введення в базу даних лише допустимих значень, а також забезпечувати несуперечність бази даних навіть у випадку системної помилки.

Надзвичайно важливою задачею СУБД є забезпечення можливості одночасної зміни бази даних декількома користувачами таким чином, щоб вони не перешкоджали один одному.

11.2. Додавання нових даних. Оператор INSERT

Оператор INSERT додає до таблиці новий рядок або набір з декількох рядків. У секції INTO вказується цільова таблиця, до якої додається новий рядок, і перелік полів стовпців, в які додаються нові значення, а в секції VALUES містяться значення даних для нового рядка. Список стовпців визначає, в які стовпці вносяться відповідні значення.

Синтаксис оператора INSERT у загальному випадку має вигляд

INSERT INTO назва_таблиці (назва_стовпця,…)

VALUES (константа **| NULL**,…**)**, …

Додавання нового рядка. При додаванні до таблиці нового рядка задається перелік назв необхідних стовпців і перелік відповідних значень. Порядок слідування стовпців і порядок слідування значень, що додаються, ідентичний. Для стовпців, назви яких відсутні у списку стовпців оператора INSERT, автоматично присвоюється значення NULL.

Приклад. Додати до таблиці Groups інформацію про нову групу "КС-981".

INSERT INTO Groups(GrpNo, EnterYear, GrpName, GrpHd, GrpCns)

VALUES (6, 1998, 'КС-981', 981621, 3)

Додавання значень NULL. При додаванні до таблиці нового рядка всім стовпцям, імена яких відсутні у списку стовпців оператора INSERT, автоматично присвоюється значення NULL.

Додавання значень всіх стовпців. Для зручності в SQL дозволяється не включати список стовпців в оператор INSERT. Якщо список стовпців відсутній, він генерується автоматично, і в ньому зліва направо перелічуються всі стовпці таблиці. Користуючись цією скороченою формою запису, оператор INSERT з попереднього прикладу можна переписати наступним чином:

```
INSERT INTO Groups
        VALUES (6, 1998, 'КС-981', 981621, 3)
```

Послідовність значень даних повинна точно відповідати порядку стовпців у таблиці. На місці невідомого значення у списку значень необхідно явно вказувати значення NULL.

Багаторядковий оператор INSERT додає до цільової таблиці одразу декілька рядків.

Тут можливі 2 варіанти:

1. У реченні VALUES вказується перелік рядків, які потрібно додати.
2. Рядки, які додаються, є результатом відповіді на запит.

Приклад використання 1-го варіанта багаторядкового оператора INSERT наведений нижче

Приклад. Додати до таблиці Groups інформацію про 3 нові групи.

```
INSERT INTO Groups(GrpNo, EnterYear, GrpName, GrpHd, GrpCns)
        VALUES(6, 1998, 'КС-981', 981621, 3),
              (7, 1998, 'КС-982', 981623, 4)
              (8, 1998, 'СК-981', 981647, 2)
```

У другому варіанті багаторядкового оператора INSERT значення даних для нових рядків явно не задаються. Джерелом нових рядків є запит на читання, що міститься всередині оператора INSERT. Зазвичай такий запит отримує дані з декількох взаємозв'язаних таблиць.

INSERT INTO назва_таблиці (назва_стовпця,…) запит на читання

Приклад. Додати до таблиці LovePhysics інформацію про всіх студентів, які склали іспити з фізики на відмінно.

```
INSERT INTO LovePhysics (GrpNo, EnterYear, GrpName, GrpHd, GrpCns)
        SELECT *
        FROM Students, Marks, Subjects
        WHERE Students.StdNmb = Marks.StdNmb
            AND Marks.SubjNo = Subjects.SubjNo
            AND Subjects.SubjName = 'Фізика'
            AND Marks.Mark = 5
```

Можна взагалі не вказувати стовпці у таблиці, що формується, але в цьому випадку порядок і типи стовпців у обох таблицях повинні співпадати. Зазвичай такі оператори використовуються для створення копії таблиці.

Приклад. Скопіювати вміст таблиці Students до таблиці StudentsCopy.

INSERT INTO StudentsCopy
SELECT * **FROM** Students

11.3. Видалення існуючих даних. Оператор DELETE

Найменшою одиницею інформації, яку можна видалити з реляційної бази даних, є один рядок.

Оператор DELETE видаляє вибрані рядки даних з однієї таблиці.

Синтаксис оператора DELETE має вигляд

DELETE FROM назва_таблиці
[WHERE умова_пошуку**]**

У секції FROM вказується таблиця, що містить рядки, які потрібно видалити. У секції WHERE вказується вимога до рядків, які повинні бути видалені.

Приклад. Видалити з таблиці Students дані про студента з кодом 981623.

DELETE FROM Students
WHERE StdNmb = 981623

Хоча секція WHERE в операторі DELETE є необов'язковою, вона майже завжди присутня. Якщо вона відсутня, то видаляються всі рядки цільової таблиці.

DELETE FROM Students

Іноді рядки необхідно вибрати, опираючись на дані з інших таблиць. Для цього можна використовувати вкладені запити.

11.4. Оновлення існуючих даних. Оператор UPDATE

Оператор UPDATE оновлює значення одного або декількох стовпців у вибраних рядках однієї таблиці.

Синтаксис оператора UPDATE має вигляд

UPDATE назва_таблиці
SET назва_стовпця = вираз,...
[WHERE умова_пошуку**]**

У операторі вказується цільова таблиця, записи якої повинні бути змінені, при цьому користувач повинен мати дозвіл на оновлення таблиці і кожного конкретного стовпця. Секція WHERE відбирає рядки таблиці, які підлягають оновленню. У секції SET вказується, які стовпці повинні бути оновлені, і для них задаються нові значення.

Приклад. Для студента з кодом 981622 змінити прізвище на Петренко:

```
UPDATE Students
SET StdName = 'Петренко'
WHERE StdNmb = 981622
```

Умови пошуку, які можуть бути задані у секції WHERE оператора UPDATE, точно відповідають умовам пошуку, доступним у операторах SELECT і DELETE.

Як і оператор DELETE, оператор UPDATE може одночасно оновлювати декілька рядків, що відповідають умові пошуку.

Приклад. Для студентки з кодом 981622 змінити прізвище на Петренко і замінити стипендію на 1500:

```
UPDATE Students
SET StdName = 'Петренко', StdStip = 1500
WHERE StdNmb = 981622
```

Секція WHERE в операторі UPDATE є необов'язковою. Якщо вона відсутня, то оновлюються всі рядки цільової таблиці.

У операторі UPDATE, так само як і в операторі DELETE, вкладені запити дають змогу відбирати рядки для оновлення, спираючись на інформацію з інших таблиць.

11.5. Контрольні запитання

1. Яка структура оператора INSERT?
2. Як додати новий рядок до таблиці?
3. Як додати до таблиці рядки зі значеннями всіх стовпців?
4. Як додати до таблиці групу рядків?
5. Як додати до таблиці результат запиту до іншої таблиці?
6. Як видалити всі рядки таблиці?
7. Як видалити рядки таблиці за умовою?
8. Яка структура оператора оновлення існуючих даних?
9. Як оновити дані за певною умовою?

11.6. Завдання для самопідготовки

Для бази даних Деканат складіть наступні запити:

Завдання 11.1. До таблиці Students додати дані про нового студента.

Завдання 11.2. До таблиці Students додати дані про 3 нових студентів.

Завдання 11.3. До таблиці Groupe_КС981 додати інформацію про всіх студентів групи КС-981.

Завдання 11.4. З таблиці Students видалти інформацію про студентів групи КС 981.

Завдання 11.5. В таблиці Subject замінти назву дисципліни "Програмування" на назву "Основи програмування".

12. ГЕНЕРАЦІЯ ВЕЛИКОЇ КІЛЬКОСТІ ДАНИХ У ТАБЛИЦЯХ

При виконанні різного роду дослідних робіт з БД, наприклад, оцінка швидкості виконання різних варіантів одного й того ж запиту, попередньо треба мати набір даних відповідного обсягу. Звичайно, іноді для цього можна використати існуючу промислову БД, але зазвичай це не допустимо, тому для подібних задач потрібно створити окрему БД, таблиці якої заповнені тисячами і десятками тисяч випадкових записів.

З цією метою PostgreSQL [5] надає спеціальні інструменти для генерації випадкових чисел, рядків та дат.

12.1. Функції генерації випадкових чисел

У таблиці 12.1 перелічено всі функції генерації випадкових чисел.

Таблиця 12.1. Функції генерації випадкових чисел

Функція	Тип результату	Опис
random()	double precision	випадкове число в діапазоні 0.0 <= x < 1.0
random_normal()	double precision	випадкове значення нормального розподілу із заданими параметрами 0.0 <= x < 1.0
setseed(dp)	void	задає відправну точку для наступних викликів random(), де -1.0 <= dp <= 1.0

Функція random() повертає випадкове число в діапазоні від =>0 до <1. Вона використовує простий лінійний конгруентний алгоритм, працює швидко, але не підходить для криптографічних додатків. Більш безпечна альтернатива є в модулі pgcrypto.

Функція random_normal() повертає випадкове число в діапазоні від =>0 до <1. Вона використовує простий лінійний конгруентний алгоритм, працює швидко, але не підходить для криптографічних додатків. Більш безпечна альтернатива є в модулі pgcrypto. Загальний формат функції має вигляд:

random_normal ([mean **[,** stddev**]])**

За замовчуванням для mean встановлено значення 0.0, а для stddev – 1.0.

Функція setseed(dp) задає початкове значення для наступних викликів функції random(), де dp приймає значення між -1.0 та 1.0 включно.

Якщо скористатися функцією setseed() и викликати її з одним і тим же аргументом, результати наступних викликів random() в поточному сеансі будуть послідовно повторюватися.

12.1.1. Випадкове десяткове число

Випадкове число в діапазоні від =>0 до <1, задається наступним чином:

SELECT random();

random double precision
0.345875730738044

Випадкове число в діапазоні від =>0 до <1000

SELECT random()*1000 або **SELECT** 1000*random()

Випадкове число в діапазоні від =>10 до <25

SELECT random()*(25-10)+10

12.1.2. Випадкове ціле число

Стандартно ціле випадкове число отримують за допомогою функції floor(). Вона повертає найближче ціле число, яке менше або рівне заданому. Тип значення, що повертається – double precision, тому іноді його треба перетворювати до іншого типу, наприклад, до int.

Синтаксис функції floor():

floor (number or decimal**)**

Приклади.

Згенерувати випадкове ціле число в діапазоні від =>1 до <=1000:

SELECT floor(random()*1000+1)

Згенерувати випадкове ціле число в діапазоні від =>1 до <=1000 з перетворенням до типу int:

SELECT floor(random()*1000+1)::int

Згенерувати випадкове ціле число в діапазоні від =>10 до <=25:

SELECT floor(random()*(25-10+1))+10

Згенерувати випадкове ціле число в діапазоні від =>10 до <=25 із перетворенням до типу int:

SELECT (floor(random()*(25-10+1)+10)::int

Примітка. Ціле випадкове число можна отримати і з використанням функції trunc(). Вона повертає число, усічене до визначеної кількості десяткових

знаків. Якщо кількість десяткових знаків не задано, то повертається ціле значення числа. Тип значення, що повертається – double precision, тому іноді його треба перетворювати до іншого типу, наприклад, до int.

Синтаксис функції trunc():

trunc(number, **[**decimal_places**])**

Приклади.

Згенерувати випадкове ціле число в діапазоні від =>1 до <=100:

SELECT trunc(random()*1000+1)

Згенерувати випадкове ціле число в діапазоні від =>1 до <=100 з перетворенням до типу int:

SELECT trunc(random()*1000+1)::int

Згенерувати випадкове ціле число в діапазоні від =>10 до <=25:

SELECT trunc(random()*(25-10+1)+10)

Згенерувати випадкове ціле число в діапазоні від =>10 до <=25 з перетворенням до типу int:

SELECT trunc(random()*(25-10+1)+10)::int

12.1.3. Випадкове значення дати, часу, дати_часу

Випадкове значення дати, часу, а також мітки часу (date/time/timestamp) між двома позначками дата/час/мітка_часу можна отримати за допомогою запитів, як у прикладах:

```
SELECT date '2020-01-10' +
        (random() * (date '2020-01-20' - date '2020-01-10'))::int;
SELECT time '10:00:00' +
        random() * (time '20:00:00' - time '10:00:00');
SELECT timestamp '2020-01-10 20:00:00' +
        random() * (timestamp '2020-01-20 20:00:00' -
                timestamp '2020-01-10 10:00:00')
```

12.2. Функції, які повертають рядки значень

У цьому розділі описуються додаткові опції, які можуть повертати не одне значення, а множину рядків. Найчастіше з їх числа використовуються функції, які генерують ряди значень, які перелічені у таблиці 12.2 та таблиці 12.3. Інші, більш спеціалізовані функції множин описано у відповідних розділах документації.

Таблиця 12.2. Функції, що генерують ряди значень

Функція	Тип аргументу	Тип результату	Опис
generate_series(start, stop)	int, bigint, numeric	setof int, setof bigint, setof numeric (визначається типом аргументу)	Видає ряд цілих чисел від *start* до *stop* з кроком *1*
generate_series(start, stop, step)	int, bigint, numeric	setof int, setof bigint, setof numeric (визначається типом аргументу)	Видає ряд цілих чисел від *start* до *stop* с заданим кроком *step*
generate_series(start, stop, step interval)	timestamp, timestamp with time zone	setof timestamp або setof timestamp with time zone (визначається типом аргументу)	Видає ряд цілих чисел від *start* до *stop* с заданим кроком *step*

Якщо заданий крок step позитивний, а start виявляється більшим, ніж stop, тоді ці функції повертають 0 рядків. Той же результат отримаємо, якщо step менше 0, а start менше stop, або якщо будь-якої аргумент дорівнює NULL. Якщо ж step буде дорівнює 0, буде згенеровано помилку.

Таблиця 12.3. Функції, що генерують індекси масивів

Функція	Тип результату	Опис
generate_subscripts(*array anyarray*, *dim int*)	setof int	Видає ряд значень для використання в якості індексу даного масиву.
generate_subscripts(array anyarray, dim int, reverse boolean)	setof int	Видає ряд значень для використання в якості індексу даного масиву. Якщо параметр reverse дорівнює true, значення видаються від більшого до меншого.

12.3. Генерація серії значень

12.3.1. Генерація серії чисел

Генерація серії чисел в заданому діапазоні реалізується виразом

SELECT * FROM generate_series(2, 4)

Результат наведено у таблиці 12.4.

Таблиця 12.4. Результат генерації серії чисел в заданому діапазоні

generate_series integer
2
3
4

Генерація серії чисел в заданому діапазоні з заданим кроком реалізується виразом

SELECT * **FROM** generate_series(5, 1, -2)

Результат генерації наведено у таблиці 12.5.

Таблиця 12.5. Результат генерації серії чисел в заданому діапазоні з заданим кроком

generate_series integer
5
3
1

12.3.2. Генерація серії дат

Генерація серії дат в заданому діапазоні з заданим кроком в 1 день. Тип даних – date. Вкладений запит повинен мати псевдонім.

SELECT gs::date **FROM** generate_series('2018-01-01', '2018-01-04',
interval '1 day') **AS** gs

Результат генерації серії дат наведено у таблиці 12.6.

Таблиця 12.6. Результат генерації серії дат

gs date
2018-01-01
2018-01-02
2018-01-03
2018-01-04

12.3.3. Генерація серії часу

Генерація серії часу в заданому діапазоні часу з заданим кроком 1 хвилина. Тип даних результату – time. Зауважимо, діапазон часу має задаватися повністю 'дата_час, дата_час'. Результатом буде час. Вкладений запит повинен мати псевдонім.

SELECT gs::time
FROM generate_series('2020-01-01 00:00:00', '2020-01-01 23:59:59',
interval '1 minute') **AS** gs

Результат генерації серії часу наведено у таблиці 12.7.

Таблиця 12.7. Результат генерації серії часу

gs time
00:00:00
00:01:00
00:02:00
00:03:00
…

12.3.4. Генерація серії дат і часу

Генерація серії дат і часу в заданому діапазоні з заданим кроком 1 хвилина. Тип даних timestamp (з урахуванням часового поясу). Вкладений запит повинен мати псевдонім.

```
SELECT *
    FROM generate_series('2022-01-01 00:00:00',  '2022-08-31 23:59:59',
                         interval '1 minute') AS gs
```

Результат генерації серії дат і часу наведено у таблиці 12.8.

Таблиця 12.8. Результат генерації серії дат і часу

gs timestamp without time zone
2022-01-01 00:00:00+02
2022-01-01 00:01:00+02
2022-01-01 00:02:00+02
2022-01-01 00:03:00+02
…

Порівняти попередній запит можна з наступним запитом, де вказано тип даних, що повертаються, як timestamp (без урахування часового поясу). Вкладений запит повинен мати псевдонім.

```
SELECT gs::timestamp
    FROM generate_series('2022-01-01 00:00:00',  '2020-01-05 00:05:00',
                         interval '1 minute') AS gs
```

Результат виконання цього запиту наведено у таблиці 12.9.

Таблиця 12.9. Результат генерації серії дат і часу з типом даних timestamp

gs timestamp without time zone
2022-01-01 00:00:00
2022-01-01 00:01:00
2022-01-01 00:02:00
2022-01-01 00:03:00
…

12.3.5. Генерація серії псевдовипадкових чисел

Генерація серії зі 100 псевдовипадкових цілих чисел реалізується виразом:

```
SELECT floor(random()*1000+1)::int
    FROM generate_series(1, 100)
```

Результат виконання цього запиту наведено у таблиці 12.10.

Таблиця 12.10. Результат генерації серії зі 100 псевдовипадкових цілих чисел

floor integer
360
790
51
…

або виразом:

```
SELECT trunc(random()*1000+1)::int
    FROM generate_series(1, 100)
```

Генерація серії псевдовипадкових цілих чисел в діапазоні 1-100 з кроком 5 реалізується виразом

```
SELECT floor(random()*1000+1)::int
    FROM generate_series(1, 100, 5)
```

або

```
SELECT trunc(random()*1000+1)::int
    FROM generate_series(1, 100, 5)
```

12.3.6. Генерація псевдовипадкових рядків

Генерація 5-ти псевдовипадкових рядків з 2-х символів реалізується виразом

```
SELECT chr(floor(65 + random()*25)::int) ||
            chr(floor(65 + random()*25)::int) ||
            chr(floor(65 + random()*25)::int)
    FROM generate_series(1, 5)
```

Результат виконання цього запиту наведено у таблиці 12.11.

Таблиця 12.11. Результат генерації 5-ти псевдовипадкових рядків з 2-х символів:

column text
TSD
NXE
GUN
HEI
KHR

12.4. Функції, що генерують індекси масивів

Функція generate_subscripts дозволяє спростити отримання всього набору індексів для зазначеної розмірності заданого масиву. Вона видає 0 рядків, якщо в масиві немає зазначеної розмірності або сам масив дорівнює NULL (хоча для елементів, рівних NULL, індекси будуть видані, як і для будь-яких інших).

Приклад, коли масив включає 3-ри елементи:

SELECT generate_subscripts(**ARRAY**['foo', 'bar', 'baz'], 1)

Результат виконання цього запиту наведено у таблиці 12.12.

Таблиця 12.12. Результат запиту з використанням функції generate_subscripts

generate_subscripts integer
1
2
3

12.5. Криптографічна хеш-функція MD5

Функція PostgreSQL MD5 – це криптографічна хеш-функція, яка використовується для перетворення заданого рядка в текстовий, що містить 32 символи. Вона застосовується в критично важливих програмах, де безпека даних є головною метою.

Функція MD5 зокрема використовується для текстового шістнадцяткового представлення контрольної суми 128 біт.

Приклад.

SELECT MD5('PostgreSQL MD5');

Результат запиту наведено у таблиці 12.13.

Таблиця 12.13. Результат запиту з використанням функції PostgreSQL MD5

Md5 text
f78fdb18bf39b23d42313edfaf7e0a44

12.6. Конструктор таблиці константних значень

Загалом вираз FROM вказує вхідні набори рядків для виразу SELECT та способи об'єднання (JOIN) декількох наборів рядків у набір рядків виразу SELECT. Окрім цього вираз FROM надає можливість вибору таблиці константних значень, створеної виразом VALUES конструктора табличних значень.

12.7. Створення та заповнення таблиці випадковими значеннями

Розглянемо декілька прикладів створення та заповнення таблиці випадковими значеннями.

Приклад. Створити таблицю з 2-х стовпчиків id та str, перший з яких є індексом, а другий вміщує випадкові рядки значень.

DROP TABLE IF EXISTS test_indx;

CREATE TABLE test_index (id integer, str text COLLATE "C")

Вкладений запит повинен мати псевдонім:

INSERT INTO test_indx (id, txt)

SELECT t_id, random()::text **FROM** generate_series(1, 100000) **AS** t_id

ORDER BY random();

SELECT * **FROM** test_indx

Поле id таблиці заповнюється серією, а поле test_txt вираховується випадково для кожного рядка таблиці. Результат має вигляд як у таблиці 12.14.

Таблиця 12.14. Результуюча таблиця з 2-х стовпчиків id та str

id integer	str character varying
66	0.76469952147454
91	0.196612752042711
92	0.371335481759161
78	0.00303955422714353
…	…

Приклад. Створити таблицю з 3-х стовпчиків id, str та date, перший з яких є індексом, другий вміщує випадкові рядки значень, а третій – випадкові значення дати/часу.

DROP TABLE IF EXISTS test_indx;

CREATE TABLE test_indx (id integer, str text **COLLATE** "C",
date timestamp);

```
INSERT INTO test_indx (id, txt, date)
SELECT trunc(random()*1000)::int,
        chr(trunc(65 + random()*25)::int) ||
        chr(floor(65 + random()*25)::int) ||
        chr(floor(65 + random()*25)::int),
       timestamp '2022-12-01 20:00:00' + random() * (timestamp '2022-12-31
       20:00:00' - timestamp '2022-12-01 10:00:00')
FROM generate_series(1, 500000);
SELECT * FROM test_indx
```

Результат має вигляд як у таблиці 12.15.

Таблиця 12.15. Результуюча таблиця з 3-х стовпчиків id, str та date

id integer	str text	date timestamp without time zone
253	ERW	"2022-12-23 06:07:52.048851"
861	BTI	"2022-12-11 20:52:06.444033"
127	QAX	"2022-12-29 20:24:39.796731"
957	GVQ	"2022-12-02 11:57:14.001222"

Приклад. Створити таблицю з 2-х стовпчиків id та str, де ключове поле id заповнюється системою, а поле str заповнюється різними підрядками заданого рядка. Кількість рядків таблиці визначається серією.

```
DROP TABLE IF EXISTS test_indx;
CREATE TABLE test_indx(id bigserial PRIMARY KEY, str varchar(255))
```

Для заповнення поля test_text підрядками заданого рядка використовується конструкція SQL, яка надає можливість вибору рядка або таблиці константних значень, створених виразом VALUES конструктора табличних значень.

```
INSERT INTO test_index(test_txt)
SELECT substr(characters,
        (random() * length(characters) + 1)::integer,  10)
FROM (VALUES (
'qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASD¬FGHJKLZXCVBNM'))
 AS  symbols(characters), generate_series(1, 1000000)  AS  q;
SELECT * FROM test_indx
```

Результат має вигляд як у таблиці 12.16.

Таблиця 12.16. Результуюча таблиця з 2-х стовпчиків id та str з різним заповненням

id bigint	str character varying
1	"xcvbnmQWER"
2	"cvbnmQWERT"
3	"KLZXCVBNM"
4	"nmQWERTYUI"
5	"dfghjklzxc"

Приклад. Створити таблицю з 2-х стовпчиків str_character та str_indexed, де обидва поля заповнюється з допомогою криптографічної хеш-функції MD5. Кількість рядків таблиці визначається серією.

DROP TABLE IF EXISTS test_indx;

CREATE TABLE test_indx (str varchar, str_indexed varchar);

Звернути увагу на необхідність аліасів 2-х заповнюваних полів symbols(characters) та q.

Кожен вкладений запит повинен мати аліас.

INSERT INTO test_indx **SELECT** md5(random()::text),
md5(random()::text)
FROM (**SELECT** * **FROM** generate_series(1, 500000) **AS** id) **AS** ser;

EXPLAIN ANALYZE SELECT * **FROM** test_index;

Результат має вигляд як у таблиці 12.17.

Таблиця 12.17. Результуюча таблиця запиту з хеш-функцією MD5

str_character varying	str_indexed character varying
c40e103edee5d00630cf350f7812d21d	f3e48b74eb06aa4ede09629178f0f168
3554bd9d7800b05be32876ced728b271	613233642813020ed75f7a698122a67c
8ff474dbf805ee23d78ecbc3b55fe72d	31af6c112d63f7e3208e14b1e7369bed
4edf2483adc2d281a215876724216f63	5049f9d832f4a6fe6bacef667d208242

12.8. Контрольні запитання

1. Перелічіть основні функції генерації випадкових чисел та їх призначення.
2. Як отримати ціле випадкове число?
3. Як отримати випадкове значення дати, часу, дати/часу?
4. Перелічіть основні функції, які повертають ряди значень та їх призначення.
5. Як згенерувати серію чисел в заданому діапазоні?
6. Як згенерувати серію дат?
7. Як згенерувати серію часу дат і часу?

8. Як згенерувати серію псевдовипадкових рядків?
9. Перелічіть основні функції, що генерують індекси масивів.
10. Для чого призначено криптографічну хеш-функцію MD5?
11. Яка структура та як функціонує конструктор таблиці константних значень?

12.9. Завдання для самопідготовки

Завдання 12.1. Сформуйте завдання на генерацію серії чисел.

Завдання 12.2. Сформуйте завдання на генерацію серії дат.

Завдання 12.3. Сформуйте завдання на генерацію серії часу.

Завдання 12.4. Сформуйте завдання на генерацію серії дата+час.

Завдання 12.5. Сформуйте завдання на генерацію серії псевдовипадкових чисел.

Завдання 12.6. Сформуйте завдання на генерацію серії псевдовипадкових рядків.

Завдання 12.7. Сформуйте завдання на створення та заповнення таблиці випадковими значеннями.

13. ПОВНОТЕКСТОВИЙ ПОШУК

Повнотекстовий пошук або пошук тексту – це можливість знаходити документи природною мовою, відповідні до запиту з можливістю додатково сортувати їх за релевантністю.

Документ – це одиниця обробки в системі повнотекстового пошуку, наприклад, технічна стаття або звіт. Система пошуку тексту повинна вміти розбирати документи і зберігати зв'язки лексем (ключових слів) з документом, який їх містить. Згодом ці зв'язки можуть використовуватися для пошуку документів за заданими ключовими словами.

Під релевантністю (Relevance) розуміють ступінь відповідності знайденого документа або набору документів інформаційним потребам користувача.

Найбільш поширене завдання повнотекстового пошуку – знайти всі документи, що містять задані слова запиту, і видати ці документи, відсортовані за ступенем відповідності запиту.

13.1. Засоби повнотекстового пошуку PostgreSQL

Історично реляційні БД використовувалися для представлення економічних, технічних і подібних до них даних, тобто даних, які мають переважно табличний вигляд. Однак, в подальшому, в зв'язку з їх широким розповсюдженням, реляційні БД почали використовувати і для представлення текстових документів. І хоча на сьогодні з'явилися документоорієнтовані БД, наприклад MongoDB, реляційні БД розвинули засоби, які в багатьох випадках дозволяють обійтися без залучення подібних спеціалізованих систем. Додаткові можливості для цього виникли в зв'язку з появою текстового формату обміну даними JSON (JavaScript Object Notation).

PostgreSQL (pgSQL) має розвинені засоби повнотекстового пошуку.

У концепції pgSQL документ – це вміст текстового поля рядка таблиці або поєднання (об'єднання) значень полів, які можуть зберігатися в одній або різних таблицях або формуватися динамічно. Наприклад, таке поєднання можна отримати, склеївши поля StName || TchName || SbjName 3-х відповідних таблиць. Таким чином, документ для індексації може створюватися з декількох частин і не зберігатися як єдине ціле.

Слова природної мови крім основоположного кореня містять префікси, суфікси, закінчення тощо. Функція to_tsvector вирізає їх і перетворює текст в набір токенів (token), за якими виконується пошук. У результаті кожен документ зводиться до його спеціального формату tsvector, який містить компактне

представлення всього документу. Пошук і ранжування виконується виключно з цим представленням документу – вхідний текст потрібно буде витягти, тільки коли документ буде відібраний для виведення користувачеві.

Функція to_tsvector має вигляд:

to_tsvector('мова', 'початковий текст'**)**

Вона нормалізує та перетворює заданий текст у значення tsvector, наприклад:

SELECT to_tsvector('ukrain', 'Здав екзамени з програмування на відмінно')

Отримаємо:

"'відмінн':6 'програмуванн':4 'здав':1 'екзам':2'"

Приведений до спеціального формату запит на повнотекстовий пошук зберігається в структурі tsquery. Значення типу tsquery містить токени пошуку, можливо об'єднані в вирази операторами AND, OR, NOT, <-> тощо.

Функції to_tsquery() та plainto_tsquery() мають вигляд:

to_tsquery('мова', 'фрагмент **[&** фрагмент2 …**]')**

Вони працюють аналогічно функції to_tsvector() і нормалізують та перетворюють заданий користувачем текст запиту в значення tsquery, наприклад:

SELECT to_tsquery ('ukrain', 'екзамен & з & програмування')

Отримаємо:

"'екзам' & 'програмуванн'"

13.2. Пошук простої відповідності

Повнотекстовий пошук в pgSQL реалізується на базі оператора відповідності @@, який повертає true, якщо документ tsvector відповідає запиту tsquery. Для цього оператора не важливо, який тип tsvector або tsquery записаний першим або другим.

Таким чином, запит з пошуком відповідності виглядає так:

SELECT to_tsvector('мова', 'початковий текст' **) @@**
to_tsquery('мова', 'фрагмент **[&** фрагмент2 …**]')**

Наприклад:

SELECT to_tsvector('ukrain', 'Здав екзамени зо програмування на відмінно, вивчав англійську мову …'**) @@**
to_tsquery('ukrain', 'екзамен & відмінно')

Отримаємо: true

13.3. Конфігурування пошуку відповідності

Вище розглянуто простий приклад повнотекстового пошуку. Функціональність повнотекстового пошуку pgSQL дозволяє робити багато іншого:

1. Пропускати певні слова (стоп-слова).
2. Обробляти синоніми.
3. Виконувати складний аналіз слів тощо.

Ці функції налаштовуються конфігураціями текстового пошуку. Зокрема, в попередніх прикладах використовувалась конфігурація мови пошуку – 'ukrain'. У разі англійської мови конфігурацію можна не ставити, вона йде за замовчуванням.

У pgSQL є набір зумовлених конфігурацій для багатьох природніх мов, але можна створювати також і власні конфігурації. (Всі доступні конфігурації можна переглянути за допомогою команди \dF).

Відповідна конфігурація для даного середовища вибирається під час установки і записується в параметрі default_text_search_config в postgresql.conf.

Більш детально питання конфігурації розглянуті в документації.

13.4. Пошук у таблиці

Повнотекстовий пошук можна виконати безпосередньо в полі таблиці. Наступний запит виводить імена студентів, в текстовому полі резюме StResume яких є слова "фізика" і "відмінно":

```
SELECT StName FROM Students
WHERE to_tsvector('ukrain', StResume) @@
                to_tsquery('ukrain', 'фізика & відмінно')
```

Отримаємо:

перелік студентів

Для прискорення текстового пошуку створюються індекси GIN або GiST за відповідним полем (див. ниче).

13.5. Ранжування результатів пошуку

Ранжування документів можна розглядати як спосіб оцінки, тобто наскільки вони релевантні заданому запиту, і впорядкувати їх так, щоб найбільш релевантні виводилися першими. У pgSQL вбудовано 2-ві функції ранжування ts_rank() і ts_rank_cd(), які враховують лексичну, позиційну і структурну інформацію; тобто, вони враховують, наскільки часто і наскільки близько

зустрічаються в документі ключові слова і яка вага частини документа, що їх містить. Однак саме поняття релевантності вельми нечітке і багато в чому визначається додатком. Вбудовані функції ранжування можна розглядати лише як приклади реалізації. Для своїх конкретних завдань користувач може розробити власні функції ранжування і/або врахувати при обробці їх результатів додаткові чинники.

13.6. Виділення результатів пошуку

Важливою вимогою до результату пошуку є виділення тієї частини або частин документа, які відповідають запиту. Зазвичай пошукові системи показують фрагменти документу із зазначеними слів, що необхідно знайти. У pgSQL для реалізації цієї можливості використовується функція ts_headline():

ts_headline([конфігурація regconfig**,]** документ text, запит tsquery
[, параметри text**])**

Функція ts_headline() приймає документ разом із запитом і повертає витяг з документа, в якій виділяються слова з запиту. Конфігурацію, що застосовується для розбору документа, можна вказати в параметрі config; якщо цей параметр відсутній, застосовується конфігурація за замовчуванням:

default_text_search_config.

Якщо в параметрах передається рядок options, він повинен складатися зі списку розділених комами пар параметр = значення. Параметри можуть бути наступними:

1. StartSel, StopSel – рядки, які будуть розмежовувати слова запиту в документі, виділяючи їх серед інших. Якщо ці рядки містять пробіли чи коми, їх потрібно взяти в лапки.
2. MaxWords, MinWords – числа, які визначають верхню і нижню межі розміру вибірки.
3. ShortWord – слова зазначеної довжини (або коротше) на початку і в кінці витягу будуть відкидатися. Значення за замовчуванням, що дорівнює трьом, виключає поширені англійські артиклі.
4. HighlightAll – логічний прапорець; якщо він дорівнює true, витягом буде весь документ, а три попередні параметра ігноруються.
5. MaxFragments – максимальне число виведених текстових витягів або фрагментів. Значення за замовчуванням, рівне 0, вибирає метод створення результату без фрагментів. При значенні більшому 0

вибирається метод з фрагментами, коли знаходяться всі фрагменти, що містять якомога більше слів запиту, а потім вони стискаються до слів запиту. Такі фрагменти можуть містити деякі ключові слова в середині і обмежуються двома словами, що необхідно знайти. При цьому фрагменти можуть містити не більше MaxWords слів, а на початку і в кінці вони будуть очищені від слів довжини ShortWord і менше. Якщо в документі знайдено не всі слова запиту, виводиться один фрагмент, що включає перші MinWords слів в документі.

6. FragmentDelimiter – якщо виводяться декілька фрагментів, вони будуть розділятися цим рядком.

Всі неявні параметри отримують такі значення за замовчуванням:

StartSel=, StopSel=,

MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,

MaxFragments=0, FragmentDelimiter=" ... "

Приклад використання:

SELECT ts_headline ('ukrain', 'Здав екзамени з програмування на відмінно, а зі схемотехніки на добре', **to_tsquery(**'ukrain', 'екзамен & відмін-но**))**

Отримаємо слова, які необхідно знайти, виділені парами тегів и :

"Здав екзамени з програмування на відмінно, а зі схемотехніки на добре"

Увага! Функція ts_headline() працює з оригінальним документом, а не з його стисненим представлення tsvector, так що вона може бути повільною і використовувати її слід обачно. Більш детально це питання розглянуто у документації.

13.7. Індекси і прискорення повнотекстового пошуку

Якщо пошук за деякою колонкою виконується регулярно, то зазвичай її індексують.

Для прискорення повнотекстового пошуку можна використовувати індекси 2-х видів:

1. Узагальнене дерево пошуку GiST (Generalized Search Tree).
2. Узагальнений інвертований індекс GIN (Generalized Inverted Index).

13.7.1. Створення та використання індексів текстових полів

Наступна команда створює індекс на базі GiST. Тут колонка може мати тип tsvector або tsquery:

CREATE INDEX назва_індексу **ON** назва_таблиці **USING GIST**(колонка)

Приклад створення індексу GiST:

CREATE INDEX StResumIdx **ON** Students
USING GIST(to_tsvector('ukrain', StResume))

Наступна команда створює індекс на базі GIN. Тут колонка повинна мати тип tsvector:

CREATE INDEX назва_індексу **ON** назва_таблиці **USING GIN**(колонка)

Приклад створення індексу GIN:

CREATE INDEX StResumIdx_1 **ON** Students
USING GIN(to_tsvector('ukrain', StResume))

Увага! Оскільки при створенні індексу використовувалась версія to_tsvector() з чотирма аргументами, цей індекс буде використовуватися тільки в запитах, де to_tsvector() викликається також з двома аргументами і в першому передається назва тієї ж конфігурації 'ukrain'.

SELECT StName
FROM Students
WHERE to_tsvector('ukrain', StResume) @@
to_tsquery('ukrain', 'фізика & відмінно')

У розглянутому вище прикладі конфігурація ukrain відноситься до значень всіх полів стовпця StResume. Цікавим є питання, а що якщо вміст окремих полів стовпця StResume представлено різними мовами, тобто має різну конфігурацію (ukrain, english тощо). В цьому випадку в таблиці створюється додатковий стовпець текстового типу, наприклад StConfigTSvect, в якому розміщується конфігурація для кожного поля стовпця StResume, а індекс можна створити наступним чином:

CREATE INDEX StResumIdx_2 **ON** Students
USING GIN(to_tsvector(StConfigTSvect, StResume))

В результаті в одному індексі будуть елементи з різними конфігураціями. Запит в цьому випадку повинен використовувати той же індекс, тобто індекс з таким же типом конфігурації:

SELECT StName
FROM Students
WHERE to_tsvector(StConfigTSvect, StResume) @@
to_tsquery(StConfigTSvect, 'фізика & відмінно')

Інший варіант створення індексів GiST або GIN полягає в наступному. У таблиці Students створюється стовпець типу tsvector, наприклад, StResumTSvect,

в якому розміщується tsvector для кожного поля індексного стовпчика StResume. Потім слід сформувати значення поля StResumTSvect і створити необхідний GiST-індекс:

```
UPDATE Students SET StResumTSvect = to_tsvector('ukrain', StResume);
CREATE INDEX StResumIdx_3 ON Students
USING GIST(StResumTSvect);
```

Тепер можна виконати повнотекстовий пошук:

```
SELECT StName
FROM Students
WHERE StResumTSvect @@ to_tsquery('ukrain', 'фізика & відмінно')
```

Коли представлення tsvector зберігається в окремому стовпчику, необхідно створити тригер, який буде підтримувати стовпчик з tsvector в актуальному стані за будь-яких змін у стовпчиках StName або StResume.

13.7.2. Порівняння способів індексації GiST та GIN

Індекси GiST і GIN значно розрізняються за швидкодією.

Індекс GiST допускає помилкові потрапляння і тому їх потрібно виключати додатково, звіряючи результат з фактичними даними таблиці. Імовірність помилкових влучень залежить від ряду факторів, наприклад, від кількості унікальних слів.

Неточність індексу призводить до зниження продуктивності через додаткові звернення до записів таблиці, для яких припущення про збіг виявляється хибним. Оскільки довільний доступ до таблиці зазвичай не буває швидким, це обмежує сферу застосувань індексів GiST.

Індекси GIN не є точними для стандартних запитів, але їх продуктивність логарифмічно залежить від числа унікальних слів. Слід зазначити, що індекси GIN зберігають тільки слова (лексеми) значень tsvector, але втрачають інформацію про їх ваги. Таким чином, для виконання запиту з вагами буде потрібно перевірити ще раз дані в таблиці.

Вибираючи між індексами GiST і GIN, необхідно врахувати такі їх відмінності з точки зору продуктивності:

1. Пошук за індексом GIN приблизно втричі швидше, ніж GiST.
2. Індекси GIN будуються приблизно втричі довше, ніж GiST.
3. Індекси GIN оновлюються трохи повільніше, ніж GiST, але якщо відключено швидке оновлення, різниця може досягати 10 разів.
4. Індекси GIN зазвичай в два-три рази більше індексів GiST.

Як правило, індекси GIN краще підходять для статичних даних, оскільки пошук з ними виконується швидше.

Для динамічних даних краще використовувати індекси GiST, оскільки вони швидше оновлюються. Точніше, індекси GiST ефективні для динамічних даних і працюють швидко, якщо число унікальних слів (лексем) не перевищує 100 000, а індекси GIN краще справляються з великою кількістю лексем, але оновлюватися будуть повільніше.

Є два напрямки збільшення швидкості пошуку з можливістю поновлення "на льоту":

1. секціонувати великі колекції документів;
2. ефективно застосовувати індекси GiST і GIN.

Секціонувати дані можна як на рівні БД, з використанням наслідування таблиць, так і розподіливши документи по різних серверах і потім збирати результати. Останній варіант можливий завдяки тому, що функції ранжування використовують тільки локальну інформацію.

13.8. Контрольні запитання

1. Опишіть призначення та можливості функції to_tsvector.
2. Опишіть призначення та можливості функції to_tsquery.
3. Як виконується пошук простої відповідності?
4. Як виконується повнотекстовий пошук безпосередньо в полі таблиці?
5. Яким чином виконується ранжування результатів пошуку?
6. Як виконується виділення результатів пошуку?
7. Якими засобами можна прискорити повнотекстовий пошук?
8. Якими типи індексів використовуються для прискорення повнотекстового пошуку?

13.9. Завдання для самопідготовки

Завдання 13.1. Сформуйте завдання на пошук простої відповідності на наявність 2-х слів у заданому рядку.

Завдання 13.2. Сформуйте завдання на повнотекстовий пошук двох слів, безпосередньо в полі таблиці.

Завдання 13.3. Сформуйте завдання на виділення результатів пошуку двох слів у заданому рядку.

Завдання 13.4. Сформуйте завдання на повнотекстовий пошук двох слів, безпосередньо в полі колонки таблиці, яке проіндексована GiST.

14. ВИЗНАЧЕННЯ БАЗИ ДАНИХ

14.1. Визначення структури даних в SQL

Визначення структури бази даних включає низку дій щодо створення, зміни та видалення основних структур бази даних – самої бази даних, таблиць, індексів, представлень та курсорів.

14.2. Створення бази даних. Оператор CREATE DATABASE

Багатокористувацьку базу даних зазвичай створює адміністратор, а користувачі безпосередньо звертаються до неї під час роботи. Але конкретний користувач мое на власному комп'ютері створити власну базу даних, наприклад, для лагоження створюваних програмних додатків.

Для визначення бази даних у SQL виділяється підмножина, яка називається *мова визначення даних (Data Definition Language – DDL).*

Оператор CREATE DATABASE дозволяє створити нову базу даних. Різні секції оператора встановлюють елементи визначень бази даних. Слід пам'ятати, що не всі параметри стандарту SQL2 присутні в усіх СУБД.

Зазначимо, що в різних СУБД директиви на створення БД можуть відрізнятися. Розглянемо їх на прикладі СУБД PostgreSQL.

CREATE DATABASE назва
[[WITH] [OWNER [=] ім'я_користувача **]**
[TEMPLATE [=] шаблон **]**
[ENCODING [=] кодування **]**
[LC_COLLATE [=] категорія_сортування **]**
[LC_CTYPE [=] категорія_типів_символів **]**
[TABLESPACE [=] табл_простір **]**
[CONNECTION LIMIT [=] обмеження_підключень **]]**

Перелік параметрів оператора CREATE DATABASE наведено у таблиці 14.1.

Таблиця 14.1. Перелік параметрів оператора CREATE DATABASE

Аргумент	Опис
DATABASE	Назва нової бази даних
OWNER	Ім'я користувача, яке буде перевірено на відповідність допустимим комбінаціям імені користувача та пароля в базі даних на сервері, де буде розміщено базу даних

Аргумент	Опис
PASSWORD	Пароль, який буде перевірено на відповідність допустимим комбінаціям імені користувача та пароля в базі даних на сервері, де буде розміщено базу даних
TEMPLATE	Назва шаблону, з якого створюється база даних. Якщо цей параметр відсутній, то використовується назва за замовчуванням DEFAULT
ENCODING	Встановлює кодування символів для бази даних. Якщо цей параметр відсутній, кодування за замовчуванням встановлюється в NONE
TABLESPACE	Назва табличного простору, який буде пов'язано з новою базою даних. Табличні простори дозволяють адміністраторам баз даних визначати місця у файловій системі, де можуть зберігатися файли, що представляють об'єкти бази даних. Після створення табличного простору до нього можна звертатися за назвою при створенні об'єктів БД. За замовчуванням pg_default
LC_COLLATE	Визначає порядок сортування рядків у запитах з ORDER BY, а також порядок індексів за текстовими колонками. За замовчуванням використовується порядок сортування, встановлений у шаблоні
LC_CTYPE	Встановлює класифікацію символів, яка буде застосовуватися в новій базі даних. Цей параметр визначає належність символів категоріям, наприклад: малий регістр, верхній регістр, цифри тощо. За замовчуванням використовується класифікація, встановлена у шаблоні
CONNECTION LIMIT	Максимальна кількість одночасних підключень до цієї бази даних. Значення "-1" (за замовчуванням) знімає обмеження.

Слід зазначити, що в різних СУБД директиви для створення бази даних можуть відрізнятися. Наведемо приклад створення бази даних у СУБД PostgreSQL

Приклад. Створити у СУБД PostgreSQL базу даних deanery:

```
CREATE DATABASE deanery
  WITH OWNER = postgres
     PASSWORD = 'qwerty'
      ENCODING = 'UTF8'
      TABLESPACE = pg_default
      LC_COLLATE = 'Ukrainian_Ukraine.1251'
      LC_CTYPE = 'Ukrainian_Ukraine.1251'
      CONNECTION LIMIT = -1;
```

Строгий синтаксис визначення бази даних і ряд прикладів можна знайти за посиланням [6].

Примітка. У конкретних СУБД визначення бази даних може відрізнятися від стандарту SQL.

14.3. Видалення БД. Оператор DROP DATABASE

Оператор DROP DATABASE видаляє поточну базу даних, включно з усіма асоційованими вторинними, фоновими та журнальними файлами. Видалення бази даних вилучає всі дані, які містяться в ній.

База даних може бути видалена її власником.

Синтаксис:

DROP DATABASE

Приклад: Видалити базу даних deanery.

DROP DATABASE deanery

14.4. Зміна опису бази даних. Оператор ALTER DATABASE

Базу даних може змінити її власник користувачем за допомогою оператора ALERT DATABASE.

Синтаксис оператора:

ALTER DATABASE назва **[[WITH]** параметр **[...]]**

Оператор ALERT DATABASE вимагає ексклюзивного доступу до бази даних.

Перелік параметрів оператора ALERT DATABASE наведено у таблиці 14.2.

Таблиця 14.2. Перелік параметрів оператора ALERT DATABASE

Параметр	Опис
CONNECTION LIMIT	Кількість одночасно дозволених підключень до цієї бази даних. Значення "-1" знімає обмеження
RENAME TO нова_назва	Нова назва бази даних
OWNER TO новий_власник	Новий власник бази даних
SET TABLESPACE новий_табл_простір	Новий основний табличний простір бази даних

Приклад. Змінити назву та власника існуючої бази даних deanery відповідно на department та sp_scs

ALTER DATABASE deanery

RENAME TO department

OWNER TO sp_scs

14.5. Контрольні запитання

1. Яка структура команди створення баз даних?
2. Як визначається користувач бази даних?
3. Навіщо і як встановлюється кодування символів для бази даних?
4. Для чого використовується директива LC_COLLATE?
5. Яка структура команди видалення бази даних?
6. Яка структура команди оновлення бази даних?
7. Як змінити назву бази даних?

14.6. Завдання для самопідготовки

Завдання 14.1. Надайте текст директиви створення власної бази даних.

Завдання 14.2. Надайте текст директиви видалення власної бази даних.

Завдання 14.3. Надайте текст директиви перейменування власної бази даних.

15. ВИЗНАЧЕННЯ ТАБЛИЦЬ

15.1. Створення таблиці. Оператор CREATE TABLE

У реляційній базі даних найважливішим елементом її структури є таблиця. У багатокористувацькій базі даних основні таблиці, як правило, створює адміністратор, а користувачі лише звертаються до них під час роботи. У власній базі даних користувач може самостійно створювати потрібні йому таблиці.

При роботі з базою даних часто зручно створювати власні таблиці та зберігати у них свої дані, а також дані, зчитані з інших таблиць. Такі таблиці можуть бути тимчасовими і існувати лише протягом одного інтерактивного SQL-сеансу, але можуть зберігатися й протягом кількох тижнів або навіть місяців.

Оператор CREATE TABLE визначає нову таблицю. Різні секції оператора задають елементи визначень таблиці. Необхідно пам'ятати, що не всі параметри стандарту SQL2 присутні в усіх СУБД.

Після виконання оператора CREATE TABLE з'являється нова таблиця, якій присвоюється назва, зазначена в операторі. Назва таблиці повинна бути ідентифікатором, допустимим у SQL, і не повинна конфліктувати з назвами існуючих таблиць.

Синтаксис команди CREATE TABLE в PostgreSQL досить обширний і складний. Фактично він містить директиви визначення:

1. Параметрів таблиці;
2. Переліку та параметрів стовпців (колонок) таблиці;
3. Параметрів первинного ключа;
4. Переліку та параметрів зовнішніх ключів;
5. Параметрів зберігання;
6. Параметрів індексів;
7. Параметрів зв'язку таблиць;
8. Умов унікальності даних;
9. Умов перевірки.

Розглянемо приклад директиви створення таблиці, згенерований pgAdmin – графічним клієнтом адміністрування PostgreSQL.

Розглянемо на прикладах використання найтиповіших директив команди CREATE TABLE. Для цього скористаємось директивою створення таблиці Students, згенерованою pgAdmin. Початковий варіант команди призводить до створення порожньої таблиці з назвою, але без стовпців:

CREATE TABLE "Students"

```
(
)
WITH (
  OIDS=FALSE
);
ALTER TABLE "Students"
  OWNER TO postgres;
```

У даному випадку буде створено команду на генерацію таблиці з назвою Students. Додатково вказується параметр зберігання OIDS=FALSE. Він означає, що ID для рядків таблиці спеціально створюватися не буде. Цю задачу розробник повинен розв’язати сам, створюючи власний первинний ключ таблиці.

Далі опис таблиці доповнюється директивою ALTER TABLE (див. нижче), в якій вказується ім'я власника таблиці OWNER TO. У даному випадку це postgres.

Крім того, можливі й інші параметри таблиці, зокрема ON COMMIT, PRESERVE ROWS, DELETE ROWS, DROP і TABLESPACE, які у даному випадку не розглядаються.

Примітка. У конкретних СУБД визначення таблиць може відрізнятися від стандарту SQL.

15.2. Створення стовпців таблиці

Визначення стовпців є частиною тексту команди створення таблиці.

Розглянемо, як виглядає текст оператора створення таблиці Students, що містить визначення всіх її стовпців.

Спочатку визначаються стовпці таблиці з їх типами і, можливо, з вимогою обов'язкового заповнення.

Потім визначаються обмеження CONSTRAINT, які встановлюють первинний ключ PRIMARY KEY, зовнішні ключі FOREIGN KEY і ті таблиці, на які посилаються REFERENCES зовнішні ключі.

Для ілюстрації створення стовпців таблиці скористаємось директивою згенерованою pgAdmin, для створення стовпців таблиці Students,.

```
CREATE TABLE "Students"
(
  "StdNmb" integer NOT NULL,
  "StdGrp" integer NOT NULL,
  "StdName" character varying(20),
  "StdSex" character varying(1),
```

```
  "StdBrthDate" date,
  "StdCity" integer,
  "StdStipend" real,
  "StdID" serial NOT NULL,
  CONSTRAINT "StdIDPK" PRIMARY KEY ("StdID"),
  CONSTRAINT "CtIDFK" FOREIGN KEY ("StdCity")
     REFERENCES "Cities" ("CtID") MATCH SIMPLE
     ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT "GrpIDFK" FOREIGN KEY ("StdGrp")
     REFERENCES "Groups" ("GrpID") MATCH FULL
     ON UPDATE CASCADE ON DELETE CASCADE
)
WITH (
  OIDS=FALSE
);
ALTER TABLE "Students"
  OWNER TO postgres;
```

Розглянемо визначення стовпців (колонок) таблиці.

Для кожного стовпця задається його назва і тип його елементів, наприклад integer, real тощо. Додатково для стовпця може бути вказано параметр NOT NULL, що забороняє внесення порожніх даних для його елементів. Для стовпців, що є первинними ключами, NOT NULL є обов'язковим.

Крім того, можливі інші параметри стовпців, зокрема CHECK NO INHERIT, DEFAULT, UNIQUE, які у даному випадку не розглядаються.

Зупинимось на визначенні неключових стовпців, серед яких можуть бути зовнішні ключі. У вказаному прикладі для стовпців з назвами StdNmb і StdGrp – відповідно номер студентського квитка і код групи, вказано NOT NULL, оскільки студентський квиток і студентська група є невід'ємною частиною визначення студента. Тому при першому заповненні таблиці даними значення StdNmb і StdGrp повинні бути введені обов'язково.

З іншого боку, при першому заповненні таблиці даними для стовпців з назвами StdBrthDate і StdCity, введення відповідної інформації не є обов'язковим і допускається NULL.

Тепер перейдемо до визначення первинного ключа – стовпець з іменем StdID. Тип даних у нього serial. Поля цього стовпця обов'язково повинні бути заповнені, тобто вони не допускають відсутності значення, і тому для ключового поля обов'язково вказується NOT NULL.

У обмеженні CONSTRAINT з назвою StdIDPK зазначено, що стовпець StdID є первинним ключем PRIMARY KEY.

CONSTRAINT "StdIDPK" **PRIMARY KEY** ("StdID")

Тепер розглянемо визначення зовнішніх ключів, що зв'язують таблицю Students з таблицями Cities і Groups. У даному прикладі це, відповідно, стовпці StdCity і StdGrp. Таким чином, необхідно додатково вказати, що ці стовпці є зовнішніми ключами і з якими таблицями вони пов'язані. Для цього використовуються відповідні обмеження CONSTRAINT.

Визначення стовпця StdCity як зовнішнього ключа має вигляд

CONSTRAINT "CtIDFK" **FOREIGN KEY** ("StdCity")
REFERENCES "Cities" ("CtID") **MATCH SIMPLE**
ON UPDATE NO ACTION ON DELETE NO ACTION

У обмеженні CONSTRAINT з іменем CtIDFK зазначено, що поле StdCity посилається REFERENCES на таблицю Cities з первинним ключем CtID. У цьому контексті таблиця Cities розглядається як батьківська для дочірньої таблиці Students. Тут варто звернути увагу, що при визначенні стовпця StdCity вказано, що при першому введенні його заповнення не є обов'язковим.

Визначення стовпця StdGrp як зовнішнього ключа має вигляд

CONSTRAINT "GrpIDFK" **FOREIGN KEY** ("StdGrp")
REFERENCES "Groups" ("GrpID") **MATCH FULL**
ON UPDATE CASCADE ON DELETE CASCADE

У обмеженні CONSTRAINT з іменем GrpIDFK зазначено, що поле StdGrp посилається REFERENCES на таблицю Groups з первинним ключем GrpID. У цьому контексті таблиця Groups розглядається як батьківська для дочірньої таблиці Students. Тут варто звернути увагу, що при визначенні стовпця StdGrp вказано, що при першому введенні заповнення цього поля є обов'язковим, тобто вказано NOT NULL.

Зупинимось на властивості збігу MATCH у описі зовнішнього ключа. Він може приймати 2 значення: SIMPLE і FULL. Значення, вставлені в стовпці зовнішніх ключів, порівнюються зі значеннями в стовпцях первинних ключів зовнішньої таблиці з урахуванням заданого типу відповідності.

У разі MATCH FULL поле зовнішнього ключа не може містити NULL. Це стосується стовпця StdGrp.

У разі MATCH SIMPLE поле зовнішнього ключа може містити NULL. Це стосується стовпця StdCity.

В заключенні розглянемо директиви ON UPDATE і ON DELETE. Вони можуть встановлювати одне з наступних значень: CASCADE, SET NULL, SET

DEFAULT або NO ACTION. Вони вказують, які дії слід виконати з рядком дочірньої таблиці, якщо зв'язаний з нею рядок батьківської таблиці оновлюється UPPDATE або видаляється DELETE.

На практиці визначення таблиці має ще ряд надзвичайно важливих параметрів. В першу чергу це визначення індексів, тригерів і збережених процедур. Вони будуть додаватися при розгляді відповідного матеріалу.

Строгий синтаксис визначення таблиці в PostgreSQL і ряд прикладів можна отримати за посиланням [7].

15.3. Зміна таблиці. Оператор ALTER TABLE

Під час роботи з таблицею іноді виникає потреба додати або змінити окремі елементи в її описі. Для зміни структури таблиці використовується оператор ALTER TABLE. За допомогою цього оператора можна змінити практично всі параметри, які визначено за допомогою оператора CREATE TABLE.

Синтаксис команди ALTER TABLE в PostgreSQL досить обширний і складний. Він містить директиви, що дозволяють додати або змінити параметри, визначені в операторі CREATE TABLE. Зокрема, за допомогою оператора ALTER TABLE можна:

1. Додати до таблиці визначення нового стовпця або видалити існуючий стовпець.
2. Змінити значення за замовчуванням для певного стовпця.
3. Додати новий або видалити існуючий первинний ключ таблиці.
4. Додати новий або видалити існуючий зовнішній ключ таблиці.
5. Додати нову або видалити існуючу умову унікальності.
6. Додати нову або видалити існуючу умову перевірки тощо.

Наведемо декілька прикладів:

Приклад. Додати в таблицю Cities стовпець поштового індексу PostCode

```
ALTER TABLE "Cities"
   ADD COLUMN "PostCode" character varying(5)
```

Приклад. У таблиці Students видалити стовпець StdStipend

```
ALTER TABLE "Students"
   DROP COLUMN "StdStipend" RESTRICT
```

Приклад. У таблиці Students перейменувати стовпець StdNmb на StdNmb:

```
ALTER TABLE "Students" RENAME COLUMN StdNmb TO StdNmb
```

Строгий синтаксис зміни параметрів таблиці та деякі приклади можна знайти за посиланням [8].

15.4. Видалення таблиці. Оператор DROP TABLE

Таблиці можна видалити з бази даних за допомогою оператора DROP TABLE.

DROP TABLE назва_таблиці **CASCADE | RESTRICT**

Оператор містить назву таблиці, яку потрібно видалити. Зазвичай користувач видаляє одну зі своїх власних таблиць і вказує у операторі неповну назву таблиці. Із відповідним дозволом можна також видалити таблицю іншого користувача, але в такому випадку необхідно вказувати повну назву таблиці.

Стандарт SQL2 вимагає, щоб оператор DROP TABLE включав параметр CASCADE або RESTRICT, які визначають, як видалення таблиці впливає на інші об'єкти в базі даних. Якщо вказано параметр RESTRICT і в базі даних є об'єкти, які містять посилання на таблицю, що видаляється, то виконання оператора DROP TABLE завершиться помилкою. У більшості комерційних СУБД дозволяється використання оператора DROP TABLE без будь-яких параметрів.

Приклад. Видалити таблицю Student

DROP TABLE Student

15.5. Визначення доменів. Оператори CREATE, ALTER та DROP DOMAIN

У реляційних базах даних існує обмеження для вмісту окремого стовпця таблиці – значення стовпця в усіх рядках таблиці повинні бути одного й того ж типу. Наприклад, всі поштові індекси міст в стовпці PostCode таблиці Cities складаються з 5 цифр.

У стандарті SQL2 визначення домену формально реалізовано як частина визначення бази даних. Згідно з цим стандартом, домен є іменованою сукупністю значень даних і широко використовується при визначенні бази даних як додатковий тип даних.

Домени корисні для абстрагування та винесення загальних характеристик різних полів в єдине місце для спрощення супроводження. Наприклад, у кількох таблицях може бути стовпець, що містить поштовий індекс, і для всіх таблиць потрібні однакові обмеження CHECK, що перевіряють синтаксис поштового індексу. У цьому випадку краще визначити домен, а не задавати для кожної таблиці окремі обмеження.

Домен створюється за допомогою оператора CREATE DOMAIN на основі базових типів даних. Після створення домену його можна використовувати всередині визначення таблиці як тип даних.

CREATE DOMAIN назва_домену **[AS]** тип_даних
[COLLATE правило_сортування **]**
[DEFAULT вираз **]**
[[CONSTRAINT назва_обмеження **]**
{ NOT NULL | NULL | CHECK (вираз**) } [...]]**

Слід звернути увагу, що при створенні домену може знадобитися задати порядок сортування його елементів за замовчуванням COLLATE. Але це можливо, якщо тип даних допускає сортування.

Приклад. Створити домен поштових індексів міст України. Індекс міста складається з 5 цифр.

CREATE DOMAIN ua_postal_code **AS** TEXT
CHECK(VALUE ~ 'd{5}')

Тут у визначенні домену використано регулярний вираз, який вказує, що поштовий індекс повинен складатися з 5 цифр.

Тепер назва домену ua_postal_code може використовуватися як тип даних у операторі CREATE TABLE при визначенні індексу міста PostCode в таблиці Cities.

ALTER TABLE "Cities"
ADD COLUMN "PostCode" ua_postal_code

Як і у випадку з іншими визначеннями бази даних, домен може бути змінений за допомогою ALTER DOMAIN або видалений за допомогою DROP DOMAIN, наприклад

DROP DOMAIN ua_postal_code

15.6. Визначення індексів. Оператори CREATE та DROP INDEX

15.6.1. Поняття індексів

Один із структурних елементів фізичної пам'яті, який присутній у більшості реляційних СУБД, є *індекс (Index)* – засіб, що забезпечує швидкий доступ до рядків таблиці на основі значення одного або декількох її стовпців.

Індекс – це об'єкт бази даних, створений для підвищення продуктивності виконання запитів. Таблиці в базі даних можуть містити велику кількість рядків, які зберігаються у довільному порядку, і пошук їх за заданим значенням шляхом послідовного перегляду таблиці рядок за рядком може займати багато часу.

Індекс формується зі значень одного (простий) або кількох (складний) стовпців таблиці та вказівників на відповідні рядки таблиці і, таким чином, дозволяє знаходити потрібні рядки за заданим значенням.

Як приклад розглянемо таблицю з даними про студентів та запит, що часто використовується, "Знайти всіх студентів, які навчаються в групі X", де X – певний параметр. За таких умов адміністратор БД може вибрати спосіб збереження даних, який схематично показано на рис. 15.1. Він базується на двох збережених файлах: файлі з даними про студентів та файлі з даними про групи; файли можуть розміщуватися в різних наборах сторінок. Передбачається, що у файлі груп використовується впорядкування за алфавітним переліком їх назв, тобто за ключовим полем GrpName (назва групи) з вказівниками на відповідні записи у файлі студентів.

Файл студентів (дані)

StNo	*GrName*	*StName*	*City*
98621	КС–981	Іваненко	Київ
98624	СП–981	Лучніченко	Львів
98622	КС–981	Петренко	Чернігів
98625	СП–981	Кузнєцов	Львів
98623	КС–981	Сидоренко	Чернігів

Файл груп (індекс)

КС–981	
КС–981	
КС–981	
СП–981	
СП–981	

Рис. 15.1. Індексування файлу студентів за полем GrpName файлу груп

Для пошуку всіх студентів з групи СП–981 можна застосувати наступну стратегію: знайти у файлі груп групу СП–981, а потім відповідно до вказівників витягти всі відповідні записи з файлу студентів.

Така стратегія буде ефективнішою порівняно з пошуком у файлі з даними студентів, оскільки СУБД відомо фізична послідовність записів у файлі груп (пошук припиниться, коли буде отримано наступна після СП–981 назва групи в алфавітному порядку). Крім того, навіть якщо доведеться переглянути файл груп повністю, для такого пошуку потрібно буде значно менше операцій вводу-виводу, оскільки фізичний розмір файлу груп менше, ніж розмір файлу з даними студентів через менший розмір записів.

У розглянутому прикладі файл груп називається індексним файлом або індексом відносно файлу студентів, і навпаки, файл студентів називається індексованим файлом відносно файлу груп.

Індексний файл – це збережений файл особливого формату, в якому кожен запис складається з двох значень, а саме даних та вказівника. Дані відповідають певному полю - *індексному полю*, з індексованого файлу, а вказівник служить для зв'язку з відповідною записом індексованого файлу. Індексне поле також називається *індексним ключем (index key)*.

Покращення роботи з використанням індексів досягається, насамперед, завдяки тому, що індекс має структуру, оптимізовану під пошук – наприклад, збалансоване дерево. Індекси фізично можуть бути реалізовані різними структурами. Найбільш використовувані B-дерева, B*-дерева, B+ дерева та хеш-таблиці.

Візьмемо наступний запит:

SELECT StdName
FROM Studens
WHERE GrpName = 'СП-981'

Для виконання такого запиту без індексу СУБД повинна перевірити поле StdStipend у кожному рядку таблиці (повний перебір). При використанні індексу СУБД просто пройде по B-дереву, поки не знайде запис СП-981. Такий прохід потребує набагато менше ресурсів, ніж повний перебір таблиці.

Індекси можуть бути визначені як унікальні, так і неунікальні. *Унікальний індекс* реалізує обмеження цілісності таблиці, виключаючи можливість вставки значень, що повторюються.

Послідовність, в якій стовпці представлені в складному індексі, є досить важливою.

15.6.2. Створення та видалення індексів

Для створення та видалення індексів використовуються оператори CREATE INDEX та DROP INDEX.

Слід також зазначити, що СУБД автоматично створює індекси для первинних ключів. Синтаксис цих операторів має наступний вигляд:

CREATE [UNIQUE] INDEX назва_індексу **ON** назва_таблиці
(назва_стовпця **[ASC | DESC],...)**
і
DROP INDEX назва_індексу

Приклад. Створити індекс StdStipendInd таблиці Students за стовпцем StdStipend.

CREATE INDEX StdStipendInd **ON** Students (StdStipend)

Приклад. Створити індекс StdStipStSexInd таблиці Students за стовпцями StdStipend та StdSex.

CREATE INDEX StdStipendInd **ON** Students (StdStipend, StdSex)

Приклад. Видалити індекс StdStipendInd таблиці Students.

DROP INDEX StdStipendInd

Примітка. Способи визначення індексів в PostgreSQL, їх типи та особливості поведінки будуть розглянуті нижче в окремому розділі.

15.7. Контрольні запитання

1. Перелічіть основні директиви команди створення таблиці.
2. Яка структура команди визначення таблиці?
3. Що є результатом виконання команди створення таблиці?
4. Яка структура визначення стовпця таблиці?
5. Яка структура визначення первинного ключа таблиці?
6. Яка структура визначення зовнішнього ключа таблиці?
7. Як встановлюється зв'язок між зовнішнім ключем таблиці та таблицею, на яку він посилається?
8. Яка структура команди зміни таблиці?
9. Яка структура команди видалення таблиці?
10. Як встановлюється зв'язок між зовнішнім ключем таблиці та таблицею, на яку він посилається?
11. Що таке домен і яка структура директиви створення домена?
12. Що таке індекс і яка структура директиви створення індексу?

15.8. Завдання для самопідготовки

Завдання 15.1. Наведіть текст директиви створення таблиці власної БД і поясніть її основні елементи.

Завдання 15.2. Наведіть текст директиви створення первинного ключа вашої таблиці і поясніть її основні елементи.

Завдання 15.3. Наведіть текст директиви створення зовнішнього ключа вашої таблиці і поясніть її основні елементи.

Завдання 15.4. Наведіть текст директиви створення індексу вашої таблиці і поясніть її основні елементи.

16. ПРЕДСТАВЛЕННЯ

16.1. Поняття представлення

Представлення – це віртуальна таблиця, яка зазвичай містить набір стовпців однієї або кількох таблиць. Фактично воно не представлене в базі даних, а задається запитом на вибірку даних SELECT. З цієї точки зору представлення – це збережений запит до БД.

Коли СУБД стикається у SQL-операторі з посиланням на представлення, вона знаходить його визначення, збережене в БД. Потім СУБД перетворює запит користувача, що посилається на представлення, в еквівалентний запит до вихідних таблиць представлення і виконує цей запит. Таким чином, СУБД створює ілюзію існування представлення у вигляді окремої таблиці, а водночас зберігає цілісність вихідних таблиць.

Застосування представлень у БД різних типів може бути корисним у найрізноманітніших ситуаціях. У БД на персональних комп'ютерах представлення використовуються для зручності та дозволяють спрощувати запити до БД. У промислових БД представлення відіграють головну роль у створенні власної структури БД для кожного користувача та забезпеченні її безпеки. Основні переваги представлень перелічені нижче:

1. Безпека. Кожному користувачеві можна дозволити доступ до невеликої кількості представлень, що містять лише ту інформацію, яку йому дозволено знати. Таким чином, можна обмежити доступ користувачів до інформації, що зберігається.
2. Простота запитів. За допомогою представлення можна зчитувати дані з кількох таблиць та представляти їх як одну таблицю, перетворюючи таким чином запит до багатьох таблиць у однотабличний запит до представлення.
3. Простота структури. За допомогою представлень для кожного користувача можна створити власну (віртуальну) структуру БД, визначивши її як множину доступних користувачу віртуальних таблиць.
4. Захист від змін. Представлення може повертати несуперечливий та незмінний образ структури БД, навіть якщо вихідні таблиці розділяються, реструктуруються або перейменовуються.
5. Цілісність даних. Якщо доступ до даних або введення даних здійснюється за допомогою представлення, СУБД може автоматично перевіряти, чи виконуються певні умови цілісності.

Попри перелічені переваги, представлення мають два суттєві недоліки:

1. Продуктивність. Представлення створює лише видимість існування відповідної таблиці, і СУБД доводиться перетворювати запит до представлення у запит до вихідних таблиць, що вимагає витрат часу.
2. Обмеження на оновлення. Коли користувач намагається оновити рядки представлення, СУБД повинна встановити їх відповідність рядкам вихідних таблиць та оновити останні. Це можливо лише для простих представлень. Складні представлення неможливо оновлювати, вони доступні лише для читання.

Зазначені недоліки означають, що не варто захоплюватися представленнями замість вихідних таблиць. В кожному конкретному випадку необхідно враховувати перелічені переваги та недоліки представлень.

16.2. Створення представлень. Оператор CREATE VIEW

Для створення представлень використовується оператор CREATE VIEW. У ньому вказуються назва представлення та запит, що є його основою. Для успішного створення представлення необхідно мати дозвіл на доступ до всіх таблиць, які входять у запит.

CREATE [OR REPLACE] VIEW назва_представлення (назва_стовпця,...) **AS** запит

За необхідності у операторі CREATE VIEW можна задати назву для кожного стовпця створюваного представлення. Якщо вказується список назв стовпців, то він повинен містити стільки елементів, скільки стовпців містить запит. Зверніть увагу на те, що задаються лише назви стовпців.

Примітка. Конструкція є розширенням SQL у PostgreSQL.

Приклад. Створити представлення, що містить прізвища студентів групи КС-981.

```
CREATE VIEW GroupKC981 (Name) AS
SELECT StdName
FROM Students INNER JOIN Groups ON Students.GrpNo = Groups.GrpNo
WHERE Groups.GrpName = 'КС-981'
```

16.3. Оновлення представлень

Оновлення представлення можна розглядати у 2-х варіантах:

1. Фактичне змінення представлення.
2. Змінення різних властивостей представлення.

Для змінення представлення використовується команда

CREATE OR REPLACE VIEW назва_представлення (назва_стовпця,...) **AS** запит

Вона фактично визначає нове представлення з тією ж назвою. При цьому наявні стовпці повинні залишатися, але можуть бути додані також і нові.

Приклад. Змінити представлення GroupKC981 та додати стовпець «стипендія».

CREATE OR REPLACE VIEW GroupKC981 (Name, Stipend) **AS**
SELECT StdName, StdStipend
FROM Students **INNER JOIN** Groups **ON** Students.GrpNo = Groups.GrpNo
WHERE Groups.GrpName = 'КС-981'

Для зміни різних властивостей представлення використовується команда ALTER VIEW. Вона має декілька варіантів. Для прикладу розглянемо наступний:

ALTER VIEW [IF EXISTS] назва **RENAME [COLUMN]** назва_стовпця

TO нова_назва_стовпця

Приклад. У представленні GroupKC981 замінити назва стовпця Name на StdName

ALTER VIEW *GroupKC981* **RENAME** Name **TO** StdName

16.4. Видалення представлень. Оператор DROP VIEW

У стандарті SQL2 було формально закріплено використання оператора DROP VIEW для видалення представлень. У ньому також деталізовано правила видалення представлень, на основі яких були створені інші представлення.

DROP VIEW [IF EXISTS] назва_представлення **CASCADE | RESTRICT**

Параметр CASCADE передбачає автоматичне видалення об'єктів, що залежать від даного представлення, наприклад, інших представлень, а також всіх об'єктів, що залежать від них.

Параметр RESTRICT передбачає відмову у видаленні представлення, якщо від нього залежать будь-які об'єкти. Цю поведінку передбачено за замовчуванням.

Приклад. Видалити представлення GroupKC981

DROP VIEW *GroupKC981*

16.5. Контрольні запитання

1. Що таке представлення?
2. Для чого використовуються представлення?
3. Яка структура директиви створення представлення?
4. Яка структура директиви визначення стовпця таблиці?
5. Які варіанти оновлення представлень ви знаєте?
6. В чому різниця між директивами REPLACE VIEW та ALTER VIEW?
7. Яка структура директиви на видалення представлення і в чому її особливість?

16.6. Завдання для самопідготовки

Завдання 16.1. Наведіть текст директиви створення представлення у власній БД і поясніть її основні елементи.

Завдання 16.2. Наведіть текст директиви зміни представлення у власній БД і поясніть її основні елементи.

Завдання 16.3. Наведіть текст директиви зміни окремих властивостей представлення у власній БД і поясніть її основні елементи.

Завдання 16.4. Наведіть текст директиви видалення представлення у власній БД і поясніть її основні елементи.

17. КУРСОРИ

17.1. Поняття курсора

Однією з сильних сторін SQL є його здатність працювати зі всіма рядками таблиці, щоб розглядати певну умову виділення рядків як блок записів, не знаючи заздалегідь, скільки таких рядків може бути. Якщо 10 рядків відповідають умові виділення, то запит може вивести всі ці 10 рядків. Якщо для виведення визначено 10000000 рядків таблиці, то всі ці 10000000 рядків будуть виведені. Однак це дещо ускладнюється, коли необхідно використовувати SQL-запити в різних мовах програмування. Наприклад, як можна призначати виведення результатів запиту для змінних, якщо заздалегідь невідомо, наскільки великим буде обсяг виводу? Рішення полягає в тому, щоб використовувати так званий *курсор*.

Курсор – це поняття мови SQL, що дозволяє за допомогою набору спеціальних операторів отримати рядковий доступ до результату запиту до БД.

Запит до реляційної БД зазвичай повертає декілька рядків (записів) даних, але програмний додаток обробляє лише один рядок за один раз. Крім того, при модифікації, видаленні або додаванні даних робочою одиницею є рядок. У цій ситуації на передній план виходить концепція курсора, і в такому контексті курсор – це вказівник на рядок. Вдалим аналогом курсора є ітератори в колекціях або контейнерах об'єктів.

Зазвичай курсори використовуються для вибору з БД певної підмножини збережених в ній даних. У кожний момент часу прикладною програмою може бути оброблено лише один рядок курсору.

Курсори часто використовуються в операторах SQL, вбудованих в прикладні програми, написані мовами процедурного типу. Деякі курсори неявно створюються сервером БД, у той час як інші визначаються програмістами.

Згідно зі стандартом SQL під час роботи з курсорами можна виділити наступні основні дії:

1. Створення або оголошення курсора.
2. Відкриття курсора, тобто наповнення його даними, які зберігаються в багаторівневій пам'яті.
3. Вибірка з курсора та зміна за допомогою нього рядка даних.
4. Закриття курсора, після чого він стає недоступним для програм користувача.
5. Звільнення курсора, тобто видалення курсора як об'єкта, оскільки його закриття необов'язково звільняє пам'ять, що асоціюється з ним.

У деяких випадках застосування курсора неминуче, наприклад, при предметній рядковій обробці даних або при реалізації каскадних змін даних. Очевидно, що курсори не дозволяють проводити операції зміни над усім обсягом даних, ці операції виконуються рядок за рядком. Тому швидкість виконання операцій обробки даних за допомогою курсора помітно нижча, ніж у стандартних засобах SQL.

За можливості слід уникати використання курсорів і працювати зі стандартними командами обробки даних: SELECT, UPDATE, INSERT і DELETE, особливо, якщо програма використовує візуальні компоненти.

Курсори поділяються на дві категорії: послідовні та з довільним напрямком доступу. Послідовні курсори дозволяють вибирати дані лише в одному напрямку – від початку до кінця. Курсори з довільним напрямком доступу надають більшу свободу дій – допускається переміщення у обох напрямках та перехід до довільного рядка результуючого набору курсору. Говорячи про курсори, не слід забувати про ізольованість транзакцій. Коли один користувач змінює запис, інший читає його за допомогою власного курсора, більше того, він може також змінити той самий запис, що вимагає дотримання цілісності даних (див. нижче).

Також розрізняють курсори *статичні* та *динамічні*. У схемі зі статичним курсором інформація читається з БД один раз і зберігається у вигляді моментального знімка (на певний момент часу), тому зміни, внесені в БД іншим користувачем, не відображаються. Під час відкриття курсора сервер встановлює блокування на всі рядки, які входять у його повний результат. Статичний курсор не змінюється після створення і завжди показує набір даних, який існував на момент його відкриття.

Якщо інші користувачі змінять дані, що включені в курсор, у початковій таблиці, це не вплине на статичний курсор.

У статичний курсор неможливо внести зміни, тому він завжди відкривається у режимі тільки для читання.

Динамічний курсор підтримує дані у "живому" стані, але це потребує витрат мережевих і програмних ресурсів. Під час використання динамічних курсорів не створюється повна копія початкових даних, а виконується динамічний вибір з початкових таблиць лише при зверненні користувача до даних. Під час вибірки сервер блокує рядки, і всі зміни, внесені користувачем до повного результату курсора, будуть видимими в курсорі. Проте, якщо інший користувач внесе зміни після вибірки даних курсором, вони не відобразяться в курсорі.

17.2. Створення та використання курсора

У стандарті SQL для створення курсора передбачено наступну команду:

DECLARE назва_курсора **[INSENSITIVE] [SCROLL] CURSOR**
FOR SELECT_оператор
[FOR { READ_ONLY | UPDATE [OF назва_стовпця **[,...n]] }]**

При використанні ключового слова INSENSITIVE буде створено статичний курсор. У цьому випадку зміни даних в тій же транзакції не дозволяються, крім того, не відображаються зміни, зроблені іншими користувачами. Якщо ключове слово INSENSITIVE відсутнє, створюється динамічний курсор.

При зазначенні ключового слова SCROLL створений курсор можна прокручувати у будь-якому напрямку, що дозволяє застосовувати будь-які команди вибірки. Якщо цей аргумент пропущений, то курсор буде послідовним, тобто його перегляд буде можливим лише в одному напрямку – від початку до кінця.

SELECT_оператор визначає тіло запиту SELECT, за допомогою якого визначається результуючий набір рядків курсору.

При вказанні аргументу FOR READ_ONLY створюється курсор лише для читання, і жодні модифікації даних не дозволяються. Він відрізняється від статичного курсора, хоча останній також не дозволяє змінювати дані.

В якості курсора, призначеного лише для читання, може бути оголошений динамічний курсор, що дозволить відображати зміни, зроблені іншим користувачем.

Створення курсора з аргументом FOR UPDATE дозволяє виконувати зміну даних в курсорі або вказаних стовпцях, або, за відсутності аргументу OF назва_стовпця, у всіх стовпцях.

Для відкриття курсора та наповнення його даними з вказаного при створенні курсору запиту SELECT використовується наступна команда:

OPEN { { [GLOBAL] назва_курсора **} | @**назва_змінної_курсору**}**

Після відкриття курсора відбувається виконання пов'язаного з ним оператора SELECT, вихідні дані якого зберігаються в багаторівневій пам'яті.

Одразу після відкриття курсора можна вибрати його вміст (результат виконання відповідного запиту) за допомогою наступної команди FETCH, яка має свій синтаксис.

Команда FETCH дозволяє переміщатися та отримувати доступ до зазначеного рядка повного результуючого набору курсору, який стає поточним рядком.

Команда має наступний вигляд

FETCH [напрямок**] [FROM | IN]** назва_курсора

Можливі такі основні напрямки переміщення:

1. перейти до першого рядка FIRST;
2. перейти до останнього рядка LAST;
3. розташуватися після поточного рядка NEXT;
4. розташуватися перед поточним рядком PRIOR;
5. перейти за абсолютним номером рядка ABSOLUTE;
6. розташуватися зі зміщенням відносно поточного рядка RELATIVE.

Для виконання змін за допомогою курсора використовується команда UPDATE.

Для видалення даних за допомогою курсора використовується команда DELETE.

Для закриття курсора використовується команда

CLOSE {назва_курсора **| @**назва_змінної_курсору**}**

Після закриття курсор стає недоступним для користувачів програми. При закритті знімаються всі блокування, встановлені в процесі його роботи. Закриття може застосовуватися тільки до відкритих курсорів. Закритий, але не звільнений курсор може бути повторно відкритий. Не допускається закривати не відкритий курсор.

Слід зазначити, що в конкретних СУБД реалізація курсора може відрізнятися від стандарту SQL.

Приклад. Оголосити курсор для таблиці Students

DECLARE st_cursor **CURSOR FOR SELECT** * **FROM** Students;

Приклад. Відкрити курсор для таблиці Students

OPEN st_cursor

Приклад. Перемістити курсор до останнього

FETCH LAST IN st_cursor

Приклад. Закрити курсор для таблиці Students

CLOSE st_cursor

17.3. Контрольні запитання

1. Що таке курсор?
2. Де і для чого використовуються курсори?
3. Яка структура директиви на створення курсора?
4. Яка процедура створення курсора?
5. Що таке послідовні курсори та курсори з довільним напрямком доступу?
6. Що таке динамічні та статичні курсори?
7. Яке призначення команди FETCH?
8. Які основні можливості надає команда FETCH?

17.4. Завдання для самопідготовки

Завдання 17.1. Створіть курсор для таблиці власної БД.

Завдання 17.2. Наведіть текст директиви на видалення курсора.

Завдання 17.3. Наведіть приклади використання команди FETCH.

18. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ

Збережена процедура (Stored Procedure) – об'єкт БД, який представляє собою набір SQL-інструкцій, що компілюється один раз і зберігається на сервері. Збережені процедури дуже схожі на звичайні процедури мов високого рівня, вони можуть мати вхідні та вихідні параметри та локальні змінні, в них можуть виконуватися числові обчислення та операції над символьними даними, результати яких можуть бути присвоєні змінним та параметрам. У збережених процедурах можуть виконуватися стандартні операції з БД (як описи даних DDL, так і маніпулювання даними DML). Крім того, у збережених процедурах можливі цикли та розгалуження, тобто вони можуть використовувати інструкції управління потоком.

18.1. Призначення та переваги збережених процедур

Збережені процедури дозволяють підвищити продуктивність, розширюють можливості програмування, а також підтримують функції безпеки даних.

Замість збереження запиту, що часто використовується, клієнти можуть посилатися на відповідну збережену процедуру. В момент виклику збереженої процедури її вміст одразу обробляється сервером.

Крім самого виконання запиту, збережені процедури також дозволяють виконувати обчислення та маніпулювати даними – змінювати, видаляти, виконувати DDL-оператори та викликати інші збережені процедури, виконувати складну транзакційну логіку. Один оператор дозволяє викликати складний сценарій, який міститься в збереженій процедурі, що дозволяє уникнути передачі через мережу сотень команд та особливо необхідності передачі великих обсягів даних від клієнта на сервер.

Збережені процедури роблять дуже простою модифікацію алгоритмів обробки даних, які відразу ж стають доступними для всіх користувачів мережі, та забезпечують можливість розширення системи без внесення змін до самої програми: достатньо змінити збережену процедуру на сервері БД. Розробнику не потрібно перекомпілювати програму, створювати її копії, а також інструктувати користувачів про необхідність роботи з новою версією. Користувачі взагалі можуть не підозрювати, що в систему внесено зміни.

Збережені процедури існують незалежно від таблиць чи будь-яких інших об'єктів БД. Вони викликаються клієнтською програмою, іншою збереженою процедурою або тригером (див. нижче). Розробник може управляти правами доступу до збереженої процедури, дозволяючи або забороняючи її виконання.

У більшості СУБД при першому запуску збереженої процедури вона компілюється (виконується синтаксичний аналіз та генерується план доступу до даних). В подальшому її обробка здійснюється швидше.

18.2. Можливості програмування

Створену збережену процедуру можна викликати в будь-який момент, що забезпечує модульність та стимулює повторне використання коду. Останнє полегшує супровід БД, оскільки вона стає ізольованою від змінюваних бізнес-правил. Модифікувати збережену процедуру відповідно до нових правил можна в будь-який момент. Після цього всі програми, які використовують її, автоматично адаптуються до нових бізнес-правил без безпосередньої модифікації.

18.3. Безпека

Використання збережених процедур дозволяє обмежити або взагалі виключити прямий доступ користувачів до таблиць БД, залишивши користувачам лише дозволи на виконання збережених процедур, які забезпечують опосередкований та строго регламентований доступ до даних. Крім того, деякі СУБД підтримують шифрування тексту збереженої процедури.

18.4. Засоби реалізації збережених процедур у промислових СУБД

Слід зазначити, що збережені процедури зазвичай пишуться мовами програмування, які є процедурними розширеннями мови SQL. Ці розширення містять такі оператори, як do while, if then else, що дозволяють описувати ширший спектр алгоритмів, ніж ті, що надає SQL.

На відміну від мови SQL, яка підпорядковується стандарту, її процедурні розширення не стандартизовані і різні СУБД використовують різні синтаксичні реалізації одних і тих же алгоритмічних конструкцій. Наприклад, для цих цілей у СУБД існують спеціалізовані мови програмування наведені в Таблиця 18.1.

У деяких СУБД можливе використання збережених процедур, написаних будь-якою мовою програмування, яка здатна створювати незалежні файли, що виконуються, наприклад, Java, C++ або Delphi (див. таблицю 18.1).

Таблиця 18.1. Мови програмування збережених процедур

СУБД	Мова збережених процедур
Oracle	PL/SQL, Java
Informix	SPL
Sybase	T-SQL, SQLJ
IBM DB2	SQL/PL подібний PL/SQL Oracle; Всі поширені мови програмування
MS SQL Server 2005 і вище	Transact-SQL, всі мови програмування в .Net
PostgreSQL	PL/pgSQL подібний PL/SQL Oracle; PL/Tcl, PL/Perl, PL/Python, PL/Java, C
MySQL	Подібний PL/SQL Oracle
InterBase	PSQL
Cache'	Cache' Object Script

У термінології Microsoft SQL Server такі процедури називаються розширеними збереженими процедурами і є просто функціями, що містяться в Win32/64-DLL.

Починаючи з версії MS SQL 2005, у СУБД з'явилась можливість писати збережені процедури будь-якою мовою .Net, а від розширених збережених процедур планується відмовитись у майбутньому.

У Oracle існує можливість писати збережені процедури мовою Java.

У IBM DB2 написання збережених процедур і функцій звичайними мовами програмування є традиційним способом, що підтримується з самого початку, а процедурне розширення SQL було додано до цієї СУБД лише в досить пізніх версіях, після його включення до стандарту ANSI.

Примітка. Також слід зазначити, що деякі ODBC-драйвери (див. далі) не підтримують виклик збережених процедур з клієнтських додатків, навіть якщо такі підтримуються самою СУБД.

18.5. Контрольні запитання

1. Що таке збережена процедура?
2. Яке призначення і переваги збережених процедур?
3. Що таке процедурне розширення мови SQL?
4. Охарактеризуйте можливості різних СУБД щодо написання збережених процедур.

19. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ ТА ФУНКЦІЇ В POSTGRESQL

Історично більшисть сучасних СУБД підтримує збережені функції чи їх аналоги.

Збережені функції (Stored Functions) СУБД PostgreSQL ведуть себе як функції, які повертають одне значення заданого типу, і як процедури, які можуть повертати декілька значень через вихідні параметри.

Однак пізніше в СУБД PostgreSQL було додано можливість підтримки *збережених процедур (Stored Procedures)*. Вони не повертають значень, але можуть видавати дані в програмний код, що їх викликає, через свої вихідні параметри.

Збережені функції та збережені процедури мають схожу структуру. Принципова відмінність між ними полягає в тому, що функцію можна викликати так само, як викликається SQL запит, і вона може бути частиною якогось SQL-запиту, а процедуру можна викликати лише за допомогою спеціального оператора

CALL назва_процедури()

Завдяки цим особливостям збережені функції використовуються набагато частіше, ніж збережені процедури.

У цьому підручнику розглядаються питання створення та використання збережених функцій, проте за традицією їх часто називають збереженими процедурами. За необхідності освоєння питань створення та використання саме збережених процедур не складе особливих труднощів.

19.1. Оператор Create Function мови PL/pgSQL

Програми PL/pgSQL складаються з блоків. Програмні блоки вводяться у командах SQL CREATE FUNCTION, які використовуються для визначення функцій PL/pgSQL в БД PostgreSQL.

Команда CREATE FUNCTION визначає назву функції, типи її аргументів і значення, що повертається.

Основний блок функції починається з секції оголошень змінних, які можуть бути ініціалізовані значеннями за замовчуванням. У оголошенні вказується назва та тип змінної. Секція оголошень позначається ключовим словом DECLARE, а кожне оголошення завершується символом крапка з комою ";".

Після оголошення змінних йде ключове слово BEGIN, яке позначає початок основного програмного блоку. За ключовим словом BEGIN йдуть оператори, які входять у блок. Кінець програмного блоку позначається ключовим словом END. Основний блок функції PL/pgSQL повинен повернути значення заданого типу, а всі вкладені блоки (блоки, що починаються всередині інших блоків) повинні бути завершені до досягнення ключового слова END.

У мінімальному варіанті синтаксису опису функції у PL/pgSQL наведено нижче:

CREATE [OR REPLACE] FUNCTION <назва_функції> ([<параметр> <тип даних>, ...])
RETURNS <тип_даних_результату> **AS** $$
[**DECLARE** оголошення; ...]
BEGIN
    <блок_операторів>; [...]
    RETURN <назва_змінної>;
END;
$$
LANGUAGE 'plpgsql';

Усі змінні, що використовуються в блоці операторів, повинні бути визначені в розділі оголошень DECLARE. Змінні можуть мати будь-який тип даних SQL, такий як integer, varchar, char.

Приклади оголошення змінних:

user_id *integer*;
avg_ball *real*;
url *varchar*;
gr_name *character varying*
data_row *tablename%ROWTYPE*;
data_field *tablename.columnname%TYPE*;
data_arrow *record*;

Є один виняток – змінна лічильника циклу FOR, яка оголошується автоматично залежно від типу циклу. Для циклу за діапазоном чисел це змінна типу integer, а для циклу за результатами курсора – це змінна типу record.

Строгий синтаксис визначення функції та ряд прикладів можна знайти за посиланням [9].

19.2. Виклик збережених функцій мови PL/pgSQL у прикладній програмі

Узагальнений виклик збереженої функції мови PL/pgSQL у прикладній програмі виглядає наступним чином:

SELECT * FROM public."Назва функції"([параметр, …]);

Далі засобами конкретної мови програмування для роботи з SQL-запитами слід підготувати вхідні параметри (якщо вони є), відправити запит на виконання, а потім послідовно виділити значення полів оператора SELECT, які повернулись.

19.3. Приклади збережених функцій та їх виклики

19.3.1. Функція без вхідних параметрів, яка повертає одне значення

Підрахувати суму витрат на стипендії всіх студентів.

Початковий текст функції getSumStipAll() наступний:

```
CREATE OR REPLACE FUNCTION "getSumStipAll"()
    --Визначення типу результату функції
RETURNS real AS
$$
    --Блок визначення змінних
DECLARE summa real;
    --Основний блок
BEGIN
  summa = (SELECT SUM("StdStipend") FROM public."Students");
  RETURN summa;
END; $$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі виглядає наступним чином:

SELECT * FROM public."getSumStipAll"();

Далі засобами конкретної мови програмування для роботи з SQL запитами слід відправити запит на виконання, а потім виділити єдине поле оператора SELECT.

19.3.2. Приклад функції з двома вхідними параметрами, яка не повертає значень

Додати новий запис у таблицю Regions.

Початковий текст функції insRegion_1():

```
CREATE OR REPLACE FUNCTION "insStudents"(num_std integer,
                                         name_std character varying)
  RETURNS void AS
$$
BEGIN
    INSERT INTO public."Students" ("StdNmb", "StdName")
    VALUES (num_std, name_std);
END;
$$;
LANGUAGE 'plpgsql'
```

Виклик цієї функції у прикладній програмі має такий вигляд:

```
SELECT * FROM public."insRegion_1"('981725', 'Петренко');
```

Як видно з тексту функції insRegion_1(), крім оператора SELECT в основному блоці функції можна використовувати інші оператори мови SQL, а також оператори мови PL/pgSQL.

Далі засобами мови програмування слід підготувати два вхідні параметри та відправити запит на виконання.

19.3.3. Приклад функції з одним вхідним параметром, яка повертає одне значення

Підрахувати суму витрат на стипендію всіх студентів групи X, де X – параметр, що задає назву групи.

Початковий текст функції getStipGroup() наступний:

```
CREATE OR REPLACE FUNCTION "getStipGroup"(gr_name character varying)
RETURNS real AS
$$
DECLARE sum real;
BEGIN
   sum = (SELECT SUM("Students"."StdStipend")
              FROM public."Students", public."Groups"
```

```
            WHERE ("Students"."StdGrp" = "Groups"."GrpID") AND
              ("Groups"."GrpName" = gr_name));
    RETURN sum;
END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

```
SELECT * FROM public."getStipGroup"('КС-981');
```

Далі засобами мови програмування слід підготувати один вхідний параметр, відправити запит на виконання, а потім виділити поле оператора SELECT, що повертається.

19.3.4. Приклад функції з одним вхідним і двома вихідними параметрами, через які повертаються результати

Наступний приклад демонструє можливість функції повертати декілька значень. В функцію передаються 3 параметри: 1 вхідний (режим IN) і 2 вихідні (режим OUT), через які функція поверне результати. Слід звернути увагу на те, що тип значення, що повертається, в цьому випадку – record.

Вивести кількість студентів групи X і середній бал за групою, де X – параметр, що задає назву групи.

Так виглядає початковий текст функції getCntAndSrBallGrp():

```
CREATE OR REPLACE FUNCTION "getCntAndAvgBallGrp"(
IN gr_name character varying,
OUT cnt_stud integer, OUT avg_ball real)
RETURNS record AS
$$
BEGIN
cnt_stud = (SELECT COUNT("Students"."StdID")
      FROM public."Students", public."Groups"
      WHERE ("Students"."StdGrp"="Groups"."GrpID") AND
            ("Groups"."GrpName"=gr_name));

avg_ball = (SELECT AVG("Students"."StdAvgBall")
      FROM public."Students", public."Groups"
      WHERE ("Students"."StdGrp"="Groups"."GrpID") AND
            ("Groups"."GrpName"=gr_name));
```

```
END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

```
SELECT * FROM public."getCntAndAvgBallGrp"('КС-981');
```

Далі засобами конкретної мови програмування для роботи з SQL запитами слід підготувати вхідний параметр, а потім послідовно виділити значення полів оператора SELECT, які повернулись.

Примітка. Тип даних record, що повертається як результат, і тип даних record не є одним і тим же. Спеціально це питання буде розглянуто нижче.

19.3.5. Повернення таблиці. Тип TABLE

Приклад функції з 1 вхідним параметром, яка повертає множину записів отриманої (тимчасової) таблиці.

Вивести інформацію про студентів групи X, де X – параметр, що задає назву групи.

Так виглядає початковий текст функції getStudGroup_2():

```
CREATE OR REPLACE FUNCTION "getStudGroup_2"(gr_name character varying)
RETURNS TABLE("StdNmb" integer, "StdName" character varying) AS
    -- RETURNS TABLE повністю відповідає поверненню
    -- при оголошенні одного або декількох вихідних параметрів OUT або
    -- вказівкою RETURNS SETOF деякий_тип (див. нижче)
$$
BEGIN
    -- Повертається звичайна таблиця з двома стовпцями.
    -- Щоб повернути велику кількість записів необхідно
    -- після ключового слова RETURN
    -- вказати ключове слово QUERY, за яким слідує текст запиту
RETURN QUERY SELECT "Students"."StdNmb", "Students"."StdName"
            FROM public."Students", public."Groups"
            WHERE (("Students"."StdGrp" = "Groups"."GrpID") AND
                    ("Groups"."GrpName" = gr_name));
END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

SELECT * FROM public."getStudGroup_2"('КС-981');

Далі засобами мови програмування слід підготувати 1 вхідний параметр, відправити запит на виконання, а потім виділити рядки, що повертаються, і поля оператора SELECT.

19.3.6. Повернення множини рядків таблиці. Тип SETOF

Приклад функції з 1 вхідним параметром, яка повертає множину записів таблиці певного типу, яка зберігається в базі даних.

У даному випадку це таблиця Students. Вивести інформацію про студентів групи X, де X – параметр, що задає назву групи.

Початковий текст функції getStudGroup_1():

```
CREATE OR REPLACE FUNCTION "getStudGroup_1"(gr_name character varying)
RETURNS SETOF "Students" AS
    -- SETOF вказує на те, що результатом, що повертається,
    -- буде множина елементів (записів) таблиці "Students"
$$
BEGIN
    -- Щоб повернути множину записів, після ключового слова return
    -- необхідно вказати ключове слово query, за яким слідує текст запиту
RETURN QUERY SELECT "Students"."StdID", "Students"."StdGrp",
                    "Students"."StdName", "Students"."StdStipend",
             "Students"."StdCity",
             "Students"."StdBrthDate", "Students"."StdSex"
             FROM public."Students", public."Groups"
                    WHERE (("Students"."StdGrp" = "Groups"."GrpID") AND
                      ("Groups"."GrpName" = gr_name));
END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

SELECT * FROM public."getStudGroup_1"('КС-981');

Далі засобами мови програмування слід підготувати 1 вхідний параметр, відправити запит на виконання, а потім виділити поля оператора SELECT, що повертаються.

19.3.7. Приклад функції з двома вхідними параметрами, яка повертає множину елементів (записів) таблиці

Додати новий запис у таблицю Regions і вивести всі записи з цієї таблиці.

Початковий текст функції insRegion_2():

```
CREATE OR REPLACE FUNCTION "insRegion_2"(reg_no integer,
                                         reg_name character varying)
  RETURNS SETOF "Regions" AS
    -- SETOF вказує на те, що результатом, що повертається,
    -- буде множина елементів (записів) таблиці "Regions"
$$
BEGIN
INSERT INTO public."Regions" ("RgNo", "RgName")
VALUES (reg_no, reg_name);
RETURN QUERY SELECT "RgNo", "RgName" FROM public."Regions";
END;
$$
LANGUAGE 'plpgsql'
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

```
SELECT * FROM public."insRegion_2"('11', 'Київська');
```

Далі засобами мови програмування слід підготувати два вхідні параметри і відправити запит на виконання, а потім виділити поля оператора SELECT, що повертаються.

19.3.8. Змінні типу даних стовпця таблиці. Конструкція %TYPE

Конструкція %TYPE з'явилась у більш пізніх версіях PL/pgSQL і надає тип даних змінної або стовпця таблиці. Цю конструкцію можна використовувати для оголошення змінних, які отримують значення з БД. Наприклад, для оголошення змінної з таким же типом, як і стовпець StdName у таблиці Students, потрібно написати:

```
std_name "Students"."StdName"%TYPE;
```

Використання конструкції %TYPE дозволяє автоматично отримувати з БД тип змінної в залежності від типу стовпця таблиці, що підвищує надійність програми. У протилежному випадку програмісту доводиться стежити, щоб типи його змінних не суперечили типам стовпців, що обробляються.

19.3.9. Приклад функції з одним вхідним параметром, яка повертає стовпець значень таблиці

Вивести імена студентів групи X, де X – параметр, що задає назву групи.

Початковий текст функції getStdNameGrp():

```
CREATE OR REPLACE FUNCTION "getStdNameGrp"(st_nmb integer)
RETURNS  "Students"."StdName"%TYPE  AS
    -- Перевірити тип повернення !!!
$$
DECLARE
std_name "Students"."StdName"%TYPE;
BEGIN
    -- Повертається одне значення.
std_name = (SELECT "Students"."StdName"
            FROM public."Students"
            WHERE "Students"."StdNmb" = st_nmb);
    RETURN std_name;

END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

```
SELECT * FROM public."getStdNameGrp"(981621);
```

Далі засобами мови програмування слід підготувати 1 вхідний параметр, відправити запит на виконання, а потім виділити рядки, що повертаються, і поля оператора SELECT.

19.3.10. Змінні типу кортеж. Конструкція %ROWTYPE

Змінна типу кортеж або змінна-кортеж є змінною складного типу, оскільки включає дані різних типів. Значенням такої змінної може бути цілий кортеж, отриманий в результаті виконання запиту SELECT або FOR, за умови, що набір стовпців запиту відповідає заявленому типу змінної.

Доступ до окремих полів значення кортежу здійснюється як зазвичай, через крапку за схемою назва_змінної.назва_поля_таблиці, наприклад std_row."StdName".

Змінна-кортеж може бути оголошеною з таким же типом, як і рядок у існуючій таблиці, використовуючи запис назва_таблиці%ROWTYPE.

Конструкція % ROWTYPE з'явилась у більш пізніх версіях PL/pgSQL і надає можливість визначити кортеж з полів рядка таблиці з їх типами. Це дає змогу звертатися до елементів кортежу як до полів окремого запису таблиці.

Важливо підкреслити, що повернений у програму кортеж зберігає свої властивості.

19.3.11. Приклад функції з одним вхідним параметром повертає кортеж значень таблиці

Вивести відомості про студента з заданим номером студентського квитка.

Початковий текст функції getStdRow():

```
CREATE OR REPLACE FUNCTION "getStdRow"( st_nmb integer)
 RETURNS text AS
$$
DECLARE
 std_row  "Students"%ROWTYPE;
BEGIN
    -- Запит повертає кортеж
 SELECT * INTO std_row FROM public."Students"
                WHERE ("Students"."StdNmb" = st_nmb);
       -- Повертає весь кортеж
       -- RETURN std_row;
       -- Повертається результат склеювання вибраних елементів кортежу
    RETURN std_row."StdNmb"||std_row."StdName";
END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

```
SELECT * FROM public."getStdRow"(981621);
```

Далі засобами мови програмування слід підготувати 1 вхідний параметр, надіслати запит на виконання, а потім виділити рядки, що повертаються, та поля оператора SELECT.

У випадку використання тексту **RETURN** std_row; ми отримаємо кортеж усіх елементів рядка у вигляді наведеному на рис. 19.1.

Рис. 19.1. Результат повернення всього кортежу

У випадку використання рядка коду **RETURN** std_row."StdNmb" || std_row."StdName"; ми отримаємо результат склеювання двох обраних елементів кортежу у вигляді наведеному на рис. 19.2.

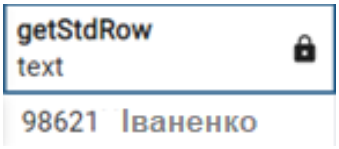


Рис. 19.2. Результат склеювання вибраних елементів

Тут немає жодних роздільників між елементами кортежу.

19.3.12. Змінні типу record

Змінні типу record схожі на змінні-кортежі, але вони не мають попередньо визначеної структури. Вони перебирають фактичну структуру від рядка, який їм присвоюється командами SELECT або FOR. Структура змінної типу record може змінюватись щоразу при присвоюванні чергового значення. Наслідком цього є те, що поки значення не призначено вперше, змінна типу record не має структури, і будь-яка спроба отримати доступ до її окремих полів призведе до помилки під час виконання.

Варто звернути увагу, що тип значення функції, що повертається в RETURN record не є тим же самим, що і змінна типу record, наприклад, std_row record, хоча така функція може використовувати змінну типу record для зберігання свого результату. У обох випадках фактична структура рядка record невідома під час створення функції.

Коли змінна типу record отримує значення, вона одразу набуває й структуру. Тепер можна звертатися до її окремих елементів, наприклад std_row.StdName. При цьому змінна типу record може змінювати свою структуру на льоту в результаті кожного наступного заповнення.

Коли функція повертає результат типу record, то структура не передається, а передається звичайний запис. Фактична структура запису визначається під час розбору запиту, який викликає її.

19.3.13. Приклад функції з одним вхідним параметром, яка повертає кортеж значень таблиці

Вивести інформацію про студента з заданим номером студентського квитка.

Початковий текст функції getStdRow():

```
CREATE OR REPLACE FUNCTION "getStdRow_2"( st_nmb integer)
 RETURNS text AS
$$
DECLARE
 std_row  record;
BEGIN
    -- Заповнюється запис std_row.
 SELECT * INTO std_row FROM public."Students"
            WHERE ("Students"."StdNmb" = st_nmb)
      -- Повертається весь запис std_row
      -- RETURN std_row;
      -- Повертається запис із вибраних елементів таблиці
      RETURN std_row."StdNmb"||std_row."StdName";
END;
$$
LANGUAGE 'plpgsql';
```

Виклик цієї функції у прикладній програмі має наступний вигляд:

```
SELECT * FROM public."getStdRow_2"(98621);
```

Наступним кроком засобами мови програмування слід підготувати один вхідний параметр, відправити запит на виконання, а потім виділити рядки, що повертаються, та поля оператора SELECT.

У випадку використання коду RETURN std_row; ми отримаємо запис з усіх елементів рядка таблиці вигляду приведеного на рис. 19.3.

Рис. 19.3. Повернення всього запису

Цей запис програма, що його викликає, повинна самостійно розібрати.

У випадку використання коду RETURN std_row."StdNmb" || std_row."StdName"; ми отримаємо результат об'єднання двох обраних елементів запису такого вигляду, як на рис. 19.4.

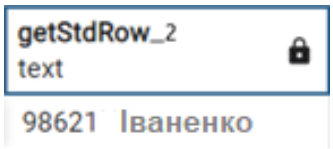


Рис. 19.4. Повернення запису із вибраних елементів таблиці

В другому випадку немає жодних роздільників між елементами.

19.4. Видалення збереженої функції. DROP FUNCTION

Видалення збереженої функції виконується командою

DROP FUNCTION [IF EXISTS] Назва **[([[** Режим_аргументу **]**
[Назва_аргументу**]** Тип_аргументу **[, ...]])] [, ...]**
[CASCADE | RESTRICT]

Користувач, який виконує цю команду, має бути власником функції. Окрім назви функції обов'язково потрібно вказати типи її аргументів, оскільки в БД можуть існувати декілька функцій з однією назвою, але з різними списками аргументів.

Примітка. Під час відлагодження функції програміст може неодноразово змінювати її текст, але, якщо він бажає змінити аргументи функції, обов'язково необхідно видалити функцію і знову її створити.

Інші суттєві параметри команди:

1. IF EXISTS – не вважати помилкою, якщо функція не існує. У цьому випадку буде видано попередження.
2. Режим_аргументу: IN, OUT, INOUT або VARIADIC. За замовчуванням мається на увазі IN. Для ідентифікації функції потрібні лише типи вхідних аргументів. Тому достатньо перерахувати лише аргументи IN, INOUT і VARIADIC.
3. Назва_аргументу – DROP FUNCTION не звертає уваги на назви аргументів, оскільки для однозначної ідентифікації функції достатньо лише типів аргументів.
4. Тип_аргументу Тип даних аргументів функції, якщо такі є.
5. CASCADE – автоматично видаляти об'єкти, залежні від цієї функції (наприклад, оператори або тригери), і, в свою чергу, всі об'єкти, що залежать від них.

6. RESTRICT – відмовити у видаленні функції, якщо від неї залежать будь-які об'єкти. Це поведінка за замовчуванням.

Розглянемо деякі приклади.

Команда видаляє функцію без параметрів:

DROP FUNCTION "getSumStipAll"()

Команда видаляє функцію з двома вхідними параметрами:

DROP FUNCTION "insRegion_1"(integer, character varying)

Команда видаляє функцію з 1-м вхідним параметром і 2-ма вихідними параметрами, які в цьому випадку не мають значення:

DROP FUNCTION "getCntAndAvgBallGrp"(**IN** character varying)

Видалення декількох функцій однією командою:

DROP FUNCTION "getStdRow"(integer), "getStdRow_2"(integer)

19.5. Контрольні запитання

1. Перелічіть основні директиви команди створення функції мови PL/pgSQL.
2. Як виконується виклик функції мови PL/pgSQL в прикладній програмі?
3. Чи може виклик функції мови PL/pgSQL замінити у програмі безпосереднє звернення до БД з SQL запитом?
4. Що таке вхідні і вихідні параметри функції?
5. Як у функцію передаються вхідні параметри?
6. Як функція повертає результати. Призначення директиви RETURNS?
7. Як функція повертає одне значення?
8. Як у функцію передаються вихідні параметри?
9. Як функція повертає значення декількох вихідних параметрів?
10. Як функція повертає множину тимчасових записів таблиці?
11. Як функція повертає множину записів таблиці певного типу, що міститься в БД?
12. Як вказати, що функція не повертає жодного значення?
13. Перелічіть основні директиви команди видалення функції мови PL/pgSQL.

19.6. Завдання для самопідготовки

Завдання 19.1. Створіть функцію без параметрів, яка повертає одне значення.

Завдання 19.2. Створіть функцію з кількома вхідними параметрами, яка повертає одне значення.

Завдання 19.3. Створіть функцію з кількома вихідними параметрами і, відповідно, яка повертає їх значення.

Завдання 19.4. Створіть функцію, яка повертає декілька записів таблиці, отриманої в результаті запиту до БД.

Завдання 19.5. Створіть функцію, яка повертає множину записів таблиці певного типу, що знаходиться в БД.

Завдання 19.6. Створіть функцію, яка не повертає значень і додає запис до таблиці БД.

Завдання 19.7. Створіть функцію, яка не повертає значень і видаляє запис з таблиці БД.

Завдання 19.8. Створіть команду видалення функції з двома вхідними параметрами.

20. ТРИГЕРИ

20.1. Поняття тригера

Тригер (Trigger – спусковий механізм, курок, защіпка) представляє собою збережену процедуру особливого типу, яка викликається автоматично (натискання на спусковий механізм) при модифікації записів БД, тобто при зміні, видаленні або додаванні записів в БД.

На відміну від збережених функцій, тригери не можна викликати з програми клієнта, а також передавати їм параметри та отримувати від них результати.

Тригер – іменований об'єкт БД, який асоціюється з таблицею та активується при настанні певної події, пов'язаної з цією таблицею. Тригер запускається сервером БД автоматично при спробі змінити дані в таблиці, з якою він пов'язаний. Усі здійснені ним модифікації даних розглядаються як виконані в транзакції, в якій виконано дію, що спричинило спрацювання тригера. Відповідно, у випадку виявлення помилки або порушення цілісності даних може відбутися відкат цієї транзакції.

Запуск тригера обумовлений настанням певної події (дії) – по суті, додавання INSERT або видалення DELETE рядка в заданій таблиці, а також при модифікації UPDATE даних в певному стовпці заданої таблиці БД.

Момент запуску тригера визначається за допомогою ключових слів BEFORE (перед), коли тригер запускається до виконання пов'язаної з ним події, наприклад, до додавання запису, або AFTER (після), коли тригер запускається після виконання пов'язаної з ним події.

У випадку, якщо тригер викликається до настання події, він може внести зміни до запису, що модифікується подією, звичайно, за умови, що подія – не видалення запису.

Тригери зазвичай використовуються для підтримки цілісності посилань, каскадної зміни або видалення даних, архівування видалених або змінених даних, відстеження змін або виклику користувацьких або системних збережених процедур та програмної реалізації, так званих бізнес-правил.

20.2. Програмування тригера в PostgreSQL

Створення тригера схоже для багатьох СУБД. Покажемо це на прикладі СУБД PostgreSQL. Детальну інформація про програмування тригерів в PostgreSQL наведено на сторінці [10]. У PostgreSQL для створення тригера

спочатку задається тригерна функція, а вже потім сам тригер, який запускає цю функцію.

Тригерна функція відрізняється від збереженої функції тим, що її результат має тип trigger і у неї немає параметрів. Проте є спеціальні можливості сформувати параметри в тригері та передати їх у функцію.

Тригерна функція повинна повернути або NULL, або запис/рядок, що відповідає структурі таблиці, для якої спрацював тригер.

Нижче наведені приклади тригерних функцій.

Синтаксис визначення власне тригера в PostgreSQL наступний:

CREATE TRIGGER Назва_тригера Час_ініціювання_тригера
Подія_тригера **ON** Назва_таблиці
Рівень_тригера
Умова
EXECUTE ROCEDURE Заголовок_тригерної_функції;

Строгий синтаксис визначення тригера можна отримати за посиланням [10].

Назва_тригера: назва тригера повинна бути унікальною серед назв тригерів схеми БД.

Час_ініціювання_тригера: при визначенні тригера надається можливість вказати момент ініціювання його виконання, а саме – чи повинен тригер бути ініційований перед виконанням події – фраза BEFORE, чи після – фраза AFTER.

Подія_тригера: у ролі подій тригера можуть виступати лише зміни станів БД. Вони, звісно ж, пов'язані з командами вставки, зміни та видалення рядків таблиці.

Назва_таблиці: назва таблиці БД, при чому лише однієї.

Рівень_тригера: розрізняють тригери рівня рядків та рівня тверджень (операторів). Рівень тригера визначається однією з наступних фраз: FOR EACH ROW або FOR EACH STATEMENT. Тим самим можна вказати, скільки разів тригер буде виконуватися щодо події, що ініціює його. Можливі 2 варіанти виконання тригера:

1. Один раз щодо кожного рядка, що обробляється, який буде підданий впливу події тригера – фраза FOR EACH ROW.
2. Один раз щодо події незалежно від того, скільки рядків при цьому оброблюються – фраза FOR EACH STATEMENT.

Заголовок_тригерної_функції: Вказується назва тригерної функції, яка буде виконуватися при ініціюванні тригера.

Спеціальні змінні. Коли спрацьовує тригерна функція, автоматично створюються декілька спеціальних змінних:

1. TG_NAME – назва тригера, що спрацював;
2. TG_OP – операція, для якої спрацював тригер: INSERT, UPDATE, DELETE або TRUNCATE;
3. TG_WHEN – час ініціювання тригера: BEFORE, AFTER або INSTEAD OF залежно від визначення тригера.
4. TG_TABLE_NAME – таблиця, для якої спрацював тригер.
5. NEW типу RECORD – це новий рядок таблиці для команд INSERT/UPDATE в тригерах рівня рядків. У тригерах рівня операторів і для команди DELETE ця змінна має значення NULL.
6. OLD типу RECORD – це старий рядок таблиці для команд UPDATE/DELETE в тригерах рівня рядків. У тригерах рівня оператора і для команди INSERT ця змінна має значення NULL.
7. TG_NARGS – кількість аргументів у команді CREATE TRIGGER, які передаються в тригерну функцію.
8. TG_ARGV[] – масив типу text, що містить аргументи від оператора CREATE TRIGGER. Індекс масиву починається з 0.

Варто звернути особливу увагу на змінні NEW і OLD. Вони використовуються в тригерах рівня рядків FOR EACH ROW, які на практиці складають більшість тригерів.

Оскільки змінна NEW відповідає доданому або зміненому рядку, то, зокрема, можна отримати доступ до необхідного стовпця такого запису, наприклад, для таблиці Students це буде NEW.StdID. Звісно, тригери DELETE не можуть посилатися на NEW.

Аналогічно змінна OLD відповідає видаленому або ще не зміненому рядку, і, зокрема, можна отримати доступ до потрібного поля такого запису, наприклад для таблиці Students це буде OLD.StdID. Звісно, тригери INSERT не можуть посилатися на OLD.

Розглянемо приклад тригера рівня рядків, пов'язаного з оновленням рядка таблиці БД.

Приклад. Під час виклику тригера визначити, яка операція викликала тригер і повідомити про це.

Спочатку необхідно написати тригерну функцію, яка визначатиме, яка операція викликала тригер, а потім вже створити та викликати тригер.

Повний код тригерної функції operatingStudents():

CREATE OR REPLACE FUNCTION "operatingStudents"()

```
  RETURNS trigger AS
$$
BEGIN
    IF TG_OP = 'DELETE' THEN
        RAISE NOTICE 'Видалити студента с ID: %', OLD.StdID;
    END IF
    IF TG_OP = 'INSERTE' THEN
        RAISE NOTICE 'Вставити студента с ID: %', NEW.StdID;
    END IF
    RETURN NEW;
EXEPTION
    WHEN OTHERS THEN
        RAISE NOTICE 'Помилка в тригері operatingStudents: %', SQLERRM;
    RETURN NULL;
END;
$$
LANGUAGE 'plpgsql' VOLATILE
COST 100;
CREATE TRIGGER modifyStudents
    AFTER INSERT OR DELETE ON "Students"
FOR EACH ROW EXECUTE PROCEDURE "operatingStudents"();
```

Текст тригеру, що викликає тригерну функцію:

```
CREATE TRIGGER modifyStudents
    AFTER INSERT OR DELETE ON "Students"
FOR EACH ROW EXECUTE PROCEDURE "operatingStudents"();
```

Приклад. Під час виклику тригера визначити кількість рядків у таблиці Students, а також парна вона чи непарна.

Спочатку необхідно написати тригерну функцію, яка визначить кількість рядків у таблиці Students, а потім вже створити та викликати тригер.

Повний код тригерної функції operatingStudents():

```
CREATE OR REPLACE FUNCTION "countStudents"()
  RETURNS trigger AS
$$
DECLARE
    record_row  RECORD;
    cnt  INTEGER
```

```
BEGIN
        cnt  := 0;
        FOR record_row  IN SELECT * FROM "Students"
                LOOP
                        cnt  := cnt + 1;
                END LOOP;
        RAISE NOTICE 'Кількість записів у таблиці Students: %', cnt;
        IF cnt % 2 = 0  THEN
                RAISE NOTICE 'Парна кількість записів;
        ELSE
                RAISE NOTICE 'Непарна кількість записів;
        END IF;
        RETURN NEW;
END;
$$
LANGUAGE 'plpgsql' VOLATILE
COST 100;
```

Текст тригера cntStudents:

```
CREATE TRIGGER cntStudents
 AFTER INSERT OR DELETE ON "Students"
        FOR EACH ROW EXECUTE PROCEDURE "countStudents"();
```

Розглянемо приклад тригера рівня рядків, пов'язаного з видаленням певних рядків таблиці БД.

Приклад. Під час виклику тригера видалити з таблиці Students відомості про студентів групи КС-983.

Спочатку необхідно написати тригерну функцію, яка видалить з таблиці Students відомості про студентів групи КС-983, а потім вже створити та викликати тригер.

Повний код тригерної функції deleteStudents():

```
CREATE OR REPLACE FUNCTION "deleteStudents"()
  RETURNS trigger AS
$$
BEGIN
 IF TG_OP = 'DELETE' THEN
        IF OLD."StdGrp" = (SELECT "GrpID" FROM "Groups"
                                WHERE "GrpName" = 'КС-983')
```

```
            THEN DELETE FROM "Students" WHERE "StdGrp" =
OLD."StdGrp";
            END IF;
            RETURN OLD;
        END IF;
        END;
        $$
        LANGUAGE 'plpgsql' VOLATILE
        COST 100;
```

Текст тригера dltStudents:

```
        CREATE TRIGGER deltStudents
         AFTER INSERT OR DELETE ON "Students"
        FOR EACH ROW EXECUTE PROCEDURE "deleteStudents"();
```

Розглянемо приклад тригера рівня рядків, пов'язаного з додаванням певних рядків у таблицю БД.

Приклад. При виклику тригера додати в таблицю Students відомості про студентів групи СП-981.

Спочатку необхідно написати тригерну функцію, яка додати в таблицю Students відомості про студентів групи СП-981, а потім вже створити та викликати тригер.

Повний код тригерної функції insertStudents():

```
        CREATE OR REPLACE FUNCTION "insertStudents"()
          RETURNS trigger AS
        $$
        BEGIN
            -- Під час роботи тригера використовується
            -- спеціальна змінна NEW
        IF TG_OP = 'INSERNT' OR TG_OP = 'UPDATE' THEN
            IF NEW."StdGrp" = (SELECT GrpID FROM "Groups"
                                    WHERE "GrpName" = 'СП-981')
            THEN INSERT INTO "Students"("StdNmb", "StdName")
                                VALUES (NEW."StdNmb", NEW."StdName");
            -- занесення вмісту всіх полів змінної NEW
                    -- THEN INSERT INTO "Students"  VALUES (NEW.*);
            END IF;
            RETURN NEW;
        END IF;
```

```
END;
$$
LANGUAGE 'plpgsql' VOLATILE
COST 100;
```

Текст тригера insertStudents:

```
CREATE TRIGGER insertStudents
 AFTER INSERT OR DELETE ON "Students"
FOR EACH ROW EXECUTE PROCEDURE "insertStudents"();
```

Розглянемо приклад тригера на рівні рядків, пов'язаного з оновленням поля у всіх рядках таблиці БД.

Приклад. При додаванні запису до таблиці Marks БД "Деканат ВНЗ" необхідно автоматично перераховувати середній бал студента (колонка SrBall в таблиці Students).

Спочатку треба написати тригерну функцію updateAvgBall, яка буде підраховувати середній бал студента за таблицею Marks і оновлювати відповідний запис у таблиці Students, а потім вже створити та викликати тригер.

Повний код тригерної функції updateAvgBall():

```
CREATE OR REPLACE FUNCTION "updateStAvgBall"()
  RETURNS trigger AS
$$
BEGIN
      -- Під час роботи тригера використовується
      -- спеціальна змінна NEW
 UPDATE "Students" SET "StAvgBall" = (SELECT AVG("Mark")
                                      FROM "Marks"
                                      WHERE "MStd" = NEW."MStd")
 WHERE "StdID" = NEW."MStd";
 RETURN NEW;
END;
$$
LANGUAGE 'plpgsql' VOLATILE
COST 100;
```

Текст тригера changeStAvgBall:

```
CREATE TRIGGER changeStAvgBall
 AFTER INSERT ON "Marks"
      FOR EACH ROW EXECUTE PROCEDURE "updateStAvgBall"();
```

20.3. Видалення тригера

Видалення визначення тригера виконується командою

DROP TRIGGER [IF EXISTS] назва_тригера **ON** назва_таблиці
[CASCADE | RESTRICT]

Користувач, який виконує цю команду, повинен бути власником таблиці, для якої визначений цей тригер.

Призначення параметрів назва_тригера та назва_таблиці очевидно.

Інші параметри:

IF EXISTS – не вважати помилкою, якщо тригер не існує;

CASCADE – автоматично видаляти об'єкти, залежні від цього тригера, а також всі об'єкти, що залежать від них;

RESTRICT – відмовити в видаленні тригера, якщо від нього залежать будь-які об'єкти. Це – поведінка за замовчуванням.

Приклад. Видалити тригер для таблиці Students.

DROP TRIGGER insertStudents **ON** "Students"

20.4. Контрольні запитання

1. Перелічіть основні елементи директиви створення тригера та їхнє призначення.
2. Що таке час ініціювання тригера? Як він встановлюється?
3. Що таке подія тригера? Перелічіть основні події.
4. Що таке рівень тригера? Чим рівні тригера відрізняються один від одного?
5. Що таке тригерна функція і чим вона відрізняється від збереженої функції?
6. Який тип значень повертає тригерна функція?
7. Що таке спеціальні змінні тригерної функції?
8. Для чого призначено і як використовується змінна NEW?
9. Для чого призначено і як використовується змінна OLD?
10. Перелічіть основні елементи директиви видалення тригера та їхнє призначення?

20.5. Завдання для самопідготовки

Завдання 20.1. Створіть тригер рівня рядків, пов'язаний з оновленням рядка таблиці БД.

Завдання 20.2. Створіть тригер рівня рядків, пов'язаний з видаленням певних рядків таблиці БД.

Завдання 20.3. Створіть тригер рівня рядків, пов'язаний з додаванням певних рядків до таблиці БД.

Завдання 20.4. Створіть тригер рівня рядків, пов'язаний з оновленням певного стовпця таблиці БД.

21. ІНДЕКСУВАННЯ ТАБЛИЦЬ БАЗ ДАНИХ В POSTGRESQL

21.1. Індекси

Індекси є ефективним засобом збільшення продуктивності БД. Використовуючи індекс, сервер БД може знаходити і вибирати потрібні рядки таблиці набагато швидше, ніж без нього. Однак з індексами пов'язана додаткове навантаження на СУБД в цілому, тому застосовувати їх слід зважено.

Індекси обов'язково створюються для всіх первинних ключів і, якщо є, то й альтернативних ключів.

Зазвичай для підвищення швидкості виконання критично важливих запитів створюються індекси по неключових полях таблиці.

На практиці вважається, що допустимо не більше 3-х індексів за неключовими полями таблиці.

Індекси, які використовуються в запитах зрідка або взагалі ніколи не використовуються, повинні бути видалені.

21.2. Створення та видалення індексів

21.2.1. Створення індексу

Створити індекс для стовпчика таблиці можна за допомогою наступної команди:

CREATE INDEX Назва_індексу **ON** Назва_таблиці (Назва_стовпчика)
CREATE INDEX Назва_індексу **ON** Назва_таблиці (Назва_стовпчика)

USING *Метод*

Команда CREATE INDEX створює індекси за вказаним стовпцем (стовпцями) заданого відношення, яким може бути таблиця або матеріалізоване представлення.

Назва індексу може бути довільною, головне, щоб вона дозволяла зрозуміти, для чого цей індекс.

Повний синтаксис команди:

CREATE [UNIQUE] INDEX [CONCURRENTLY]
[[IF NOT EXISTS] Назва**] ON** Назва_таблиці **[USING** Метод **]**
({ Назва_стовпчика **| (Вираз) } [COLLATE** Правило_сортування **]**
[Клас_операторів **] [ASC | DESC] [NULLS { FIRST | LAST }] [, ...])**
[WITH (Параметр_зберігання **[=** Значення**] [, ...])]**

[TABLESPACE Табл_простір **]**
[WHERE Предикат **]**

Команда CREATE INDEX є достатньо складною і її параметри та виконання пов'язане з особливостями організації і функціонування бази даних. Тому нижче будуть розглянуті найбільш загальні параметри команди, а за поясненнями інших параметрів треба звернутися до документації.

Параметри:

1. UNIQUE. Вказує, що система повинна контролювати повторювані значення в таблиці при створенні індексу (якщо в таблиці вже є дані) і при кожному додаванні даних. Спроби вставити або змінити дані, за наявності яких буде порушено унікальність індексу, завершуватимуться помилкою.
2. CONCURRENTLY. З цим зазначенням PostgreSQL побудує індекс, не встановлюючи жодних блокувань, які б запобігали додаванню, зміні або видаленню записів в таблиці, тоді як за замовчуванням операція побудови індексу блокує запис (але не читання) даних в таблиці до свого завершення. Зі створенням індексу в цьому режимі пов'язаний ряд особливостей (див. документацію).
3. Для тимчасових таблиць CREATE INDEX завжди виконується більш простим, неблокуючим способом, оскільки вони не можуть використовуватися жодними іншими сеансами.
4. IF NOT EXISTS. Не вважати помилкою, якщо індекс з такою назвою вже існує. У цьому випадку буде видано повідомлення.
5. Назва. Назва створюваного індексу. Вказівка на схему при цьому не допускається; індекс завжди відноситься до тієї ж схеми, що і батьківська таблиця. Якщо назва відсутня, PostgreSQL формує відповідну назву з назви батьківської таблиці та назв індексованих стовпців.
6. Назва_таблиці. Назва таблиці, що індексується (можливо, доповнена схемою).
7. Метод. Назва методу індексу. Можливі варіанти: btree, hash, gist, spgist, gin і brin. За замовчуванням мається на увазі метод btree.
8. Назва_стовпчика. Назва стовпчика таблиці.
9. Вираз. Вираз з одним або декількома стовпцями таблиці. Зазвичай вираз має записуватися в дужках, однак дужки можна не вказувати, якщо вираз має вигляд виклику функції.

10. Правило_сортування. Назва правила сортування, що застосовується для індексу. За замовчуванням використовується правило сортування, задане для індексованого стовпчика, або отримане для результату вирази індексу. Індекси з нестандартними правилами сортування можуть бути корисні для запитів, що включають вирази з такими правилами.
11. Клас_операторів. Назва класу операторів (див. документацію):
    11.1. ASC. Вказує порядок сортування за зростанням (за замовчуванням).
    11.2. DESC. Вказує порядок сортування як спадаючій.
    11.3. NULLS FIRST. Вказує, що значення NULL після сортування розташовуються перед іншими. Це поведінка за замовчуванням з порядком сортування DESC.
    11.4. NULLS LAST. Вказує, що значення NULL після сортування розташовуються після інших. Це поведінка за замовчуванням з порядком сортування ASC.
12. Параметр_зберігання. Назва специфічного для індексу параметра зберігання (див. документацію).
13. Табл_простір. Табличний простір, в якому буде створено індекс. Якщо не визначено, вибирається default_tablespace, або temp_tablespaces, при створенні індексу тимчасової таблиці.
14. Предикат. Вираз обмеження для часткового індексу.

21.2.2. Видалення індексу

Для видалення індексу використовується команда

DROP INDEX Назва

Команда DROP INDEX є достатньо складною і її параметри та виконання пов'язані з особливостями організації і функціонування бази даних. Тому нижче будуть розглянуті найбільш загальні параметри команди, а за поясненнями інших параметрів треба звернутися до документації.

Повний синтаксис команди

DROP INDEX [CONCURRENTLY] [IF EXISTS] *назва* **[, ...] [CASCADE | RESTRICT]**

Параметри:

1. CONCURRENTLY. З цим зазначенням індекс видаляється, не блокуючи одночасні операції вибірки, додавання, зміни і видалення даних в таблиці індексу. Звичайний оператор DROP INDEX запитує виняткове блокування для таблиці, не допускаючи інші звернення до неї до завершення видалення. Якщо ж додано цю вказівку, команда, навпаки, буде чекати завершення конфліктуючих транзакцій.

Застосовуючи вказівку CONCURRENTLY, треба враховувати декілька особливостей. Зокрема, при цьому можна задати назву тільки одного індексу, а параметр CASCADE не підтримується. (Таким чином, індекс, що підтримує обмеження UNIQUE або PRIMARY KEY, таким чином видалити не можна). Крім того, звичайну команду DROP INDEX можна виконати в блоці транзакції, а DROP INDEX CONCURRENTLY – ні.

Для тимчасових таблиць DROP INDEX завжди виконується більш простим, неблокуючим способом, оскільки вони не можуть використовуватися жодними іншими сеансами.

2. IF EXISTS. Не вважати помилкою, якщо індекс не існує. У цьому випадку буде видано повідомлення.
3. Назва. Назва індексу, можливо доповнена схемою, що підлягає видаленню.
4. CASCADE. Автоматично видаляти об'єкти, залежні від даного індексу, і, в свою чергу, всі залежні від них об'єкти.
5. RESTRICT. Відмовити у видаленні індексу, якщо від нього залежать будь-які об'єкти. Це – поведінка за замовчуванням.

21.2.3. Поведінка та використання індексів

Додавати і видаляти індекси можна в будь-який час існування БД.

Коли індекс створено, жодні додаткові дії не потрібні. PostgreSQL буде сам оновлювати індекс при зміні даних в таблиці і буде сам використовувати цей індекс в запитах, де, на його думку, це буде ефективніше, ніж сканування всієї таблиці.

Для ефективного використання індексів в запитах, необхідно періодично запускати команду ANALYZE для поновлення статистичних даних, на основі яких планувальник запитів приймає рішення.

Індекси можуть застосовуватися в пошуку зі з’єднанням. Таким чином, індекс, визначений для стовпчика, який бере участь в умові з'єднання, може значно прискорити запити з наявними виразами JOIN.

Індекси можуть бути корисні також при виконанні команд UPDATE і DELETE з умовами пошуку.

21.3. Типи індексів

PostgreSQL підтримує такі типи індексів – B-дерево, хеш, GiST, SP-GiST, GIN і BRIN. Для різних типів індексів використовуються різні алгоритми впорядкування, орієнтовані на певні типи запитів.

За замовчуванням команда CREATE INDEX створює індекси типу B-дерево, що є ефективними в більшості випадків.

21.3.1. B-дерева

B-дерева можуть функціонувати в умовах пошуку на рівність: дорівнює, не дорівнює тощо, а також у перевірках діапазонів з даними типу "більше A і менше B" або аналогічних їм конструкціях BETWEEN і IN.

B-дерево є типом індексу за замовчуванням і не вимагає явного вказування. Створити такий індекс можна за допомогою такої команди

CREATE INDEX Назва_індексу **ON** Назва_таблиці(Назва_стовпця)

21.3.2. Хеш-індекси

Хеш-індекси (Hash Indexes) працюють тільки з простими умовами порівняння "дорівнює". Створити такий індекс можна за допомогою такої команди:

CREATE INDEX Назва_індексу **ON** Назва_таблиці
USING HASH (Назва_стовпця**)**

Однією з особливостей є його орієнтація на роботу з деяким типом структурованих даних. Так стандартний дистрибутив PostgreSQL надає можливість роботи з декількома типами двовимірних геометричних даних, що використовуються в різних областях картографії. Для можливості індексування таких типів сервер має декілька інструментів.

21.3.3. Індекси GiST, SP-GiST і GiN

Індекси GiST, SP-GiST і GiN являють собою інфраструктуру, що дозволяє реалізувати різні стратегії індексування в залежності від заданого набору (класу) операторів.

21.3.4. Індекси GiST

Узагальнене пошукове дерево – індекси GiST (Generalized Search Tree) це структура, яка є узагальненим різновидом R-дерева – RD-деревом, і надає стандартні методи навігації по дереву і його оновлення – розщеплення і видалення вузлів.

Примітка 1. R-дерево є подібнім до B-дерева, але використовується для організації доступу до просторових даних, тобто для індексації багатовимірної інформації, наприклад, географічних даних. Ця структура даних розбиває багатовимірний простір на множину ієрархічно вкладених і, можливо, тих, що перетинаються, прямокутників, прямокутних паралелепіпедів тощо.

Примітка 2. RD-дерево є адаптацією R-дерева, що використовує природну аналогію між просторовими об'єктами та множинами.

Індекс GiST є збалансованим за висотою RD-деревом, кінцеві вузли (листя) якого містять пари (key, rid), де key – ключ, a rid – покажчик на відповідний запис на сторінці даних. Внутрішні вузли дерева містять пари елементів, що виконують роль навігаторів з доступом до необхідного листу дерева.

Створити GiST індекс можна командою:

CREATE INDEX Назва_індексу **ON** Назва_таблиці
USING GiST (Назва_стовпця**)**

Для GiST-дерева визначено базові методи Search, Insert, Delete, і інтерфейс для написання користувацьких методів, з допомогою яких можна управляти роботою цих (базових) методів.

Стандартний дистрибутив PostgreSQL включає класи операторів GiST для двовимірних типів геометричних даних.

Індекси GiST і GIN (див. нижче) широко використовуються при повнотекстовому пошуку PostgreSQL. Це означає, що для кожного вектору tsvector, що описує всі лексеми документа, створюється сигнатура, що описує, які з лексем входять в даний tsvector.

21.3.5. Індекси SP-GiST

Індексація з розбивкою простору SP-GiST (Space-Partitioned GiST), полягає в розбитті області значень на неперекриваючі підобласті, кожна з яких, в свою чергу, також може бути поділеною. Таке розбиття породжує незбалансовані дерева на відміну від B-дерев GiST.

Індекси SP-GiST зіставляють вузли дерева пошуку зі сторінками на диску так, що при пошуку потрібно звертатися тільки до декількох сторінок на диску, навіть якщо при цьому потрібно переглянути множину вузлів.

Створити SP-GiST індекс можна за допомогою такої команди:

CREATE INDEX Назва_індексу **ON** Назва_таблиці
USING SP-GiST (Назва стовпця)

Стандартний дистрибутив PostgreSQL включає класи операторів SP-GiST для точок в двовимірному просторі.

21.3.6. Індекси GIN

Індекси GIN (Generalized Inverted Index) використовують з типами даних, значення яких не є атомарними, а складаються з окремих елементів. При цьому індексуються не самі значення, а їх окремі елементи. Кожен елемент посилається на ті значення, в яких він зустрічається. Прикладом таких значень є масиви.

Інвертований індекс може ефективно функціонувати в запитах, які перевіряють присутність елемента в неатомарних значеннях полів таблиці. Ці індекси можуть підтримувати різні, визначені користувачем, стратегії і в залежності від них можуть застосовуватися з різними операторами.

Індекси GIN зберігаються у вигляді B-дерева. До кожного елементу цього дерева прив'язаний упорядкований набір посилань на рядки таблиці, що містять значення з цим елементом.

Створити GIN-індекс можна за допомогою такої команди:

CREATE INDEX Назва _індексу **ON** Назва_таблиці
USING GIN (Назва_стовпця)

Стандартний дистрибутив PostgreSQL включає клас операторів GIN для масивів як значень полів таблиці.

Як уже відмічалося, індекси GiST (див. вище) і GIN широко використовуються при повнотекстовому пошуку PostgreSQL. Це означає, що для кожного вектору tsvector, що описує всі лексеми документа, створюється сигнатура, що описує, які з лексем входять в даний tsvector.

Основна область застосування індексів GIN – це прискорення повнотекстового пошуку. Для цього в структурі індексів з кожної лексемою зіставляється відсортований список номерів документів, в яких вона зустрічається. Пошук у такій структурі набагато ефективніший, ніж при використанні GiST, однак процес додавання нового документа досить тривалий.

Ще одним прикладом складного типу даних, для якого є вбудовано GIN-підтримку, є значення в форматі JSON. Для роботи зі значеннями JSON в даний час визначено ряд операторів і функцій, частина з яких може бути прискорена за допомогою індексів.

21.3.7. Індекси BRIN

Індекси зон блоків BRIN (Block Range Indexes) з'явилися, починаючи з PostgreSQL 9.5, і зберігають узагальнені відомості про значення, що знаходяться в фізично послідовно розташованих блоках таблиці.

Індекси BRIN можуть підтримувати певні визначені користувачем стратегії, і в залежності від них застосовуватися з різними операторами. Для типів даних, що мають лінійний порядок сортування, записам в індексі відповідають мінімальні і максимальні значення даних в стовпчику для кожної зони блоків.

Індекси BRIN призначено для обробки великих таблиць, в яких значення індексованого стовпчика має деяку природну кореляцію з фізичним розташуванням рядка в таблиці, тобто значення в стовпчику мають лінійний порядок сортування. Наприклад, це характерно для зберігання збереження квитанцій на оплату комунальних послуг, які пишуться послідовно, а тому вже на фізичному рівні впорядковані за датою / номером, і в той же час такі таблиці з такими даними зазвичай розростаються до гігантських розмірів. Аналогічно це характерно для зберігання замовлень інтернет-магазину тощо.

Завдяки своїй природі індекс BRIN має дуже малий розмір. В результаті сканування індексу додає невеликі накладні витрати, тому індекси BRIN володіють такими якостями таблиць, як швидка вставка рядка і швидке створення індексу.

Створити індекс BRIN можна за допомогою такої команди:

CREATE INDEX Назва_індексу **ON** Назва_таблиці
USING BRIN (Назва_стовпця)

21.4. Дослідження результатів пошуку за індексованим стовпчиком

Для дослідження результатів пошуку за індексованим стовпчиком слід використовувати конструкцію типу

EXPLAIN ANALYZE SELECT * FROM deanery.Students
WHERE StdNmb = 981527;

21.5. Складені індекси

Якщо пошук виконується за декількома стовпчиками таблиці, то можна створити складений індекс за ними.

Створити індекс за кількома стовпчиками таблиці можна за допомогою наступної команди:

CREATE INDEX Назва_індексу **ON** Назва_таблиці (Назва_стовпця_1, Назва_стовпця_2, ...)

В даний час складеними можуть бути тільки індекси типів B-дерево, GiST, GIN і BRIN. Число стовпців в індексі обмежується 32.

21.6. Контрольні запитання

1. В чому полягає ідея використання індексів?
2. Які стовпчики таблиці індексуються обов’язково?
3. Перелічіть основні елементи команди на створення індексу та їх призначення.
4. Перелічіть основні елементи команди на видалення індексу та їх призначення.
5. Перелічіть основні типи індексів в PostgreSQL.
6. Опишіть ідею побудови та використання B-tree індексів.
7. Опишіть ідею побудови та використання Hash індексів.
8. Опишіть ідею побудови та використання GiST індексів.
9. Опишіть ідею побудови та використання SP-GiST індексів.
10. Опишіть ідею побудови та використання GIN індексів.
11. Опишіть ідею побудови та використання BRIN індексів.
12. Опишіть ідею складених індексів і області їх використання.

21.7. Завдання для самопідготовки

Згенерувати таблицю, яка вміщує 100000 рядків. Визначити стовпчик, за яким буде проведено дослідження на швидкість запиту на пошук, коли стовпчик не індексовано та коли стовпчик індексовано відповідним типом індексу.

Завдання 21.1. Виконайте пошук даних при обмеженні на значення не індексованого стовпчика.

Завдання 21.2. Проведіть Hash індексацію визначеного стовпчика та виконайте пошук даних при обмеженні на значення.

Завдання 21.3. Проведіть B-Tree індексацію визначеного стовпчика та виконайте пошук даних при обмеженні на значення.

Завдання 21.4. Проведіть GiST індексацію визначеного стовпчика та виконайте пошук даних при обмеженні на значення.

Завдання 21.5. Проведіть SP-GiST індексацію визначеного стовпчика та виконайте пошук даних при обмеженні на значення.

Завдання 21.6. Проведіть GIN індексацію визначеного стовпчика та виконайте пошук даних при обмеженні на значення.

Завдання 21.7. Проведіть BRIN індексацію визначеного стовпчика та виконайте пошук даних при обмеженні на значення.

Завдання 21.8. Порівняйте вплив індексів на швидкість пошуку за індексованим стовпчиком та поясніть розбіжності в швидкості пошуку.

22. РОЛІ. ПРИВІЛЕЇ. ДИРЕКТИВИ GRANT І REVOKE

22.1. Ролі

PostgreSQL використовує концепцію ролей (Roles) для управління доступом до БД. Роль можна розглядати як користувача або як групу користувачів БД, залежно від того, як роль налаштовано. Ролі можуть володіти об'єктами БД (наприклад, таблицями та функціями) і надавати іншим ролям дозволи доступу до цих об'єктів, управляючи тим, хто має доступ і до яких об'єктів. Крім того, можна надати одній ролі членство в іншій ролі, таким чином, одна роль може використовувати права інших ролей.

Примітка. Ролі користувачів операційної системи та користувачів БД жодним чином не пов'язано, хоча на практиці між ними може підтримуватися повна або часткова відповідність.

Для створення ролі використовується команда

CREATE ROLE Назва

Для видалення ролі використовується команда

DROP ROLE Назва

Отримати список існуючих ролей можна, звернувшись до pg_roles системного каталогу, наприклад:

SELECT rolname **FROM** pg_roles;

Отримати список існуючих ролей, які можуть підключатися до БД, можна за допомогою команди:

SELECT rolname **FROM** pg_roles **WHERE** rolcanlogin;

Для початкового налаштування БД система відразу після ініціалізації завжди містить одну певну роль із можливістю підключення до БД. Ця роль завжди є суперкористувачем (Superuser). Зазвичай ця роль називається postgres, але при встановленні БД можна визначити і іншу назву ролі.

Для створення інших ролей спочатку потрібно підключитися до БД в ролі суперкористувача.

Кожне підключення до сервера БД виконується під назвою конкретної ролі, і ця роль визначає початкові права доступу для команд, які виконуються в цьому з'єднанні.

22.2. Привілеї та права доступу

Відповідно до ідеології мови SQL контроль прав доступу даного користувача до таблиць БД і їх складових здійснюється на основі *механізму привілеїв*. Фактично, цей механізм полягає в тому, що для виконання будь-якої дії над таблицею користувач повинен мати відповідні права доступу. Фактично всі можливі дії описуються фіксованим стандартним набором привілеїв.

Користувач, який створив таблицю, автоматично стає власником усіх можливих привілеїв виконання операцій над цією таблицею. До цих привілеїв входить і привілей на передачу всіх або деяких привілеїв відносно даної таблиці іншому користувачеві, включаючи привілей на передачу привілеїв. Іноді підтримується і зворотна операція вилучення привілеїв від користувача, який раніше їх отримав.

У SQL механізм привілеїв реалізується командами GRANT (допуск) та REVOKE (скасування). У багатьох СУБД роздача привілеїв можлива тільки при визначенні таблиці. Користувач, який отримав деякі привілеї від інших користувачів, може передати їх лише при визначенні схеми.

22.3. Команда GRANT

Команда GRANT (допуск) надає права доступу певним ролям користувачів.

Синтаксис команди GRANT має декілька варіантів, залежно від типу об'єкта БД, до якого надається доступ – сама БД, таблиця, стовпець тощо, всі елементи БД.

Синтаксис команди GRANT для доступу до таблиці має вигляд

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
[, ...] | ALL [PRIVILEGES] }
ON { [TABLE] Назва_таблиці **[, ...] |**
ALL TABLES IN SCHEMA Назва_схеми **[, ...] }**
TO Назва_ролі **[, ...] [WITH GRANT OPTION]**
[GRANTED BY Назва_ролі **]**

Ця команда надає окремим операціям – SELECT, INSERT тощо доступ до таблиці, групи таблиць або всіх таблиць одній, декільком або всім ролям.

Синтаксис команди GRANT для доступу до стовпців таблиці має вигляд

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } (

Назва_стовпця **[, ...])**
[, ...] | ALL [PRIVILEGES] (Назва_стовпця **[, ...]) }**
ON [TABLE] Назва_таблиці **[, ...]**
TO Назва_ролі **[, ...] [WITH GRANT OPTION]**
[GRANTED BY Назва_ролі **]**

Ця команда надає окремим операціям – SELECT, INSERT тощо доступ до стовпця або групи стовпців таблиці одній, декільком або всім ролям.

Тут мають місце наступні два параметри:

1. WITH GRANT OPTION вказує, що отримувач права, у свою чергу, може передавати його іншим. Без цього вказання розпоряджатися своїм правом він не зможе.
2. GRANTED BY реалізований для сумісності зі стандартом SQL і на практиці не використовується.

Як вже зазначалося, деякі ролі можуть передавати свої права доступу іншим ролям. Синтаксис команди GRANT для передачі прав доступу іншим ролям

GRANT Назва_ролі **[, ...] TO** Назва_ролі **[, ...]**
[WITH { ADMIN | INHERIT | SET }{ OPTION | TRUE | FALSE }]
[GRANTED BY Назва_ролі **]**

Приклад. Надати користувачу Professor всі права на таблицю Students.

GRANT ALL PRIVILEGES ON "Students" **TO** "Professor";

Приклад. Дозволити всім ролям читати записи в таблиці Students.

GRANT SELECT ON "Students" **TO PUBLIC**;

Приклад. Дозволити користувачу Professor змінювати значення стовпця StNmb в таблиці Students.

GRANT UPDATE "StNmb" **ON** "Students" **TO** "Teacher";

Приклад. Передати права ролі Professor ролі Assistant.

GRANT "Professor" **TO** "Assistant";

22.4. Команда REVOKE

Команда REVOKE (скасування) відкликає права доступу певним ролям користувачів.

Синтаксис команди REVOKE, так само як і синтаксис команди GRANT, має декілька варіантів, залежно від типу об'єкта БД, доступ до якого відкликається – сама БД, таблиця, стовпець тощо.

Синтаксис команди REVOKE для відклику доступу до таблиці має вигляд

REVOKE [GRANT OPTION FOR]
{ { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
[, ...] | ALL [PRIVILEGES] }
ON { [TABLE] Назва_таблиці **[, ...] |**
ALL TABLES IN SCHEMA назва_схеми **[, ...] }**
FROM Назва_ролі **[, ...]**
[GRANTED BY Назва_ролі **]**
[CASCADE | RESTRICT]

Ця команда відкликає доступ окремим операціям – SELECT, INSERT тощо доступ до таблиці, групи таблиць або всіх таблиць однієї, кількох або всіх ролей.

Синтаксис команди REVOKE для відклику доступу до стовпців таблиці має вигляд

REVOKE [GRANT OPTION FOR]
{ { SELECT | INSERT | UPDATE | REFERENCES }
(Назва_стовпця **[, ...]) [, ...] | ALL [PRIVILEGES]**
(Назва_стовпця **[, ...]) }**
ON [TABLE] Назва_таблиці **[, ...]**
FROM Назва_ролі **[, ...]**
[GRANTED BY Назва_ролі **]**
[CASCADE | RESTRICT]

Синтаксис команди REVOKE для відклику у окремих ролей наданих їм раніше прав доступу іншими ролями.

REVOKE [{ ADMIN | INHERIT | SET } OPTION FOR]
Назва_ролі **[, ...] FROM** Назва_ролі **[, ...]**
[GRANTED BY Назва_ролі **]**
[CASCADE | RESTRICT]

Тут параметр CASCADE або RESTRICT визначає, як впливає відкликання вказаних ролей на інші ролі в БД.

Якщо задано параметр CASCADE, то видалення прав доступу виконується за всім ланцюжком успадкування цих прав.

Якщо задано параметр RESTRICT, і в БД є ролі, які успадкували свої права доступу від ролей, що вилучаються, то виконання оператора REVOKE закінчиться неуспішно. У більшості комерційних СУБД допускається застосування оператора DROP TABLE без будь-яких параметрів.

Приклад. Позбавити користувача Teacher всіх прав на таблицю Students.

REVOKE ALL PRIVILEGES ON "Students" **TO** "Teacher";

Приклад. Позбавити всіх користувачів права додавати запису в таблицю Students.

REVOKE INSERT ON "Students" **TO PUBLIC**;

Приклад. Позбавити користувача Teacher права змінювати значення стовпця StNmb в таблиці Students.

REVOKE UPDATE "StNmb" **ON** "Students" **TO** "Teacher";

Приклад. Відкликати права ролі Professor у ролі Assistant.

REVOKE "Professor" **FROM** "Assistant";

22.5. Контрольні запитання

1. Що таке роль користувача БД?
2. Як створити і видалити роль?
3. Як отримати список існуючих ролей, які можуть підключатися до БД?
4. Що таке привілеї і якими командами вони управляються?
5. Перелічіть основні елементи команди надання прав доступу певним ролям користувачів.
6. Перелічіть основні елементи команди відклику прав доступу у певної ролі користувачів.
7. Як передати права доступу від окремих ролей іншим ролям?
8. Як відкликати права доступу у окремих ролях, передані їм іншими ролями?

4. Що таке привілеї та якими командами вони керуються?

22.6. Завдання для самопідготовки

Завдання 22.1. Надайте певній ролі права вставки нового запису у таблицю.

Завдання 22.2. Відкликайте у певній ролі права вставки нового запису у таблицю.

Завдання 22.3. Передайте права доступу від певної ролі Role_1 до іншої ролі Role_2.

Завдання 22.4. Відкликайте у ролі Role_2 права доступу, надані їй від ролі Role_1.

23. СИСТЕМНИЙ КАТАЛОГ АБО СЛОВНИК ДАНИХ

23.1. Поняття системного каталогу/словника даних

Системним каталогом або словником даних називається сукупність спеціальних таблиць БД. Їх створює, супроводжує та володіє ними сама СУБД. Ці системні таблиці містять інформацію, яка описує структуру БД.

Таблиці системного каталогу створюються автоматично при створенні БД. Зазвичай вони об'єднуються під спеціальним системним ідентифікатором користувача з такими назвами, як SYSTEM, MASTER або DBA.

При обробці операторів SQL СУБД постійно звертається до даних системного каталогу. Наприклад, щоб обробити двотабличний оператор SELECT, СУБД повинна:

1. Перевірити, чи існують дві вказані таблиці.
2. Переконатися, що користувач має дозвіл на доступ до цих таблиць.
3. Перевірити, чи існують стовпці, на які є посилання в даному запиті.
4. Встановити, до яких таблиць відносяться неповні назви стовпців.
5. Визначити тип даних кожного стовпця.

Якщо б системні таблиці служили лише для задоволення внутрішніх потреб СУБД, то для користувачів БД вони не представляли б практично жодного інтересу. Однак системні таблиці, як правило, доступні також і для користувачів – запити до системних каталогів дозволені майже в усіх СУБД для персональних комп'ютерів та великих серверів. За допомогою запитів до системних каталогів користувач може отримати інформацію про структуру БД, навіть якщо ніколи раніше з нею не працював.

Користувачі можуть лише зчитувати інформацію з системного каталогу. СУБД забороняє користувачам модифікувати системні таблиці безпосередньо, оскільки це може порушити цілісність БД. СУБД сама вставляє, видаляє та оновлює рядки системних таблиць під час модифікації структури БД. Зміни в системних таблицях також відбуваються в результаті виконання таких операторів DDL, як CREATE, ALTER, DROP.

23.2. Системний каталог та стандарт ANSI/ISO

У стандарті SQL1 нічого не говориться про структуру та зміст системного каталогу. Стандарт фактично не вимагає навіть наявності самого системного каталогу. Однак у всіх основних реляційних СУБД у тій чи іншій формі він

створюється. Структура каталогу та таблиці, що в ньому містяться, значно відрізняються в різних СУБД.

У зв'язку з зростаючим значенням інструментальних програм загального призначення, які спрямовано на роботу з БД і вимагають доступу до системного каталогу, в стандарті SQL2 включено специфікацію набору представлень, які забезпечують стандартизований доступ до інформації, яка зазвичай міститься в системному каталозі. СУБД, що відповідає стандарту SQL2, повинна підтримувати ці представлення, які узагальнено позначаються як INFORMATION_SCHEMA (інформаційна схема).

23.3. Вміст системного каталогу

Кожна таблиця системного каталогу містить інформацію про окремий структурний елемент БД. До складу майже всіх комерційних реляційних СУБД входять, з невеликими відмінностями, системні таблиці, кожна з яких описує один із наступних п'яти елементів:

Таблиці. В каталозі описується кожна таблиця БД. Для окремої таблиці вказується її назва, власник, кількість стовпців у ній, їх розмір тощо.

Стовпці. В каталозі описується кожний стовпець таблиць БД. Для окремого стовпця зазначають назву стовпця, назву таблиці, до якої він належить, тип даних стовпця, його розмір, чи дозволені значення NULL та інше.

Домени. В каталозі описується кожен домен БД. Для окремого домену зазначаються тип, умова сортування, умова контролю та інше.

Представлення. В каталозі описується кожне представлення, що є у БД. Для окремого представлення вказують його назву, ім'я власника, запит, який є визначенням представлення, та інше.

Користувачі. В каталозі описується кожен зареєстрований користувач БД. Вказуються ім'я користувача, його пароль у зашифрованому вигляді та інші дані.

Привілеї. В каталозі описується кожен набір привілеїв, наданих в БД. Зазначаються імена тих, кому привілеї надано, і тих, хто їх надав, вказуються самі привілеї, об'єкти, на які вони поширюються, та інше.

23.4. Інформаційна схема в стандарті SQL2

У стандарті SQL2 не визначено форму системного каталогу, яку мають підтримувати реляційні СУБД. Оскільки на момент прийняття стандарту SQL2 вже існувала широка розбіжність характеристик комерційних СУБД різних типів та величезні відмінності в їх системних каталогах, було неможливо досягти згоди

щодо стандартної специфікації системного каталогу. Замість цього автори стандарту надали визначення ідеалізованого системного каталогу, який постачальники СУБД могли б використовувати при розробці нових СУБД, відповідних стандарту SQL2. Системні таблиці цього ідеалізованого системного каталогу, який у стандарті називається схемою визначень, наведені у таблиці 23.1.

Таблиця 23.1. Ідеалізований системний каталог – словник даних, описаний у стандарті SQL2

Системна таблиця	Вміст
USERS	Один рядок для кожного ідентифікатора користувача у каталозі.
SCHEMATA	Один рядок для кожної інформаційної схеми у каталозі.
DATA_TYPE_DESCRIPTOR	Один рядок для кожного домену або стовпця, який має якийсь тип даних.
DOMAINS	Один рядок для кожного домену.
DOMAIN_CONSTRAINTS	Один рядок для кожної обмежувальної умови, накладеної на домен.
TABLES	Один рядок для кожної таблиці або представлення.
VIEWS	Один рядок для кожного представлення.
COLUMNS	Один рядок для кожного стовпця у кожній визначеній таблиці чи представленні.
VIEW_TABLE_USAGE	Один рядок кожної таблиці, на яку є посилання у якомусь визначенні представлення. Якщо визначенням представлення є багатотабличний запит, кожна таблиця буде представлена окремим рядком.
VIEW_COLUMN_USAGE	Один рядок для кожного стовпця, на який є посилання в якомусь представленні.
TABLE_CONSTRAINTS	Один рядок для кожної обмежувальної умови, заданої в якомусь визначенні таблиці.
KEY_COLUMN_USAGE	Один рядок для кожного стовпця, на який накладено умову унікальності і який є у визначенні первинного або зовнішнього ключа. Якщо у визначенні ключа чи умови унікальності вказано декілька стовпців, то це визначення буде представлено декількома рядками.
REFERENTIAL_ CONSTRAINTS	Один рядок для кожного визначення зовнішнього ключа, що є у визначенні таблиці.
CHECK_CONSTRAINTS	Один рядок для кожної умови перевірки, заданої у визначенні таблиці.
CHECK_TABLE_USAGE	Один рядок для кожної таблиці, на яку є посилання в умові перевірки, обмежувальній умові для домену або твердженні.
CHECK_COLUMN_USAGE	Один рядок для кожного стовпця, на який є посилання в умові перевірки, обмежувальній умові для домену або твердженні.
ASSERTIONS	Один рядок для кожного заданого твердження.

Системна таблиця	Вміст
TABLE_PRIVILEGES	Один рядок для кожного привілею, наданого на якусь таблицю.
COLUMN_PRIVILEGES	Один рядок для кожного привілею, наданого на будь-який стовпець.
USAGE_PRIVILEGES	Один рядок для кожного привілею, наданого на будь-який домен, набір символів тощо.
CHARACTER_SETS	Один рядок для кожного заданого набору символів.
COLLATIONS	Один рядок для кожної заданої послідовності порівняння.
TRANSLATIONS	Один рядок для кожного заданого перетворення.
SQL_LANGUAGES	Один рядок для кожної мови, наприклад PASCAL, C++ тощо, що підтримується СУБД даного типу.

Слід зазначити, що стандарт SQL2 не вимагає, щоб конкретні СУБД підтримували таблиці системного каталогу, збережені у Таблиця 23.1 або будь-які інші. Замість цього, стандарт SQL2 визначає ряд стандартних представлень, заснованих на цих системних таблицях.

Представлення, засновані на системних таблицях, містять ті об'єкти бази даних, які мають бути доступні для звичайного користувача. Ці представлення у стандарті SQL називаються *інформаційною схемою (Information Schema)*. Для того щоб СУБД відповідала стандарту SQL2, вона має підтримувати обов'язкові представлення. Такий підхід надає користувачеві стандартний спосіб отримання інформації про доступні для нього об'єкти бази даних за допомогою стандартних запитів до стандартних представлень системного каталогу.

На практиці комерційні реляційні СУБД підтримують стандартні представлення системного каталогу шляхом створення відповідних представлень на основі реальних таблиць своїх власних системних каталогів. Інформація в системних каталогах більшості СУБД досить близька до тієї, яка потрібна у стандарті, тому визначення стандартних представлень каталогу, створюваних в цих СУБД, є відносно простими.

Представлення системного каталогу, які вимагає стандарт SQL2, подано у таблиці 23.2. У ній наведено короткий опис інформації, яка міститься у кожному представленні.

Таблиця 23.2. Представлення системного каталогу – словника даних, встановлені стандартом SQL2

Представлення в системному каталозі	Вміст
INFORMATION_SCHEMA_CATALOG_NAME	Один рядок з назвою БД для кожного користувача – каталогу з термінології стандарту SQL2, який описує дана інформаційна схема.

Представлення в системному каталозі	Вміст
SCHEMATA	Один рядок для кожної інформаційної схеми БД, що належить поточному користувачеві. Містить назву схеми, набір символів за замовчуванням тощо.
DOMAINS	Один рядок для кожного домену, доступного для поточного користувача. Містить назву домену, базовий тип даних, набір символів, максимальну довжину, ступінь, точність тощо.
DOMAIN_CONSTRAINTS	Один рядок для кожного обмежуючого умови домену; містить назву умови та її характеристики.
TABLES	Один рядок для кожної таблиці або представлення, доступного користувачу. Містить назву та позначення того, чи йдеться про таблицю чи про представлення.
VIEWS	Один рядок для кожного представлення, доступного користувачу. Містить назву, інформацію про режим контролю та можливості оновлення.
COLUMNS	Один рядок для кожного стовпця, доступного користувачу. Містить назву стовпця, назву таблиці або представлення, які містять даний стовпець, тип даних, що містяться в ньому, ступінь, точність, набір символів тощо.
TABLE_PRIVILEGES	Один рядок для кожного привілею на таблицю, наданого користувачеві або наданому ним іншому користувачеві. Містить назву таблиці, тип привілею, вказівку на те, хто надав привілей, кому він наданий та чи має користувач право надання цього привілею.
COLUMN_PRIVILEGES	Один рядок для кожного привілею на стовпець, наданого користувачеві або наданого ним іншому користувачеві. Містить назву таблиці та стовпця, тип привілею, вказівку на те, хто надав привілеї, кому він наданий та чи має користувач право надання цього привілею.
USAGE_PRIVILEGES	Один рядок для кожного привілею, наданого користувачеві або користувачем на будь-який домен, набір символів тощо.
TABLE_CONSTRAINTS	Один рядок на кожну обмежувальну умову: первинний ключ – PRIMARY KEY; зовнішній ключ – FOREIGN KEY; умова унікальності – UNIQUE; умова перевірки – CHECK, задане для таблиці, якою володіє користувач. Містить назву умови та таблиці, тип умови та її характеристики.
REFERENTIAL_ CONSTRAINTS	Один рядок для кожного обмеження за посиланнями (визначення зовнішнього ключа) на таблицю, якою володіє користувач; містить назву обмеження, назву таблиці-нащадка та назву таблиці-предка.
CHECK__CONSTRAINTS	Один рядок для кожної умови перевірки для таблиці, якою володіє користувач.

Представлення в системному каталозі	Вміст
KEY_COLUMN_USAGE	Один рядок для кожного стовпця первинного або зовнішнього ключа, на який (стовпець) накладено умову унікальності і який входить до таблиці, що належить користувачу. Рядок містить назву таблиці, назву стовпця та позицію стовпця в ключі.
ASSERTIONS	Один рядок для кожного твердження, яким володіє користувач. Містить назву твердження та його характеристики.
CHARACTER_SETS	Один рядок для кожного визначення набору символів, доступного користувачу.
COLLATIONS	Один рядок для кожного визначення послідовності порівняння, доступного користувачу.
TRANSLATIONS	Один рядок для кожного визначення перетворення, доступного користувачу.
VIEW_TABLE_USAGE	Один рядок для кожної таблиці, на яку є посилання в визначеннях представлень, що належать користувачу. Рядок містить назву таблиці.
VIEW_COLUMN_USAGE	Один рядок для кожного стовпця, на який є посилання в представленнях, що належать користувачеві. Рядок містить назву стовпця та таблиці, до якої входить стовпець.
CONSTRAINT_TABLE_ USAGE	Один рядок для кожної таблиці, на яку є посилання в умові перевірки, умові унікальності, твердженні та визначення зовнішнього ключа, що належать користувачу.
CONSTRAINT_COLUMN_ USAGE	Один рядок для кожного стовпця, на який є посилання в умові перевірки, умові унікальності, твердження та визначення зовнішнього ключа, що належать користувачу.
SQL_LANGUAGES	Один рядок для кожної мови (наприклад, COBOL, C тощо), що підтримується СУБД даного типу. У рядку вказується рівень відповідності мови стандарту SQL2, тип діалекту SQL, що підтримується, тощо.

У стандарті SQL2 також визначені 3 домени, які використовуються представленнями системного каталогу і доступні користувачам. Ці домени наведені у таблиці 23.3.

Таблиця 23.3. Домени, визначені у стандарті SQL2

Системний домен	Область значень домену
SQL_IDENTIFIER	Домен всіх символьних рядків змінної довжини, які є допустимими ідентифікаторами SQL відповідно до стандарту SQL2. Будь-яке значення, взяте з цього домену, є допустимою назвою таблиці, назвою стовпця тощо.
CHARACTER_DATA	Домен всіх символьних рядків змінної довжини, довжина яких може бути від нуля до максимального значення, що підтримується СУБД. Значення, взяте з цього домену, є допустимим символьним рядком.

Системний домен	Область значень домену
CARDINAL_NUMBER	Домен всіх невід'ємних чисел від нуля до максимального цілого числа, з яким може працювати СУБД. Значення, взяте з цього домену, є нулем чи допустимим позитивним числом.

У практичному плані слід звернути увагу на наступні таблиці та представлення:

TABLES – один рядок для кожної таблиці або представлення;

VIEWS – один рядок для кожного представлення;

TABLE_CONSTRAINTS – один рядок на кожну обмежувальну умову, задану для таблиці: первинний ключ – PRIMARY KEY; зовнішній ключ – FOREIGN KEY; умова унікальності – UNIQUE; умова перевірки – CHECK;

COLUMNS – один рядок для кожного стовпця в кожному визначенні таблиці або представлення;

KEY_COLUMN_USAGE – один рядок для кожного стовпця, на який накладено умову унікальності UNIQUE і який присутній в визначенні первинного PRIMARY KEY або зовнішнього FOREIGN KEY ключа;

CONSTRAINT_COLUMN_USAGE – один рядок для кожного стовпця, на який є посилання в твердженні та визначенні зовнішнього ключа PRIMARY KEY, умови перевірки CHECK або умови унікальності UNIQUE.

Наведемо декілька прикладів запитів для отримання інформації про структуру БД з представлень системного каталогу, визначених у стандарті SQL2:

Приклад. Вивести назви всіх таблиць та представлень користувача, який працює з БД у поточний момент.

```
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
```

Приклад. Вивести назву, позицію та тип даних для кожного стовпця в таблиці Students.

```
SELECT COLUMN_NAME, ORDINAL_POSITION, DATA_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE (INFORMATION_SCHEMA.COLUMNS.TABLE_NAME =
        'Students')
```

Приклад. Визначити, скільки стовпців є в таблиці Students.

```
SELECT COUNT(*) FROM INFORMATION_SCHEMA.COLUMNS
WHERE (INFORMATION_SCHEMA.COLUMNS.TABLE_NAME =
         'Students')
```

Приклад. Вивести назву, позицію та тип даних для кожного стовпця у всіх представленнях.

```
SELECT TABLE_NAME, COLUMN_NAME, ORDINAL_POSITION,
       DATA_TYPE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE (INFORMATION_SCHEMA.COLUMNS.TABLE_NAME
    IN (SELECT TABLE_NAME FROM VIEWS))
```

23.5. Офіційні стандарти мови SQL

Перший офіційний стандарт мови SQL, так званий SQL-86, був прийнятий у 1986 році та дещо уточнений у 1989 році. Подальший розвиток мови від постачальників СУБД змусив прийняти у 1992 році новий розширений стандарт SQL-92 або просто SQL2. Наступним стандартом став SQL:1999 або SQL3. На даний момент діє стандарт, прийнятий у 2003 році – SQL:2003 з невеликими модифікаціями, внесеними пізніше (таблиця 23.4).

Таблиця 23.4. Офіційні стандарти мови SQL

Рік	Назва	Інша назва	Зміни
1986	SQL-86		Перший варіант стандарту, прийнятий ANSI у 1986 р. та схвалений ISO у 1987 р. Стандарт ISO 9075
1992	SQL-92	SQL2	Значні зміни ISO 9075
1999	SQL:1999	SQL3	Додано підтримку регулярних виразів, рекурсивних запитів, підтримка тригерів, базові процедурні розширення, нескалярні типи даних та деякі об'єктно-орієнтовані можливості.
2003	SQL:2003		Введено розширення для роботи з XML-даними, віконні функції, що застосовуються для роботи з OLAP базами даних, генератори послідовностей і типи даних, що ґрунтуються на них.
2006	SQL:2006		Розширено функціональність роботи з XML-даними. З'явилась можливість спільного використання у запитах SQL та XQuery.
2008	SQL:2008		Поліпшено можливості віконних функцій
2011	SQL:2011		Реалізовано підтримку хронологічних баз даних (PERIOD FOR), підтримка конструкції FETCH
2016	SQL:2016		Реалізовано захист на рівні рядків, поліморфні табличні функції, JSON
2023	SQL:2023		Реалізовано операції над графами. Агрегатна функція ANY_VALUE(). Введена підтримка 16-ти, 2-ух та 8-ми розрядних літералів

23.6. Контрольні запитання

1. Що таке системний каталог або словник даних? Для чого він призначений?

2. Хто створює системний каталог?
3. Яка стандартна послідовність взаємодії СУБД з системним каталогом при обробці оператора SELECT?
4. Чи має користувач можливість звертатися з запитом до системного каталогу?
5. Чи має користувач можливість вносити зміни до системного каталогу?
6. Коли і за яких умов вносяться зміни до таблиць системного каталогу?
7. Які елементи входять до складу системного каталогу?

23.7. Завдання для самопідготовки

Завдання 23.1. Вивести назви таблиць заданої БД.

Завдання 23.2. Вивести назву та тип даних кожного стовпця заданої таблиці БД.

Завдання 23.3. Вивести вміст таблиці TABLE_CONSTRAINTS і проаналізувати її склад для заданої таблиці БД.

Завдання 23.4. Вивести вміст таблиці COLUMNS і проаналізувати її склад для заданої таблиці БД.

Завдання 23.5. Вивести вміст таблиці CONSTRAINT_COLUMN_USAGE і проаналізувати її склад для відповідних стовпців заданої таблиці БД.

Завдання 23.6. Вивести вміст таблиці KEY_COLUMN_USAGE і проаналізувати її склад для відповідних стовпців заданої таблиці БД.

24. КЛІЄНТ/СЕРВЕРНІ ТЕХНОЛОГІЇ СУБД

24.1. Системи типу клієнт/сервер

Системи управління базами даних (СУБД, Database Manager Systems, DBMS) – це програмні засоби, призначені для створення, наповнення, оновлення, пошуку та видалення БД.

За своєю архітектурою системи БД діляться на однорівневу, дворівневу, трирівневу (рис. 24.1).

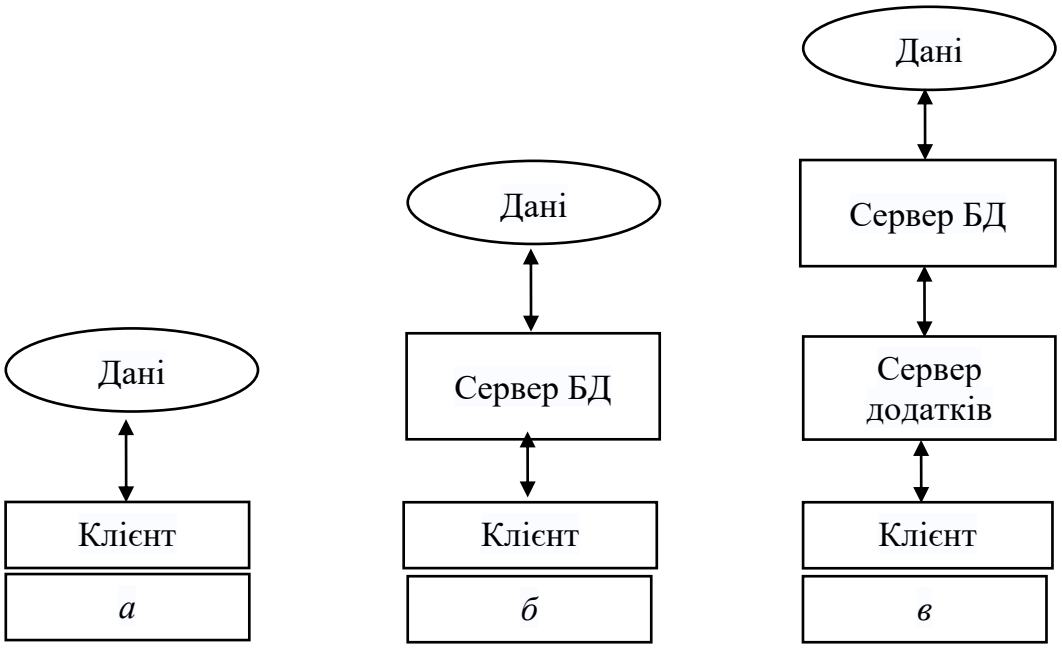


Рис. 24.1. Архітектура систем БД: однорівнева (а), дворівнева (б), трирівнева (в)

В однорівневій архітектурі використовується єдиний рівень (клієнт), що забезпечує необхідну логіку управління даними та їх візуалізацію.

У дворівневій архітектурі значну частину логіки управління даними бере на себе сервер БД, тоді як клієнт переважно займається відображенням даних у зручному для користувача вигляді. При цьому клієнт має знати, де розташований сервер.

У трирівневій архітектурі використовується проміжний рівень – сервер додатків, що є посередником між клієнтом та сервером БД. Сервер додатків призначено щодо управління даними та забезпечення зв'язку з сервером БД.

Залежно від розташування окремих частин СУБД відрізняють локальні та мережеві СУБД.

Всі складові локальної СУБД розміщуються на комп'ютері користувача БД.

До мережевих відносяться файл-серверні, клієнт-серверні та розподілені СУБД. Невід'ємною ознакою цих систем є мережа, що забезпечує апаратний

зв'язок комп'ютерів і дає можливість корпоративної роботи багатьох користувачів з одними й тими ж даними.

У файл-серверних СУБД всі дані зазвичай розміщуються у одному або декількох каталогах досить потужної машини, спеціально виділеної для цих цілей і постійно підключеної до мережі. Такий комп'ютер називається файл-сервером. Безумовною перевагою СУБД цього типу є відносна простота її створення та обслуговування – фактично все зводиться лише до розгортання локальної мережі та встановлення на підключених до неї комп'ютерах мережевих операційних систем.

Між локальними та файл-серверними варіантами СУБД немає особливих відмінностей, оскільки в них всі складові саме СУБД (крім даних) знаходяться на комп'ютері клієнта. За архітектурою вони зазвичай є одноярусними, але в деяких випадках можуть використовувати сервер додатків. Недоліком файл-серверних систем є значне навантаження на мережу, оскільки між сервером та клієнтом доводиться пересилати цілі файли, а обробка файлів виконується на клієнтських машинах.

Історично файл-серверні СУБД були першими, які надавали можливість колективного використання даних. На сьогодні вони, як і раніше, підтримуються, але використовуються переважно для додатків, які були створені раніше.

Клієнт-серверні (Client-Server) – дворівневі системи, що значно зменшують навантаження на мережу, оскільки клієнт спілкується з даними через спеціалізованого посередника – сервер БД, який розміщується на машині з даними. Сервер БД (BackEnd) приймає запит від клієнта (FrontEnd), знаходить у даних потрібний запис та передає його клієнту. Таким чином, мережею передаються відносно короткий запит і потрібні записи, навіть якщо відповідний файл з даними містить сотні тисяч записів.

Запит до сервера формується мовою структурованих запитів SQL, тому часто сервери БД називаються SQL-серверами. Сервери БД представляють собою відносно складні програми, розроблені різними фірмами. До них відносяться, наприклад, Oracle, DB2, Microsoft SQL Server, Sybase SQL Server, Informix, MySQL Server, PostgreSQL Server тощо.

Клієнт-серверні СУБД масштабуються до сотень і тисяч клієнтських місць.

Переваги клієнт-серверних технологій:

1. Дозволяє організувати мережу з великою кількістю комп'ютерів;
2. Централізоване управління обліковими записами;
3. Ефективний доступ до мережевих ресурсів.

Недоліки клієнт-серверних технологій:

1. Збої в роботі сервера може спричинити відмову мережі;
2. Адміністрування даної системи вимагає від персоналу високої кваліфікації;
3. Висока вартість обладнання.

Систему клієнт/сервер можна розглядати як розподілену систему, в якій одні вузли є клієнтами, а інші – серверами, всі дані зберігаються на серверах, а всі додатки виконуються клієнтами.

Розподілені СУБД можуть містити декілька десятків і сотень серверів БД. Кількість клієнтських місць у них може досягати десятків і сотень тисяч.

24.2. Сервери баз даних

Зазвичай для визначення всієї СУБД, що базується на архітектурі клієнт-сервер, включаючи і серверну, і клієнтську частини, використовується термін "сервер БД". Такі системи призначені для збереження та забезпечення доступу до БД.

Доступ до БД прикладної програми або користувача здійснюється шляхом звернення до клієнтської частини системи. Основним інтерфейсом між клієнтською та серверною частинами виступає мова SQL.

У типовому на сьогоднішній день випадку на стороні клієнта СУБД працює тільки таке програмне забезпечення, яке не має безпосереднього доступу до БД, а звертається для цього до серверу з використанням мови SQL.

Слід зазначити й інші варіанти цієї основної теми:

1. Декілька клієнтів можуть використовувати один і той же сервер (це досить характерна ситуація).
2. Один клієнт може здійснювати доступ до декількох серверів. Цю ситуацію можна реалізувати двома шляхами:
   2.1. Клієнт у певний момент часу може здійснювати доступ лише до одного сервера, тобто кожен окремий запит до БД повинен бути спрямований лише до одного сервера і всередині одного запиту не можна комбінувати дані, розміщені на двох або більше серверах. Крім того, користувач повинен знати, які дані і на яких серверах зберігаються;
   2.2. Клієнт у певний момент часу може здійснювати доступ до декількох серверів, тобто в окремому запиті до БД можуть комбінуватися дані, розміщені на двох або більше серверах. Це означає, що дані для користувача виглядають так, ніби вони дійсно

були розміщені на одному сервері. У такому випадку користувачеві не потрібно знати, які дані і на яких серверах зберігаються.

Другий спосіб фактично є формулюванням істинної розподіленої системи, сенс якої не відповідає поширеному визначенню терміну "клієнт/сервер".

24.3. Концепція відкритих систем

Реальне поширення архітектури клієнт-сервер стало можливим завдяки розвитку та широкому впровадженню в практику концепції відкритих систем (Open Systems).

Основним змістом підходу відкритих систем є спрощення комплексування обчислювальних систем за рахунок міжнародної та національної стандартизації апаратних та програмних інтерфейсів.

Ключовою особливістю відкритих систем є незалежність від конкретного постачальника. Орієнтуючись на продукцію компаній, що дотримуються стандартів відкритих систем, споживач може нарощувати потужності своєї системи шляхом придбання продуктів будь-якої компанії, що дотримується стандартів.

24.4. Відкритий зв'язок із базою даних. ODBC

Відкритий зв'язок з БД (ODBC, Open DataBase Connectivity) – це програмний інтерфейс (API, Application Programs Interface) доступу до БД. ODBC дозволяє працювати з різними джерелами даних з використанням одного рівня, абстрагуючись від особливостей взаємодії в кожному конкретному випадку.

На зламі 80–90-х років компанія Microsoft запропонувала стандарт для взаємодії з БД на платформі Windows. Цей комутаційний інтерфейс отримав назву Open Database Connectivity (ODBC). Основою для створення послужив стандарт SQL X/Open Call Level Interface.

ODBC API дозволяє розробникам створювати додатки, які працюють з БД різних виробників.

На рис. 24.2 показано вміст панелі адміністрування Windows. Зокрема, вона надає доступ до налаштувань з'єднань з 32- та 64-бітними серверами БД.

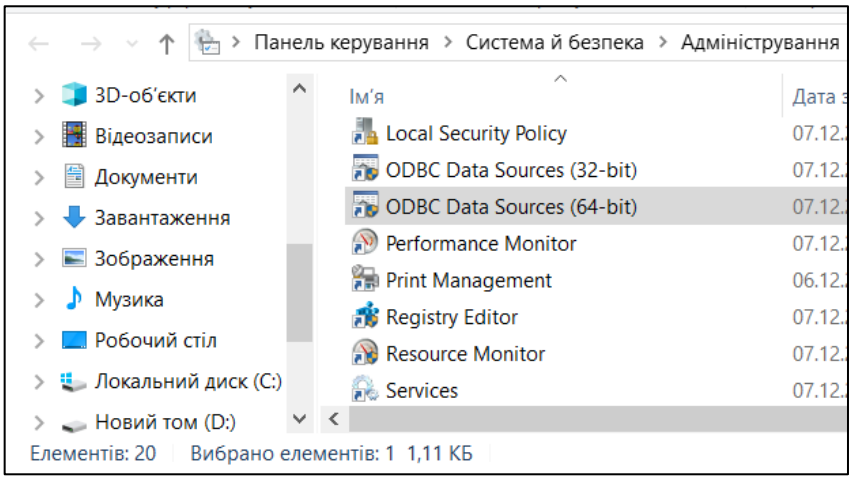


Рис. 24.2. Панель Windows адміністрування джерел даних ODBC

На рис. 24.3 показано перелік драйверів у вікні адміністратора джерел даних ODBC.

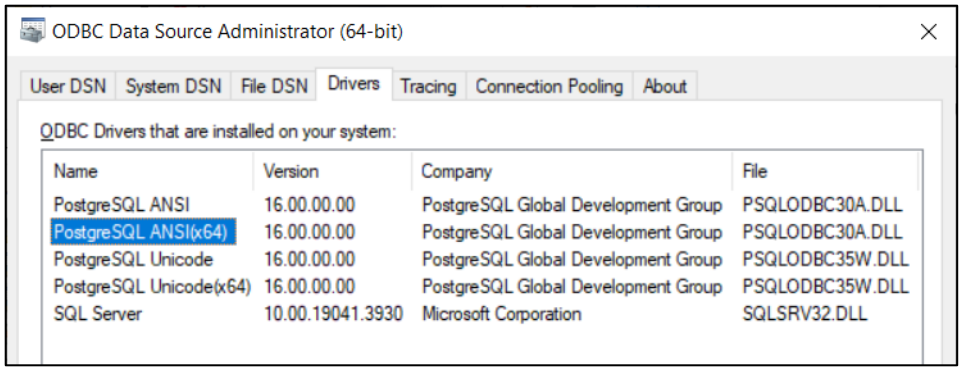


Рис. 24.3. Адміністратор джерел даних ODBC – перелік драйверів ODBC

На рис. 24.4 показано налаштування DSN – назви джерела даних користувача Deanery для БД deanery у СУБД PostgreSQL.

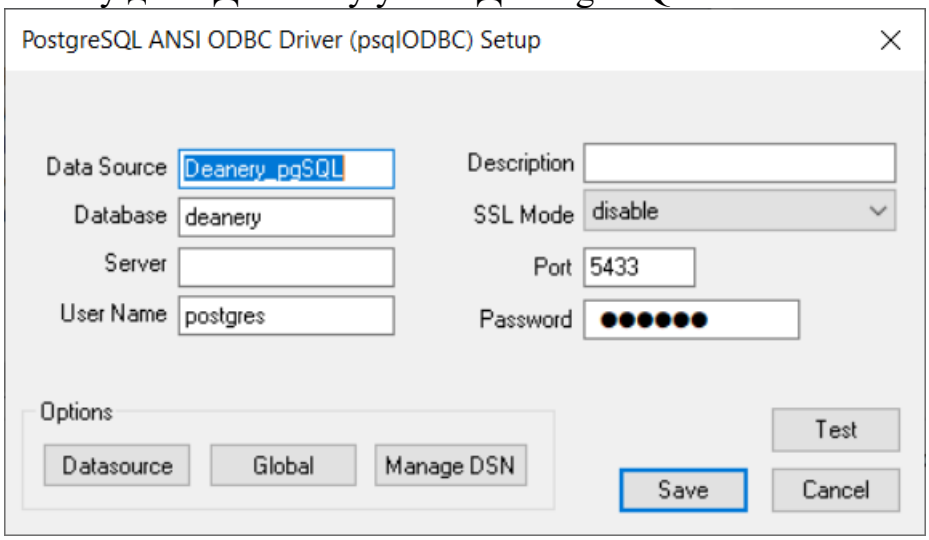


Рис. 24.4. Визначення DSN – назви джерела даних, з БД deanery

Після визначення DSN користувач у своєму додатку може звертатися до свого джерела даних – Deanery, не фокусуючи увагу на тому, з якою СУБД він працює. Якщо знадобиться перейти до аналогічної БД, але, наприклад у MySQL, достатньо внести зміни у налаштування на панелі рис. 24.5. При цьому жодних змін у додатку робити не потрібно. У результаті додаток не залежить від фізичного джерела даних.

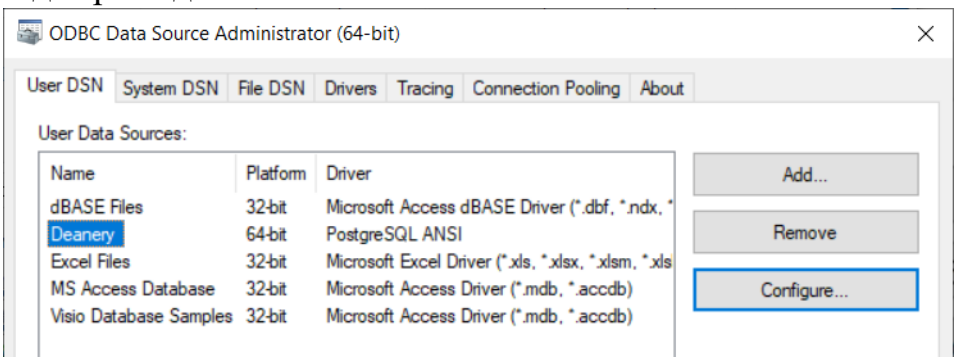


Рис. 24.5. DSN користувача для БД deanery

Засоби ODBC дуже популярні, але вони представляють собою "важкі" процеси і при зростанні обсягу БД вони втрачають у продуктивності. Кажуть, що вони орієнтовані на створення "товстого клієнта", коли обробка даних виконується на машині клієнта. Крім того, ODBC працює тільки з реляційними БД.

У подальшому компанія Microsoft розробила нові стандарти, що забезпечують створення "легких" процесів для зв'язку з БД та підтримку моделей даних, відмінних від реляційних. Ці стандарти дозволяють досягти хороших результатів при роботі з Windows. Кажуть, що вони орієнтовані на створення "тонкого клієнта", коли обробка даних виконується на сервері БД. Один з таких стандартів є ADO для Windows та ADO.Net для .NET.

Примітка. Слід підкреслити, що якщо нова СУБД з'являється на ринку, вона обов'язково повинна надавати засоби ODBC. Справа в тому, що не всі мови програмування мають засоби прямої взаємодії з конкретними СУБД, але вони завжди мають можливість звернутися до цих СУБД за допомогою ODBC. Для цього створюються спеціальні конектори, наприклад, для мови Java це jdbc:odbc міст, а для мови Python це модуль pyodbc.

24.5. Особливості технології доступу до даних ADO

OLE DB і ADO – частина універсального механізму доступу до даних Microsoft (Microsoft Universal Data Access), що дозволяє здійснювати доступ до

реляційних та нереляційних джерел даних, таких як файлова система, дані електронної пошти, багатовимірні сховища даних і інше.

Microsoft ActiveX Data Objects (ADO) – це набір бібліотек, що містять COM-об'єкти, які реалізують прикладний програмний інтерфейс для доступу до таких даних і використовуються в клієнтських додатках. ADO використовує бібліотеки OLE DB, які надають низькорівневий інтерфейс для доступу до даних. OLE DB надає доступ до даних за допомогою COM-інтерфейсів. Також можна використовувати OLE DB безпосередньо, оминаючи ADO.

Для доступу до джерела даних за допомогою OLE DB необхідно, щоб на комп'ютері, де використовується клієнтський додаток, був встановлений OLE DB-провайдер для відповідної СУБД. OLE DB-провайдер є DLL, що завантажується в адресний простір клієнтського додатка і використовується для доступу до джерела даних. Для кожного типу СУБД потрібен власний OLE DB-провайдер, оскільки ці провайдери базуються на функціях клієнтських API, різних для різних СУБД.

Серед OLE DB-провайдерів для різних джерел даних є спеціальний провайдер Microsoft OLE DB Provider for ODBC Drivers. Цей провайдер використовує не API клієнтської частини якої-небудь СУБД, а інтерфейс ODBC API, тому він застосовується разом з ODBC-драйвером для вибраної СУБД.

Структура взаємодії додатка з БД наведена на рис. 24.6.

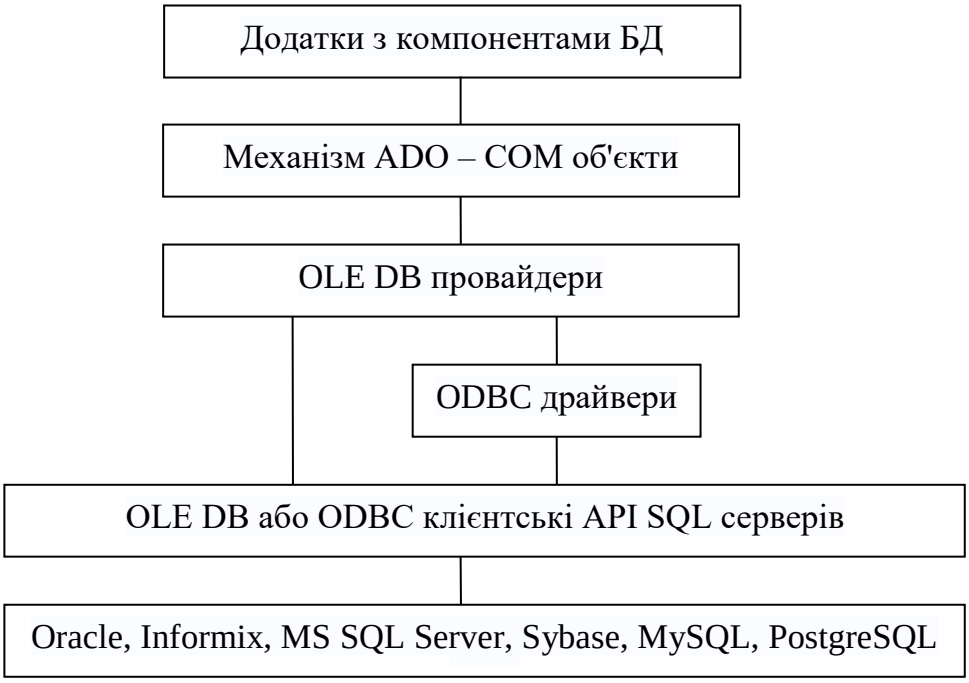


Рис. 24.6. Структура взаємодії додатка з БД за допомогою ADO

Важливо підкреслити, що ADO надає програмісту свій варіант імітації всіх елементів, які властиві реальним (фізичним) БД. Таким чином, взаємодія з

реальною БД підміняється взаємодією зі спеціально створеними компонентами роботи з БД, які через ADO "вміють" взаємодіяти з реальними елементами відповідних БД. Компоненти ADO легко зв'язуються з відповідними графічними компонентами MS Visual Studio.

Основною перевагою технології ADO є її природна орієнтація на створення полегшеного (тонкого) клієнта.

На машині сервера БД створюється і розміщується джерело даних. Якщо використовується промисловий сервер БД типу Oracle або MS SQL Server, дані не потребують попередньої підготовки, а в якості провайдера використовується відповідно Microsoft OLE DB Provider for Oracle або Microsoft OLE DB Provider for SQL Server.

На жаль, технологію ADO не можна використовувати, якщо для відповідної структури даних не створено потрібний провайдер або ODBC драйвер. Так, наприклад, для деяких серверів БД – InterBase, Informix, DB2 та інших на сьогоднішній день відсутні потрібні OLE DB провайдери. Однак для деяких з них, наприклад Informix, можна знайти відповідні ODBC-драйвери.

24.6. Особливості технології доступу до даних ADO.Net

ADO.Net (ActiveX Data Objects .Net) є набором класів, які реалізують програмні інтерфейси для спрощення підключення до БД з додатка, незалежно від особливостей реалізації конкретної СУБД та від структури самої БД, а також незалежно від місця розташування цієї БД – зокрема, в розподіленому середовищі на стороні сервера.

ADO.Net широко використовується спільно з технологією веб-програмування за допомогою об'єктів ASP.Net для доступу зі сторони клієнта до серверних БД.

Вирішення навіть найпростішої задачі, пов'язаної з даними, передбачає використання багатьох різноманітних об'єктів – представників класів ADO.Net, які знаходяться між собою в досить складних взаємовідносинах.

24.6.1. Приєднаний та від'єднаний доступ до даних

У додатках, що працюють з БД, до недавнього часу застосовувався доступ до даних через постійне з'єднання з джерелом даних – модель доступу до приєднаних даних. Додаток відкривав з'єднання з БД і не закривав його, принаймні, до завершення роботи з джерелом даних. Протягом цього часу з'єднання з джерелом підтримувалось постійно. Однак будь-яке з'єднання з БД

потребує виділення системних ресурсів, і якщо БД розташована на сервері, то при великій кількості клієнтів це може бути критично для сервера. Хоча постійне з'єднання і дозволяє трохи прискорити роботу додатка, загальний збиток від розтрати системних ресурсів зводить нанівець перевагу в швидкості виконання додатка.

Факт поганого масштабування додатків з постійним з'єднанням відомий давно. З'єднання з парою клієнтів обслуговується додатком добре, 10 клієнтів обслуговуються гірше, 100 – набагато гірше тощо. Недоліки такого підходу почали особливо проявлятися після появи додатків, орієнтованих на Інтернет.

Для усунення зазначених вище недоліків в ADO.Net використовується модель доступу до від'єднаних даних. При цьому з'єднання з даними встановлюється лише на той час, який необхідний для проведення певної операції на сервері БД.

Модель доступу ADO.Net – компромісна модель. У ряді випадків вона програє за продуктивністю традиційній моделі, і для цих випадків рекомендується замість ADO.Net використовувати ADO.

24.6.2. Концепція доступу до даних в ADO.Net

Концепція доступу до даних в ADO.Net базується на використанні двох компонентів:

1. Набору даних, що виступає як локальне тимчасове сховище даних, і представляється об'єктом класу DataSet, який забезпечує взаємодію додатка і БД зі сторони клієнта;
2. Провайдера даних, що виступає як посередник, який забезпечує взаємодію додатка і БД зі сторони локальної БД або сервера БД, і представляється об'єктом класу DataProvider.

Схему доступу до бази даних в ADO.Net показано на рис. 24.7.

Рис. 24.7. Доступ до БД в ADO.Net

24.6.3. Об'єктна модель ADO.Net

Об'єктна модель ADO.Net передбачає використання двох наборів класів – класу під'єднаних і класу від'єднаних об'єктів, що виконують чітко визначені завдання при роботі з БД (рис. 24.8).

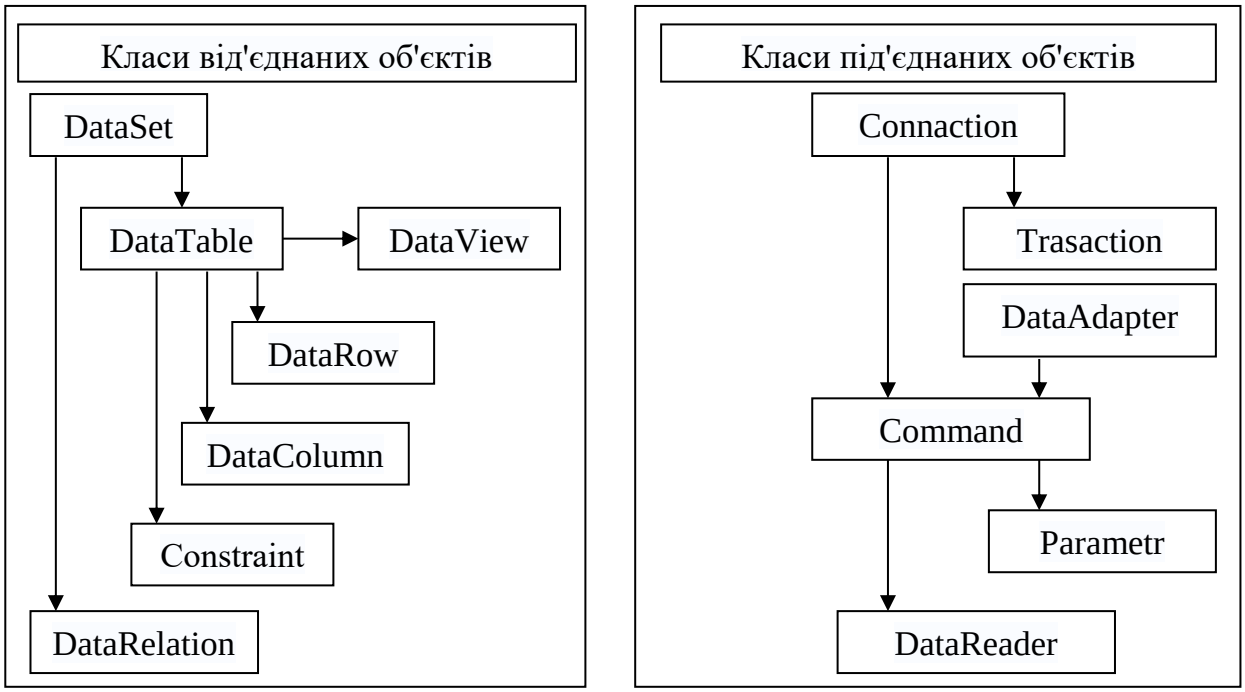


Рис. 24.8. Об'єктна модель ADO.Net

Класи під'єднаних об'єктів забезпечують встановлення з'єднання з БД і управління цією БД зі сторони додатка.

Класи від'єднаних об'єктів забезпечують збереження, використання і перетворення отриманих від БД інформації на стороні додатка.

Основна обробка даних виконується на рівні класів від'єднаних об'єктів.

24.6.4. Класи від'єднаних об'єктів

24.6.4.1. Об'єкт DataTable

Кожен об'єкт DataTable представляє копію однієї таблиці БД. Таблиця характеризується:

1. Схемою таблиці;
2. Вмістом таблиці.

При цьому схема таблиці – структура об'єкта DataTable, визначається двома наборами:

1. Набором стовпців таблиці – набір DataColumns, що складається з множини об'єктів DataColumn;
2. Набором обмежень таблиці – набір Constraints, що складається з множини об'єктів Constraint.

24.6.4.2. Об'єкт DataColumn

У кожного об'єкта DataTable є набір колонок Columns, який являє собою контейнер об'єктів DataColumn. Набір (колекція) Columns задає схему таблиці, визначаючи тип даних кожної колонки.

Таким чином, DataColumn містить не дані, які зберігаються в об'єкті DataTable, а інформацію про структуру стовпця, яка відповідає інформації про структуру стовпця в БД.

24.6.4.3. Об'єкт DataRow

Вміст таблиці (безпосередньо дані) задається набором Rows – це конкретна множина рядків таблиці, кожен з яких є об'єктом класу DataRow.

За допомогою набору Rows реалізується можливість посилання на будь-який запис таблиці. До будь-якого запису можна звернутися безпосередньо, тому не потрібні методи позиціонування та переміщення записами таблиці.

24.6.4.4. Об'єкт DataRelations

У класі DataSet визначається властивість Relations – набір об'єктів класу DataRelations. Кожен такий об'єкт визначає зв'язки між об'єктами (таблицями) DataTable, що складають об'єкт DataSet. Якщо в DataSet є більше одного набору DataTable, набір DataRelations буде містити декілька об'єктів типу DataRelation.

Кожен об'єкт класу DataRelations визначає зв'язки між таблицями – DataTable. Таким чином, в об'єкті DataSet реалізується повний набір елементів для управління даними, включаючи самі таблиці, обмеження та відношення між таблицями.

Крім того, об'єкти надають властивості, що дозволяють забезпечити цілісність посилань. Наприклад, DataRelation можна налаштувати так, щоб зміна значення первинного ключа батьківського запису автоматично копіювалась дочірнім записам. Об'єкт DataRelation також можна налаштувати так, що при видаленні запису в батьківському об'єкті автоматично видаляються відповідні записи дочірнього об'єкта DataTable.

24.6.4.5. Об'єкт Constraints

Об'єкти класу Constraint в наборі Constraints об'єкта DataTable дозволяють задати на множині об'єктів DataTable різноманітні обмеження. Наприклад, можна створити об'єкт Constraint, що гарантує, що значення поля або декількох полів будуть унікальні в межах DataTable.

24.6.4.6. Об'єкт DataView

Вибравши результати запиту в об'єкт DataTable, можна за допомогою об'єкта DataView переглядати цей вміст різними способами, задаючи різні варіанти сортування та фільтрації.

Переглядати вміст одного об'єкта DataTable можна одночасно за допомогою декількох об'єктів DataView.

Варто підкреслити, що DataView не є засобом перетворення вихідної інформації, зафіксованої в таблиці. Він надає засоби лише динамічного перегляду набору даних.

24.6.4.7. Об'єкт DataSet

В рамках від'єднаної моделі ADO.NET об'єкт DataSet стає важливим елементом технології від'єднаного доступу. Об'єкт DataSet представляє множину таблиць.

DataSet можна розглядати як контейнер об'єктів DataTable (які зберігаються в наборі Tables об'єкта DataSet). Іноді потрібно звернутися до компонента на сервері та вибрати вміст декількох таблиць. Замість того щоб багато разів звертатися до сервера та вибирати дані по одній таблиці за раз, можна помістити всі дані в об'єкт DataSet і повернути його за один виклик. Проте, об'єкт DataSet – це не просто контейнер об'єктів DataTable.

Дані в об'єкті DataSet від'єднані від БД. Всі зміни даних просто кешуються в об'єктах DataRow. Коли настане час передати ці зміни в БД, ймовірно, буде неефективним пересилати серверу весь об'єкт DataSet. Можна скористатися методом GetChanges і вилучити з DataSet лише змінені дані. В результаті зменшиться обсяг даних, які передаються між процесами та серверами.

24.7. Доступ до баз даних у мовах програмування

Типовий спосіб доступу до БД у різних мовах програмування базується на концепції драйверів, які дозволяють отримувати з'єднання з БД за допомогою спеціально описаного URL – уніфікованого локатора ресурсу (Uniform Resource Locator).

Драйвери доступу до БД можуть завантажуватись динамічно. Пізніше вони викликаються автоматично, коли програма вимагає URL, що містить протокол, за який відповідає драйвер.

Використовуючи відповідний драйвер до будь-якої БД, можна підключитися через її URL-рядок, за яким драйвер визначає, де, чим і з чим встановлювати з'єднання. У загальному випадку рядок URL має один із двох наступних форматів:

1. host/database;
2. host:port/database,

де:

1. host – адреса сервера, де розташована СУБД;
2. port – tcp-порт, на якому буде встановлюватися з'єднання;
3. database – назва БД, яка буде використовуватися.

Для більшості СУБД передбачено стандартні номери портів, які встановлюються автоматично при розгортанні СУБД на сервері. Однак при установці адміністратор може задати інший вільний номер порту. Це можливо, якщо на сервері необхідно підтримувати різні версії однієї і тієї ж СУБД.

Наприклад, для доступу до БД deanery на локальному комп'ютері рядок URL буде виглядати так:

1. Для PostgreSQL:

localhost/deanery

localhost:5432/deanery

2. Для MySQL:

localhost/deanery

localhost:3036/deanery

Для повної ідентифікації доступу до БД крім URL необхідно вказати ім'я користувача та його пароль. Таким чином, рядок доступу матиме вигляд:

url/user/password

Наприклад, для користувача за замовчуванням:

1. Для PostgreSQL:

localhost/deanery, postgres, qwerty

localhost:5432/deanery, postgres, qwerty

2. Для MySQL:

localhost/deanery, root, qwerty

localhost:3306/deanery, root, qwerty

Примітка. Щоб програма була незалежною від конкретного драйвера, доцільно задати драйвер, url, user та password в окремому текстовому файлі та завантажувати його на початку роботи програми.

24.8. Доступ до бази даних в Java. JDBC

У Java доступ до бази даних базується на концепції драйверів, які дозволяють отримувати з'єднання з базою даних за спеціально описаним URL. З'єднання з БД на Java – JDBC (Java DataBase Connectivity) – це незалежний від платформи промисловий стандарт взаємодії Java-додатків з різними СУБД, реалізований у вигляді пакету java.sql, що входить до складу Java (рис. 24.9).

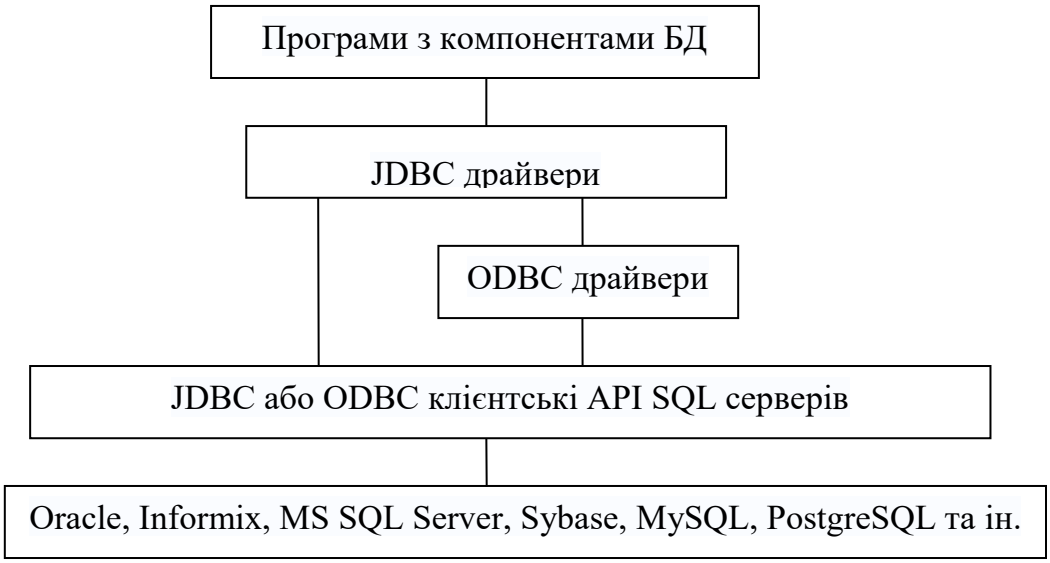


Рис. 24.9. Доступ до БД із прикладних програм у JDBC

У випадку, якщо JDBC не містить драйвера до конкретної СУБД, можна використовувати драйвер-міст JDBC-ODBC, а драйвери ODBC є практично до всіх СУБД.

Будь-яка програма, яка планує використовувати JDBC, повинна імпортувати пакети java.sql.*:

import java.sql.*;

Варто підкреслити, що при завантаженні драйвера з програми вона стає залежною від конкретної СУБД. Для переходу до іншої СУБД у цьому випадку потрібно перекомпілювати програму.

Для завантаження драйвера JDBC з програми слід виконати:

1. Для PostgreSQL:

Class.forName ("org.postgresql.Driver");

2. Для MySQL:

Class.forName ("com.mysql.jdbc.Driver");

3. Для драйвера-моста JDBC-ODBC:

Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");

До будь-якої БД можна підключитися через її URL. З'єднання з БД описується класом, що реалізує інтерфейс java.sql.Connection:

Connection connect = DriverManager.getConnection(url, user, password);

Тут url – рядок, за яким JDBC визначає з чим, де і чим встановлювати з'єднання. Він має один із наступних форматів:

1. Для прямого звернення до відповідних серверів url буде мати вигляд:
   1.1. PostgreSQL:
   jdbc:postgresql://host/database
   jdbc:postgresql://host:port/database
   1.2. MySQL:
   jdbc:mysql://host/database
   jdbc:mysql://host:port/database
2. Для мосту jdbc:odbc сервер вказувати не потрібно. Він задається в odbc:
   jdbc:odbc: database

Наприклад, для звернення до БД deanery на сервері PostgreSQL з'єднання матиме вигляд:

```
Connection connect =
   DriverManager.getConnection(jdbc:postgresql://localhost:5432/deanery,
                               postgresr, qwerty);
```

Для звернення до БД deanery на сервері MySQL з'єднання матиме вигляд:

```
Connection connect =
   DriverManager.getConnection(jdbc:mysql://localhost:3306/deanery, root,
                               qwerty);
```

Примітка. Щоб програма була незалежною від конкретного драйвера, доцільно задати драйвер, url, user та password в окремому текстовому файлі та завантажувати його на початку програми.

24.9. Доступ до бази даних в Python

У Python, так само як і у Java, доступ до БД ґрунтується на концепції драйверів, що дозволяють отримувати з'єднання з БД за допомогою спеціально описаного URL. Однак у Python відсутній єдиний стандарт щодо взаємодії з БД, і використовуються окремі модулі, які спочатку необхідно встановлювати, а потім імпортувати в конкретну програму. Наприклад, для роботи з PostgreSQL необхідно встановити модуль psycopg2, а для MySQL – модуль mysql.connector.

У разі відсутності в Python драйвера до конкретної СУБД, можна використовувати модуль-міст pyodbc, який забезпечує доступ до драйверів ODBC, а драйвери ODBC існують до всіх СУБД.

Для роботи Python з БД використовується набір правил та інтерфейсів Python DB-API. Окремі деталі реалізації для різних БД можуть відрізнятися, але загальні принципи дозволяють використовувати один і той же підхід при роботі з різними БД.

На рис. 24.10 наведено основні інтерфейси та їх використання з СУБД PostgreSQL. Така схема має місце для PostgreSQL, MySQL, sqlite3 та ще до деяких СУБД.

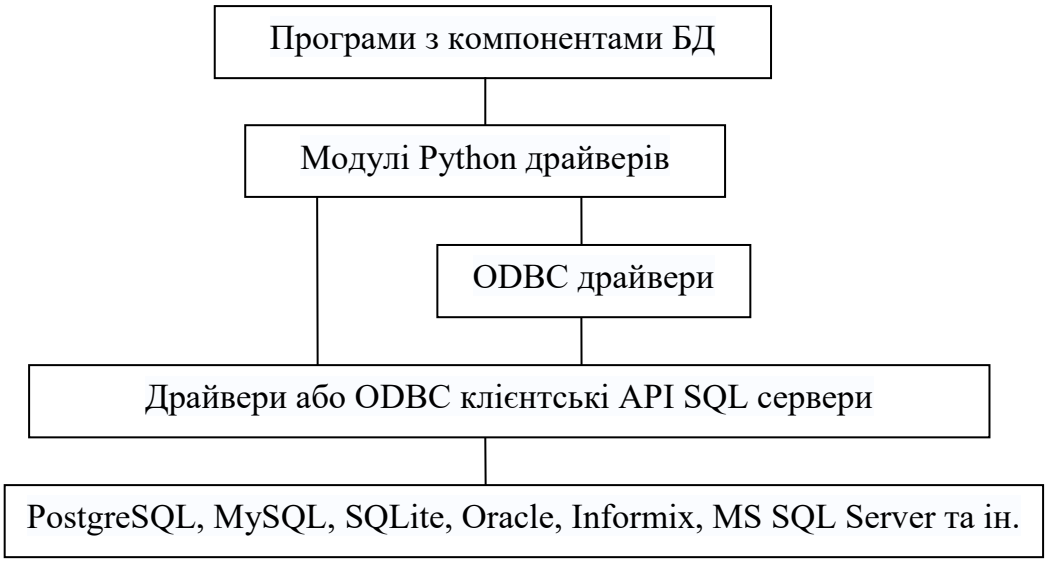


Рис. 24.10. Доступ до БД із прикладних програм в Python

У разі, якщо JDBC не має драйвера до конкретної СУБД, можна використати драйвер-міст JDBC-ODBC, а драйвери ODBC існують практично до всіх СУБД.

Будь-яка програма, яка передбачає роботу з БД PostgreSQL, повинна імпортувати відповідний модуль – драйвер. Наприклад, для PostgreSQL це модуль psycopg2:

```
import psycopg2;
```

Потім необхідно встановити з'єднання з БД:

connection = psycopg2.connect (url, user, password);

Наприклад, для звернення до БД deanery на сервері PostgreSQL з'єднання матиме вигляд:

```
connection = psycopg2.connect(host="localhost", port="5432",
                database="deanery", user="postgres", password="qwerty")
```

Аналогічно для звернення до серверу MySQL необхідно встановити, а потім імпортувати в програму модуль конектор:

import mysql.connector;

Для звернення до БД deanery на сервері MySQL з'єднання матиме вигляд:

```
connection = mysql.connector.connect(host="localhost", port="3306",
                database="deanery", user="root", password="qwerty")
```

Для звернення до довільної СУБД через ODBC необхідно встановити, а потім імпортувати в програму модуль-міст:

import pyodbc;

У цьому випадку всі налаштування з'єднання описано в ODBC, тому для звернення до бази даних необхідно просто вказати назву джерела даних, наприклад Deanery:

connection = pyodbc.Deanery

24.10. Доступ до бази даних в C#

У навчальних цілях тут розглядається найпростіший спосіб доступу до даних, а саме з використанням класів підключених об'єктів. Він ґрунтується на використанні провайдерів доступу, повністю сумісних з ADO.Net, які дозволяють отримувати з'єднання з БД за спеціально описаним URL.

Схему доступу до бази даних в ADO.Net показано на рис. 24.11.

Рис. 24.11. Доступ до БД з прикладних програм в ADO.Net

Для роботи з PostgreSQL можна використовувати провайдер доступу Npgsql, а для роботи з MySQL – провайдер доступу MySqlConnection.

Будь-яка програма, яка передбачає роботу з БД із PostgreSQL, повинна імпортувати відповідний пакет доступу – драйвер. Наприклад, для PostgreSQL це пакет Npgsql:

```
using Npgsql;
```

Потім необхідно встановити з'єднання з БД:

```
connection = new NpgsqlConnection (url, user, password);
```

Наприклад, для звернення до БД deanery на сервері PostgreSQL з'єднання матиме вигляд:

```
connection = new NpgsqlConnection (host="localhost", port=5432,
                database="deanery", user="postgres", password="qwerty");
```

Після чого необхідно відкрити з'єднання:

```
connection.Open();
```

Аналогічно для звернення до серверу MySQL необхідно імпортувати конектор MySqlConnection:

```
using MySqlConnection;
```

Для звернення до БД deanery на сервері MySQL з'єднання матиме вигляд:

```
connection = new MySqlConnection (host="localhost", port=3306,
                database="deanery", user="root", password="qwerty");
```

Після чого необхідно відкрити з'єднання:

```
connection.Open();
```

Для звернення до довільної СУБД через ODBC необхідно імпортувати в програму модуль-міст:

using System.Data.Odbc;

Для звернення до БД deanery на сервері PostgreSQL через ODBC з'єднання матиме вигляд:

```
connection = new OdbcConnection(host="localhost", port=5432,
                                database="deanery", user="postgres", password="qwerty");
```

Після чого необхідно відкрити з'єднання:

```
connection.Open();
```

24.11. SQL і мови програмування

У сучасних системах програмування MS Visual Studio та інших у рамках ADO і ADO.Net для взаємодії з БД і формування SQL-запитів створені спеціальні компоненти.

У мовах Java, Python та інших через їх орієнтацію на підтримку декількох платформ відсутні спеціальні компоненти для роботи з БД. Для взаємодії з БД використовуються об'єкти типу Statement (Java), cursor (Python) і подібні, що використовуються для виконання запитів до БД мовою SQL.

24.11.1. Взаємодія з базою даних в Java

Для роботи з БД в Java необхідно виконати наступні дії:

1. Підключити до програми потрібний драйвер.
2. Виконати з'єднання з БД.
3. Створити об'єкт типу Statement, через який виконуватиметься подальша взаємодія з БД.
4. Сформувати запит – об'єкт типу ResultSet.
5. Отримати та обробити результат.

Існують 3 види об'єкта Statement, що відрізняються за призначенням:

java.sql.Statement – служить для виконання непараметричних запитів загального призначення;

java.sql.PreparedStatement – служить для виконання запитів, що містять параметри, що підставляються (позначаються символом '?' у тілі запиту);

java.sql.CallableStatement – призначений для виклику збережених процедур (див. нижче).

Інтерфейс java.sql.ResultSet дозволяє легко обробляти результати запиту.

Розглянемо на прикладі запит до БД в Java:

```
import java.sql.*;
// Завантаження драйвера
Class.forName ("org.postgresql.Driver");
//Class.forName ("com.mysql.jdbc.Driver");
//Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
// Формування URL
String URL = "jdbc:postgresql://localhost:5432/deanery"; // для БД в PostgreSql
//String URL = "jdbc:mysql://localhost:3306/deanery"; // для БД в MySql
//String URL = "jdbc:odbc:Deanery"; // ODBC для БД в PostgreSql
// Створення з'єднання з PostgreSQL
Connection dbc = DriverManager.getConnection(URL, "postgres", "qwerty");
// Виконання непараметричного запиту
// Об'єкт типу Statement
Statement st = dbc.createStatement();
// Виконання та отримання результату запиту
ResultSet rs = st.executeQuery("select * from students where std_stipend > 600");
// Обробка та вивід вмісту 2-х стовпців
while (rs.next()) {
        System.out.println(rs.getString(1));
        System.out.println(rs.getString("std_name")+"\t"+rs.getString("std_stipend
        "));
}
rs.close();
st.close();
// Виконання параметричного запиту
// Створення PreparedStatement
PreparedStatement pst = dbc.createPreparedStatement();
// Формування тексту запиту з двома параметрами
String SQLTxt := "select  std_name, std_brth_date, std_stipend  +
                  "from students "  +
                  "where std_stipend > ? and grp_nmb = ?";
```

Перед виконанням об'єкта PreparedStatement необхідно встановити значення всіх його параметрів за допомогою методів setXXX, де XXX – це тип

параметра. Наприклад, якщо параметр має тип int у Java, використовується метод setInt.

Перший аргумент методів setXXX – це порядковий номер параметра, а другий – значення, яке треба встановити. Наприклад, наступний код встановлює перший параметр у значення 600, а другий – у значення 2:

```
pst.setLong(1, 600);
pst.setInt(2, 2);
```

Після встановлення параметра його можна використовувати при багаторазовому виконанні виразу доти, поки він не буде очищений методом clearParameters.

Приклад виконання та результату запиту до БД

```
//виконання та результат запиту

ResultSet rs = pst.executeQuery(SQLTxt);

/ / Обробка. Виведення вмісту 2-х стовпців

while (rs.next()) { System.out.println(rs.getString("std_name") + "\t" +
                    rs.getString("std_stipend")); }
rs.close();
pst.close();
```

24.11.2. Взаємодія з базою даних у Python

Для роботи в Python з БД PostgreSQL треба виконати наступні дії:

1. Імпортувати у програму пакет psycopg2.
2. З'єднатися з БД.
3. Отримати курсор. Через курсор проходитиме подальше спілкування з БД.
4. Створити запит та обробити його результати.

Приклад запиту до БД в Python:

```
   # Імпорт у програму пакету psycopg2
import psycopg2
   # З’єднання з БД
connection = psycopg2.connect(host=" localhost", port="5432",
database="deanery", user="postgres", password="qwerty")
   # Отримання курсору. Через курсор проходить подальше
   # спілкування з БД
cursor = connection.cursor()
```

```
   # Створення та виконання запиту
cursor.execute("select st_did, stdname, std_nmb from students
                    order by std_name")
   # Отримання результату запиту
row = cursor.fetchone()
   # Обробка результату запиту
while row is not None:
        print(row)
     ...
     row = cursor.fetchone()
...
```

24.11.3. Взаємодія з базою даних в C#

Для роботи в C# з БД PostgreSQL треба виконати наступні дії:

1. Імпортувати у програму пакет Npgsql.
2. З'єднатися з БД.
3. Відкрити з'єднання. Через нього проходитиме подальше спілкування з БД.
4. Створити запит та обробити його результати.

Приклад запиту до БД в C #:

```
   // Імпорт в програму пакету Npgsql
using System;
using System.Data;
using Npgsql;
class Sample1
{
   // Отримати з’єднання з БД
  NpgsqlConnection con = new NpgsqlConnection ("Host=127.0.0.1;
                                  Port=5432; Database=deanery;
                                  Username="postgres"; Password="qwerty");
  con.Open();
   // Створити та виконати запит
  NpgsqlCommand cmd = new NpgsqlCommand("select std_id, std_name,
                    Std_nmb from students order by std_name ", con);
  Exequte(cmd);
```

```
   NpgsqlDataReader dr = cmd.ExequteReader();
    // Обробити результат запиту
   while (dr.Read())
     Console.Write("dr[0], dr[1], dr[2]");
     // Console.WriteLine($"dr.GetInt32(0), dr.GetString(1),
                                  dr.GetString(2)");
con.Close();
}
```

24.11.4. Виклик збережених функцій в Java

Виклик збережених функцій та процедур в Java-додатках практично не відрізняється від виконання звичайного SQL-запиту на вибірку даних. Інтеграція з JDBC – це перевага для збережених процедур, оскільки для їх виклику з програми не потрібно використовувати жодні конфігураційні файли.

JDBC підтримує виклик збережених процедур за допомогою класу CallableStatement. Цей клас є підкласом класу PreparedStatement. Рядок, який передається як параметр методу prepareCall() – це специфікація виклику процедури. Вона визначає назву процедури, що викликається, та символи ‘?’, які визначають необхідні параметри. Можливі два варіанти специфікації виклику збереженої функції:

1. Якщо функція повертає значення, тобто працює як функція, то виклик виглядає:

{?= call <procedure-name>[(<arg1>,<arg2>, ...)]}

2. Якщо функція не повертає значення, тобто працює як процедура, то виклик виглядає:

{call <procedure-name>[(<arg1>,<arg2>, ...)]}

Виклик збереженої процедури очевидний. Він аналогічний виклику функції, яка не повертає значення.

Розглянемо варіанти виклику збережених функцій, що наведено у пункті 19.3.

Приклад. Збережена функція має декілька вхідних параметрів і не повертає значень. Додати дані про нового студента:

insStudent(st_no integer, st_name character varying)

connection.setAutoCommit(false);

CallableStatement proc = connection.prepareCall("{call insStudent(? , ?) }");

```
proc.setInt(1, 958977);
proc.setString(2, "Іваненко");
proc.execute();
```

У цій функції використовуються два параметри. Для їх встановлення використовуються методи setInt(1, 958977) і setString(2, "Іваненко"). Тут значення 1-го параметра методу, наприклад setInt, вказує порядковий номер параметра збереженої процедури insStudent, а значення 2-го параметра цього методу є значенням, яке присвоюється відповідному параметру збереженої процедури insStudent. Таким чином, метод setInt(1, 958977) присвоює 1-му параметру процедури insStudent код студента типу даних «ціле число».

Приклад. Збережена функція не має вхідних параметрів і повертає одне значення. Підрахувати суму витрат на стипендію всіх студентів:

```
getSumStipAll()
connection.setAutoCommit(false);
CallableStatement proc = connection.prepareCall("{ ? = call getSumStipAll() }");
proc.registerOutParameter(1, Types.REAL);
proc.execute();
float sum = proc.getFloat(1);
```

За допомогою метода registerOutParameter() реєструється SQL-тип вихідного параметра. У Java замість типу real використовується тип float, тому для отримання результату функції використовується метод getFloat().

Приклад. Функція з одним вхідним параметром і повертає одне значення. Підрахувати суму витрат на стипендію всіх студентів групи X, де X – параметр, що визначає назву групи:

```
getStipGroup(gr_name character varying)
connection.setAutoCommit(false);
CallableStatement proc = connection.prepareCall("{? = call getStipGroup(?)}");
proc.registerOutParameter(1, Types.REAL);
proc.setString(2, "КС-971");
proc.execute();
float sum = proc.getFloat(1);
```

У цій функції використовується один параметр. Для його встановлення використовується метод setString(2, "КС-971"), де значення 2 вказує, що другому параметру збереженої процедури, тобто першому параметру функції getStipGroup, необхідно присвоїти рядкове значення.

Приклад. Функція з одним вхідним параметром і повертає множину елементів (записів) таблиці:

```
getStudGroup(gr_name character varying)
connection.setAutoCommit(false);
CallableStatement proc = connection.prepareCall("{? = call getStudGroup(?)}");
proc.registerOutParameter(1, Types.OTHER);
proc.setString(2, "КС-971");
proc.execute();
ResultSet rs = (ResultSet) proc.getObject(1);
while (rs.next()){
    String name = rs.getString(3);
    float stip = proc.getFloat(4);
    System.out.println("Студент" + name + "стипендія" + stip);
}
rs.close();
```

Результатом цієї функції є множина рядків з таблиці Student. Множину рядків представляє об'єкт класу ResultSet. За допомогою метода registerOutParameter(1, Types.OTHER) реєструється тип вихідного параметра як Types.OTHER. У Java для зберігання об'єктів цього типу використовується клас Object. Після виклику збереженої процедури необхідно виконати зворотне перетворення до типу ResultSet. Далі використовується стандартний спосіб обходу записів об'єкта ResultSet.

24.11.5. Виклик збережених функцій у Python

У Python можна викликати збережені функції PostgreSQL за допомогою бібліотеки psycopg2, яка надає інтерфейс для взаємодії з PostgreSQL.

Приклад. Виклик збереженої функції PostgreSQL у Python за допомогою psycopg2:

```
import psycopg2
from psycopg2 import Error

try:
    # З'єднання з базою даних
    connection = psycopg2.connect(
        user="your_username",
```

```
        password="your_password",
        host="your_host",
        port="your_port",
        database="your_database"
    )

    cursor = connection.cursor()

    # Виклик збереженої функції
    cursor.callproc('your_stored_function', (param1, param2, ...))  # Передайте параметри, які потрібно передати у функцію

    # Отримати результат
    result = cursor.fetchone()
    print("Result:", result)

    # Підтвердити транзакцію
    connection.commit()

except (Exception, Error) as error:
    print("Error while working with PostgreSQL", error)

finally:
    # Закрити з'єднання з базою даних
    if connection:
        cursor.close()
        connection.close()
        print("PostgreSQL connection is closed")
```

У цьому прикладі your_stored_function - це назва збереженої функції у PostgreSQL, і param1, param2, ... - це параметри, які передаються у функцію.

24.11.6. Виклик збережених функцій в C#

Виклик збережених функцій та процедур у додатках C# практично не відрізняється від виконання звичайного SQL запиту на вибірку даних.

C# підтримує виклик збережених процедур за допомогою властивості CommandType.StoredProcedure об'єкта команди – CommandType.StoredProcedure. Ця команда визначає назву процедури, що викликається.

Розглянемо варіанти виклику збережених процедур, що наведені у пункті 19.3.

Приклад. Збережена функція має декілька вхідних параметрів і не повертає значень. Додати дані про нового студента: insStudent(st_no integer, st_name character varying)

У цій функції використовуються 2 параметри. Для їх встановлення використовуються відповідні методи

cmd.Parmetrs["num_std"].Value = 981755;

cmd.Parmetrs["name_std"].Value = 'Петренко';

Виклик процедури insStudent:

```
using System;
using System.Data;
using Npgsql;
   class Sample1
  {
    static void Main(string[] args)
    {
       // З’єднання з БД в PostgreSQL
       NpgsqlConnection conn = new NpgsqlConnection("Server=127.0.0.1;
              User Id=postgres; "Password=qwerty; Database=deanery;");
       con.Open();
       //  Старт транзакції
       NpgsqlTransaction tran = conn.BeginTransaction();
       // Визначення команди для виклику процедури.
       NpgsqlCommand cmd = new NpgsqlCommand("insStudent", con);
       // Визначення типу команди – виклик процедури.
       cmd.CommandType = CommandType.StoredProcedure;
       // Задаємо параметри процедури.
       cmd.Parmetrs.Add(new SqlParametr("num_std", SqlDbType.Int));
       cmd.Parmetrs.Add(new SqlParametr("name_std",
                                    SqlDbType.NVarChar, 20));
       // Задаємо значення параметрів процедури.
       //Тут використано передачу параметрів за їх іменами.
       //Можна було передати параметри за номерами їх позицій
```

```
        cmd.Parmetrs["num_std"].Value = 981755;
        cmd.Parmetrs["name_std"].Value = 'Петренко';
        // Виконання процедури
        cmd.ExecuteNonQuery();
        //  Зупинка транзакції
        tran.Commit();
        conn.Close();
      }
    }
```

Приклад. Збережена функція не має вхідних параметрів і повертає одне значення: Підрахувати суму витрат на стипендію всіх студентів – getSumStipAll().

Виклик процедури getSumStipAll():

```
using System;
using System.Data;
using Npgsql;
  class Sample2
  {
    static void Main(string[] args)
    {
      // З'єднання з БД в PostgreSQL
      NpgsqlConnection conn = new NpgsqlConnection("Server=127.0.0.1;
              User Id=postgres; "Password=qwerty; Database=deanery;");
      con.Open();
      //  Старт транзакції
      NpgsqlTransaction tran = conn.BeginTransaction();
      // Визначення команди для виклику процедури.
       NpgsqlCommand cmd = new NpgsqlCommand("getSumStipAll", con);
       cmd.CommandType = CommandType.StoredProcedure;
       // Виконання процедури та отримання набору результатів
       Exequte(cmd);
       NpgsqlDataReader dr = cmd.ExecuteReader();
       // Вивід значення
         Console.Write(dr);
       tran.Commit();
```

```
        conn.Close();
      }
    }
```

Приклад. Функція з одним вхідним параметром і повертає одне значення: Підрахувати суму витрат на стипендію всіх студентів групи X, де X – параметр, що визначає назву групи: getStipGroup(gr_name character varying)

Виклик процедури getStipGroup:

```
using System;
using System.Data;
using Npgsql;
  class Sample1
  {
    static void Main(string[] args)
    {
       // З'єднання з БД в PostgreSQL
       NpgsqlConnection conn = new NpgsqlConnection("Server=127.0.0.1;
              User Id=postgres; "Password=qwerty; Database=deanery;");
       con.Open();
       //  Старт транзакції
       NpgsqlTransaction tran = conn.BeginTransaction();
       // Визначення команди для виклику процедури.
       NpgsqlCommand cmd = new NpgsqlCommand("getStipGroup", con);
       // Визначення типу команди – виклик процедури.
       cmd.CommandType = CommandType.StoredProcedure;
       // Задаємо параметри процедури.
       cmd.Parmetrs.Add(new SqlParametr("name_grp",
                                    SqlDbType.NVarChar, 6));
       // Задаємо значення параметрів процедури.
       //Тут використано передачу параметрів за їх іменами.
       //Можна було передати параметри за номерами їх позицій
       cmd.Parmetrs["name_grp"].Value = 'КС-982';
       // Виконання процедури та отримання набору результатів
       Exequte(cmd);
       NpgsqlDataReader dr = cmd.ExecuteReader();
       // Вивід значення
```

```
        Console.Write(dr);
      //  Стоп транзакції
      tran.Commit();
      conn.Close();
     }
    }
```

Приклад. Функція з одним вхідним параметром і повертає множину елементів (записів) таблиці: getStudGroup(gr_name character varying).

Виклик процедури getStudGroup:

```
using System;
using System.Data;
using Npgsql;
  class Sample1
  {
   static void Main(string[] args)
   {
      // З'єднання з БД в PostgreSQL
      NpgsqlConnection conn = new NpgsqlConnection("Server=127.0.0.1;
            User Id=postgres; "Password=qwerty; Database=deanery;");
      con.Open();
      //  Старт транзакції
      NpgsqlTransaction tran = conn.BeginTransaction();
      // Визначення команди для виклику процедури.
      NpgsqlCommand cmd = new NpgsqlCommand("getStipGroup", con);
      // Визначення типу команди – виклик процедури.
      cmd.CommandType = CommandType.StoredProcedure;
      // Задаємо параметри процедури.
      cmd.Parmetrs.Add(new SqlParametr("name_grp",
                                  SqlDbType.NVarChar, 6));
      // Задаємо значення параметрів процедури.
      //Тут використано передачу параметрів за їх іменами.
      //Можна було передати параметри за номерами їх позицій
      cmd.Parmetrs["name_grp"].Value = 'КС-982';
      // Виконання процедури та отримання набору результатів
      NpgsqlDataReader dr = cmd.ExecuteReader();
```

```
    // Виконання процедури та отримання набору результатів
    Exequte(cmd);
    NpgsqlDataReader dr = cmd.ExecuteReader();
   // Обробити результат запиту
   while (dr.Read())
     Console.Write("dr[0], dr[1], dr[2]");
    //  Стоп транзакції
    tran.Commit();
    conn.Close();
  }
}
```

24.12. Контрольні запитання

1. Що таке СУБД?
2. Що таке системи типу клієнт/сервер?
3. В чому відмінність одно, дво- і трирівневих архітектур систем БД?
4. Які переваги і недоліки клієнт/серверних систем БД?
5. Що таке сервер БД?
6. В чому полягає концепція відкритих систем?
7. Які особливості відкритого зв'язку з БД у технології ODBC?
8. Які особливості відкритого зв'язку з БД у технології ADO?
9. Які особливості відкритого зв'язку з БД у технології ADO.Net?
10. В чому полягає концепція типового способу доступу до БД в різних мовах програмування?
11. Як реалізується доступ до БД на Java?
12. Як реалізується доступ до БД на Python?
13. Як реалізується доступ до БД на C#?
14. Опишіть процес взаємодії з БД в Java.
15. Опишіть процес взаємодії з БД в Python.
16. Опишіть процес взаємодії з БД в C#.
17. Опишіть процес виклику збережених процедур в Java.
18. Опишіть процес виклику збережених процедур в Python.
19. Опишіть процес виклику збережених процедур в C#.
20. Охарактеризуйте SQL і мови програмування та виклик збережених процедур.

24.13. Завдання для самопідготовки

Завдання 24.1. Для обраної мови програмування і БД розробіть програму на зчитування даних з БД.

Завдання 24.2. Для обраної мови програмування і БД розробіть програму на внесення нового запису в обрану таблицю БД.

Завдання 24.3. Для обраної мови програмування і БД розробіть програму на зміну деякого запису в обраній таблиці БД.

Завдання 24.4. Для обраної мови програмування і БД розробіть програму на видалення деякого запису в обраній таблиці БД.

Завдання 24.5. Для обраної мови програмування і БД розробіть програму на виклик деякої збереженої процедури БД.

25. ТРАНЗАКЦІЇ. ПРИЗНАЧЕННЯ І ВИКОРИСТАННЯ

25.1. Транзакції

Транзакція (Transaction) – в інформатиці, група послідовних операцій з чітко визначеними початковою і кінцевою операціями, яка є логічною одиницею роботи з даними.

Іншими словами, транзакція – це операція перенесення БД з одного збереженого стану в інший збережений стан.

Транзакцію можна виконати або повністю і успішно, дотримуючись цілісності даних і незалежно від інших транзакцій, що паралельно виконуються, або не виконати взагалі, тоді вона не повинна мати жодного ефекту.

Транзакції обробляються транзакційними системами, в процесі роботи яких створюється історія транзакцій.

Наведемо ілюстративний приклад транзакції. Перевести з рахунку мобільного телефону номер 067-XXXXXXX на рахунок номер 067-YYYYYYY суму в 10 грошових одиниць. Цього можна досягти, наприклад, наступною послідовністю дій:

1. Розпочати транзакцію.
2. Прочитати баланс на рахунку за номером 067-XXXXXXX (SELECT).
3. Зменшити баланс на рахунку за номером 067-XXXXXXX на 10 грошових одиниць.
4. Зберегти новий баланс рахунку за номером 067-XXXXXXX (UPDATE).
5. Прочитати баланс на рахунку за номером 067-YYYYYYY (SELECT).
6. Збільшити баланс рахунку за номером 067-YYYYYYY на 10 грошових одиниць.
7. Зберегти новий баланс рахунку за номером 067-YYYYYYY (UPDATE).
8. Завершити транзакцію.

Ці дії є логічною одиницею роботи з переносом суми між рахунками, і, таким чином, є транзакцією. Якщо перервати цю транзакцію, наприклад, посередині і не анулювати всі зміни, легко залишити власника рахунку за номером 067-XXXXXXX без 10 одиниць, тоді як власник рахунку за номером 067-YYYYYYY їх не отримає.

Всі операції, що виконуються з даними на SQL сервері, відбуваються в контексті транзакцій.

Таким чином, транзакція – це групова операція, тобто набір дій з БД. Найсуттєвішим для транзакцій є правило або все, або нічого. Якщо під час виконання даного набору дій на якомусь етапі неможливо виконати наступну

дію, то потрібно виконати повернення БД до початкового стану – відкат транзакції. Такий механізм обробки транзакцій забезпечує цілісність БД.

Розглянемо інший приклад. Припустимо, що відношення Students включає додатковий атрибут SumBall, що представляє суму балів (оцінок) студента, за результатами здачі поточної сесії. Значення SumBall для будь-якого студента припускається рівним сумі всіх значень Ball з таблиці Marks, отриманих цим студентом у поточному семестрі.

Складається враження, що обчислення суми балів є одиночною (атомарною) операцією. Насправді додавання нової оцінки в таблицю Marks – це виконання двох оновлень в БД – додавання нової оцінки (INSERT) і обчислення (оновлення) нового значення суми балів (UPDATE). Дійсно, в БД між цими двома оновленнями тимчасово порушується вимога, що значення SumBall для конкретного студента дорівнює сумі всіх значень поля Ball для цього студента у поточному семестрі. Таким чином, неприпустимо, щоб одне з оновлень було виконане, а інше ні, оскільки БД залишиться в неузгодженому стані. У ідеальному випадку мають бути виконані обидва оновлення. Однак не можна забезпечити 100% гарантію, що так і станеться. Не виключена ймовірність того, що система, наприклад, перейду у аварійний стан між двома оновленнями, або ж на другому оновленні відбудеться арифметичне переповнення тощо.

Отже, логічна одиниця роботи – транзакція, не просто одиночна операція системи БД, а скоріше узгодження декількох таких операцій. Загалом, це перетворення одного узгодженого стану БД в інший, при цьому в проміжних точках БД знаходиться в неузгодженому стані.

Система, яка підтримує транзакції, гарантує, що якщо під час виконання деяких оновлень сталася помилка, то всі ці оновлення будуть анульовані. Таким чином, транзакція або виконується повністю, або повністю скасовується.

25.2. Схема управління транзакціями в SQL

Для управління транзакціями в SQL використовуються 3 оператори:

1. **SET TRANSACTION** – встановлення атрибутів транзакції;
2. **COMMIT** – підтвердження виконання транзакції;
3. **ROLLBACK** – відмова від виконання транзакції.

Оператор SET TRANSACTION визначає групу SQL-операторів для пошуку або модифікації даних.

Оператор COMMIT сигналізує про успішне завершення транзакції. Він повідомляє адміністратору транзакцій, що логічна одиниця роботи завершена успішно, БД знову знаходиться (або буде знаходитися) в узгодженому стані, а всі

оновлення, виконані логічною одиницею роботи, тепер можуть бути зафіксовані, тобто стати постійними.

Оператор ROLLBACK сигналізує про неуспішне завершення транзакції. Він повідомляє адміністратору транзакцій, що сталась якась помилка, БД знаходиться в неузгодженому стані і всі оновлення можуть бути скасовані, тобто анульовані.

У деяких СУБД, наприклад, в ORACLE, використовуються додаткові оператори:

1. **SAVEPOINT** – дозволяє створити в транзакції точку збереження, щоб була можливість скасування лише частини роботи, виконаної в транзакції;
2. **ROLLBACK TO** <точка збереження> – відкат до точки збереження. Використовується разом із SAVEPOINT.

Примітка. Аналогічні оператори SAVE TRANSACTION і модифікований ROLLBACK <точка збереження> є у СУБД SYBASE.

Принцип використання транзакцій показаний на рис. 25.1.

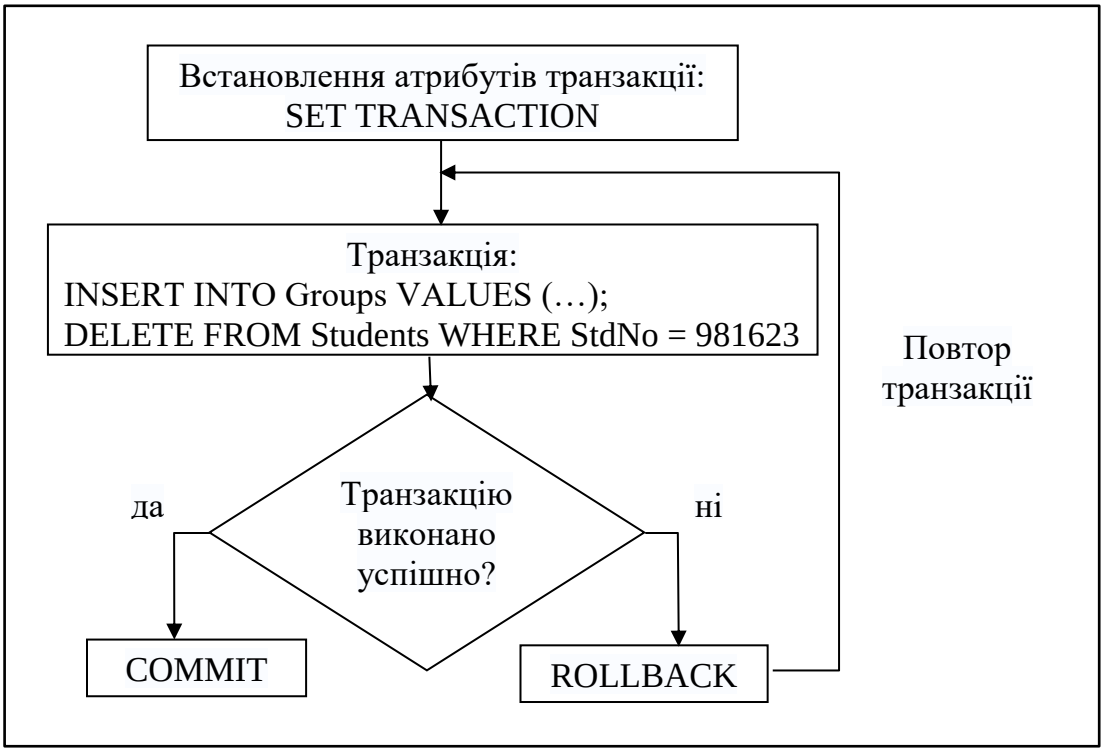


Рис. 25.1. Принцип використання транзакцій

Узагальнена модель транзакцій ANSI/ISO має вигляд як на рис. 25.2.

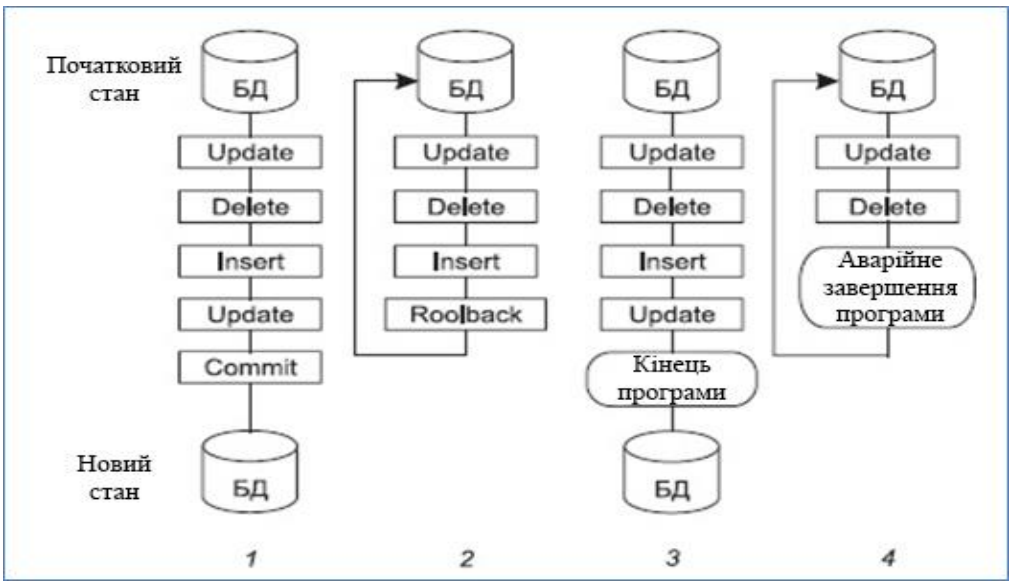


Рис. 25.2. Модель транзакцій ANSI/ISO

25.3. Адміністрування транзакцій

Системний компонент, що забезпечує атомарність (неподільність транзакцій) називається адміністратором або диспетчером транзакцій, а ключами до його виконання є оператори COMMIT і ROLLBACK.

Для скасування оновлень система підтримує журнал (файл) реєстрації змін БД (журнал транзакцій – ЖТ), де записуються деталі всіх операцій оновлення, зокрема, нове і старе значення модифікованого об'єкта. Таким чином, за необхідності скасування деякого оновлення система може використовувати ЖТ для повернення об'єкта в початковий стан.

Загальними принципами відновлення БД після неуспішного завершення транзакції є:

1. Результати зафіксованих транзакцій повинні бути збережені у відновленому стані БД, тобто повинна підтримуватися властивість довготривалості (Durability) транзакцій;
2. Результати незафіксованих транзакцій повинні бути відсутні у відновленому стані БД. В іншому випадку стан БД може залишитися не цілісним.

Це, власне, і означає, що відновлюється останній за часом узгоджений стан БД.

Ще один важливий момент. Система повинна гарантувати, що окремі SQL-оператори є атомарними (тобто виконуються повністю або не виконуються зовсім). Це особливо важливо для реляційних систем, де оператори багаторівневі і зазвичай працюють з множиною кортежів одночасно. Такий оператор просто

не може бути порушений посеред операції і привести систему в неузгоджений стан. Іншими словами, якщо сталась помилка під час роботи такого оператора, БД має залишитися повністю незмінною. Більше того, це має відбутись навіть у тому випадку, коли дії оператора є причиною додаткової, наприклад, каскадної операції.

25.4. Виконання транзакції

Нова транзакція починається на початку кожної сесії роботи з БД. Потім всі SQL-оператори, що виконуються, будуть входити в одну транзакцію до тих пір, поки не буде виконано оператор COMMIT або ROLLBACK.

Коли всі дії, що складають транзакцію, успішно виконано або виникла помилка, транзакцію має бути завершено, щоб БД знаходилась в несуперечливому стані.

Оператор COMMIT завершує транзакцію і встановлює точку фіксації, яка відповідає кінцю логічної одиниці роботи і, отже, точці, в якій БД знаходиться в узгодженому стані.

Встановлення нової точки фіксації виконується, коли всі оновлення, здійснені програмою з моменту встановлення попередньої точки фіксації, успішно виконано. Під час виконання всі такі оновлення можуть розглядатися як пробні, у тому сенсі, що вони можуть бути не виконані, наприклад, скасовані адміністратором.

Оператор ROLLBACK повертає БД в стан, в якому вона була під час операції SET TRANSACTION, тобто у попередню точку фіксації. Транзакція завершується і жодних змін, створених у рамках транзакції, у БД не вноситься. Ця операція виконується при виникненні помилки під час виконання операції (наприклад, неможливості оновити запис).

Система може виконати відкат транзакції у 3 випадках:

1. Індивідуальний відкат транзакції. Тривіальною ситуацією відкату транзакції є її явне завершення оператором ROLLBACK. Також можливі ситуації, коли відкат транзакції ініційовано системою. Прикладами можуть бути виникнення виключної ситуації в прикладній програмі, наприклад, ділення на нуль, або вибір транзакції як жертви при руйнуванні синхронізаційного тупика. Для відновлення узгодженого стану БД при індивідуальному відкаті транзакції потрібно усунути наслідки операторів модифікації БД, які виконувалися в цій транзакції.
2. Відновлення після м'якого збою – раптової втрати вмісту оперативної пам'яті. Така ситуація може виникнути при аварійному вимкненні

електричного живлення, при виникненні непереборного збою процесора (наприклад, спрацювання контролю основної пам'яті) тощо. Ситуація характеризується втратою тієї частини БД, яка на момент збою містилась в буферах оперативної пам'яті СУБД. Відновлення вмісту БД виконується за вмістом ЖТ, що зберігаються на дисках.

3. Відновлення після жорсткого збою – поломки основного зовнішнього носія БД. Ця ситуація при достатньо високій надійності сучасних пристроїв зовнішньої пам'яті може виникати порівняно рідко, але, тим не менш, СУБД повинна бути в змозі відновити БД навіть у цьому випадку. Основою відновлення є архівна копія і ЖТ БД, що зберігаються на непошкоджених зовнішніх носіях.

У всіх трьох випадках основою відновлення є збереження надлишкових даних. Ці надлишкові дані зберігаються в ЖТ, що містить послідовність записів про зміни БД.

Примітка. Більшість комерційних СУБД дозволяє встановлювати режим автоматичної фіксації змін – автокоміт. Для встановлення цього режиму використовується оператор SET AUTOCOMMIT ON, а для скасування режиму – SET AUTOCOMMIT OFF.

Зі сказаного вище видно, що транзакції – це не лише логічні одиниці роботи, але також і одиниці відновлення при невдачі виконання операцій. При успішному завершенні транзакції система гарантує, що оновлення успішно зафіксовано в БД, навіть якщо система зазнає краху в наступний момент.

25.5. Журналювання транзакцій

Можливо, що в системі відбудеться збій після успішного виконання COMMIT, але перед тим, як оновлення будуть фізично записані в БД (вони все ще можуть залишатися в буфері оперативної пам'яті і, таким чином, можуть бути втрачені в момент збою системи).

Навіть якщо подібне сталося, процедура перезавантаження системи все одно повинна встановлювати ці оновлення в БД, враховуючи відповідні записи в ЖТ. З цього випливає, що ЖТ повинен бути фізично записаний перед завершенням операції COMMIT.

Це важливе правило ведення ЖТ відоме як протокол WAL (Write Ahead Log) – протокол попереднього запису в ЖТ, тобто запис про операцію здійснюється перед її виконанням. Таким чином, процедура перезавантаження зможе відновити будь-які успішно завершені транзакції, хоча їх оновлення не

були записані фізично до аварійного збою системи. Отже, як зазначалося раніше, транзакція дійсно є одиницею відновлення.

Можливі два основних варіанти ведення ЖТ. У 1-му варіанті для кожної транзакції підтримується окремий локальний ЖТ. Ці локальні ЖТ використовуються для індивідуальних відкатів транзакцій і можуть підтримуватися в основній – вірніше сказати, в віртуальній пам'яті СУБД. Крім того, підтримується загальний ЖТ, який використовується для відновлення стану БД після м'яких і жорстких збоїв.

Даний підхід дозволяє швидко виконувати індивідуальні відкати транзакцій, але призводить до дублювання інформації в локальних і загальному ЖТ.

У другому варіанті, який використовується частіше, підтримується лише загальний ЖТ, що використовується при виконанні індивідуальних відкатів.

Загальна структура ЖТ умовно може бути представлена у вигляді деякого послідовного файлу, в якому фіксується кожна зміна БД, що відбувається під час виконання транзакції. Усі транзакції мають свої внутрішні номери, тому в єдиному ЖТ фіксуються всі зміни, які здійснюються всіма транзакціями.

Кожен запис у ЖТ позначається номером транзакції, до якої він відноситься, та значеннями атрибутів, які він змінює. Крім того, для кожної транзакції в ЖТ фіксується команда початку та завершення транзакції (рис. 25.3).

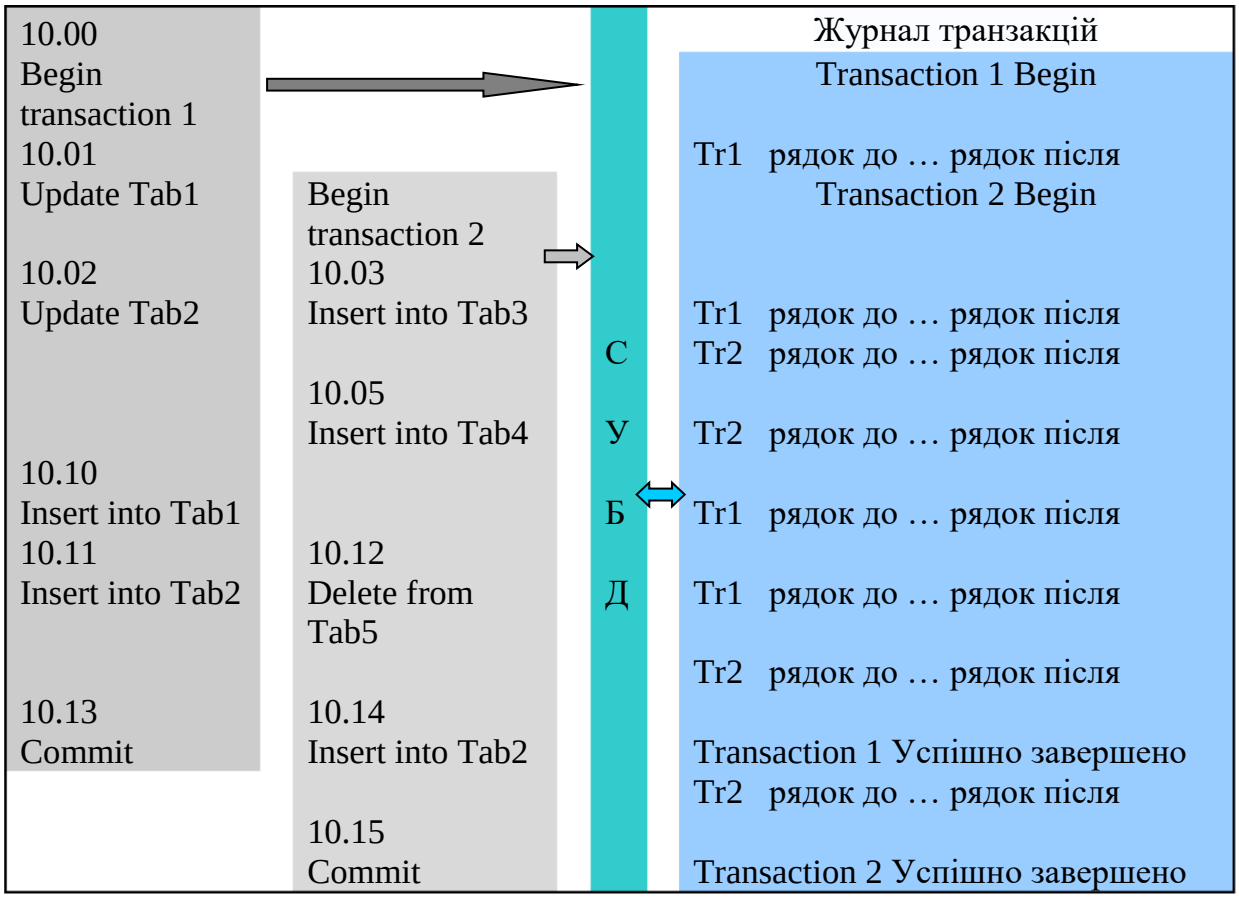


Рис. 25.3. Журнал транзакцій

Для більшої надійності ЖТ часто дублюється системними засобами комерційних СУБД, саме тому обсяг зовнішньої пам'яті в декілька разів перевищує реальний обсяг даних, які зберігаються в сховищі.

Існують 2 альтернативні варіанти ведення ЖТ:

1. Протокол з відкладеними оновленнями.
2. Протокол з негайними оновленнями.

Ведення ЖТ за принципом відкладених оновлень передбачає наступний механізм виконання транзакцій:

1. Коли транзакція T_1 починається, в протокол заноситься запис <T_1 Begin transaction>.
2. Протягом виконання транзакції в протоколі для кожного запису (кортежу), що змінюється, записується нове значення: < T_1, rid, атрибут, нове значення … >. Тут rid – унікальний номер запису в зовнішній пам'яті. Фактично в протокол заносяться і інші дані, наприклад, час запуску операції, користувач, термінал тощо.
3. Якщо всі дії, з яких складається транзакція T_1, успішно виконано, то транзакція частково фіксується і в протокол заноситься < T_1 Commit>.

4. Після того як транзакція T_1 фіксується, записи протоколу, що до неї відносяться, використовуються для внесення відповідних змін в БД.
5. Якщо відбувається м'який або жорсткий збій, то СУБД переглядає протокол і визначає, які транзакції потрібно переробити. Транзакцію T_1 потрібно переробити, якщо протокол містить обидва записи <T_1 Begin transaction> і <T_1 Commit>. БД може знаходитися в неузгодженому стані, однак усі нові значення змінених елементів даних містяться в протоколі, і це вимагає повторного виконання транзакції. Для цього використовується деяка системна процедура, типу REDO(), яка замінює всі значення елементів даних на нові, переглядаючи протокол у прямому порядку.
6. Якщо в протоколі не міститься команда фіксації транзакції Commit, то жодних дій проводити не потрібно, а транзакція запускається знову.

Альтернативний механізм з негайними оновленнями передбачає внесення змін відразу в БД, а в протокол заносяться не лише нові, але й усі старі значення атрибутів, що змінюються, тому кожен запис виглядає наступним чином

< T_1, rid, атрибут, нове значення, старе значення …>

Тут rid – унікальний номер запису в зовнішній пам'яті.

Фактично в протокол заносяться і інші дані, наприклад, час запуску операції, користувач, термінал тощо.

При цьому запис у ЖТ передує безпосередньому виконанню операції над БД. Коли транзакція фіксується, тобто зустрічається команда <T_1 Commit> і вона виконується, то всі зміни вже внесені в БД і немає потреби в подальших діях щодо цієї транзакції.

При відкаті транзакції виконується системна процедура типу UNDO(), яка повертає всі старі значення в скасованій транзакції, послідовно проходячи по протоколу, починаючи з команди Begin transaction.

При відкаті транзакції виконується системна процедура типу UNDO(), яка повертає всі старі значення у скасованій транзакції, послідовно проходячи протоколом, починаючи з команди Begin transaction.

Для відновлення при збої використовується наступний механізм:

1. Якщо транзакція містить команду початку транзакції <T_1 Begin transaction>, але не містить команду фіксації з підтвердженням її виконання <T_1 Commit>, то виконується послідовність дій, як при відкаті транзакції, тобто відновлюються старі значення.

2. Якщо збій відбувся після виконання останньої команди зміни БД, але перед виконанням команди фіксації <T_1 Commit>, то команда фіксації виконується, а у БД жодних змін не відбувається. Робота відбувається лише на рівні протоколу.

Однак слід зауважити, що проблеми відновлення виглядають набагато складніше за наведені раніше алгоритми з урахуванням того, що зміни як в ЖТ, так і в БД заносяться не відразу, а після буферизації.

25.6. Журналювання транзакцій та буферизація сторінок БД

Журналювання змін тісно пов'язане не лише з управлінням транзакціями, але й з буферизацією сторінок БД в оперативній пам'яті.

Якби запис про зміну БД, яка має потрапити в ЖТ при виконанні будь-якої операції модифікації БД, реально негайно записувався б у зовнішню пам'ять, це призвело б до значного уповільнення роботи системи. Тому записи в ЖТ також буферизуються. При нормальній роботі наступна сторінка «виштовхується» у зовнішню пам'ять ЖТ лише при повному заповненні записами.

Проблема полягає в розробці певної загальної політики виштовхування буфера ЖТ та буферів сторінок БД, яка забезпечувала б можливість відновлення стану БД після збоїв.

Проблема не виникає при індивідуальних відкатах транзакцій, оскільки в цих випадках вміст оперативної пам'яті не втрачено і можна скористатися вмістом як буфера ЖТ, так і буферів сторінок БД. Але якщо відбувся м'який збій і вміст буферів втрачено, для відновлення БД необхідно мати певний узгоджений стан ЖТ та БД у зовнішній пам'яті.

Основним принципом узгодженої політики виштовхування буфера ЖТ та буферів сторінок БД є те, що запис про зміну об'єкта БД повинен потрапляти у зовнішню пам'ять ЖТ раніше, ніж змінений об'єкт опиняється у зовнішній пам'яті БД. Відповідний протокол журналювання (і управління буферизацією) називається Write Ahead Log (WAL) – "пиши спершу у ЖТ" і полягає в тому, що якщо потрібно записати у зовнішню пам'ять змінений об'єкт БД, то перед цим потрібно гарантувати запис у зовнішню пам'ять ЖТ запису про його зміну.

Інакше кажучи, якщо у зовнішній пам'яті БД знаходиться певний об'єкт БД, стосовно якого виконано операцію модифікації, то у зовнішній пам'яті ЖТ обов'язково знаходиться запис, який відповідає цій операції. Зворотне невірно, тобто, якщо у зовнішній пам'яті ЖТ знаходиться запис про певну операцію зміни об'єкта БД, то сам змінений об'єкт може бути відсутнім у зовнішній пам'яті БД.

Додаткова умова на виштовхування буферів накладається вимогою, що кожна транзакція, що успішно завершилась, повинна бути реально зафіксована у зовнішній пам'яті. Який би збій не стався, система повинна бути в змозі відновити стан БД, який містить результати усіх зафіксованих до моменту збою транзакцій.

Простим рішенням було б виштовхування буфера ЖТ, за яким слідує масове виштовхування буферів сторінок БД, що змінювалися даною транзакцією. Досить часто так і роблять, але це призводить до значних накладних витрат при виконанні операції фіксації транзакції.

Виявляється, що мінімальною вимогою, яка гарантує можливість відновлення останнього узгодженого стану БД, є виштовхування при фіксації транзакції у зовнішню пам'ять ЖТ всіх записів про зміни БД цією транзакцією. Водночас останнім записом у ЖТ, який виробляється від імені даної транзакції, є спеціальний запис про кінець транзакції.

Розглянемо тепер, як можна виконувати операції відновлення БД в різних ситуаціях, якщо у системі підтримується загальний для всіх транзакцій ЖТ з загальною буферизацією записів, що підтримується відповідно до протоколу WAL.

25.7. ACID властивості транзакцій

Транзакції мають чотири важливі властивості: атомарність, узгодженість, ізоляція та стійкість. Їх звичайно називають властивостями АСІД або ACID.

1. *Атомарність (Atomicity)*. Транзакції виконуються за принципом «все або нічого».
2. *Узгодженість (Consistency)*. Транзакції захищають БД узгоджено. Це означає, що транзакції переводять один узгоджений стан БД до іншого без обов'язкової підтримки узгодженості в усіх проміжних точках.
3. *Ізоляція (Isolation)*. Транзакції відокремлено одна від одної. Це означає, що проміжні результати певної транзакції повинні бути закриті від інших активних транзакцій, доки ця транзакція виконується.
4. *Довговічність, стійкість (Durability)*. Коли транзакцію виконано, її оновлення зберігаються, навіть якщо в наступний момент відбудеться збій системи. Це підтримується за допомогою ЖТ.

25.8. Контрольні запитання

1. Що таке транзакції і в чому особливості їх виконання?
2. В чому полягає схема управління транзакціями і якими операторами SQL вона реалізується?
3. Що таке журнал транзакцій?
4. В чому полягають загальні принципи відновлення БД після неуспішного завершення транзакції?
5. Якою є процедура виконання транзакцій?
6. В яких випадках виконується відкат транзакції?
7. Що таке журналювання транзакцій?
8. Охарактеризуйте правила ведення журналу транзакцій.
9. Що таке буферизація сторінок БД в оперативній пам'яті?
10. Що таке ACID властивості транзакцій?
11. Дайте визначення поняття атомарність транзакцій.
12. Дайте визначення поняття узгодженості транзакцій.
13. Дайте визначення поняття довговічність чи стійкість транзакцій.

26. ПАРАЛЕЛІЗМ ТРАНЗАКЦІЙ. БЛОКУВАННЯ. ВІДНОВЛЕННЯ БАЗИ ДАНИХ

26.1. Проблеми паралелізму транзакцій

Термін паралелізм транзакцій означає можливість одночасної обробки в СУБД багатьох транзакцій з доступом до одних і тих самих даних, при чому в один і той же час. У такій системі для коректної обробки паралельних транзакцій без виникнення конфліктних ситуацій необхідно використовувати деякі методи управління паралелізмом.

Кожен метод управління паралелізмом призначено для вирішення конкретної задачі запобігання конфліктних ситуацій. Тим не менше, при обробці правильно складених транзакцій виникають конфліктні ситуації, які можуть призвести до отримання неправильного результату через взаємні перешкоди серед деяких транзакцій. При цьому транзакція, яка створює перешкоду, може бути правильною. *Неправильний кінцевий результат виникає через безконтрольне чергування операцій із двох правильних транзакцій*.

При паралельному використанні транзакцій можуть виникати наступні 4 проблеми:

1. Неповторюване читання (Non-repeatable Read);
2. Брудне читання (Dirty Read);
3. Втрачене оновлення (Lost Update);
4. Фантомна вставка (Phantom Insert).

Розглянемо ситуації, в яких можливе виникнення даних проблем.

26.1.1. Неповторюване читання

Проблема неповторюваного читання є наслідком неодноразового зчитування транзакцією одних і тих самих даних. Під час виконання транзакції T_1 транзакція T_2 може внести в дані зміни, тому при повторному читанні транзакція T_1 отримає вже інший набір даних, що призводить до порушення їх цілісності або логічної неузгодженості.

Припустимо, що є 2 транзакції, відкриті різними додатками, в яких виконані SQL-оператори, що наведено у таблиці 26.1.

Таблиця 26.1. Втрата в момент часу t_4 результатів оновлення, виконаного транзакцією T_1

Транзакція T_1	Час	Транзакція T_2
SELECT F_2 **FROM** Tb_1 **WHERE** F_1=1;	t_1	
Очікування	t_2	**SELECT** F_2 **FROM** Tb_1 **WHERE** F_1=1;
UPDATE Tb_1 **SET** F_2=F_2+1 **WHERE** F_1=1;	t_3	Очікування
Очікування	t_4	**SELECT** F_2 **FROM** Tb_1 **WHERE** F_1=1;

У транзакції T_2 вибирається значення поля F_2, потім у транзакції T_1 змінюється значення поля F_2. При повторної спробі вибору значення з поля F_2 у транзакції T_2 буде отримано інший результат. Ця ситуація особливо неприйнятна, коли дані зчитуються з метою їх часткової зміни та зворотного запису в БД.

26.1.2. Брудне читання або незафіксовані залежності

Проблема брудного читання (незафіксованих залежностей) можлива в тому випадку, якщо користувач виконує складні операції обробки даних, які вимагають множинного змінення даних перед тим, як вони набудуть логічно вірного стану. Якщо під час зміни даних інший користувач їх зчитує, то може виявитися, що він отримає логічно невірну інформацію.

Для уникнення подібних проблем необхідно проводити зчитування даних після завершення всіх змін. Іншими словами, *брудне читання або проблема незафіксованої залежності – це читання даних, що були записані транзакцією, для якої потім було виконано відкат*.

Припустимо, що є дві транзакції, відкриті різними додатками, в яких виконані SQL-оператори, наведено у таблиці 26.2.

Таблиця 26.2. Транзакція T_2 стає залежною від невиконаної зміни на момент часу t_2

Транзакція T_1	Час	Транзакція T_2
SELECT F_2 **FROM** Tb_1 **WHERE** F_1=1;	t_1	
UPDATE Tb_1 **SET** F_2=F_2+1 **WHERE** F_1=1;	t_2	
Очікування	t_3	**SELECT** F_2 **FROM** Tb_1 **WHERE** F_1=1;
ROLLBACK;	t_4	Очікування

У транзакції T_1 змінюється значення поля F_1, а потім у транзакції T_2 вибирається значення поля F_2. Після цього відбувається відкат транзакції T_1. У результаті значення, отримане транзакцією T_2, буде відрізнятися від значення, що зберігається в БД. Транзакція T_2 стає залежною від невиконаної зміни у момент часу t_2.

Дещо змінений приклад наведено у таблиці 26.3.

Таблиця 26.3. Транзакція T_2 оновлює невиконану зміну на момент часу t_2, і результати цього оновлення втрачаються на момент часу t_2

Транзакція T_1	Час	Транзакція T_2
UPDATE Tb$_1$ **SET** $F_2=F_2+1$ **WHERE** $F_1=1$;	t_1	
Очікування	t_2	**UPDATE** Tb$_1$ **SET** $F_2=F_2+2$ **WHERE** $F_1=1$;
ROLLBACK;	t_3	Очікування

Тепер транзакція T_2 оновлює невиконану зміну у момент часу t_2, і результати цього оновлення втрачаються у момент часу t_3. У цьому випадку не лише транзакція T_2 стає залежною від зміни, не виконаної у момент часу t_2, але також у момент часу t_3 фактично втрачається результат оновлення, оскільки скасування виконання транзакції T_1 у момент часу t_3 призводить до відновлення кортежу p до початкового значення у момент часу t_1. Це є ще одним варіантом проблеми втрати результатів оновлення.

26.1.3. Втрачене оновлення

Проблема втраченого оновлення або останньої зміни виникає, коли декілька користувачів змінюють один і той же рядок таблиці, ґрунтуючись на його початковому значенні. У цьому випадку частину даних рядка буде втрачено, оскільки кожна наступна транзакція перезапише зміни, зроблені попередньою. Вихід з цієї ситуації полягає у послідовному внесенні змін.

Припустимо, що є дві транзакції, відкриті різними додатками, в яких виконано деякі SQL-оператори (див. таблицю 26.4).

Таблиця 26.4. Втрата в момент часу t_4 результатів оновлення, виконаного транзакцією T_1

Транзакція T_1	Час	Транзакція T_2
SELECT F_2 **FROM** Tb_1 **WHERE** F_1=1;	t_1	
Очікування	t_2	**SELECT** F_2 **FROM** Tb_1 **WHERE** F_1=1;
UPDATE Tb_1 **SET** F_2=20 **WHERE** F_1=1;	t_3	Очікування
Очікування	t_4	**UPDATE** Tb_1 **SET** F_2=25 **WHERE** F_1=1;
SELECT F_2 **FROM** Tb_1 **WHERE** F_1=1;	t_5	

У транзакції T_1 змінюється значення поля F_2, а потім у транзакції T_2 також змінюється значення поля F_2. У результаті зміни поля F_2, виконані транзакцією T_1, буде втрачено і, якщо у транзакції T_1 знову відбудеться зчитування поля F_2, то виникне ситуація читання, що не повторюється.

26.1.4. Фантомна вставка або проблема несумісного аналізу

Проблема фантомної вставки або читання фантомів виникає після того, як одна транзакція вибирає дані з таблиці, а інша транзакція вставляє або видаляє рядки до завершення першої транзакції. Вибрані з таблиці значення будуть некоректні.

Припустимо, що є дві транзакції, відкриті різними додатками, в яких виконані SQL-оператори наведено у таблиці 26.5.

Таблиця 26.5. Транзакція T_2 виконала несумісний аналіз

Транзакція T_1	Час	Транзакція T_2
	t_1	**SELECT** SUM(F_2) **FROM** Tb_1;
UPDATE Tb_1 **SET** F_2=20 **WHERE** F_1=1;	t_2	Очікування
	t_3	**SELECT** SUM(F_2) **FROM** Tb_1;

У транзакції T_2 виконується SQL-оператор, що використовує всі значення поля F_2. Потім у транзакції T_1 виконується зміна поля F_2 у одному або декількох кортежах, що призводить до того, що повторне виконання SQL-оператора у транзакції T_2 згенерує інший результат, який очевидно невірний, і, якщо він буде записаний в БД, то в ній може виникнути проблема несумісності. У такому випадку говорять, що транзакція T_2 зіткнулась з несумісним станом і на його основі було виконано несумісний аналіз.

Така ситуація називається фантомною вставкою або проблемою несумісного аналізу і є окремим випадком читання, що не повторюється. При

цьому, якщо виконуваний SQL-оператор вибирає не всі значення поля F_2, а тільки значення одного рядка таблиці (використовується предикат WHERE), то виконання оператора INSERT не приведе до ситуації фантомної вставки.

26.2. Блокування транзакцій

Блокуванням (Locks) називаються механізми, що використовуються для управління паралельними змінами даних.

Якщо для виконання певної транзакції необхідно, щоб певний об'єкт – зазвичай кортеж бази даних – не змінювався непередбачувано без відома цієї транзакції, цей об'єкт блокується.

Отже, ефект блокування полягає в тому, щоб заблокувати доступ до конкретного елемента з боку інших транзакцій, тим самим запобігаючи непередбачуваній зміні цього об'єкта. Таким чином, вихідна транзакція може виконати всю необхідну обробку з урахуванням того, що об'єкт, який обробляється, залишається в стабільному стані настільки довго, наскільки це потрібно.

Існують наступні логічні та фізичні рівні блокування:

1. Блокування на рівні рядків (Row-Level Locking);
2. Блокування на рівні таблиць (Table-Level Locking);
3. Блокування на рівні елемента таблиці (Item-Level Locking);
4. Блокування на рівні БД (DBSpace-Level Locking).

Крім того, деякі СУБД підтримують додаткові види блокування. Наприклад, СУБД Oracle підтримує наступні додаткові види блокування:

1. Блокування на рівні табличного простору (Tablespace-Level Locking).
2. Блокування на рівні блока або сторінки (Page-Level Locking) – фізичної одиниці дискового простору.

Управління блокуваннями на SQL-сервері здійснює менеджер блокувань, який контролює їх застосування та розв’язує конфлікти.

Транзакції та блокування тісно пов'язані між собою. Транзакції накладають блокування на дані, щоб забезпечити виконання вимог ACID. Без використання блокувань декілька транзакцій можуть змінювати одні й ті самі дані.

Виділяють два типи блокувань:

1. Оптимістичні блокування (Optimistic Locks) – уникнення конфліктних ситуацій шляхом відкату транзакції за необхідності (такі блокування не підтримуються в стандарті SQL92);

2. Песимістичні блокування (Pessimistic Locks) – уникнення конфліктних ситуацій, блокуючи одночасний доступ до даних для одночасних транзакцій.

Якщо при песимістичному блокуванні виконується SQL-оператор, який може призвести до конфліктної ситуації для іншого SQL-оператора, то виконання другого SQL-оператора буде призупинено.

При оптимістичному блокуванні можуть виконуватися будь-які SQL-оператори, але у разі виникнення конфліктної ситуації всі зміни буде втрачено.

У SQL розв'язання проблем, що виникають через можливість паралельного виконання транзакцій, базується на блокуваннях, які діляться на:

1. Блокування з взаємним доступом або роздільні блокування (S-locks, Shared Locks), які можуть встановлюватися одночасно декількома користувачами;
2. Блокування без взаємного доступу або виключні блокування (X-locks, Exclusive Locks), які встановлюються лише одним користувачем, який отримує ексклюзивний доступ до даних.

Примітка. X-блокування та S-блокування іноді називають блокуваннями запису та читання відповідно.

Примітка. У комерційних СУБД присутні також інші види блокувань.

Механізм функціонування X-блокувань та S-блокувань у випадку, коли об'єктом, що блокується, є кортеж, має наступний вигляд:

1. Якщо транзакція T_1 блокує кортеж p без можливості взаємного доступу (X блокування), то запит іншої транзакції T_2 з блокуванням цього кортежу p буде скасовано.
2. Якщо транзакція T_1 блокує кортеж p з можливістю взаємного доступу (S блокування), то:
   2.1. запит зі сторони деякої транзакції T_2 на X блокування кортежу буде відхилено.
   2.2. Запит зі сторони деякої транзакції T_2 на S блокування кортежу p буде прийнято, тобто транзакція T_2 також буде блокувати кортеж p за допомогою S блокування.

Слід підкреслити, що більшість SQL-серверів використовують технологію блокувань за будь-якої модифікації даних. Іншими словами, дані, змінені в транзакції одним з користувачів, недоступні для інших користувачів до завершення цієї транзакції.

Зовсім уникнути блокувань неможливо, проте можна зменшити тривалість транзакцій:

1. Перед виконанням транзакції для неї повинні бути вже підготовлені всі необхідні дані. Усередині транзакції не повинно бути жодних звернень до зовнішніх пристроїв – файлів, мережі тощо, а мають бути звернення лише до БД.
2. Ситуація, коли всередині транзакції відбувається діалог з користувачем, тим паче є недопустимою.
3. Транзакції повинні бути якомога коротшими, що дає можливість більшому числу користувачів одночасно працювати з БД, не конфліктуючи один з одним.

26.3. Вирішення проблем паралелізму транзакцій

Блокування – це метод управління паралельними процесами, при якому об'єкт бази даних не може бути змінений без відома транзакції, тобто відбувається блокування доступу до об'єкта з боку інших транзакцій, що виключає непередбачувані зміни об'єкту. Як показано вище, існують 2 види блокувань:

1. Блокування запису – транзакція блокує рядки в таблицях таким чином, що запит іншої транзакції до цих рядків буде скасовано;
2. Блокування читання – транзакція блокує рядки так, що запит з боку іншої транзакції на блокування запису цих рядків буде відхилено, а на блокування читання – прийнято.

26.3.1. Протокол доступу до даних – протокол блокування

У СУБД використовують протокол доступу до даних, який дозволяє уникнути проблеми паралелізму. Він полягає в наступному:

1. Транзакція, результатом дії якої на рядок даних в таблиці є її вилучення, зобов'язана накласти S-блокування читання на цей рядок;
2. Транзакція, призначена для модифікації рядка даних, накладає на нього X-блокування запису;
3. Якщо блокування, що запитується, на рядок відхиляється через вже існуюче блокування, то транзакція переводиться в режим очікування до того моменту, поки блокування не буде знято;
4. X-блокування запису зберігається до кінця виконання транзакції.

Вирішення проблеми паралельної обробки БД полягає в тому, що рядки таблиць блокуються, а наступні транзакції, які модифікують ці рядки, відхиляються і переводяться в режим очікування. У зв'язку з властивістю

збереження цілісності БД транзакції є відповідними одиницями ізоляції користувачів. Дійсно, якщо кожна сесія взаємодії з БД реалізується транзакцією, то користувач починає з того, що звертається до узгодженого стану БД – стану, в якому вона могла б знаходитися, навіть якщо б користувач працював з нею один.

Якщо в СУБД не реалізовано механізми блокувань, то при одночасному читанні та зміні тих самих даних декількома користувачами можуть виникнути розглянуті вище проблеми одночасного доступу – неповторного читання, брудного читання, останньої зміни та читання фантомів.

Для вирішення перелічених проблем в спеціально розробленому стандарті SQL92 визначено 4 рівні блокувань, що задаються рівнем ізоляції транзакції. Він визначає, чи можуть інші (конкуруючі) транзакції вносити зміни в дані, змінені поточною транзакцією, а також чи може поточна транзакція бачити зміни, зроблені конкуруючими транзакціями, і навпаки.

Кожен наступний рівень ізоляції транзакції підтримує вимоги попереднього і накладає додаткові обмеження:

Рівень 0 – заборона "забруднення" даних. Цей рівень вимагає, щоб змінювати дані могла лише одна транзакція; якщо іншій транзакції потрібно змінити ті ж дані, вона повинна чекати завершення першої транзакції.

Рівень 1 – заборона "брудного" читання. Якщо транзакція почала змінювати дані, то жодна інша транзакція не зможе прочитати їх до завершення першої.

Рівень 2 – заборона неповторного читання. Якщо транзакція зчитує дані, то жодна інша транзакція не зможе їх змінити. Таким чином, при повторному читанні вони будуть знаходитися в початковому стані.

Рівень 3 – заборона фантомів. Якщо транзакція звертається до даних, то жодна інша транзакція не зможе додати нові або видалити існуючі рядки, які можуть бути зчитані при виконанні транзакції. Реалізація цього рівня блокування виконується за допомогою використання блокувань діапазону ключів. Таке блокування накладається не на конкретні рядки таблиці, а на рядки, що задовольняють певній логічній умові.

Розглянемо приклади розв'язання проблем паралелізму за допомогою механізму блокувань.

26.3.2. Брудне читання

Нижче наведено зміни у відповідних прикладах з таблиці 26.2 та таблиці 26.3. Вони демонструють чергування виконання операцій згідно з описаним

вище протоколом блокування. Операцію для транзакції T_2 в момент часу t_2 не буде виконано. Річ у тім, що вона є неявним запитом з встановленням блокування для кортежу p, а цей запит конфліктує з X-блокуванням, вже встановленим транзакцією T_1. Таким чином, транзакція T_2 переходить у стан очікування до тих пір, поки не буде припинено виконання транзакції T_1 (до операції завершення або відміни виконання транзакції T_1). Потім встановлене транзакцією T_1 блокування буде знято, і транзакцію T_2 можна буде виконати. При цьому транзакція T_2 матиме справу з певним фіксованим значенням (або існуючим до виконання транзакції T_1 при її скасуванні, або отриманим після виконання транзакції T_1). У будь-якому разі транзакція T_2 більше не залежить від незафіксованого оновлення (таблиця 26.6).

Таблиця 26.6. Брудне читання або проблема незафіксованої залежності

Транзакція T_1	Час	Транзакція T_2
SELECT F_2 **FROM** Tb_1 **WHERE** F_1=1;	t_1	
UPDATE Tb_1 **SET** F_2=F_2+1 **WHERE** F_1=1; (встановлення X-блокування для p)	t_2	
Очікування	t_3	**SELECT** F_2 **FROM** Tb_1 **WHERE** F_1=1; (встановлення S-блокування для p)
ROLLBACK; (зняття X-блокування для p)	t_4	Очікування завершення T_1
		Підсумок: Вилучення кортежу p (встановлення S-блокування для p)

Тепер транзакцію T_2 захищено від виконання операцій з незафіксованим зміненням в момент часу t_2.

Аналогічною є таблиця 26.7 для другого прикладу.

Таблиця 26.7. Брудне читання або проблема незафіксованої залежності

Транзакція T_1	Час	Транзакція T_2
UPDATE Tb_1 **SET** F_2=F_2+1 **WHERE** F_1=1; (встановлення X-блокування для p)	t_1	
Очікування	t_2	**UPDATE** Tb_1 **SET** F_2=F_2+2 **WHERE** F_1=1; (встановлення X-блокування для p)
ROLLBACK; (зняття X-блокування для p)	t_3	Очікування завершення T_1
	t_4	Підсумок: Оновлення кортежу p (встановлення X-блокування для p)

Тепер транзакція T_2 є захищеною від виконання операцій з незафіксованим зміненням в момент часу t_2.

26.3.3. Втрачене оновлення

У таблиці 26.8 наведено модифіковану версію процесу, показаного у таблиці 26.5, з урахуванням застосування протоколу блокування для чергових операцій. Операцію оновлення для транзакції T_1 в момент часу t_3 не буде виконано, оскільки вона є неявним запитом з встановленням X-блокування для кортежу p, а цей запит конфліктує з S-блокуванням, вже встановленим транзакцією T_2. Таким чином, транзакція T_1 переходить у стан очікування. З аналогічних причин транзакція T_2 переходить у стан очікування в момент часу t_4. Оновлення тепер не втрачаються, проте виникає нова проблема – нескінченне очікування або тупикова ситуація. Способи вирішення цієї проблеми розглядаються нижче.

Таблиця 26.8. Хоча оновлення не втрачаються, але в момент часу t_4 виникає тупикова ситуація

Транзакція T_1	Час	Транзакція T_2
SELECT F_2 **FROM** Tb_1 **WHERE** F_1=1; (встановлення S-блокування для p)	t_1	
Очікування	t_2	**SELECT** F_2 **FROM** Tb_1 **WHERE** F_1=1; (встановлення S-блокування для p)
UPDATE Tb_1 **SET** F_2=20 **WHERE** F_1=1; (встановлення X-блокування для p)	t_3	Очікування
	t_4	**UPDATE** Tb_1 **SET** F_2=25 **WHERE** F_1=1; (встановлення X-блокування для p)
Очікування		Очікування

26.4. Тупикові ситуації

Як було показано вище, блокування можна використовувати для вирішення 4-х основних проблем, які виникають при паралельній обробці кортежів. На жаль, використання блокувань призводить до виникнення іншої проблеми – тупикової ситуації.

У таблиці 26.9 показано узагальнений приклад проблеми тупикової ситуації, в якому p_1 і p_2 представляють будь-які об'єкти, що блокуються, необов'язково кортежі БД, а вирази типу "блокування ... без взаємного доступу"

представляють будь-які операції з накладанням блокування без взаємного доступу, задані як явно, так і неявно.

Таблиця 26.9. Приклад тупикової ситуації

Транзакція T_1	Час	Транзакція T_2
Блокування p_1 без взаємного доступу	t_1	–
Очікування	t_2	Блокування p_2 без взаємного доступу
Блокування p_2 без взаємного доступу	t_3	Очікування
Очікування	t_4	Блокування p_1 без взаємного доступу
Очікування		Очікування

Тупикова ситуація виникає тоді, коли 2 або більше транзакції одночасно перебувають у стані очікування, при чому для продовження роботи кожна з транзакцій очікує припинення виконання іншої транзакції.

Для виявлення тупикової ситуації слід виявити цикл в діаграмі станів очікування, тобто в переліку транзакцій, які очікують завершення виконання інших транзакцій. Пошук виходу з тупикової ситуації полягає в виборі однієї з заблокованих транзакцій в якості жертви і скасуванні її виконання. Таким чином, з неї знімається блокування, і виконання іншої транзакції може бути відновлено. При виборі транзакції, яку необхідно пожертвувати, сервер виходить з урахуванням мінімальної вартості.

На практиці не всі системи здатні виявляти тупикову ситуацію.

В деяких з них використовується хронометраж виконання транзакцій, і повідомлення про виникнення тупикової ситуації надходить, якщо транзакція не виконується за певний заздалегідь визначений час.

При обробці тупикових ситуацій в деяких системах передбачено автоматичний перезапуск транзакції з початку за умови, що обставини, які призвели до тупикової ситуації, не повторяться знову.

В інших системах до програми, пов'язаної з даною транзакцією, просто надсилається повідомлення про транзакцію-жертву, що викликала тупикову ситуацію, для обробки цієї ситуації у самій програмі.

З точки зору програмування додатків кращим є перший з цих підходів. Але, незважаючи на це, завжди рекомендується вирішувати цю проблему з точки зору користувача.

26.5. Рівні ізоляції транзакцій

Стандарт SQL92 визначає 4 рівні ізоляції транзакцій, які наведено у таблиці 26.10, встановлення яких запобігає розглянутим вище конфліктним ситуаціям при паралельному виконанні транзакцій:

1. SERIALIZABLE – послідовне виконання, використовується за замовчуванням. Цей рівень гарантує запобігання всіх описаних вище конфліктних ситуацій, але при цьому спостерігається найнижчий рівень паралелізму. В результаті, транзакція бачить лише стан БД, що був на момент початку цієї транзакції плюс зміни, внесені самою транзакцією. Це один з двох найчастіше вживаних рівнів ізоляції.
2. REPEATABLE READ – читання, що повторюється. На цьому рівні дозволено виконання операторів INSERT, що призводять до конфліктної ситуації фантомною вставкою. Цей рівень слід використовувати, якщо SQL-оператори, що виконуються, не залежать від додавання нових рядків.
3. READ COMMITTED – фіксоване читання. Цей рівень дозволяє транзакції отримувати різні результати для однакових запитів, але лише після фіксації іншої транзакції, яка призвела до зміни даних. Це один з двох найчастіше вживаних рівнів ізоляції.
4. READ UNCOMMITTED – незафіксоване читання. Тут можливе отримання різних результатів для однакових запитів без врахування фіксації транзакції.

Таблиця 26.10. Формальний опис рівнів ізоляції

Рівень ізоляції транзакції	Запобігання конфліктній ситуації			
	Неповторюване читання (non-repeatable read)	Брудне читання (dirty read)	Втрачене оновлення (lost update)	Фантомна вставка (phantom insert)
SERIALIZABLE	+	+	+	+
REPEATABLE READ	+	+	+	-
READ COMMITTED	-	+	+	-
READ UNCOMMITTED	-	-	+	-

26.6. Визначення параметрів транзакції в SQL

Визначення параметрів транзакції здійснюється оператором SET TRANSACTION, який має наступний формальний опис:

TRANSACTION
{ **ISOLATION LEVEL**
{ **READ UNCOMMITTED**
| **READ COMMITTED**
| **REPEATABLE READ**
| **SERIALIZABLE** }
| { **READ ONLY** | **READ WRITE** }
} .,… ;

Фраза ISOLATION LEVEL вказує відповідний рівень ізоляції:

1. READ UNCOMMITED – незавершене читання або допустиме брудне читання. Це найнижчий рівень ізоляції, що відповідає рівню 0. Він гарантує лише фізичну цілісність даних. Якщо декілька користувачів одночасно змінюють один і той же рядок, то у кінцевому варіанті рядок матиме значення, визначене користувачем, який останнім змінив запис. По суті, для транзакції не встановлюється жодного блокування, яке гарантувало б цілісність даних. Для встановлення цього рівня використовується команда:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. READ COMMITTED – завершене читання, коли відсутнє брудне читання. Тем не менше, під час роботи однієї транзакції інша може бути успішно завершена, а зміни, зроблені нею, зафіксовані. У результаті перша транзакція буде працювати з іншим набором даних. Це проблема неповторного читання. Його установка реалізується командою:

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. REPEATABLE READ – читання, що повторюється. Повторне читання рядка поверне спочатку зчитані дані, незважаючи на будь-які оновлення, зроблені іншими користувачами до завершення транзакції. Тим не менш, на цьому рівні ізоляції можливе виникнення фантомів. Його встановлення реалізується командою:

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. SERIALIZABLE – серіалізованість. Читання заборонено до завершення транзакції. Це максимальний рівень ізоляції, який забезпечує повну ізоляцію транзакцій один від одного. Він встановлюється командою:

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

У кожен момент часу можливий лише один рівень ізоляції.

Фраза READ ONLY встановлює режим, за якого дозволяється лише читання. Цей режим встановлюється за замовчуванням, якщо рівень ізоляції визначено як READ UNCOMMITTED. При режимі READ ONLY на дані не встановлюються жодні блокування.

Фраза READ WRITE встановлює режим, який дозволяє читання та запис даних. При цьому режимі рівень ізоляції не може бути встановлений як READ UNCOMMITTED. Режим READ WRITE встановлюється за замовчуванням для будь-якого рівня, відмінного від READ UNCOMMITTED.

Приклад. Для визначення транзакції, що запобігає всім описаним вище конфліктним ситуаціям, слід виконати SQL-оператор

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE.

26.7. Управління транзакціями в мовах програмування

Зазвичай транзакції реалізуються в прикладних програмах і, відповідно, їх управління залежить від конкретної мови програмування. Для ілюстрації розглянемо управління транзакціями в Delphi та Java.

26.7.1. Управління транзакціями в Java

У Java підтримується два механізми управління транзакціями – JDBC транзакції та JTA (Java Transaction API) транзакції.

JDBC-транзакція управляється менеджером транзакцій конкретної СУБД. Щоб закодувати JDBC-транзакцію, потрібно викликати методи commit та rollback інтерфейсу java.sql.Connection. Початок транзакції відбувається неявно. Транзакція починається з першого SQL оператора, який слідує за останнім оператором commit або rollback. Загальна схема управління JDBC-транзакціями має вигляд:

```
Connection con = DriverManager.getConnection(url, user, password);
            try {
  con.setAutoCommit(false);
          // оператори транзакції
   # Створення та виконання запиту
        cursor.execute("insert into students(std_name) values('Науменко')")
        cursor.execute("insert into students(std_name) values('Василенко')")
  con.commit();
```

```
} catch (Exception ex) {
        try {
            con.rollback();
            throw new EJBException("Transaction failed: " +
        ex.getMessage());
            } catch (SQLException sqx) {
              throw new EJBException("Rollback failed: " +
         sqx.getMessage());
        }
  }
```

JTA-транзакції не залежать від реалізації менеджера транзакцій СУБД. Вони керуються менеджером транзакцій J2EE. JTA-транзакція може оновлювати декілька БД від різних постачальників, тоді як менеджер транзакцій конкретної СУБД може не працювати з гетерогенними БД. Однак менеджер транзакцій J2EE має одне обмеження – він не підтримує вкладені транзакції. Іншими словами, він не може почати транзакцію для екземпляра, поки попередня транзакція не завершиться. Загальна схема управління JTA-транзакціями має вигляд:

```
Transaction tx = session.beginTransaction(); // активація транзакції
try {
tx.begin();
// оператори транзакції
tx.commit();
} catch (Exception ex) {
try {
    tx.rollback();
    } catch (SystemException syex) {
            throw new EJBException ("Rollback failed: " + syex.getMessage());
            }
throw new EJBException ("Transaction failed: " + ex.getMessage());
 }
```

Рівень ізоляції транзакцій можна встановити програмно, використовуючи API, наданий СУБД, що застосовується. Наприклад, СУБД може дозволити виконувати непідтверджені читання за допомогою виклику методу setTransactionIsolation:

```
Connection con;
...
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

26.7.2. Управління транзакціями в Python

Транзакції PostgreSQL обробляються об'єктом з'єднання connection. Він відповідає за збереження змін або відкат у разі невдачі.

За замовчуванням з'єднання connection працює в режимі автоматичного збереження, тобто властивість autocommit дорівнює True. Це означає, що при успішному виконанні запиту зміни негайно зберігаються в БД, а відкат транзакції стає неможливим.

Для виконання запитів в режимі транзакції властивість autocommit потрібно вимкнути. Для цього потрібно виконати наступне налаштування:

connection.autocommit=False

Це дає можливість у разі невдачі виконати відкат до початкового стану.

Якщо всі операції в транзакції успішно завершено, для збереження змін в БД використовується директива connection.commit().

Якщо в транзакції хоча б одна операція завершується невдало, то скасувати зміни можна за допомогою директиви connection.rollback():

```
# Імпорт в програму пакету psycopg2
import psycopg2
   # З’єднання з БД
connection = psycopg2.connect(host="localhost", port="5432",
                    database="deanery", user="postgres", password="qwerty")
   # Отримати курсор
try
    connection.autocommit=False
    cursor = connection.cursor()
     #  Виконати 2-ві різні команди вставки
    cursor.execute("insert into students(std_name) values('Науменко')")
    cursor.execute("insert into students(std_name) values('Василенко')")
    #  Проведення транзакції
   connection.commit();
   print("Транзакція успішно завершена")
except (Exception, psycopg2.DatabaseError) as error :
                print ("Помилка в транзакції. Відміна всіх інших операцій
            транзакції", error)
    #  Відкат транзакції
   connection.rollback()
finally:
```

```
    if connection:
        cursor.close()
        connection.close()
        print("З’єднання з PostgreSQL зачинено")
          ...
```

26.7.3. Управління транзакціями в C#

У ADO.NET управління транзакціями здійснюється за допомогою об'єкта Connection. Ініціювати транзакцію можна за допомогою методу BeginTransaction.

Метод BeginTransaction повертає посилання на транзакцію. Це посилання призначається об'єктам SqlCommand, який є прикріпленим до транзакції.

Після початку транзакції за допомогою властивості Transaction об'єкта Command до неї можна прикріпити команду. Потім в залежності від успіху або помилки компонентів транзакції можна зафіксувати або виконати відкат змін, зроблені в джерелі даних.

Область дії транзакції обмежено з'єднанням.

Приклад транзакції в C#:

```
using System;
using System.Data;
using Npgsql;
  class Sample1
  {
    static void Main(string[] args)
     {
       //  З’єднання з БД в PostgreSQL
       NpgsqlConnection con = new NpgsqlConnection("Server=127.0.0.1;
                User Id=postgres; "Password=qwerty; Database=deanery;");
      con.Open();
       //  Оголошення та старт транзакції tran
       NpgsqlTransaction trans = conn.BeginTransaction();
       //  Визначення команди cmd
       NpgsqlCommand cmd = new NpgsqlCommand("insStudent", con);
       //  Команда cmd прикріпляється до транзакції trans через властивість
```

```
// Transaction
cmd.Transaction = trans;
try
  {
    // Виконати 2-ві різні команди вставки
    command.CommandText =
          "Insert into students(std_name) values('Науменко')";
    command.ExecuteNonQuery();
    command.CommandText =
          "insert into students(std_name) values('Василенко')";
    command.ExecuteNonQuery();

    //  Підтвердження транзакції
    trans.Commit();
    Console.WriteLine("Обидва записи занесено до БД.");
  }
  catch (Exception ex)
{
   // Обробка винятку, якщо транзакцію не вдалося зафіксувати
  Console.WriteLine(ex.Message);
  try
   {
     // Спроба відкотити транзакцію
     trans.Rollback();
   }
   catch (Exception exRollback)
   {
     // Викликає InvalidOperationException, якщо з'єднання
     // закрито або для транзакції вже було виконано відкат на сервері
      Console.WriteLine(exRollback.Message);
   }
```

26.8. Відновлення БД

Відновлення у СУБД передбачає насамперед, відновлення самої БД, тобто повернення БД до правильного стану, якщо будь-який збій зробив поточний стан неправильним або підозрілим.

Основний принцип, на якому будується відновлення БД, – це надмірність даних. Збереження даних організовується на фізичному рівні. Таку надмірність буде приховано від користувача, і, отже, її буде приховано на логічному рівні. Іншими словами, якщо будь-яка частина інформації, що міститься в БД, може бути відновлена з іншої надлишкової інформації, що зберігається в системі, то БД є відновлюваною.

Система має бути готовою до відновлення не лише після невеликих локальних порушень, таких як невиконання операції в межах певної транзакції, але й після глобальних порушень типу збоїв у живленні обчислювального пристрою та інших.

Локальне порушення за визначенням впливає лише на транзакцію, в якій воно виникло.

Глобальне порушення впливає одразу на всі транзакції і, отже, призводить до значних наслідків для системи.

Існує 2 види глобальних порушень:

1. Відмови системи (наприклад, збої в живленні), які впливають на всі транзакції, що виконуються в даний момент, але фізично не порушують БД в цілому. Такі порушення в системі також називають аварійною відмовою програмного забезпечення.
2. Відмови носіїв (наприклад, поломки голівок дискового накопичувача), які можуть загрожувати БД або будь-якій її частині і впливати, принаймні, на ті транзакції, які використовують цю частину БД. Відмови носіїв також називають аварійною відмовою апаратури.

Відновлення системи нерозривно пов'язане з журналюванням. Нижче розглядаються особливості відновлення після різних видів відмов.

26.8.1. Індивідуальний відкат транзакції

Для того, щоб можна було виконати за загальним журналом транзакції індивідуальний відкат транзакції, всі записи в журналі транзакцій зв'язуються у зворотний список. Початком списку для незавершених транзакцій є запис про останню зміну БД, виконану даною транзакцією. Для завершених транзакцій (індивідуальні відкати яких вже неможливі) початком списку є запис про кінець

транзакції, яка обов'язково виштовхнута у зовнішню пам'ять журналу транзакцій. Кінцем списку завжди є перший запис про зміну БД, зроблену даною транзакцією. Зазвичай у кожному записі вказується унікальний ідентифікатор транзакції, щоб можна було відновити прямий список записів про зміни БД даною транзакцією.

Отже, індивідуальний відкат транзакції (ще раз підкреслимо, що це можливо лише для незавершених транзакцій) виконується наступним чином:

1. Вибирається черговий запис із списку даної транзакції.
2. Виконується протилежна за змістом операція. Замість операції Insert виконується відповідна операція Delete, замість операції Delete виконується Insert і замість прямої операції Update – зворотна операція Update, що відновлює попередній стан об'єкта БД.
3. Будь-яка з цих зворотних операцій також записується в журнал транзакцій. Власне, для індивідуального відкату це не потрібно, але при виконанні індивідуального відкату транзакції може відбутися м'який збій, після відновлення якого потрібно відкотити таку транзакцію, для якої не повністю виконаний індивідуальний відкат.
4. При успішному завершенні відкату в журнал транзакцій записується запис про кінець транзакції. З точки зору журналу транзакцій така транзакція є зафіксованою.

26.8.2. Відновлення після відмов системи

Розглянемо докладніше відновлення після відмов системи. Критичною точкою відмови системи є втрата вмісту частини оперативної пам'яті (зокрема, робочих буферів БД). Оскільки точний стан будь-якої транзакції, яка виконувалась в момент відмови, невідомий, транзакція може не завершитися успішно і, отже, буде скасованою під час перезавантаження системи.

Більше того, можливо буде потрібно повторно виконати певну успішно завершену до аварійної відмови транзакцію під час перезавантаження системи, якщо не були фізично виконані оновлення цієї транзакції.

Для визначення того, яку транзакцію скасувати під час перезавантаження, а яку повторно виконати, система в певний визначений інтервал (коли в журналі транзакцій накопичується певна кількість записів) автоматично приймає контрольну точку. Прийняття контрольної точки включає фізичний запис вмісту робочих буферів БД безпосередньо в БД і спеціальний фізичний запис контрольної точки, який надає список всіх транзакцій, що здійснюються в даний момент.

На рис. 26.1 розглянуто можливі варіанти виконання транзакцій до аварійного збою системи:

1. Відмова системи відбулась в момент часу t_f.
2. Найближча на момент часу t_f контрольна точка була прийнята в момент часу t_c.
3. Транзакцію T_1 успішно завершено до моменту часу t_c.
4. Транзакцію T_2 розпочато до моменту часу t_c і успішно завершено після моменту часу t_c, але до моменту часу t_f.
5. Транзакцію T_3 також розпочато до моменту часу t_c, але не завершено до моменту часу t_f.
6. Транзакцію T_4 розпочато після моменту часу t_c і успішно завершено до моменту часу t_f.
7. Транзакцію T_5 також розпочато після моменту часу t_c, але не завершено до моменту часу t_f.

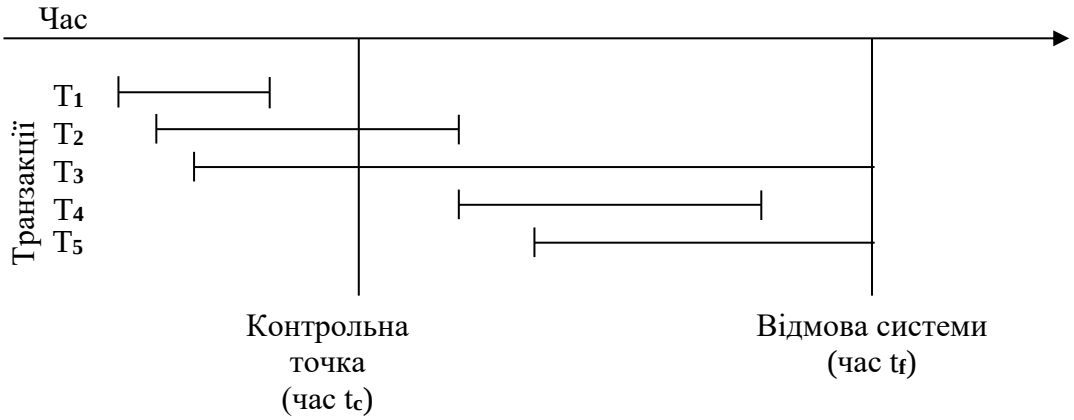


Рис. 26.1. Варіанти виконання п'яти транзакцій

Очевидно, що під час перезавантаження системи транзакції типу T_3 та T_5 мають бути скасовані, а транзакції типу T_2 та T_4 – виконані повторно. Проте слід зауважити, що транзакції типу T_1 зовсім не включаються в процес перезавантаження, оскільки оновлення потрапили до БД ще до моменту часу tc (тобто зафіксовано ще до прийняття контрольної точки). Також слід відзначити, що транзакції, які завершилися невдачею (включаючи скасовані) до моменту часу t_f, взагалі не будуть включено в процес перезавантаження.

26.8.3. Відновлення після м'якого збою

До числа основних проблем відновлення БД після м'якого збою відноситься те, що одна логічна операція зміни БД може змінювати декілька фізичних блоків БД, наприклад, сторінку даних і декілька сторінок індексів.

Сторінки БД буферизуються в оперативній пам'яті і виштовхуються незалежно. Попри використання протоколу WAL, після м'якого збою набір сторінок зовнішньої пам'яті БД може виявитися неузгодженим, тобто частина сторінок зовнішньої пам'яті відповідає об'єкту до зміни, а частина – після зміни. До такого стану об'єкта не можна застосовувати операції логічного рівня.

Стан зовнішньої пам'яті БД називається *фізично узгодженим*, якщо набори сторінок всіх об'єктів узгоджено, тобто відповідають стану об'єкта, або до його зміни, або після зміни.

Вважатимемо, що в ЖТ відзначаються *точки фізичної узгодженості БД TPC (Time of Physical Consistency)* – моменти часу, в які в зовнішній пам'яті містяться узгоджені результати операцій, що завершилися до відповідного моменту часу, і відсутні результати операцій, які не завершилися, а буфер ЖТ виштовхнуто у зовнішню пам'ять. Нижче розглянемо, як можна досягти фізичної узгодженості. Назвемо такі точки – точки фізичної узгодженості. Тоді до моменту м'якого збою можливі наступні стани транзакцій:

1. Транзакцію успішно завершено, тобто, виконана операція підтвердження транзакції Commit і для всіх операцій транзакції отримано підтвердження її виконання в зовнішній пам'яті;
2. Транзакцію успішно завершено, але для деяких операцій не отримано підтвердження їх виконання в зовнішній пам'яті;
3. Транзакція отримала і виконала команду відкату Rollback;
4. Транзакцію не завершено.

26.8.4. Фізична узгодженість БД

Постає питання, яким чином можна забезпечити наявність точок фізичної узгодженості TPC, та як відновити стан БД у момент TPC? Для цього використовуються два основних підходи:

1. Використання тіньового механізму.
2. Журналювання посторінкових змін БД.

При відкритті файлу таблиця відображення номерів його логічних блоків в адреси фізичних блоків зовнішньої пам'яті зчитується в оперативну пам'ять. При модифікації будь-якого блока файлу в зовнішній пам'яті виділяється новий блок. При цьому поточна таблиця відображення (в оперативній пам'яті) змінюється, а тіньова – зберігається незмінною. Якщо під час роботи з відкритим файлом відбувається збій, в зовнішній пам'яті автоматично зберігається стан файлу до

його відкриття. Для явного відновлення файлу достатньо повторно зчитати в оперативну пам'ять тіньову таблицю відображення.

Загальна ідея тіньового механізму показана на рис. 26.2.

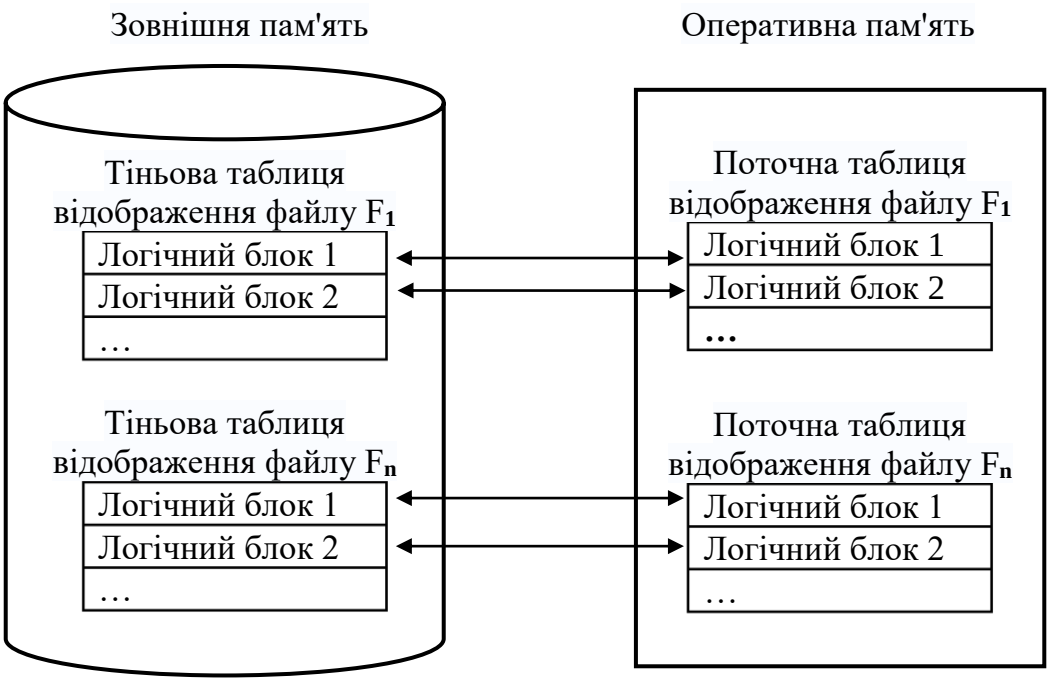


Рис. 26.2. Використання тіньових таблиць відображення інформації

У контексті БД тіньовий механізм використовується наступним чином. Періодично виконуються операції встановлення точки фізичної узгодженості БД (checkpoints). Для цього всі логічні операції завершуються, всі буфери оперативної пам'яті, вміст яких не відповідає вмісту відповідних сторінок зовнішньої пам'яті, виштовхуються. Тіньова таблиця відображення файлів БД замінюється на поточну – поточна таблиця відображення записується на місце тіньової.

Відновлення до ТРС відбувається миттєво. Поточна таблиця відображення замінюється на тіньову (при відновленні просто зчитується тіньова таблиця відображення). Усі проблеми відновлення вирішуються, але за рахунок занадто великої витрати зовнішньої пам'яті. На межі може знадобитися вдвічі більше зовнішньої пам'яті, ніж реально необхідно для зберігання БД.

Отже, тіньовий механізм – це надійний, але досить грубий засіб. Забезпечується узгоджений стан зовнішньої пам'яті в один загальний для всіх об'єктів момент часу. Насправді достатньо мати сукупність узгоджених наборів сторінок, кожному з яких може відповідати свої часові відмітки.

Для виконання такої слабшої вимоги водночас з логічним журналюванням операцій зміни БД проводиться журналювання посторінкових змін. Перший етап

відновлення після м'якого збою полягає в посторінковому відкаті незакінчених логічних операцій. Подібно до того, як це робиться з логічними записами відносно транзакцій, останнім записом про посторінкові зміни від однієї логічної операції є запис про кінець операції.

У цьому підході є 2 варіанти вирішення проблеми. При використанні 1-го методу підтримується загальний журнал логічних і сторінкових операцій. Зрозуміло, наявність двох видів записів, які інтерпретуються абсолютно по-різному, ускладнює структуру журналу. Крім того, записи про посторінкові зміни, актуальність яких має локальний характер, суттєво (і не дуже логічно) збільшують журнал.

Тому дедалі популярнішим стає підтримка окремого (короткого) журналу посторінкових змін. Така техніка використовується, наприклад, в СУБД Informix Online.

Припустимо, що якимось чином вдалося відновити зовнішню пам'ять БД до стану на момент часу tpc (нижче показано, як це можна зробити). Тоді:

1. Для транзакції T_1 жодних дій проводити не потрібно. Вона завершилась до моменту ТРС, і всі її результати відображено в зовнішній пам'яті БД.
2. Для транзакції T_2 потрібно повторно виконати частину операцій (redo), що залишилась. Справді, в зовнішній пам'яті повністю відсутні сліди операцій, які виконувалися в транзакції T_2 після моменту ТРС. Отже, повторна пряма інтерпретація операцій T_2 є коректною і приведе до логічно узгодженого стану БД (оскільки транзакція T_2 успішно завершилась до моменту м'якого збою, в журналі містяться записи про всі зміни, здійснені цією транзакцією).
3. Для транзакції T_3 потрібно виконати у зворотному напрямку першу частину операцій (undo). Справді, в зовнішній пам'яті БД повністю відсутні результати операцій T_3, які були виконані після моменту ТРС. З іншого боку, в зовнішній пам'яті гарантовано присутні результати операцій T_3, які були виконані до моменту ТРС. Отже, зворотна інтерпретація операцій T_3 є коректною і призведе до узгодженого стану БД (оскільки транзакція T_3 не завершилась до моменту м'якого збою, при відновленні необхідно усунути всі наслідки її виконання).
4. Для транзакції T_4, яка встигла розпочатися після моменту ТРС і завершитися до моменту м'якого збою, потрібно виконати повну повторну пряму інтерпретацію операцій (redo).
5. Нарешті, для транзакції T_5, яка почалась після моменту ТРС і не встигла завершитися до моменту м'якого збою жодних дій робити не потрібно.

Результати операцій цієї транзакції повністю відсутні в зовнішній пам'яті БД.

26.8.5. Відновлення після жорсткого збою

Очевидно, що для відновлення останнього узгодженого стану БД після жорсткого збою потрібне журналювання транзакцій. Основою відновлення в цьому випадку є журнал транзакцій та архівна копія БД.

Відновлення розпочинається з копіювання БД з архівної копії. Потім для всіх завершених транзакцій виконується redo, тобто операції повторно виконуються в прямому порядку.

Точніше, відбувається наступне:

1. У журналі транзакцій в прямому напрямку виконуються всі операції.
2. Для транзакцій, які не завершилися на момент збою, виконується відкат.

Фактично, оскільки жорсткий збій не супроводжується втратою буферів оперативної пам'яті, можна відновити БД до такого рівня, щоб можна було продовжити виконання навіть незавершених транзакцій. Але зазвичай цього не робиться через те, що відновлення після жорсткого збою – це досить тривалий процес.

Хоча до ведення журналу транзакцій висуваються особливі вимоги щодо надійності, в принципі можлива і його втрата. Тоді єдиним способом відновлення БД є повернення до архівної копії. Звичайно, у цьому випадку не вдасться отримати останній узгоджений стан БД, але це краще, ніж нічого.

Останнє питання, яке ми коротко розглянемо, стосується створення архівних копій БД. Найпростіший спосіб – архівувати БД під час переповнення журналу транзакцій. У журналі вводиться так звана жовта зона, досягнення якої призводить до тимчасового блокування створення нових транзакцій. Коли всі транзакції завершать свою роботу і, відповідно, БД прийде до узгодженого стану, можна провести її архівацію, після чого починати заповнення журналу транзакцій знову.

Можна виконувати архівацію БД рідше, ніж переповнення журналу транзакцій. При переповненні журналу транзакцій і завершенні всіх розпочатих транзакцій необхідно архівувати сам журнал транзакцій. Оскільки такий архівований журнал транзакцій потрібен лише для відтворення архівної копії БД, журнальна інформація при архівації може бути значно стиснута.

26.9. Контрольні запитання

1. Дайте визначення транзакції.
2. В чому полягає проблема неповторного читання?
3. В чому полягає проблема брудного читання?
4. В чому полягає проблема втраченого оновлення або останньої зміни?
5. В чому полягає проблема фантомної вставки або читання фантомів?
6. У чому полягає причина та суть блокування доступу до конкретного елемента БД з боку інших транзакцій?
7. Які існують рівні блокування транзакцій?
8. Що таке оптимістичні та песимістичні блокування?
9. Вирішення проблем яких типів блокувань передбачає стандарт SQL?
10. В чому полягає вирішення проблем паралельних транзакцій?
11. В чому полягає протокол доступу до даних, який дозволяє уникнути проблем паралельних транзакцій?
12. Назвіть рівні блокування транзакцій, передбачені стандартом SQL.
13. Охарактеризуйте поняття брудного читання.
14. Охарактеризуйте поняття втраченого оновлення.
15. В чому полягає проблема тупикової ситуації та способи її вирішення?
16. Назвіть рівні ізоляції транзакцій, передбачені стандартом SQL.
17. Охарактеризуйте схему управління транзакціями в Java.
18. Охарактеризуйте схему управління транзакціями в C#.
19. Охарактеризуйте схему управління транзакціями в Python.
20. Охарактеризуйте процеси відновлення БД після м'якого збою.
21. Яким чином можна забезпечити фізичну узгодженість БД?
22. Охарактеризуйте процеси відновлення БД після жорсткого збою.

СПИСОК ЛІТЕРАТУРИ

1. DB-Engines Ranking per database model category [Електронний ресурс] // DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems. – Режим доступу: https://db-engines.com/en/ranking_categories (дата звернення: 22.04.2024). – Назва з екрана.
2. C. J. Date. Database Design and Relational Theory: Normal Forms and All That Jazz. – New York : Apress, 2019. – 474 p.
3. A., Beaulieu. Learning SQL: Generate, Manipulate, and Retrieve Data / Alan Beaulieu. – Sebastopol, California : O'Reilly Media, 2020. – 377 p.
4. WITH Queries (Common Table Expressions) [Електронний ресурс] // PostgreSQL Documentation. – Режим доступу: https://www.postgresql.org/docs/16/queries-with.html (дата звернення: 23.04.2024). – Назва з екрана.
5. Ferrari L. Learn PostgreSQL / Luca Ferrari, Enrico Pirozzi. – Birmingham, UK : Packt Publishing, 2023. – 744 p.
6. CREATE DATABASE [Електронний ресурс] // PostgreSQL Documentation. – Режим доступу: https://www.postgresql.org/docs/current/sql-createdatabase.html (дата звернення: 23.04.2024). – Назва з екрана.
7. CREATE TABLE [Електронний ресурс] // PostgreSQL Documentation. – Режим доступу: https://www.postgresql.org/docs/current/sql-createtable.html (дата звернення: 23.04.2024). – Назва з екрана.
8. ALTER TABLE [Електронний ресурс] // PostgreSQL Documentation. – Режим доступу: https://www.postgresql.org/docs/current/sql-altertable.html (дата звернення: 23.04.2024). – Назва з екрана
9. CREATE FUNCTION [Електронний ресурс] // PostgreSQL Documentation. – Режим доступу: http://www.postgresql.org/docs/current/static/sql-createfunction.html (дата звернення: 22.04.2024). – Назва з екрана.
10. CREATE TRIGGER [Електронний ресурс] // PostgreSQL Documentation. – Режим доступу: http://www.postgresql.org/docs/current/static/sql-createtrigger.html (дата звернення: 22.04.2024). – Назва з екрана.

ПРЕДМЕТНИЙ ПОКАЖЧИК

A

ACID властивості, 258
ADO, 222
ALL, 93
ALTER DATABASE, 144
ALTER TABLE, 150
ANY, 91

B

BackEnd, 217
BETWEEN, 64
BRIN, 196
B-дерева, 196

C

C#, 241
CALL, 168
COMMIT, 249, 251
CONCURRENTLY, 195
CONSTRAINT, 149
CREATE DATABASE, 142
CREATE INDEX, 154, 192
CREATE TABLE, 146
CREATE TRIGGER, 184
CREATE VIEW, 157
CUBE, 77

D

Database, 15, 27, 216, 219, 237, 242, 243, 244, 245, 276, 286
DataColumn, 226
DECLARE, 163
DELETE, 119
DROP DATABASE, 144
DROP FUNCTION, 180
DROP INDEX, 194
DROP TABLE, 151
DROP TRIGGER, 190

E

EXCEPT, 96

F

FETCH, 163
FOR EACH ROW, 189
foreign key, 38
FROM, 62

G

generate_series, 125
GIN, 140, 198
GiST, 140
GRANT, 203
GROUP BY, 75
GROUPING SETS, 77

H

HAVING, 81

I

INSERT, 117
INTERSECT, 95

J

Java, 238
JDBC, 229

L

LIKE, 65
Locks, 264

O

ODBC, 219
OLE DB, 221
Open DataBase Connectivity, 219
ORDER BY, 74

P

PL/pgSQL, 168
primary key, 37
Python, 231, 236

R

random, 122
READ UNCOMMITTED, 273
REPEATABLE READ, 271
REVOKE, 204
ROLLBACK, 249, 250
ROLLUP, 77
ROWTYPE, 176

S

SELECT, 60
SELECT INTO, 114
SOME, 91
SP-GiST, 197
SQL, 52
SQL2, 212
SQL-92, 214
Stored Procedure, 165

T

Transaction, 248
Trigger, 183

U

UNION, 95
UPDATE, 119

W

WHERE, 63
WITH, 100

А

Агрегатна функція, 73
Адміністратор бази даних, 16
Апаратне забезпечення, 15, 16

Б

База даних, 15
Брудне читання, 267

В

Віднімання, 43
Відновлення, 278, 284
Відношення, 33, 34, 35, 36, 38, 48
ВКЛАДЕНІ ЗАПИТИ, 83
Внутрішній рівень, 18, 20
Втрачене оновлення, 262, 269

Д

Дані, 15, 77, 153, 227
Декартовий добуток, 33, 43
Динамічний курсор, 161
Ділення, 47, 51
Домен, 32, 151, 212, 213

Ж

Журналювання, 257

З

Збережена процедура, 165
Значення NULL, 40, 66, 80
Зовнішні з'єднання, 110
зовнішній ключ, 38, 39, 40, 54, 68, 71, 150, 211, 213
Зовнішній рівень, 18, 19

І

Ієрархічна модель даних, 25
Індекси, 192

К

Концептуальний рівень, 18, 20
кортеж, 34, 35, 37, 38, 39, 48, 49, 176, 177, 179, 264, 265
Курсор, 160

М

Мережева модель даних, 27
МОДЕЛІ БД, 23

О

Об'єднання, 41, 108

П

первинний ключ, 34, 37, 40, 54, 68, 147, 150, 211, 213
Перетин множин, 42
Перехресні з'єднання, 112
Повнотекстовий пошук, 134
потенційний ключ, 36, 37, 38, 39, 40
Представлення, 156
Природне з'єднання, 46, 51, 68
Програмне забезпечення, 15, 16
Проєкція, 45, 51

Р

Рекурсивні запити, 101
Реляційна алгебра, 13, 41
Реляційна повнота SQL, 113
Реляційне числення, 50
Роль, 202

С

самооб'єднання, 69
системи баз даних, 14
СИСТЕМНИЙ КАТАЛОГ, 207
СУБД, 15, 19, 24, 117, 183

Т

Тип SETOF, 174
Типи даних, 55
Транзакція, 248
Тригер, 183
Тупикові ситуації, 269

Ф

Фізичний рівень, 18

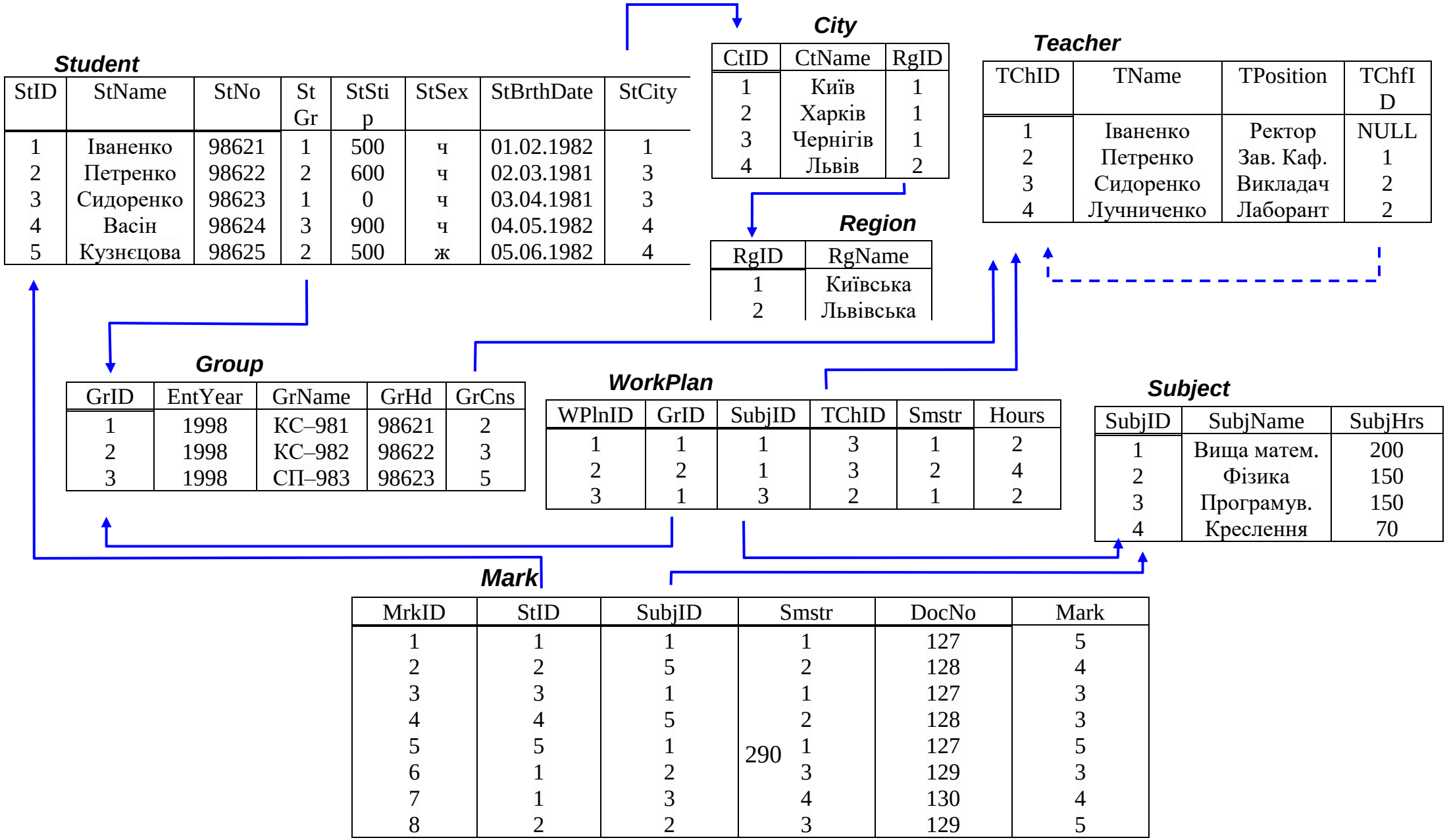
Х

Хеш-індекси, 196

Ц

Цілісність, 31, 39, 40, 156

ДОДАТОК А. СТРУКТУРА НАВЧАЛЬНОЇ БД "ДЕКАНАТ ВНЗ"



Student

StID	StName	StNo	St Gr	StSti p	StSex	StBrthDate	StCity
1	Іваненко	98621	1	500	ч	01.02.1982	1
2	Петренко	98622	2	600	ч	02.03.1981	3
3	Сидоренко	98623	1	0	ч	03.04.1981	3
4	Васін	98624	3	900	ч	04.05.1982	4
5	Кузнєцова	98625	2	500	ж	05.06.1982	4

City

CtID	CtName	RgID
1	Київ	1
2	Харків	1
3	Чернігів	1
4	Львів	2

Region

RgID	RgName
1	Київська
2	Львівська

Teacher

TChID	TName	TPosition	TChfI D
1	Іваненко	Ректор	NULL
2	Петренко	Зав. Каф.	1
3	Сидоренко	Викладач	2
4	Лучниченко	Лаборант	2

Group

GrID	EntYear	GrName	GrHd	GrCns
1	1998	КС–981	98621	2
2	1998	КС–982	98622	3
3	1998	СП–983	98623	5

WorkPlan

WPlnID	GrID	SubjID	TChID	Smstr	Hours
1	1	1	3	1	2
2	2	1	3	2	4
3	1	3	2	1	2

Subject

SubjID	SubjName	SubjHrs
1	Вища матем.	200
2	Фізика	150
3	Програмув.	150
4	Креслення	70

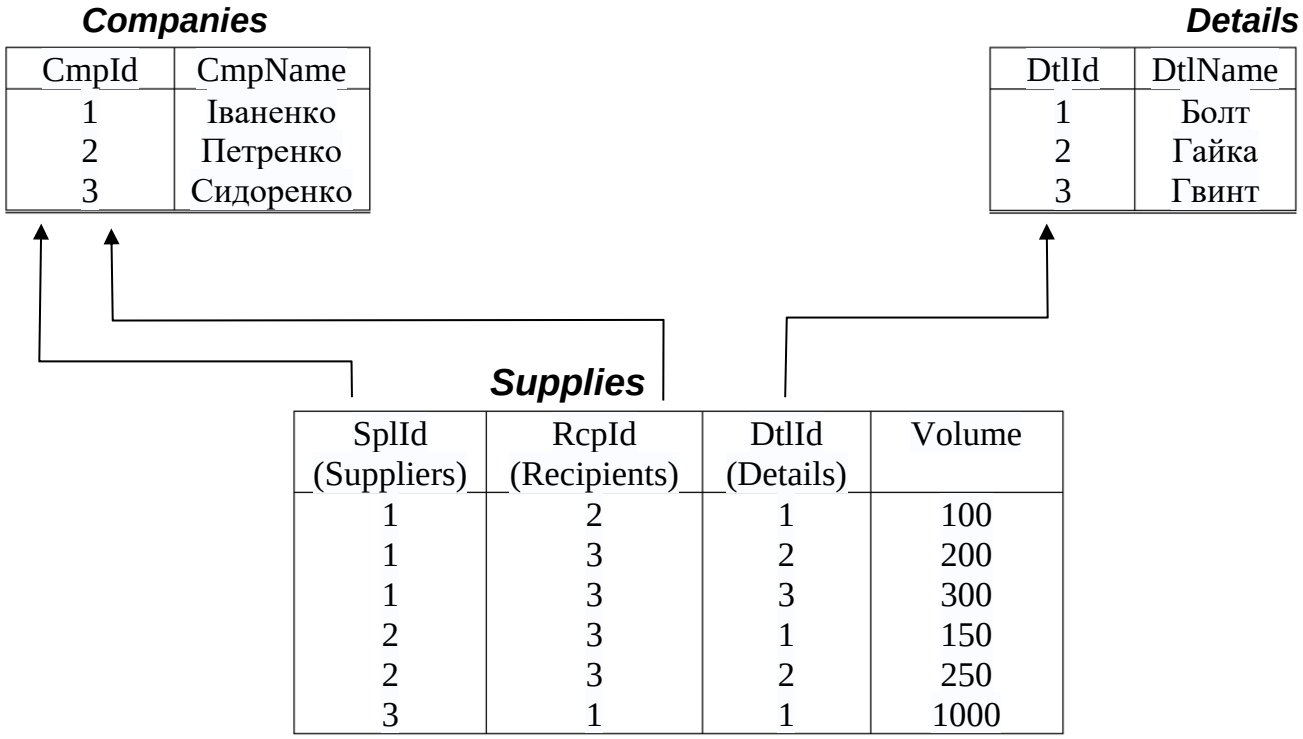
Mark

MrkID	StID	SubjID	Smstr	DocNo	Mark
1	1	1	1	127	5
2	2	5	2	128	4
3	3	1	1	127	3
4	4	5	2	128	3
5	5	1	1	127	5
6	1	2	3	129	3
7	1	3	4	130	4
8	2	2	3	129	5

290

Таблиця А.1 – Опис структури навчальної БД "Деканат ВНЗ"

ВІДНОШЕННЯ	АТРИБУТ	ТИП (РОЗМІР)
Відношення "***Student***" Містить інформацію про студентів	***StdID*** – унікальний ID студента в БД ***StdNmb*** – номер залікової книжки *StdGrp* – ID групи студента *StdName* – прізвище студента *StdSex* – стать студента *StdBrthDate* – дата народження студента *StdStip* – стипендія студента *StdCity* – ID міста студента	Числовий Текстовий (6) Числовий Текстовий (30) Текстовий (1) Дата Грошовий Числовий
Відношення "***Group***" Містить інформацію про групи	***GrpID*** – унікальний ID групи *EntYear* – рік створення групи *GrpName* – назва групи *GrpHd* (*GrpHead*) – код старости групи *GrpCns (GrpCounsel)* – код куратора групи	Числовий Текстовий (4) Текстовий (6) Числовий Числовий
Відношення "***Subject***" Містить інформацію про предмети	***SubjID*** – унікальний ID предмета *SubjName* – назва предмета *SubjHrs* – кількість годин предмета	Числовий Текстовий (20) Числовий
Відношення "***Teacher***" Містить інформацію про співробітників та викладачів	***TchID*** – унікальний ID співробітника (викладача) *TchName* – прізвище викладача *TchPosition* – посада викладача *TchChfID* – ID співробітника, що є керівником (Chief) для даного викладача	Числовий Текстовий (30) Текстовий (30) Числовий
Відношення "***Mark***" Містить інформацію про оцінки студентів	***MrkID*** – унікальний ID оцінки *StdID* – ID студента *SubjID* – ID дисципліни *Smstr* - семестр *DocNo* – номер документа (екз. відомість, дозвіл на перескладання тощо) *Mark* - оцінка	Числовий Числовий Числовий Числовий Числовий Числовий

Відношення "***WorkPlan***" Містить інформацію про рабочій план. Робочий план визначає дисципліни, що викладаються: ким, кому, в якому обсязі і коли.	***WPlnID*** – унікальний ID рядка робочого плану *GrpID* – ID групи *SubjID* – ID дисципліни Sem (Semester) - семестр *TchID* – ID викладача *Hours* – кількість годин	Числовий Числовий Числовий Числовий Числовий Числовий
Відношення "***City***" Містить інформацію про міста	***CytID*** – унікальний ID міста *CytName* – назва міста *RgnID* – ID області міста	Числовий Текстовий (20) Числовий
Відношення "***Region***" Містить інформацію про області	***RgnID*** – унікальний ID області *RgnName* – назва області	Числовий Текстовий (20)

ДОДАТОК Б. СТРУКТУРА БД "ПОСТАЧАННЯ"