Частина II
Структурована мова запитів SQL

2.1. Вступ в SQL

2.1.1. З чого почати?

Зростання кількості даних, необхідність їх зберігати і обробляти привели до того, що виникла потреба у створенні стандартної мови БД, яка могла б функціонувати у великій кількості різних видів комп'ютерних систем. Дійсно, така стандартна мова дозволяє користувачам маніпулювати даними незалежно від того, чи працюють вони на персональному

комп'ютері, мережевій робочій станції, або на універсальній ЕОМ.

SQL (Structured Query Language) – це скорочена назва структурованої мови запитів, яка надає засоби створення і обробки даних в реляційних БД. Незалежність від специфіки комп'ютерних технологій, а також підтримка SQL лідерами промисловості у області технології реляційних баз даних зробили її основною стандартною мовою БД.

Всі мови маніпулювання даними, створені до появи реляційних БД, розроблені для багатьох СУБД, були орієнтовані на операції із даними, представленими у вигляді логічних записів файлів. Зрозуміло, це вимагало від користувача детального знання організації зберігання даних і серйозних зусиль для вказівки того, які дані необхідні, де вони розміщуються і як їх отримати.

Дана мова SQL орієнтована на операції з даними, представленими у вигляді логічно взаємозв'язаних сукупностей таблиць-відношень. Найважливіша особливість структур цієї мови полягає в орієнтації на кінцевий результат обробки даних, а не на процедуру цієї обробки. SQL сама визначає, де знаходяться дані, індекси і навіть які найбільш ефективні послідовності операцій слід використовувати для отримання результату, тому не треба вказувати ці деталі в запиті до БД.

Поява теорії реляційних БД дала поштовх до розробки ряду мов запитів, які можна віднести до двох класів:

- мови, алгебри, що дозволяють виражати запити засобами спеціалізованих операторів, що використовуються до відношень;
- мови обчислення предикатів, що є набором правил для запису виразу, що визначає нове відношення із заданої сукупності існуючих відношень. Отже, обчислення предикатів є метод визначення того відношення, яке бажано отримати, як відповідь на запит із відношень, вже наявних в БД.

У 1987 році SQL стала стандартом мов для професійних реляційних СУБД і почала впроваджуватися у всі поширені системи. Це пов'язано з рядом наступних моментів. Постійне зростання швидкодії, а також зниження енергоспоживання, розмірів і вартості комп'ютерів привели до різкого розширення можливих ринків їх збуту, кола користувачів, різноманітності типів і цін. Природно, що розширився попит на різноманітне програмне забезпечення. У боротьбі за покупця фірми, що виготовляють програмне забезпечення, стали випускати на ринок все більш інтелектуальні, а значить, об'ємні програмні комплекси. Купуючи їх, багато організацій і окремі користувачі часто не могли розмістити їх на власних ЕОМ. Для обміну інформацією і її розповсюдження були створені мережі ЕОМ, де узагальнюючі програми і дані стали розмішати на спеціальних файлових серверах.

СУБД, що працюють із файловими серверами, дозволяють безлічі користувачів різних ЕОМ, розташованих достатньо далеко один від одного, діставати доступ до одних і тих же БД. При цьому спрощується розробка різних автоматизованих систем управління організаціями, навчальних комплексів, інформаційних і інших систем, де безліч співробітників або студентів повинні використовувати спільні дані і обмінюватися створюваною в процесі роботи інформацією. Проте при такому підході вся обробка запитів із програм або з терміналів призначених для користувача ЕОМ на них і виконується, тому для реалізації навіть простого запиту необхідно зчитувати з файлового сервера або записувати на сервер цілі файли, а це веде до конфліктних ситуацій і перевантаження мережі. Для виключення вказаних недоліків була запропонована технологія клієнт/сервер, проте при цьому потрібна єдина мова спілкування з сервером – і в його якості була вибрана SQL.

Реалізація в SQL концепції операцій, орієнтованих на табличне представлення даних, дозволила створити компактну мову із невеликим набором речень. SQL може використовуватися як для виконання запитів, так і для побудови прикладних програм. У ній існують:

- речення визначення даних – визначення БД, а також визначення і знищення таблиць і

індексів;
- запити на вибір даних – речення SELECT;
- речення модифікації даних – додавання, видалення і зміна даних;
- речення управління даними – надання і відміна привілеїв на доступ до даних, управління транзакціями та інші.

Крім того. SQL надає можливість виконувати в цих реченнях:
- арифметичні обчислення, включаючи різноманітні функціональні перетворення, обробку текстових рядків і виконання операцій порівняння значень арифметичних виразів і текстів;
- впорядкування рядків або стовпців при виведенні вмісту таблиць на друк або екран дисплея;
- створення уявлень, що дозволяють користувачам інтерпретувати дані без збільшення їх об'єму в БД;
- зберігання вмісту таблиці, що виводиться по запиту, декількох таблиць або уявлення в іншій таблиці;
- групування даних і застосування до цих груп таких операцій, як середнє, сума, максимум, мінімум, число елементів і т.п.

Стандарт SQL визначається ANSI (американським національним інститутом стандартів) і зараз також приймається ISO (міжнародною організацією по стандартизації). Проте більшість комерційних програм БД розширюють SQL без повідомлення ANSI, додаючи різні інші особливості в цю мову, які, як вони вважають, будуть дуже корисні. Іноді це дещо порушує стандарт мови, хоча добрі ідеї мають тенденцію розвиватися і незабаром стають стандартами.

Мова SQL є основою багатьох СУБД, оскільки вона відповідає за фізичну структуризацію і запис даних на диск, а також за фізичне читання даних з диска і дозволяє приймати SQL-запити від інших компонентів СУБД і призначених для користувача додатків. Таким чином, SQL є могутнім інструментом, який забезпечує користувачам, програмам і обчислювальним системам доступ до інформації, що міститься в реляційних БД.

Основні достоїнства мови SQL полягають в наступному:
- стандартність мови SQL – як вже було сказано, її використання в програмах стандартизоване міжнародними організаціями;
- незалежність від конкретних СУБД – всі поширені СУБД використовують SQL, оскільки реляційну БД і програми, які з нею працюють, можна перенести з однієї СУБД на іншу з мінімальними доопрацюваннями;
- можливість переносу з однієї обчислювальної системи на іншу – СУБД може бути орієнтована на різні обчислювальні системи, проте додатки, створені за допомогою SQL, допускають використання як для локальних БД, так і для великих розрахованих на багато користувачів систем;
- реляційна основа мови – SQL є мовою реляційних БД, тому вона стала популярною тоді, коли популярною стала реляційна модель представлення даних. Таблична структура реляційної БД добре зрозуміла, тому мова SQL є простою і легкою для вивчення;
- можливість створення інтерактивних запитів – SQL забезпечує користувачам негайний доступ до даних, при цьому в інтерактивному режимі можна отримати результат запиту за дуже короткий час без написання складної програми;
- можливість програмного доступу до БД – мова SQL може бути легко використана в додатках, яким необхідно звертатися до БД. Одні і ті ж оператори SQL використовуються як для інтерактивного, так і для програмного доступу, тому частини програм, що містять звернення до БД, можна спочатку перевірити в інтерактивному режимі, а потім вбудовувати в програму;
- забезпечення різного представлення даних – за допомогою SQL можна передбачити

таку структуру даних, що той або інший користувач бачитиме різні представлення даних. Крім того, дані з різних частин БД можуть бути скомбіновані і представлені користувачу у вигляді однієї простої таблиці, а значить, уявлення можна використовувати для посилення захисту БД і її настройки під конкретні вимоги окремих користувачів;

- можливість динамічної зміни і розширення структури БД – мова SQL навіть під час звернення до вмісту дозволяє маніпулювати структурою БД. Ця велика перевага перед мовами статичного визначення даних, які забороняють доступ до БД під час зміни її структури. Таким чином, SQL забезпечує гнучкість з погляду пристосованості БД до вимог предметної області, що змінюються, не перериваючи при цьому роботу додатку, що її виконує в реальному масштабі часу;
- підтримка архітектури клієнт/сервер – SQL один з кращих засобів для реалізації додатку на платформі клієнт/сервер. При цьому SQL служить сполучною ланкою між клієнтською системою, що взаємодіє з користувачем, і серверною системою, БД, що управляє, дозволяючи кожній з них зосередитися на виконанні своїх прямих функцій.

2.1.2. Історія розвитку SQL

Системи управління реляційними БД свою особливу популярність набули в кінці 80-х років. Оскільки інформація в таких БД зберігається в простій табличній формі, це дає багато переваг в порівнянні з іншими моделями представлення даних. Оскільки SQL є мовою реляційних БД, то її історія тісно пов'язана з розвитком цього способу представлення інформації.

Як вже було сказано, поняття реляційної БД введене на початку 70-х років Э. Ф. Коддом, науковим співробітником компанії IBM. Після цього почалися дослідження у області реляційних баз даних, включаючи великий дослідницький проект компанії IBM, названий System/R, в якому перевірялася і доводилася працездатність реляційної моделі. Окрім розробки самої СУБД, в рамках проекту проводилася робота над створенням мов запитів до БД, одна з яких була названа SEQUEL (Structured English Query Language – структурована англійська мова запитів).

В кінці 70-х років здійснена друга реалізація проекту System/R, внаслідок чого система була встановлена на комп'ютерах декількох замовників компанії IBM для дослідної експлуатації. Це принесло перший реальний досвід роботи із СУБД System/R і її мовою БД, яка пізніше була перейменована в SQL. В результаті IBM зробила висновок, що реляційні БД цілком працездатні і можуть служити основою для створення програмних продуктів.

Проект System/R і створена в його рамках мова роботи з БД привернули пильну увагу фахівців у всьому світі. Так, в 1977 році була організована компанія Relational Software Inc. (нині Oracle Corporation), щоб створити реляційну СУБД, засновану на SQL. Ця СУБД, названа Oracle, першою вийшла на ринок в 1979 році.

Аналогічно дослідницькій групі компанії IBM, в університеті міста Берклі (штат Каліфорнія) створили прототип реляційної СУБД, назвавши свою систему Ingres, який включав мову запитів QUEL. У 1980 році декілька фахівців покинули Берклі і заснували компанію Relational Technology, Inc., щоб створити комерційну версію системи Ingres, поставки якої на ринок почалися в 1981 році. Компанія Relational Technology, перейменована в 1989 році в Ingres Corporation, залишається провідним постачальником реляційних СУБД, внісши в систему ряд удосконалень. Наприклад, в 1986 році первинна мова запитів QUEL була з успіхом замінена на SQL.

Фірма IBM в 1981 році випустила комерційний продукт на основі System/R, який одержав назву SQL/Data System (SQL/DS), а в 1983 році з'явилася ще одна реляційна СУБД – Database 2 (DB2). DB2 була провідною реляційною СУБД компанії IBM, і мова SQL цієї системи стала фактичним стандартом для БД. Технологія, реалізована DB2, потім була

використана в програмних продуктах всіх напрямів компанії IBM, від персональних комп'ютерів до мережевих серверів і великих ЕОМ.

Протягом першої половини 80-х років, постачальники реляційних БД боролися за визнання своїх продуктів, оскільки порівняно з традиційною архітектурою БД, реляційні програмні продукти мали декілька недоліків, наприклад, продуктивність реляційних БД була нижча, ніж у традиційних. Проте у реляційних систем була велика перевага: їх мови реляційних запитів дозволяли виконувати запити до БД без написання програм і негайно одержувати результати.

У другій половині 80-х років реляційні БД вже стали вважатися технологією майбутнього, з'явилися нові версії СУБД Ingress і Oracle, продуктивність яких була в два-три рази вище, ніж у попередніх версій. Публікація в 1986 році стандарту SQL, прийнятого ANSI/ISO, офіційно закріпило за SQL статус стандарту. Крім того, даний статус був визначений для комп'ютерних систем на основі ОС UNIX, зростання популярності яких припало на кінець 80-х.

Із збільшенням потужності комп'ютерів і об'єднання їх в локальні мережі, виникла необхідність в складних СУБД для персональних комп'ютерів. Постачальники таких СУБД при створенні нових систем поклали в їх основу SQL, а постачальники СУБД для міні-комп'ютерів вийшли на ринок локальних обчислювальних мереж, що зароджуються. На початку 90-х удосконалення реалізації SQL і поява могутніших процесорів дозволили застосовувати цю мову в додатках для обробки транзакцій. Тепер SQL став ключовою частиною архітектури клієнт/сервер, що зв'язує мережевий сервер і персональні комп'ютери в систему, в якій обробка інформації відбувається з порівняно невеликими затратами. З цієї причини у всі нові СУБД стали вводити підтримку SQL, адже вона стала як офіційним, так і фактичним стандартом для реляційних БД.

Найважливішим кроком до визнання SQL стала поява стандартів на цю мову. Традиційно при згадці стандарту SQL мають на увазі офіційний стандарт, затверджений ANSI/ISO. Проте існують і інші важливі стандарти SQL, включаючи реалізацію в системі DB2 компанії IBM, і стандарт X/OPEN для SQL в середовищі UNIX.

У 1982 році почалася робота ANSI над офіційним стандартом мови реляційних БД, і комітет зупинив свій вибір на SQL. У основу стандарту була покладена SQL системи DB/2, не дивлячись на те, що вона містить в собі ряд істотних відмінностей від цього діалекту мови. Після декількох доопрацювань, в 1986 році стандарт був офіційно затверджений як стандарт ANSI номер X3.135, а в 1987 році – як стандарт ISO. Потім стандарт ANSI/ISO був прийнятий урядом США, як FIPS – федеральний стандарт США по обробці інформації, який дещо перероблений в 1989 році, звичайно називають стандартом SQL-89 або SQL1. Відмінності між діалектами SQL в СУБД різних виробників не перешкодили комітету: він обійшов деякі з них, не стандартизувавши певні частини мови. Це дозволило оголосити велике число реалізацій SQL сумісними із стандартом, однак зробило сам стандарт відносно слабким. З цієї причини ANSI продовжив свою роботу і створив проект нового, жорсткішого стандарту SQL2, а для наступного за ним стандарту SQL3 були запропоновані інші серйозні зміни. В результаті запропоновані стандарти SQL2 і SQL3 виявилися ще більш суперечливими, ніж початковий стандарт. Стандарт SQL2 затверджений в ANSI і остаточно прийнятий в жовтні 1992 року.

2.1.3. Управління базами даних за допомогою SQL

Як вже було відмічено вище, реляційна БД – це зв'язана інформація, що зберігається в двовимірних таблицях з рядками і стовпцями. Кожен рядок, в термінах СУБД звичайно називається записом, відповідатиме певній особливості даних, наприклад, набору інформації про успішність конкретного студента по тому або іншому навчальному предмету. Кожен стовпець таблиці, називається полем, міститиме значення для кожного типу даних – наприклад, імені студента, що представляється в кожному рядку. Таким чином, основою реляційної БД є

двовимірна таблиця з інформацією.

Проте реляційні БД рідко складаються з однієї таблиці. Для виконання складніших і могутніших операцій з даними створюють декілька таблиць взаємозв'язаної інформації. Потужність БД залежить від зв'язку, який визначають між фрагментами інформації, а не від самого цього фрагмента. При цьому для підтримки максимальної гнучкості системи, рядки таблиці не повинні знаходитися ні в якому певному порядку. У системах з реляційною БД є могутня функція впорядковування (індексування) інформації з можливістю подальшого її відновлення.

У таблицях БД необхідно мати стовпець, який би унікально ідентифікував кожен рядок. Звичайно цей стовпець містить номер, наприклад, номер студентського квитка. Такий унікальний стовпець (або унікальна група стовпців), який використовується, щоб ідентифікувати кожен рядок і зберігати всі рядки окремо, називається первинним ключем таблиці.

Первинні ключі таблиці важливий елемент в структурі БД. Вони є основою системи запису у файл; і коли виникає необхідність знайти певний рядок в таблиці, то до неї посилаються по цьому первинному ключу. Крім того, первинні ключі гарантують, що дані мають певну цілісність. Якщо первинний ключ правильно використовується і підтримується, то завжди буде доступна інформація про те, що немає порожніх рядків таблиці і що кожен рядок відрізняється від будь-якого іншого рядка.

На відміну від записів, поля таблиці упорядковуються і іменуються. Це означає, що кожен стовпець даної таблиці повинен мати унікальне ім'я, щоб уникнути неоднозначності. Краще всього, якщо ці імена вказують на зміст поля.

SQL звичайно працює в комп'ютерних системах, які мають більше одного користувача, і, отже, виникає необхідність розрізняти їх між собою. Звичайно в такій системі кожен користувач має якийсь код перевірки прав, який його ідентифікує. На початку сеансу з комп'ютером, користувач входить в систему (реєструється), повідомляючи комп'ютер визначений ID (ідентифікатор). Будь-яка кількість людей, що використовують той же самий ID доступу, є окремими користувачами; і аналогічно, одна людина може представляти велику кількість користувачів у різний час, використовуючи різні ідентифікатори.

Дії в більшості середовищ SQL дозволені відповідно до спеціального ідентифікатора, який точно визначає користувача. Таблиця або інший об'єкт належить користувачу, який має над ним повну владу. При цьому користувач може мати привілеї для виконання дій над об'єктом. В більшості випадків в прикладах, що наводяться, вважатимемо, що будь-який користувач має привілеї, необхідні для виконання будь-якої дії.

2.1.4. Опис навчальної бази даних

У подальшому викладі використовуватимемо як приклад невелику БД, що відображає облік успішності студентів ВНЗу, дещо модифіковану у порівнянні з тою, що використовувалася в першій частині. У ній міститься чотири таблиці.

У таблиці СТУДЕНТИ (STUDENTS) міститься п'ять полів з інформацією про студентів:

- SNUM – номер студентського квитка;
- SPRIZ – прізвище студента;
- SIMA – ім'я студента;
- SBAT – по батькові студента;
- STYP – розмір одержуваної студентом стипендії.

Таблиця 2.1 Студенти.

STUDENTS				
SNUM	**SPRIZ**	**SIMA**	**SBAT**	**STYP**
3412	Поляченко	Анатолій	Олексійович	250.50
3413	Старченко	Любов	Михайлівна	170.00

3414	Грищенко	Володимир	Миколайович	0.00
3415	Котенко	Анатолій	Миколайович	0.00
3416	Нагірний	Євген	Васильович	250.00

Таблиця 2.2 Предмети.

PREDMET				
PNUM	**PNAME**	**TNUM**	**HOURS**	**COURS**
2001	Фізика	4001	34	1
2002	Хімія	4002	68	1
2003	Математика	4003	68	1
2004	Філософія	4005	17	2
2005	Економіка	4004	17	3

Таблиця 2.3 Викладачі.

TEACHERS				
TNUM	**TPRIZ**	**TIMA**	**TBAT**	**TDATE**
4001	Вікулін	Валентина	Іванівна	01/04/1984
4002	Костиркін	Олег	Володимирович	01/09/1997
4003	Казанко	Віталій	Володимирович	01/09/1988
4004	Позняк	Любов	Олексіївна	01/09/1988
4005	Загарийчук	Ігор	Дмитрович	10/05/1989

Таблиця 2.4 Успішність.

USP				
UNUM	**OCINKA**	**UDATE**	**SNUM**	**PNUM**
1001	5	10/06/2005	3412	2001
1002	4	10/06/2005	3413	2003
1003	3	11/06/2005	3414	2005
1004	4	12/06/2005	3412	2003
1005	5	12/06/2005	3416	2004

У таблиці ПРЕДМЕТИ (PREDMET), що складається з п’яти полів, міститься інформація про навчальні предмети. Призначення полів таблиці наступне:

- PNUM – номер (код) навчального предмету;
- PNAME – найменування навчального предмету;
- TNUM – номер (код) викладача;
- HOURS – тривалість навчальної дисципліни в годинах;
- COURS – курс, на якому ведеться даний навчальний предмет.

У таблиці ВИКЛАДАЧІ (TEACHERS), що складається з п'яти полів, міститься інформація про викладачів. Поля таблиці наступні:

- TNUM – код викладача;
- TPRIZ – прізвище викладача;
- TIMA – ім'я викладача;
- TBAT – по батькові викладача;
- TDATE – дата прийняття викладача на роботу.

У таблиці УСПІШНІСТЬ (USP), що складається з п'яти полів, зберігається інформація про успішність студентів по навчальних дисциплінах. Поля таблиці наступні:

- UNUM – код факту здачі навчальної дисципліни;
- OCINKA – оцінка, отримана студентом по навчальному предмету;

- UDATE – дата здачі;
- SNUM – номер студентського квитка;
- PNUM – код навчального предмету.

2.1.5. Структура операторів і базові елементи мови

У кожного об'єкту в БД є унікальне ім'я. Імена використовуються в операторах SQL і вказують, над яким об'єктом бази даних оператор повинен виконати дію. У стандарті ANSI/ISO визначено, що імена є у таблиць, стовпців і користувачів. У багатьох діалектах SQL підтримуються також додаткові іменовані об'єкти, такі як процедури, що зберігаються, іменовані відношення і форми для введення даних.

Відповідно до стандарту ANSI/ISO, в SQL імена повинні містити від 1 до 18 символів, починатися з букви і не містити пропуски або спеціальні символи пунктуації. У стандарті SQL2 максимальне число символів в імені збільшене до 128, при цьому на практиці в різних СУБД дозволене використання в іменах таблиць спеціальних символів. Тому для підвищення переносимості краще робити імена порівняно короткими і уникати використання в них спеціальних символів.

Якщо в будь-якому операторі вказане ім'я таблиці, SQL припускає, що відбувається звернення до однієї з власних таблиць користувача. Маючи відповідний дозвіл, можна звертатися до таблиць, власниками яких є інші користувачі, за допомогою повного імені таблиці.

Повне ім'я складається з імені власника таблиці і власне її імені, розділених крапкою. Наприклад, повне ім'я таблиці USP, власником якої є користувач на ім'я DENIS, має наступний вигляд:

```
DENIS.USP
```

Повне ім'я можна використовувати замість простого імені таблиці у всіх операторах SQL. Якщо в операторі задається ім'я поля, SQL сам визначає, в якій з вказаних в цьому ж операторі таблиць міститься дане поле. Проте, якщо в оператор потрібно включити два поля з різних таблиць, але з однаковими іменами, необхідно вказати повні імена, які однозначно визначають їх місцезнаходження. Повне ім'я поля складається з імені таблиці, що містить стовпець, і імені поля, розділених крапкою. Наприклад, повне ім'я поля OCINKA з таблиці USP має наступний вигляд:

```
USP.OCINKA
```

Якщо поле знаходиться в таблиці, власником якої є інший користувач, то в повному імені слід також вказати ім'я користувача. Наприклад, повне ім'я поля OCINKA з таблиці USP, власником якої є користувач DENIS, має наступний вигляд:

```
DENIS.USP.OCENKA
```

Повне ім'я поля можна використовувати замість простого імені у всіх операторах SQL; про виключення говориться при описі конкретних операторів.

Типи даних, які згідно стандарту ANSI/ISO можуть бути присутніми в мові SQL, складаються з символів і різних типів чисел. У свою чергу, останні можна розділити на точні числа і приблизні числа. Точні числові типи – це номери з десятковою крапкою або без такої. Приблизні числові типи – це номери в показниковому записі. Для інших типів даних відмінності не є істотними.

Часто типи даних використовують значення, яке називають розміром аргументу, точний формат і значення якого міняється залежно від конкретного типа. Значення за замовчуванням забезпечені для всіх типів даних, якщо розмір аргументу відсутній.

Для текстових даних використовується тип даних CHAR(довжина) (або CHARACTER) – це рядок тексту, причому розмір аргументу тут ненегативне ціле число, яке визначає максимальну довжину рядка. Значення цього типа, повинні бути поміщені в одиночні лапки, наприклад 'text'. Дві поряд одиночні лапки (‘’) всередині рядка розумітимуться як одна одиночна

лапка (').

У SQL2 допускається використання типу VARCHAR(довжина) (або CHARACTER VARYNG), що дозволяє задавати рядки змінної довжини. Крім того, в цьому стандарті існують типи даних NCHAR(довжина) (або NATIONAL CHARACTER) – рядки символів постійної довжини національних алфавітів (локалізованих символів) і NCHAR VARYNG (довжина) (або NATIONAL CHARACTERVARYNG) – рядки локалізованих символів змінної довжини.

Для точних чисел можуть бути використані наступні типи даних:

- DEC(точність, степінь) (або DECIMAL) – це десяткове число, тобто число, яке може мати десяткову крапку. Тут аргумент розміру має дві частини: точність і степінь, причому степінь не може перевищувати точність. Точність вказує на те, скільки значущих цифр має число. Степінь вказує максимальне число цифр праворуч від десяткової крапки. Якщо степінь рівна нулю, то буде отриманий еквівалент цілого числа;
- NUMERIC – аналог DECIMAL, за винятком того, що максимальне десяткове не може перевищувати аргументу точності;
- INT (або INTEGER) – є число без десяткової крапки. Фактично є еквівалентом DECIMAL, але без цифр праворуч від десяткової крапки, тобто із степенем, рівною нулю. Тут аргумент розміру не використовується;
- SMALLINT – аналог INTEGER, за винятком того, що, залежно від реалізації, розмір за замовчуванням може бути менше, ніж INTEGER.

Приблизні числа можуть бути описані такими типами:

- FLOAT(точність) – число з плаваючою крапкою на основі показникової функції. Аргумент точність складається з одного числа, що визначає мінімальну точність;
- REAL – аналог FLOAT, за винятком того, що ніякого аргументу розміру не використовується, а задана точність встановлюється за замовчуванням;
- DOUBLE PRECISION (або DOUBLE) – еквівалент REAL, за винятком того, що точність для DOUBLE PRECISION повинна перевищувати задану точність REAL.

Для зберігання даних, що характеризують час, використовується тип TIME(точність), а згідно SQL2 може бути використаний спеціальний тип TIMESTAMP(точність) для даних, що містять дату і час. Тут точність визначає представлення часу, наприклад, десяті долі секунди. Нарешті, для зберігання тимчасового інтервалу можна використовувати тип INTERVAL.

Для зчитування і зберігання неструктурованих потоків байтів можна використовувати типи даних BIT(довжина) і BIT VARYNG(довжина), відповідно для рядків бітів постійної і змінної довжини. Ці типи можуть бути використані, наприклад, для зберігання графічних зображень або виконуваного коду.

Якщо є вкладення SQL в інші мови програмування, то значення, використовувані і проведені командами SQL, звичайно зберігаються в змінних головної мови, а значить, ці змінні повинні мати тип даних, сумісний із значеннями SQL, які вони одержуватимуть. У доповненнях, які не є частиною офіційного SQL стандарту, ANSI забезпечує підтримку при використанні вкладення SQL в чотири мови: Паскаль, PL/I, КОБОЛ, і ФОРТРАН.

У SQL є ряд операторів, за допомогою яких реалізуються всі можливості мови. Тепер поговоримо детальніше про реалізацію операторів при роботі з даними.

2.2. Вибірка, або читання даних

2.2.1. Синтаксис оператора SELECT

Запити – це момент, який найбільш часто використовується в SQL, адже ця мова для них і була створена. Запит є деякою командою, яка звертається до БД і повідомляє її, щоб вона відобразила певну інформацію з таблиць в пам'ять. Ця інформація звичайно виводиться безпосередньо на екран комп'ютера, термінал, посилається принтеру, зберігається у файлі або

служить початковими даними для іншої команди або запиту.

Всі запити в SQL складаються з одиночної команди SELECT із достатньо простою структурою, проте шляхом її використання можна виконати складну обробку даних. У найпростішій формі, команда SELECT просто звертається до БД, щоб витягнути інформацію із таблиці. Наприклад, можна вивести таблицю студентів, давши наступний запит:

```
SELECT SNUM,SPRIZ,SIMA,SBAT,STYP FROM STUDENTS;
```

Вивід для цього запиту показаний нижче:

```
SNUM      SPRIZ     SIMA       SBAT           STYP
3412      Поляченко Анатолій   Олексійович    грн.250,50
3413      Старченко Любов      Михайлівна     грн.170,00
3414      Грищенко  Володимир  Миколайович    грн.0,00
3415      Котенко   Анатолій   Миколайович    грн.0,00
3416      Нагірний  Євген      Васильович     грн.250,00
```

Іншими словами, ця команда просто виводить всі дані з таблиці. Більшість програм, що працюють з мовою SQL, видають заголовки полів, тому надалі результати наводитимуться саме в такій формі.

Детально пояснимо кожну частину цієї команди:

SELECT – ключове слово, яке повідомляє БД, що ця команда є запитом, тобто всі запити починаються цим словом.

SNUM,SPRIZ,SIMA,SBAT,STYP – список полів з таблиці, які вибираються запитом. Поля, не перераховані тут, не будуть включені у виведення команди, але це. зрозуміло, не означає, що вони будуть видалені або інформація в них буде стерта з таблиці. Запит не впливає на інформацію в таблицях; він тільки показує дані.

FROM STUDENTS – ключове слово, подібно до SELECT, яке повинне бути представлене в кожному запиті. Воно супроводжується пропуском і потім ім'ям таблиці яка використовується як джерело інформації. В даному випадку – це таблиця студентів STUDENTS.

Крапка з комою (;) використовується у всіх інтерактивних командах SQL для повідомлення БД, що команда заповнена і готова виконатися, а в деяких системах слеш (\), в рядку є індикатором кінця команди.

Очевидно, запит такого характеру не обов'язково упорядковуватиме вивід будь-яким вказаним способом. Та ж сама команда, виконана з тими ж самими даними, але у різний час не зможе вивести результат в однаковому порядку. Звичайно рядки виявляються в тому порядку, в якому вони знайдені в таблиці, а оскільки він довільний, то зовсім не обов'язково зберігатиметься той порядок, в якому дані вводилися або зберігалися. Допускається упорядковувати вивід командами SQL за допомогою спеціальної пропозиції, про яку піде мова нижче, а зараз необхідно мати на увазі, що у відсутність явного впорядкування немає ніякого певного порядку у виведенні результатів запиту.

Якщо необхідно отримати кожне поле таблиці, є необов'язкове скорочення у вигляді символу "зірочка" (*), яке можна використовувати для виведення повного списку полів таким чином:

```
SELECT * FROM STUDENTS;
```

що приведе до того ж результату, що і попередня команда.

У загальному випадку запит починається з ключового слова SELECT, що супроводжується пропуском. Після цього повинен слідувати список розділених комами імен

полів, які необхідно вивести.

Ключове слово FROM, що йде наступним, супроводжується пропуском і ім'ям таблиці, запит до якої робиться. На закінчення, крапка з комою повинна використовуватися для того, щоб закінчити запит і вказати що команда готова до виконання.

Команда SELECT здатна витягнути строго певну інформацію із таблиці. Наприклад, при необхідності виведення тільки певних полів таблиці, просто із списку виключаються не потрібні поля. Наприклад, запит

```
SELECT SNUM, SPRIZ, STYP FROM STUDENTS;
```

буде здійснювати наступний вивід:

```
SNUM       SPRIZ      STYP
3412       Поляченко  грн.250,50
3413       Старченко  грн.170,00
3414       Грищенко   грн.0,00
3415       Котенко    грн.0,00
3416       Нагірний   грн.250,00
```

Цей спосіб дозволяє працювати з таблицями, які мають велику кількість полів, що містять дані, не потрібні в даний момент користувачу.

Не дивлячись на те, що поля таблиці, за визначенням, впорядковані, це зовсім не означає, що їх вивід повинен бути тільки в тому ж порядку. Звичайно, зірочка (*) покаже всі поля в їх природному порядку, але, якщо вказати поля окремо, можна отримати їх в необхідній послідовності.

Наприклад, запит

```
SELECT SPRIZ, SNUM, STYP FROM STUDENTS;
```

буде здійснювати в новій послідовності вивід показаний нижче:

```
SPRIZ      SNUM STYP
Поляченко  3412 грн.250,50
Старченко  3413 грн.170,00
Грищенко   3414 грн.0,00
Котенко    3415 грн.0,00
Нагірний   3416 грн.250,00
```

При роботі з даними дуже часто виникає потреба у видаленні надмірних даних. Це реалізується з використанням DISTINCT – аргумент, який забезпечує можливість усувати значення, що повторюються, з пропозиції SELECT.

Припустимо, що необхідно дізнатися, які студенти в даний час здавали навчальні предмети, причому не потрібне уточнення отриманої оцінки і предмету, що складається. Запит

```
SELECT SNUM FROM USP;
```

надасть наступний вивід проте в ньому є записи-дублікати:

```
SNUM
3412 3413 3414 3412 3416
```

Для отримання списку результатів без дублікатів в даному випадку доцільно скористатися наступним:

```
SELECT DISTINCT SNUM FROM USP;
```

внаслідок чого буде отримано:

```
SNUM
3412 3413 3414 3416
```

Іншими словами, DISTINCT проглядає значення, які були виведені раніше, і не дає їм дублюватися в списку. Це – корисний спосіб уникнути надмірності даних, проте варто уважно стежити за його застосуванням, оскільки можна приховати деяку потрібну інформацію. Наприклад, якщо в таблиці студентів появляться однофамільці, то використання DISTINCT може привести до того, що про існування однофамільців користувач знати не буде.

Слід мати на увазі, що DISTINCT може вказуватися тільки один раз в даній пропозиції SELECT. Якщо пропозиція вибирає багаточисельні поля, DISTINCT опускає записи, де всі вибрані поля ідентичні. Якщо замість DISTINCT вказати ALL, то це матиме протилежний ефект і дублювання рядків виводу збережеться.

2.2.2. Використання умов пошуку для відбору рядків

З часом таблиці стають дуже великими, оскільки збільшується кількість записів, що додаються в них. Звичайно зі всіх записів цікавлять тільки певні, тому SQL дає можливість встановлювати критерії вибору записів для виводу.

WHERE – пропозиція команди SELECT, яка дозволяє встановлювати предикати, умова яких може бути або вірна або невірна для будь-якого запису таблиці. Команда витягує тільки ті записи із таблиці, для якої таке твердження істинне. Припустимо, що необхідно вибрати прізвища і розміри стипендії студентів, при цьому цікавлять тільки такі, які отримують стипендію у розмірі 250.50. Такий запит матиме вигляд:

```
SELECT SPRIZ, STYP FROM STUDENTS WHERE STYP='250,50';
```

Вивід для цього запиту буде наступний:

```
SPRIZ          STYP
Поляченко грн.250,50
```

Коли пропозиція WHERE має місце, СУБД проглядає всю таблицю по одному запису, щоб визначити, чи є предикат істинним. Отже, для запису про студента Поляченко система розгляне поточне значення поля STYP, визначить, що воно рівне 250,50, і включить цей рядок у вивід і т.д.

Пропозиція WHERE сумісна з вже розглянутими фразами, що використовуються в SELECT, тобто можна використовувати найменування полів, усувати дублікати, або переупорядковувати поля. Проте допускається змінювати порядок стовпців для імен тільки в пропозиції SELECT, але не в пропозиції WHERE.

Таким чином, існує декілька способів примусити таблицю надавати ту інформацію, яка необхідна користувачу, а не просто виводити весь її зміст. Найбільш важливо і те, що можна встановлювати предикат, що визначає наявність виводу вказаного рядка таблиці. Предикати можуть ставати дуже складними, надаючи високу точність у вирішенні, які рядки вибирати за

допомогою запиту.

Взагалі кажучи, часто в предикатах потрібно не тільки оцінювати рівність оператора як істинного або помилкового, але і здійснювати інші види зв'язків. Це реалізується за допомогою булевих операторів і знаків відношення, причому предикат може містити необмежене число умов.

В цілому, реляційний оператор – це математичний символ, який вказує на певний тип порівняння між двома значеннями, при цьому SQL має в своєму розпорядженні наступний їх набір:

= рівний чому-небудь;
> більше ніж;
< менше ніж;
>= більше ніж або рівне;
<= менше ніж або рівне;
<> не рівне.

Ці оператори мають стандартні значення для числових даних, а для символьних їх визначення залежить від кодів ASCII символів – вони слідують в алфавітному порядку, причому заголовні букви мають менший код, ніж рядкові, тому, наприклад, "Z"<"a".

Припустимо, що необхідно вивести список студентів, які отримують стипендію, тобто для яких STYP>0. Для цього скористаємося наступним запитом:

```
SELECT * FROM STUDENTS WHERE STYP>'0';
```

Результат запиту буде наступний:

```
SNUM      SPRIZ          SIMA      SBAT            STYP
3412;     Поляченко      Анатолій  Олексійович     грн.250,50
3413      Старченко      Любов     Михайлівна      грн.170,00
3416      Нагірний       Євген     Васильович      грн.250,00
```

Стандартними булевими операторами, які використовуються в SQL, є AND, OR і NOT. Нагадаємо, як вони працюють:

- AND використовує два операнди у формі A AND B і оцінює їх по відношенню до істини: чи вірні вони обидва;
- OR використовує два операнди у формі A OR B і оцінює на істинність: чи вірний один з них;
- NOT використовує один операнд у формі NOT A і замінює його значення з ІСТИНА на БРЕХНЮ, або навпаки.

Пов'язуючи предикати з булевими операторами, можна значно збільшити можливості вибірки даних. Наприклад, по таблиці з даними про успішність можна отримати інформацію про всіх студентів, що склали предмет з кодом 2003:

```
SELECT * FROM USP WHERE OCINKA >=3 AND PNUM = 2003;
```

В результаті буде отримано наступне:

```
UNUM      OCINKA    UDATE          SNUM      PNUM
1002      4         2005-10-06     3413      2003
1004      4         2005-12-06     3412      2003
```

Якщо в аналогічному запиті використовувати OR, то буде отримана інформація про всіх студентів, що мають оцінки 3 і вище, або що складали (незалежно від оцінки) навчальний

предмет з кодом 2003:

```
SELECT * FROM USP WHERE OCINKA >=3 OR PNUM = 2003;
```

Вивід для цього запиту представлений нижче:

```
UNUM       OCİNKA     UDATE           SNUM       PNUM
1001       5          2005-10-06      3412       2001
1002       4          2005-10-06      3413       2003
1003       3          2005-11-06      3414       2005
1004       4          2005-12-06      3412       2003
1005       5          2005-12-06      3416       2004
```

Умова NOT може використовуватися для інвертування логічних значень. Наприклад, для виведення інформації про студентів, у яких оцінки не є 3, можна скористатися наступним запитом:

```
SELECT * FROM USP WHERE NOT (OCINKA = 3);
```

Вивід цього запиту наступний:

```
UNUM       OCENKA     UDATE           SNUM       PNUM
1001       5          2005-10-06      3412       2001
1002       4          2005-10-06      3413       2003
1004       4          2005-12-06      3412       2003
1005       5          2005-12-06      3416       2004
```

Слід мати на увазі, що булевий оператор поміщається перед реляційним оператором, на якого він діє, а при необхідності розширення дії використовуються дужки. Наприклад, для виведення інформації про студентів, у яких оцінки не є 3 і в той же час по навчальному предмету з кодом, не рівним 2005, можна скористатися таким запитом:

```
SELECT * FROM USP WHERE NOT(OCINKA = 3 AND PNUM = 2005);
```

Вивід цього запиту буде таким:

```
UNUM       OCINKA     UDATE           SNUM       PNUM
1001       5          2005-10-06      3412       2001
1002       4          2005-10-06      3413       2003
1004       4          2005-12-06      3412       2003
1005       5          2005-12-06      3416       2004
```

В цьому випадку SQL розуміє круглі дужки як що означають, що все всередині них оцінюватиметься першим і буде оброблятися як єдиний вираз за допомогою того оператора, який знаходиться зовні.

У пропозиції SELECT на додаток до традиційних реляційних і булевих операторів, розглянутих вище, можуть бути використані спеціальні оператори IN, BETWEEN, LIKE, і IS NULL.

Оператор IN визначає набір значень, в який дане значення повинно бути включене. Наприклад, якщо відповідно до навчальної БД виникає необхідність у виведенні інформації про всіх студентів, ім'я яких Анатолій або Володимир, потрібно використовувати наступний запит:

```
SELECT *
FROM STUDENTS
WHERE SIMA = 'Анатолій' OR
SIMA = 'Володимир';
```

Результат цього запиту показаний нижче:

```
SNUM      SPRIZ          SIMA      SBAT           STYP
3412      Поляченко      Анатолій  Олексійович    грн.250,50
3414      Грищенко       Володимир Миколайович    грн.0,00
3415      Котенко        Анатолій  Миколайович    грн.0,00
```

Проте є і простіший спосіб отримати ту ж інформацію за допомогою запиту:

```
SELECT *
FROM STUDENTS
WHERE SIMA IN ('Анатолій', 'Володимир');
```

Звідси ясно, що IN визначає набір значень за допомогою списку, поміщеного в круглі дужки з роздільниками у вигляді ком. Він перевіряє різні значення вказаного поля, намагаючись знайти збіг із значеннями із набору. Якщо це трапляється, то предикат вірний. Якщо набір містить числові значення, а не символьні, то одиночні лапки опускаються. Прикладом такого запиту може служити пошук всіх студентів, що мають стипендію 170,00 і 250,50:

```
SELECT *
FROM STUDENTS
WHERE STYP IN ('170,00', '250,50');
```

Вивід для цього запиту наступний:

```
SNUM      SPRIZ          SIMA      SBAT           STYP
3412      Поляченко      Анатолій  Олексійович    грн.250,50
3413      Старченко      Любов     Михайлівна     грн.170,00
```

Оператор BETWEEN дещо схожий на IN, але на відміну від визначення з набору, BETWEEN визначає діапазон значень, в який повинні поміщатися шукані значення, що і робить предикат вірним. Структура оператора BETWEEN наступна: вводиться початкове значення, ключове слово AND і кінцеве значення. На відміну від IN, BETWEEN розрізняє порядок значень, і, отже, перше із них в пропозиції повинно бути першим по алфавітному або числовому порядку.

Наступний приклад витягуватиме з таблиці успішності номера і оцінки всіх студентів, оцінки яких поміщені між 3 і 5:

```
SELECT SNUM, OCINKA
FROM USP
WHERE OCINKA BETWEEN 3 AND 5;
```

Результат запиту наступний:

```
SNUM      OCINKA
```

```
3412      5
3413      4
3414      3
3412      4
3416      5
```

Зверніть увагу на те, що для оператора BETWEEN значення, співпадаючі з будь-яким з двох значень межі, робить предикат вірним. SQL не володіє безпосередньою підтримкою виключення значень меж для BETWEEN, тому необхідно означати значення так, щоб включаюча інтерпретація оператора була прийнятна, або можна скористатися конструкцією такого типа:

```
SELECT SNUM, OCINKA
FROM USP
WHERE (OCINKA BETWEEN 3 AND 5)
AND NOT OCINKA IN (3, 5);
```

Результат запиту буде наступний:

```
SNUM      OCINKA
3413      4
3412      4
```

BETWEEN може працювати з символьними полями в еквівалентах ASCII, що означає можливість використання BETWEEN для вибору фрагмента з впорядкованих за абеткою значень. Для прикладу приведемо запит, що вибирає всіх студентів, чиї прізвища потрапили в певний алфавітний діапазон:

```
SELECT SPRIZ, SIMA, SBAT
FROM STUDENTS
WHERE SPRIZ BETWEEN 'K' AND 'C';
```

Результати цього запиту показані нижче:

```
SPRIZ     SIMA      SBAT
Поляченко Анатолій  Олексійович
Котенко   Анатолій  Миколайович
Нагірний  Євген     Васильович
```

Відмітимо, що Старченко в результатах відсутній – це відбувається внаслідок того, що BETWEEN порівнює рядки нерівної довжини, а рядок 'C' коротший, ніж рядок Старченко, тобто BETWEEN виводить 'C' із пропусками, які передують символам в алфавітному порядку. Це дуже важливо пам'ятати при використанні BETWEEN для витягання значень з алфавітних діапазонів.

Оператор LIKE застосовний тільки до полів типу CHAR або VARCHAR, в яких він шукає підрядки, тобто він шукає символи і перевіряє, чи співпадають вони з умовою. Як умову оператор використовує групові символи – спеціальні символи, які відповідають чому-небудь. Існує два типу группових символів, що використовуються з LIKE:

■ символ підкреслення заміщає будь-який одиночний символ, наприклад, 'М_Л' відповідатиме словам 'МОЛ' або 'МЕЛ', але не відповідатиме 'МЕТАЛ';

■ знак відсотка заміщає послідовність будь-якого числа символів, зокрема нульової

довжини. Наприклад, '%М%Л' відповідатиме словам 'МЕЛ' або 'ПОМЕЛ', але не відповідає 'МОЛОКО'.

Як приклад знайдемо всіх викладачів, чиї прізвища починаються з літери К.

```
SELECT TPRIZ, TIMA, TBAT
FROM TEACHERS
WHERE TPRIZ LIKE 'К%';
```

Результат запиту буде наступний:

```
TPRIZ          TIMA      TBAT
Костиркін      Олег      Володимирович
Казанко        Віталій   Володимирович
```

Оператор LIKE може бути корисний, наприклад, при пошуку значення, якщо точне його написання невідоме. Наприклад, якщо неясно, як пишеться прізвище викладача Казанко, Казанчиків або Козанко, то можна використовувати відому частину і групові символи для знаходження всіх можливих значень:

```
SELECT TPRIZ, TIMA, TBAT
FROM TEACHERS
WHERE TPRIZ LIKE 'К_занко%';
```

Результат запиту приведений нижче:

```
TPRIZ     TIMA      TBAT
Казанко   Віталій   Володимирович
```

Груповий символ % в кінці рядка необхідний в більшості випадків, якщо довжина оцінюваного рядка невідома або довжина поля більше, ніж число символів в оцінюваному рядку.

Нарешті, виникає закономірне питання, пов'язане з необхідністю пошуку знаку відсотка або підкреслення в рядку. У LIKE предикаті можна визначити будь-який одиночний символ, як символ ESC. Символ ESC використовується відразу перед відсотком або підкресленням в предикаті і означає, що відсоток або підкреслення інтерпретуватиметься як звичайний, а не як груповий символ. Наприклад, наступний запит дозволяє знайти знак підкреслення в даних про прізвище викладача:

```
SELECT TPRIZ, TIMA, TBAT
FROM TEACHERS
WHERE TPRIZ LIKE '%/_%'ESCAPE'/';
```

Оскільки в даних про прізвище знаку підкреслення немає, то результат виводу не міститиме даних. В даному прикладі рядок порівнюється із вмістом будь-якої послідовності символів (перший знак %), що супроводжуються символом підкреслення (/). і будь-якою послідовністю символів в кінці рядка (другий знак %).

Іноді в таблиці виникають записи, які не мають жодних значень для кожного поля, наприклад тому що інформація не завершена або тому що це поле просто не заповнювалося. SQL враховує такий варіант, дозволяючи вводити порожнє значення NULL в поле замість значення. Коли значення поля рівне NULL, це означає, що СУБД спеціально позначила це поле, як що не має жодного значення для цього запису. Це принципово відрізняється від простого

значення в полі, значення нуля або пропуску, які БД оброблятиме так само, як і будь-яке інше значення. Оскільки NULL не є наочним значенням, він не має і типу даних і може поміщатися в будь-який тип поля.

Наприклад, в таблицю успішності можна ввести інформацію про студента, навчальний предмет і відому дату іспиту, проте, оскільки оцінка ще не виставлена, в полі OCINKA знаходитиметься значення NULL до тих пір, поки оцінка не стане відома.

Оскільки NULL вказує на відсутність значення, не можна знати, який буде результат будь-якого порівняння з використанням NULL. Дійсно, коли NULL порівнюється з будь-яким значенням, результат буде невідомий, зокрема, запис, що дав невідоме значення в предикаті не буде вибраний запитом.

Зрозуміло, що досить часто необхідно розрізняти невірне і невідоме значення, для чого в SQL використовують спеціальний оператора IS із ключовим словом NULL.

Знайдемо, наприклад, всі записи в таблиці із інформацією про успішність із значеннями NULL в полі OCINKA:

```
SELECT *
FROM USP
WHERE OCINKA IS NULL;
```

Даних як результату цього запиту не буде, тому що в таблиці відсутні значення NULL в полі OCINKA.

Спільно з розглянутими операторами, можуть бути використані і традиційні функції логічного відношення AND, OR і NOT.

Зокрема, для отримання предикатів, зміст яких протилежний розглянутим реляційним операторам, використовують заперечення NOT. Наприклад, якщо необхідно усунути NULL з виводу інформації про успішність, можна скористатися запитом, що містить NOT в предикаті:

```
SELECT *
FROM USP
WHERE OCINKA NOT NULL;
```

В результаті буде отримано наступне:

UNUM	OCINKA	UDATE	SNUM	PNUM
1001	5	2005-10-06	3412	2001
1002	4	2005-10-06	3413	2003
1003	3	2005-11-06	3414	2005
1004	4	2005-12-06	3412	2003
1005	5	2005-12-06	3416	2004

Тобто за відсутності значень NULL буде виведена вся таблиця успішності. До речі, розглянутий запит аналогічний наступному, такому, що дає такий же результат:

```
SELECT *
FROM USP
WHERE NOT OCINKA IS NULL;
```

Допускається використовувати NOT з IN, наприклад, для виведення даних про успішність студентів, номер квитка яких не 3412 і 3413, причому запропонуємо це зробити двома способами:

```
SELECT *
FROM USP
WHERE SNUM NOT IN (3412, 3413);
```

або

```
SELECT *
FROM USP
WHERE NOT SNUM IN (3412, 3413);
```

що в обох випадках надасть наступний вивід:

```
UNUM      OCINKA    UDATE          SNUM      PNUM
1003      3         2005-11-06     3414      2005
1005      5         2005-12-06     3416      2004
```

Аналогічним способом можна використовувати NOT BETWEEN і NOT LIKE, тобто можна використовувати весь набір стандартних математичних і спеціальних операторів.

2.2.3. Отримання підсумкових даних

Взагалі кажучи, запити можуть проводити узагальнену групову обробку значень полів, що реалізується за допомогою агрегатних функцій. Агрегатні функції проводять одиночне значення для всієї групи таблиці. У SQL допускаються наступні агрегатні функції:

- COUNT – робить підрахунок кількості рядків або не-NULL значень полів, які вибрав запит;
- SUM – розраховує арифметичну суму всіх вибраних значень даного поля;
- AVG – проводить усереднювання всіх вибраних значень даного поля;
- MAX – знаходить і повертає найбільше зі всіх вибраних значень даного поля;
- MIN – знаходить і повертає найменше зі всіх вибраних значень даного поля.

Агрегатні функції використовуються подібно до імен полів в пропозиції SELECT запиту, але з врахуванням того, що вони беруть імена поля як аргументи. Варто мати на увазі, що із SUM і AVG використовуються тільки числові поля, а з COUNT, MAX і MIN можуть використовуватися числові або символьні поля. Коли функції записуються з символьними полями, MAX і MIN використовують їх в еквівалент ASCII, відповідно до якого вибирається максимальне або мінімальне значення.

Щоб знайти суму всієї виплаченої стипендії в таблиці з даними про студентів, можна використовувати наступний запит:

```
SELECT SUM (STYP)
     FROM STUDENTS;
```

вивід якого складається з одного числа грн.670,50.

Дія агрегатних функцій, як видно, відрізняється від вибору поля, при якому повертається, наприклад, одиночне значення. З цієї причини агрегатні функції і поля не можуть спільно використовуватися без пропозиції GROUP BY, про яку мова піде нижче.

Функція COUNT дещо відрізняється від інших – як вже було відмічено, вона рахує кількість значень в даному стовпці, або число рядків в таблиці. Зокрема, для підрахунку кількості значень в стовпці, вона використовується з DISTINCT. Наприклад, можна підрахувати кількість студентів, що складали навчальні предмети по таблиці успішності за допомогою наступного запиту:

```
SELECT COUNT (DISTINCT SNUM)
```

```
FROM USP;
```

Як результат цього запиту буде отримане значення 4. Зверніть увагу на обов'язкову вимогу того, щоб DISTINCT був поміщений в круглі дужки на відміну від попередніх прикладів. Крім того, допускається можливість використання DISTINCT з будь-якими агрегатними функціями, але найчастіше він використовується з COUNT.

Для того, щоб підрахувати загальне число рядків в таблиці, використовують функцію COUNT із зірочкою замість імені поля, наприклад як в наступному прикладі:

```
SELECT COUNT (*)
     FROM STUDENTS;
```

що як результат дасть значення 5.

COUNT із зірочкою включає записи з NULL значеннями, а також дублікати, з цієї причини DISTINCT в даному випадку не може бути використаний.

Агрегатні функції в більшості реалізацій допускають використання аргументу ALL, який поміщається перед ім'ям поля аналогічно DISTINCT, проте означає протилежну дію – включати дублікати. Необхідно пояснити відмінності між ALL і *, коли вони використовуються з COUNT – ALL використовує ім'я поля як аргумент і не може підрахувати значення NULL. При цьому варто пам'ятати, що функції, відмінні від COUNT, ігнорують значення NULL у будь-якому випадку. Наступний приклад підрахує кількість не-NULL значень в полі SNUM, включаючи повторення:

```
SELECT COUNT (ALL SNUM)
     FROM USP;
```

Як результат цього запиту буде отримане значення 5.

У SQL допускається використовувати агрегатні функції з аргументами, які складаються з виразів, що включають одне або більше полів, при цьому команда DISTINCT не дозволяється. Припустимо, що необхідно знайти максимальну величину проіндексованої (у прикладі, збільшеної вдвічі) стипендії. Для кожного рядка таблиці такий запит повинен множити STYP на 2 і вибирати найбільше значення, яке буде знайдене. Для цього можна скористатися наступним:

```
SELECT MAX (STYP*2)
     FROM STUDENTS;
```

Як результат тут буде отримане число грн.501,00. Команда GROUP BY дозволяє визначати підмножину значень в полі в термінах іншого поля, і застосовувати функцію агрегату до такої підмножини. Це дає можливість об'єднувати поля і агрегатні функції в єдиній пропозиції SELECT. Наприклад, якщо виникає необхідність у визначенні найменшої оцінки, отриманої кожним студентом, то можна зробити з використанням GROUP BY наступний запит:

```
SELECT SNUM, MIN (OCINKA)
     FROM USP
     GROUP BY SNUM;
```

Вивід для цього запиту показаний нижче:

```
SNUM
3416 5
3414 3
```

```
3413 4
3412 4
```

Команда GROUP BY може застосовуватися з агрегатними функціями незалежно від серій груп, які визначаються за допомогою значення поля в цілому. В цьому випадку кожна група складається зі всіх рядків з тим же самим значенням поля SNUM, і MIN функція застосовується окремо для кожної такої групи. Значення поля, до якого застосовується GROUP BY, має тільки одне значення на групу виводу, так само як це робить агрегатна функція. Тому в результаті і з'являється сумісність, яка дозволяє агрегатним функціям і полям об'єднатися.

В принципі, допускається використання GROUP BY одночасно з декількома полями. Для прикладу модифікуємо попередній випадок так, щоб виводилося найменше значення оцінки за кожен день. При цьому запит буде наступний:

```
SELECT SNUM, UDATE, MIN (OCINKA)
     FROM USP
     GROUP BY SNUM, UDATE;
```

Вивід для цього запиту буде наступний:

```
SNUM      UDATE
3412      2005-10-06     5
3413      2005-10-06     4
3416      2005-12-06     5
3412      2005-12-06     4
3414      2005-11-06     3
```

Тепер припустимо, що в попередньому прикладі, необхідно побачити тільки мінімальну оцінку, меншу 5. Безпосередньо використовувати агрегатну функцію в пропозиції WHERE в цьому випадку не можна, тому що предикати оцінюються в термінах одиночного рядка, а агрегатні функції – груп рядків. Тому необхідно скористатися пропозицією HAVING, що визначає критерії, що використовуються для видалення певних груп із виводу, на зразок тому, як це робить пропозиція WHERE для індивідуальних рядків. При цьому запит буде наступний:

```
SELECT SNUM, UDATE, MIN (OCINKA)
     FROM USP
     GROUP BY SNUM, UDATE
     HAVING MIN (OCINKA)<5;
```

Вивід для цього запиту приведений нижче:

```
SNUM      UDATE
3413      2005-10-06     4
3412      2005-12-06     4
3414      2005-11-06     3
```

Аргументи в пропозиції HAVING підкоряються тим же самим правилам, що і в пропозиції SELECT, що складається з команд, які використовують GROUP BY: вони повинні мати одне значення на групу виводу. Наприклад, наступна команда буде заборонена:

```
SELECT SNUM, MIN (OCINKA)
     FROM USP
```

```
GROUP BY SNUM,
HAVING UDATE='10/06/2005';
```

Поле UDATE не може бути викликане пропозицією HAVING, тому що воно може мати більше ніж одне значення на групу виводу, отже, для уникнення такої ситуації, пропозиція HAVING повинна посилатися тільки на агрегатні функції і поля, вибрані в GROUP BY. Правильний спосіб зробити розглянутий запит наступний:

```
SELECT SNUM, MIN (OCINKA)
     FROM USP
     WHERE UDATE = '10/06/2005'
     GROUP BY SNUM;
```

Результат такого запиту наступний:

```
SNUM
3412      5
3413      4
```

Оскільки поле UDATE відсутнє у виводі, то для підвищення легкості читання в результати варто включити текст, що пояснює, що це за дані.

Як говорилося вище, HAVING може використовувати тільки аргументи, які мають одне значення на групу виводу. Практично ж посилання на поля, вибрані за допомогою GROUP BY, також допустимі. Наприклад, для виводу найменших оцінок для студентів з номерами 3412 і 3413 можна скористатися наступним:

```
SELECT SNUM, MIN (OCINKA)
     FROM USP
     GROUP BY SNUM
     HAVING SNUM IN (3412, 3413);
```

Вивід для цього запиту наступний:

```
SNUM
3413      4
3412      4
```

На закінчення відмітимо, що слід бути уважним при використанні таких функцій: не можна використовувати агрегатну функцію від агрегатної функції.

З врахуванням вищесказаного, з даного моменту можна використовувати запити для отримання певних значень і формувати вивід даних відповідно до вимог поточного завдання користувача.

2.2.4. Сортування результатів запиту

Більшість БД, що працюють з SQL, надають спеціальні засоби, що дозволяють удосконалювати виведення запитів.

Припустимо, що є необхідність виконати прості числові обчислення із даними, що виводяться як результат запиту. SQL дозволяє помішати вирази і константи серед вибраних полів. Ці вирази можуть доповнювати або заміщати поля в пропозиціях SELECT, при цьому вони можуть включати одне або більше вибраних полів. Наприклад, якщо необхідно проглянути проіндексовану стипендію, збільшивши її в два рази, то можна скористатися запитом:

```
SELECT SPRIZ, SIMA, SBAT, STYP*2
     FROM STUDENTS;
```

Вивід цього запиту буде таким:

```
SPRIZ      SIMA       SBAT
Поляченко  Анатолій   Олексійович     грн.501,00
Старченко  Любов      Михайлівна      грн.340,00
Грищенко   Володимир  Миколайович     грн.0,00
Котенко    Анатолій   Миколайович     грн.0,00
Нагірний   Євген      Васильович      грн.500,00
```

Зверніть увагу на те, що останній стовпець без найменування, тому що це – стовпець виводу, тобто це – стовпці даних, створені запитом способом, іншим, ніж просто видобування їх з таблиці. Такі стовпці створюються кожного разу, коли використовуються функції, константи або вирази в пропозиції SELECT запиту. Оскільки ім’я стовпця – один з атрибутів таблиці, стовпці які з'являються не із таблиць, не мають жодних імен.

Достатньо часто виникає необхідність в розміщенні тексту у виведенні запиту. Наприклад, для підвищення зручності роботи з результатами попереднього запиту, можна вставити короткий коментар щодо проіндексованої стипендії – станом на який місяць, що виконується наступним запитом:

```
SELECT SPRIZ, SIMA, SBAT, 'листопад', STYP*2
     FROM STUDENTS;
```

Вивід цього запиту буде наступний:

```
SPRIZ      SIMA       SBAT
Поляченко  Анатолій   Олексійович     листопад  грн.501,00
Старченко  Любов      Михайлівна      листопад  грн.340,00
Грищенко   Володимир  Миколайович     листопад  грн.0,00
Котенко    Анатолій   Миколайович     листопад  грн.0,00
Нагірний   Євген      Васильович      листопад  грн.500,00
```

Варто мати на увазі, що всі символи, в т.ч. пропуски, в рядку тексту також вставляються у вивід, тому запропонований спосіб можна використовувати для маркіровки виводу разом з коментарями, що вставляються. Проте необхідно пам'ятати, що цей же самий коментар буде надрукований в кожному рядку виводу, а не просто один раз для всієї таблиці. Припустимо, що необхідний звіт про кількість студентів, одержуючих ту або іншу стипендію, тоді можна запропонувати наступний запит:

```
SELECT COUNT (DISTINCT SNUM),
'студенти отримують стипендію',
STYP, 'за листопад'
FROM STUDENTS
GROUP BY STYP;
```

В результаті буде отримано:

```
2    студенти отримують стипендію  грн.0,00  за листопад
```

```
1    студенти отримують стипендію  грн.170,00     за листопад
1    студенти отримують стипендію  грн.250,00     за листопад"
1    студенти отримують стипендію  грн.250,50     за листопад
```

Некоректність виведення тексту для стипендії грн.170,00; грн.250,00 не можна уникнути, не створивши складнішої конструкції для виводу, ніж запропонована. Як можна бачити, незмінний коментар для кожного рядка таблиці може бути дуже корисний, але він має ряд обмежень. Іноді корисно вивести один коментар для всього виводу в цілому або проводити свій – власний коментар для кожного рядка, проте це забезпечують різні програми, що використовують SQL і що мають засоби генератора звітів.

Для впорядкування виводу полів таблиць SQL використовує команду ORDER BY, дозволяючи сортувати вивід запиту згідно значень в тій або іншій кількості вибраних стовпців. Якщо вказується декілька полів, то стовпці виводу упорядковуються один всередині іншого, при цьому можна визначати зростання (ASC) або спадання (DESC) для кожного стовпця. За замовчуванням встановлене зростання.

Як приклад використаємо запит, що виводить таблицю з інформацією про студентів в алфавітному порядку прізвищ:

```
SELECT * FROM STUDENTS
ORDER BY SPRIZ ASC;
```

Вивід цього запиту приведений нижче:

```
SNUM      SPRIZ     SIMA      SBAT           STYP
3414      Грищенко  Володимир Миколайович    грн.0,00
3415      Котенко   Анатолій  Миколайович    грн.0,00
3416      Нагірний  Євген     Васильович     грн.250,00
3412      Поляченко Анатолій  Олексійович    грн.250,50
3413      Старченко Любов     Михайлівна     грн.170,00
```

Приведемо приклад для впорядковування інформації по декількох стовпцях. Наприклад, інформацію з таблиці із даними про студентів упорядкуємо по зменшенню розміру стипендії, а для студентів, що мають однаковий її розмір – в алфавітному порядку їх прізвищ. Для цього скористаємося запитом:

```
SELECT * FROM STUDENTS
ORDER BY STYP DESC, SPRIZ ASC;
```

Результати запиту наступні:

```
SNUM      SPRIZ     SIMA      SBAT           STYP
3412      Поляченко Анатолій  Олексійович    грн.250,50
3416      Нагірний  Євген     Васильович     грн.250,00
3413      Старченко Любов     Михайлівна     грн.170,00
3414      Грищенко  Володимир Миколайович    грн.0,00
3415      Котенко   Анатолій  Миколайович    грн.0,00
```

Аналогічним чином допускається використовувати ORDER BY відразу з будь-яким числом стовпців, проте поля, по яких відбувається впорядковування, повинні бути вказані в SELECT. Тому запит вигляду:

```
SELECT SNUM, STYP FROM STUDENTS
     ORDER BY SPRIZ ASC;
```

буде заборонений, оскільки поле SPRIZ не було вибраним полем, і GROUP BY не зміг його знайти для впорядкування виводу.

ORDER BY може використовуватися з GROUP BY для впорядкування груп, при цьому ORDER BY повинен бути останнім. Наприклад, виведемо вже розглянутий нами звіт про кількість студентів, що одержують ту або іншу стипендію, але з впорядковуванням по спаданню розмірів їх стипендій:

```
SELECT COUNT (DISTINCT SNUM),
'студ. отримують стипендію',
STYP, 'за листопад'
FROM STUDENTS
GROUP BY STYP
ORDER BY STYP DESC;
```

В результаті буде отримано:

```
                                    STYP
1    студ. отримують стипендію      грн.250,50     за листопад
1    студ. отримують стипендію      грн.250,00     за листопад
1    студ. отримують стипендію      грн.170,00     за листопад
2    студ. отримують стипендію      грн.0,00   за листопад
```

Замість імен стовпців, можна вказувати їх порядкові номери для вказівки поля, що використовується у впорядкуванні виводу. Ці номери можуть посилатися не на порядок стовпців в таблиці, а на їх порядок у виводі. Таким чином, поле, згадане в пропозиції SELECT першим, для ORDER BY має номер 1 незалежно від того, яким по порядку воно стоїть в таблиці. Наприклад, запит, що виводить інформацію про студентів в алфавітному порядку прізвищ, можна записати так:

```
SELECT SNUM, SPRIZ, SIMA
     FROM STUDENTS
     ORDER BY 2 ASC;
```

Вивід цього запиту приведений нижче:

```
SNUM       SFAM       SIMA
3414       Грищенко   Володимир
3415       Котенко    Анатолій
3416       Нагірний   Євген
3412       Поляченко  Анатолій
3413       Старченко  Любов
```

Основна мета ключового слова ORDER BY – дати можливість використовувати цю команду із стовпцями виводу так само, як і із стовпцями таблиці – адже інколи потрібно провести впорядковування виводу по стовпцях, що виробляються агрегатною функцією, константами або виразами в пропозиції SELECT запиту. Наприклад, спробуємо розглянути звіт про кількість студентів, що одержують ту або іншу стипендію, але з впорядковуванням по спаданню кількості студентів:

```
SELECT COUNT (DISTINCT SNUM),
     'студ. отримують стипендію',
     STYP, 'за листопад'
     FROM STUDENTS
     GROUP BY STYP
     ORDER BY 1 DESC;
```

В результаті буде наступний вивід:

		STYP	
2	студ. отримують стипендію	грн.0,00	за листопад
1	студ. отримують стипендію	грн.250,00	за листопад
1	студ. отримують стипендію	грн.250,50	за листопад
1	студ. отримують стипендію	грн.170,00	за листопад

Отже, необхідно використовувати номер стовпця, оскільки стовпець виводу не має імені; і не можна використовувати саму агрегатну функцію.

Таким чином, використовуючи команду ORDER BY можна примусити запити впорядковувати вивід даних для підвищення зручності користування ними.

2.2.5. Прості запити і правила їх виконання

Вище нами розглянуті запити, які використовували як джерело даних тільки одну таблицю даних – так звані однотабличні запити. Проте часто виникає необхідність у виборі інформації з декількох таблиць – одним із варіантів здійснення такого виводу є об'єднання результатів декількох запитів, що виконуються незалежно один від одного.

Для розміщення декількох запитів разом і об'єднання їх виводу використовують пропозицію UNION. Пропозиція UNION об'єднує вивід двох або більше SQL запитів в єдиний набір рядків і стовпців. Наприклад, для отримання списку всіх студентів і викладачів, прізвища яких поміщені між літерами 'К' і 'С', можна скористатися запитом:

```
SELECT SPRIZ, SIMA, SBAT
     FROM STUDENTS
     WHERE SPRIZ BETWEEN 'К' AND 'С'
     UNION
     SELECT TPRIZ, TIMA, TBAT
     FROM TEACHERS
     WHERE TPRIZ BETWEEN 'К' AND 'С';
```

Результати цього запиту показані нижче:

```
Казанко   Віталій   Володимирович
Костиркін Олег      Володимирович
Котенко   Анатолій  Миколайович
Нагірний  Євген     Васильович
Позняк    Любов     Олексіївна
Поляченко Анатолій  Олексійович
```

Звідси можна зробити висновок, що записи, вибрані двома командами, виведені так, як би вона була одна. Природно, що заголовки стовпців виключені, тому що жоден із стовпців, виведених об'єднанням, не був витягнутий безпосередньо з тільки однієї таблиці, отже, всі ці стовпці виводу не мають жодних імен. Зверніть увагу на те, що тільки останній запит

закінчується крапкою з комою – відсутність крапки з комою дає зрозуміти SQL, що є ще один або більше запитів.

Коли два або більше запити піддаються об'єднанню, їх стовпці виводу повинні бути сумісні для об'єднання, що нами вже розглядалося вище. Нагадаємо, що це означає для кожного запиту необхідність приєднання однакового числа стовпців в тому ж порядку, що і перший, другий, третій, і т. д., і при цьому повинна бути присутньою сумісність типів. Наприклад, символьні поля повинні мати однакове число символів. Інше обмеження на сумісність – це коли пусті значення NULL заборонені в будь-якому стовпці об'єднання, тоді ці значення необхідно заборонити і для всіх відповідних стовпців в інших запитах об'єднання. Нарешті, не можна використовувати агрегатні функції в пропозиції SELECT запиту в об'єднанні.

Варто звернути увагу на той факт, що UNION автоматично виключатиме дублікати рядків із виводу. Якщо, наприклад, в таблиці STUDENTS появиться ще один студент із прізвищем Поляченко, то запит

```
SELECT SPRIZ
     FROM STUDENTS;
```

дасть наступний вивід:

```
SPRIZ
Поляченко
Старченко
Грищенко
Котенко
Нагірний
Поляченко
```

Тут є дублювання значень SPRIZ = Поляченко, тому що не вказано, щоб SQL усунув дублікати. Проте, при використанні UNION в комбінації цього запиту з йому подібним в таблиці викладачів, надлишкова інформація буде усунена.

```
SELECT SPRIZ
     FROM STUDENTS
     UNION
          SELECT TPRIZ
          FROM TEACHERS;
```

дає наступні результати без дублювання прізвища Поляченко:

```
SPRIZ
Вікулін
Грищенко
Загарийчук
Казанко
Костиркін
Котенко
Нагірний
Позняк
Поляченко
Старченко
```

Іноді виникає необхідність вставляти константи і вирази в пропозиції SELECT, що використовуються з UNION, що є корисною можливістю. Очевидно, що константи і вирази які використовуються повинні зустрічати сумісні типи даних. Ця властивість корисна, наприклад, щоб відобразити коментарі, що вказують на те, який запит вивів даний рядок. Модифікуємо попередній запит таким чином:

```
SELECT 'Студент', SPRIZ
     FROM STUDENTS
     UNION
          SELECT 'Викладач', TPRIZ
          FROM TEACHERS;
```

вивід цього запиту наступний:

```
Викладач   Вікулін
Викладач   Загарийчук
Викладач   Казанко
Викладач   Костиркін
Викладач   Позняк
Студент    Грищенко
Студент    Котенко
Студент    Нагірний
Студент    Поляченко
Студент    Старченко
```

Тепер необхідно обговорити те, що дані багаточисельних запитів виводитимуться в якомусь особливому порядку. Для цього можна використовувати пропозицію ORDER BY, щоб впорядкувати вивід з об'єднання, аналогічно тому, як це робиться в однотабличних запитах. Наприклад, переглянемо останній запит у зв'язку з тим, щоб впорядкувати прізвища:

```
SELECT 'Студент', SPRIZ
     FROM STUDENTS
     UNION
          SELECT 'Викладач', TPRIZ
          FROM TEACHERS
     ORDER BY 2 ASC;
```

Вивід цього запиту буде наступний:

```
Викладач  Вікулін
Студент   Грищенко
Викладач  Загарийчук
Викладач  Казанко
Викладач  Костиркін
Студент   Котенко
Студент   Нагірний
Викладач  Позняк
Студент   Поляченко
Студент   Старченко
```

Допускається впорядковувати вивід за допомогою декількох полів, одне всередині

іншого і вказувати фрази ASC або DESC для кожного, точно так, як це робиться для одиночних запитів. Необхідно відмітити, що номер 2 в пропозиції ORDER BY вказує, який стовпець з пропозиції SELECT буде впорядкований, оскільки стовпці об'єднання – це стовпці виводу, а значить, вони не мають імен і повинні визначатися по номеру.

Крім того, при виконанні об'єднання більш ніж двох запитів, можна використовувати круглі дужки для того, щоб визначити порядок оцінки. Тобто замість просто запиту

```
query A UNION query B UNION query C;
```

можна вказати

```
(query A UNION query B) UNION query C;
```

або

```
query A UNION (query B UNION query C);
```

Це може принципово вплинути на результати запиту, оскільки об'єднання здійснюється спочатку всередині дужок, а потім –ззовні. У такий спосіб, наприклад, запити можуть бути скомбіновані для видалення одних дублікатів і залишення інших.

2.2.6. Особливості багатотабличних запитів

Найважливішою особливістю запитів SQL є їх здатність визначати зв'язки між декількома таблицями і виводити інформацію з них в термінах цих зв'язків. Такі операції називаються об'єднанням, яке є одним з видів операцій в реляційних БД – адже це є основою реляційного підходу до зберігання даних в таблицях. Використовуючи об'єднання, відбувається безпосереднє зв'язування інформації із будь-яким номером таблиці, і, таким чином, створюються зв'язки між порівнянними частинами даних.

При багатотабличному запиті, таблиці, представлені у вигляді списку в пропозиції FROM, відокремлюються одна від одної комами. Предикат запиту може посилатися до будь-якого стовпця будь-якої зв'язаної таблиці і, отже, може використовуватися для зв'язку між ними. Звичайно предикат порівнює значення в стовпцях різних таблиць, щоб визначити, чи задовольняє WHERE встановленій умові. До цього імена таблиць в запитах опускалися, тому запрошувалася тільки одна таблиця одночасно. Навіть при запиті із декількох таблиць допускається опускати їх імена, якщо, звичайно, вони різні. Тепер же виникає необхідність використання імен стовпців і таблиць, оскільки в багатотабличному запиті можуть виникати неоднозначності.

Припустимо необхідно поставити у відповідність викладачу навчальні предмети, які він веде. Фактично SQL доведеться вибирати з таблиці викладачів відповідний йому код і, проглядаючи таблицю предметів, здійснювати пошук відповідного коду. Це можна реалізувати наступним запитом:

```
SELECT TEACHERS.TPRIZ, PREDMET.PNAME
     FROM TEACHERS, PREDMET
     WHERE TEACHERS.TNUM = PREDMET.TNUM;
```

Вивід цього запиту представлений нижче:

```
TPRIZ          PNAME
Вікулін        Фізика
Костиркін      Хімія
Казанко        Математика
```

```
Загарийчук      Філософія
Позняк          Економіка
```

Оскільки поле TNUM є і в таблиці викладачів, і в таблиці предметів, то імена таблиць повинні використовуватися як префікси, хоча це необхідно тільки у тому випадку, коли два або більше полів мають одне і те ж ім'я. Проте у будь-якому випадку включати ім'я таблиці в запити з об'єднанням зручно для кращого розуміння і несуперечності отримуваних результатів.

При виконанні багатотабличного запиту, SQL досліджує кожну комбінацію рядків двох або більше таблиць і перевіряє ці комбінації по їх предикатах. Якщо комбінація проводить таке значення, яке робить предикат вірним, то значення буде вибрано для виводу.

У попередньому прикладі був встановлений зв'язок між двома таблицями в об'єднанні, але, взагалі кажучи, ці таблиці, вже були з'єднані через поле TNUM. Цей зв'язок називається станом довідкової цілісності, і, використовуючи таке об'єднання в багатотабличному запиті, можна витягувати дані в термінах цього зв'язку. Значить в розглянутому прикладі об'єднання стовпці використовуються для визначення предиката запиту, при цьому TNUM стовпці видалені із виводу для обох таблиць, оскільки і без цього зрозуміло, які викладачі ведуть той або інший навчальний предмет.

У свою чергу, об'єднання в багатотабличних запитах, які використовують предикати, що базуються на рівності, називаються об'єднаннями по рівності. Об'єднання по рівності це найбільш загальний вид об'єднання, проте існують і інші види об'єднань – фактично можна використовувати будь-який з реляційних операторів. Прикладом іншого виду об'єднання може служити наступний запит:

```
SELECT TEACHERS.TPRIZ, PREDMET.PNAME
     FROM TEACHERS, PREDMET
     WHERE TEACHERS.TPRIZ < PREDMET.PNAME
          AND TEACHERS.TPRIZ BETWEEN 'K' AND 'C';
```

Вивід цього запиту такий:

```
TPRIZ      PNAME
Костиркін Фізика
Казанко    Фізика
Позняк     Фізика
Костиркін Хімія
Казанко    Хімія
Позняк     Хімія
Костиркін Математика
Казанко    Математика
Костиркін Філософія
Казанко    Філософія
Позняк     Філософія
```

В даному випадку запит навряд чи матиме практичне значення – з його допомогою виводиться інформація про викладачів, прізвище яких за абеткою знаходиться раніше, ніж найменування навчального предмету з таблиці PREDMET, при цьому вибираються викладачі з прізвищем, яке поміщене між літерами К і С.

Допускається також створювати запити, об'єднуючи більше двох таблиць. Наприклад, необхідно вивести список оцінок, виставлених тим або іншим викладачем. Тоді запит об'єднуватиме відразу три таблиці:

```
SELECT TEACHERS.TPRIZ, USP.OCINKA
     FROM TEACHERS, PREDMET, USP
     WHERE TEACHERS.TNUM = PREDMET.TNUM
          AND PREDMET.PNUM = USP.PNUM;
```

В результаті буде отримано:

```
TPRIZ          OCINKA
Вікулін        5
Казанко        4
Позняк         3
Казанко        4
Загарийчук     5
```

Таким чином, при об'єднанні таблиць, добре проглядається логіка зв'язку між даними і можна більше не обмежуватися переглядом однієї таблиці в кожен момент часу. Крім того, об'єднання дозволяє робити складні порівняння між будь-якими полями будь-якого числа таблиць і використовувати отримані результати для того, щоб вирішувати – яку інформацію хотілося б бачити.

2.2.7. Об'єднання таблиць

Вище було показано, що в багатотабличному запиті можна об'єднувати дані таблиць, проте цікаво і те, що та ж сама методика може використовуватися для об'єднання разом двох копій одиночної таблиці. Для об'єднання таблиці з собою можна зробити кожен рядок таблиці одночасно і комбінацією її з собою і комбінацією з кожним іншим рядком таблиці, а потім оцінити кожну комбінацію в термінах предиката. Це дозволяє легко створювати певні види зв'язків між різними елементами всередині одиночної таблиці. Наприклад, допускається зобразити об'єднання таблиці з собою, як об'єднання двох копій однієї і тієї ж таблиці, причому вона насправді не копіюється, але SQL виконує команду так, як якби це було зроблено.

Використання команди для об'єднання таблиці з собою аналогічно тому прийому, який використовується для об'єднання декількох таблиць. Коли об'єднується таблиця з собою, всі повторювані імена стовпця заповнюються префіксами імені таблиці. Щоб посилатися до цих стовпців всередині запиту, необхідно мати два різні імена для цієї таблиці. Це можна зробити за допомогою визначення тимчасових імен, які називаються псевдонімами, що означаються в пропозиції FROM запиту. Синтаксис в цьому випадку наступний: після імені таблиці залишають пропуск, а потім повинен слідувати псевдонім для неї.

Наприклад, для пошуку студентів, що мають однаковий розмір стипендії, можна скористатися наступним запитом:

```
SELECT FIRST.SPRIZ, SECOND.SPRIZ, FIRST.STYP
     FROM STUDENTS FIRST, STUDENTS SECOND
     WHERE FIRST.STYP = SECOND.STYP;
```

Результат такого запиту буде наступний:

```
Грищенко  Грищенко  грн.0,00
Грищенко  Котенко   грн.0,00
Котенко   Грищенко  грн.0,00
Котенко   Котенко   грн.0,00
Старченко Старченко грн.170,00
Нагірний  Нагірний  грн.250,00
```

```
Поляченко Поляченко грн.250,50
Поляченко Поляченко грн.250,50
Поляченко Поляченко грн.250,50
Поляченко Поляченко грн.250,50
```

У даному прикладі SQL поводиться так, як якби вона сполучала дві різні таблиці, які називаються FIRST і SECOND, тобто псевдоніми дозволяють одній і тій же таблиці бути обробленою незалежно. Зверніть увагу на те, що псевдоніми можуть використовуватися в пропозиції SELECT до їх оголошення в пропозиції FROM, проте SQL спочатку допускатиме будь-які псевдоніми і може відхилити команду, якщо вони не будуть означені далі в запиті. Крім того, необхідно пам'ятати, що псевдонім існує тільки тоді, коли команда виконується, а після завершення запиту псевдоніми, що використовуються в ньому, більше не мають жодного значення.

Вивід останнього прикладу має два значення для кожної комбінації прізвищ, причому другий раз в зворотному порядку – це пов'язано з тим, що кожне значення показане перший раз в кожному псевдонімі і другий раз в предикаті, тобто поточне значення в першому псевдонімі спочатку вибирається в комбінації із значенням в другому псевдонімі, а потім навпаки. Наприклад, в нашому випадку Грищенко вибрався разом з Котенко, а потім Котенко вибрався разом з Грищенко і т.д. Крім того, кожен рядок був порівняний сам із собою, наприклад Поляченко з Поляченко.

Кращий спосіб уникнути цього полягає в накладенні порядку на два значення так, щоб один міг бути менше, ніж інший або передував йому в алфавітному порядку. Це робить предикат асиметричним щодо зв'язку, тому ті ж самі значення у зворотному порядку не будуть вибрані знову. Отже, приклад можна модифікувати таким чином:

```
SELECT FIRST.SPRIZ, SECOND.SPRIZ, FIRST.STYP
     FROM STUDENTS FIRST, STUDENTS SECOND
     WHERE FIRST.STYP = SECOND.STYP
     AND FIRST.SPRIZ < SECOND.SPRIZ;
```

Результат цього запиту буде таким:

```
Грищенко  Котенко   грн.0,00
```

Зокрема, Грищенко передує Котенко в алфавітному порядку, тому комбінація задовольняє обом умовам предиката і з'являється у виводі. Якщо та ж сама комбінація з'являється в зворотному порядку, тобто Котенко в псевдонімі першої таблиці порівнюється з Грищенко в другій таблиці, то друга умова не виконується. З аналогічної причини у вивід не потрапляє порівняння з самим собою. Якщо ж виникла необхідність порівняння рядків з ними ж, то в запитах варто використовувати < = замість <.

Таким чином, можна використовувати цю особливість SQL для перевірки певних видів помилок. Наприклад, якщо вважати, що навчальний предмет може вести тільки один викладач, то кожен раз в таблиці PREDMET необхідна перевірка на цю умову. При цьому кожного разу, коли код предмету з'являється в таблиці PREDMET, він повинен співпадати з відповідним номером викладача. Наступна команда визначатиме будь-які неузгодженості в цій області:

```
SELECT FIRST.PNUM, FIRST.TNUM,
          SECOND.PNUM, SECOND.TNUM
     FROM PREDMET FIRST, PREDMET SECOND
     WHERE FIRST.PNUM = SECOND.PNUM
     AND FIRST.TNUM <> SECOND.TNUM;
```

Виводу для даного прикладу не буде, оскільки даних, що задовольняють предикату в даній таблиці немає. Логіка цього запиту досить проста: з таблиці вибиратиметься черговий рядок і запам'ятовуватися під першим псевдонімом. Після цього SQL почне перевіряти її в комбінації з кожним рядком таблиці під другим псевдонімом. Якщо комбінація рядків задовольняє предикату, то вона вибирається для виводу, тобто якщо буде знайдений навчальний предмет, у якого виявиться неспівпадіння номера викладача в таблиці під першим і другим псевдонімом.

Об'єднання таблиці з собою – це ситуація, що найбільш часто зустрічається, коли використовуються псевдоніми, проте їх можна використовувати у будь-який час для створення альтернативних імен для таблиць в запиті, наприклад, у випадку, якщо таблиці мають дуже довгі і складні імена.

Більше того, допускається використовувати будь-яке число псевдонімів для однієї таблиці в запиті, хоча використання більше двох в одній пропозиції SELECT часто буде надлишком. Наприклад, для призначення стипендії на наступний семестр необхідно проглянути всі варіанти комбінацій студентів з різними розмірами стипендії: 250,00; 250,50; 170,00 і 0,00 грн. Тоді такий запит виглядатиме таким чином:

```
SELECT FIRST.SPRIZ, SECOND.SPRIZ, THIRD.SPRIZ, FOURTH.SPRIZ
     FROM STUDENTS FIRST, STUDENTS SECOND,
          STUDENTS THIRD, STUDENTS FOURTH
     WHERE FIRST.STYP = '250,50'
     AND SECOND.STYP = '170,00'
     AND THIRD.STYP = '0,00'
     AND FOURTH.STYP = '250,00';
```

Вивід для цього запиту приведений нижче:

```
SPRIZ     SPRIZ     SPRIZ     SPRIZ
Поляченко Старченко Грищенко  Нагірний
Поляченко Старченко Котенко   Нагірний
Поляченко Старченко Грищенко  Нагірний
Поляченко Старченко Котенко   Нагірний
```

Як видно з результату, цей запит знаходить всі комбінації студентів з трьома різними розмірами стипендії, тому перший стовпець виводу складається із студентів із стипендією 250,50, другий з 170,00, третій з 0,00 і четвертий – 250,00, які повторюються у всіх можливих комбінаціях. Цікаво, що такий запит не може бути виконаний з GROUP BY або ORDER BY, оскільки вони порівнюють значення тільки в одному стовпці виводу.

У пропозиції SELECT допускається не використовувати кожен псевдонім або таблицю, які згадувалися в пропозиції FROM запиту, тобто псевдонім або таблиця використовуються тільки для створення предиката. Розглянемо чисто ілюстративний приклад наступного запиту:

```
SELECT FIRST.SPRIZ, FIRST.SIMA, FIRST.SBAT
     FROM STUDENTS FIRST, STUDENTS SECOND
     WHERE FIRST.STYP ='250,50'
     AND SECOND.STYP = '0,00';
```

Вивід даного запиту наступний:

```
SPRIZ     SIMA      SBAT
```

```
Поляченко Анатолій  Олексійович
Поляченко Анатолій  Олексійович
Поляченко Сергій    Олексійович
Поляченко Сергій    Олексійович
```

Фактично тут дані про студентів, що мають стипендію 250,50 грн., повторюються стільки разів, скільки існує їх поєднань із студентами, що не одержують стипендію (тобто для яких STYP = 0.00).

У SQL передбачається створення об'єднання, яке включає і різні таблиці, і псевдоніми одиночної таблиці. Наступний запит об'єднує таблицю з даними про успішність для того, щоб знайти навчальні предмети, які вже складені більш ніж одним студентом, і таблицю навчальних предметів:

```
SELECT PREDMET.PNAME, FIRST.SNUM, SECOND.SNUM
     FROM USP FIRST, USP SECOND, PREDMET
     WHERE FIRST.PNUM = SECOND.PNUM
     AND PREDMET.PNUM = FIRST.PNUM
     AND FIRST.SNUM < SECOND.SNUM;
```

Результати цього запиту наступні:

```
PNAME           SNUM       SNUM
Математика      3412       3413
```

У виводі маємо пари номерів студентів, що склали той або інший предмет, що визначений по назві з таблиці навчальних предметів.

Як вже було сказано, операція об'єднання в SQL поєднує інформацію із двох таблиць, формуючи пари зв'язаних рядків з них. Об'єднану таблицю утворюють пари тих рядків з різних таблиць, у яких в зв'язаних стовпцях містяться однакові значення. Якщо рядок однієї з таблиць не має пари, то об'єднання може привести до неочікуваних результатів.

Припустимо, що в таблиці TEACHERS з'явився запис {NULL, 'Федченко', 'Світлана', 'Геннадіївна', '01/09/1999'}, в якому табельний номер ще не відомий. Неочікувані результати можуть вийти при використанні наступних запитів:

```
SELECT TPRIZ, TIMA, TBAT
     FROM TEACHERS;
```

Вивід цього запиту наступний:

```
TPRIZ           TIMA       TBAT
Вікулін         Валентина  Іванівна
Костиркін       Олег       Володимирович
Казанко         Віталій    Володимирович
Позняк          Любов      Олексіївна
Загарийчук      Ігор       Дмитрович
Федченко        Світлана   Геннадіївна
```

і запиту

```
SELECT TEACHERS.TPRIZ, PREDMET.PNAME
     FROM TEACHERS, PREDMET
     WHERE TEACHERS.TNUM = PREDMET.TNUM;
```

Вивід цього запиту такий:

```
TPRIZ          PNAME
Вікулін        Фізика
Костиркін      Хімія
Казанко        Математика
Загарийчук     Філософія
Позняк         Економіка
```

Здавалося б, що ці два запити повинні давати однакову кількість рядків, але результати першого запиту налічують шість рядків, а другого – тільки п'ять. Це пов'язано з тим, що викладач Федченко зараз ще не отримала номери і відповідний запис має значення NULL в полі TNUM, отже, це значення не співпадає ні з одним ідентифікатором навчального предмету в таблиці PREDMET. Значить, виводу для цього рядка в другому запиті не буде – вона просто зникає з об'єднання. Таким чином, стандартне SQL-об'єднання може привести до втрати інформації, якщо об'єднувані таблиці містять незв'язані рядки.

Другий запит можна модифікувати таким чином:

```
SELECT TEACHERS.TPRIZ, PREDMET.PNAME
     FROM TEACHERS,  PREDMET
     WHERE TEACHERS.TNUM *= PREDMET.TNUM;
```

і його вивід буде повніший:

```
TPRIZ          PNAME
Вікулін        Фізика
Костиркін      Хімія
Казанко        Математика
Загарийчук     Філософія
Позняк         Економіка
Федченко       NULL
```

Такі результати запиту виходять за допомогою іншої операції об'єднання, яка називається зовнішнім об'єднанням таблиць, яке в пропозиції WHERE позначається символом *=. Зовнішнє об'єднання є розширенням стандартного об'єднання, описаного вище і іноді називається внутрішнім об'єднанням таблиць.

Майте на увазі, що в стандарті SQL1 дане означення тільки внутрішнього об'єднання, а поняття зовнішнього об'єднання в ньому відсутнє.

Розглянемо повне зовнішнє об'єднання таблиць на прикладі наступного запиту:

```
SELECT STUDENTS.SPRIZ, STUDENTS.SNUM, USP.PNUM, USP.SNUM
     FROM STUDENTS, USP
     WHERE STUDENTS.SNUM = USP.SNUM;
```

Вивід цього запиту приведений нижче:

```
SPRIZ      SNUM      PNUM      SNUM
Поляченко  3412      2001      3412
Старченко  3413      2003      3413
Грищенко   3414      2005      3414
Поляченко  3412      2003      3412
```

```
Нагірний  3416      2004      3416
```

Цей запит формує інформацію про студентів і коди навчальних предметів, які ними складені. Це внутрішнє об'єднання дає п'ять рядків виводу, показуючи відповідні пари студент-код предмету. Для студента Котенко жодного виводу немає, оскільки він жодних дисциплін не складав.

Тепер припустимо, що необхідно вивести ту ж інформацію, але з врахуванням студентів, яким не відповідає жоден навчальний предмет. Такий запит, побудований за правилами повного зовнішнього об'єднання, виглядатиме так:

```
SELECT STUDENTS.SPRIZ, STUDENTS.SNUM, USP.PNUM, USP.SNUM
     FROM STUDENTS, USP
     WHERE STUDENTS.SNUM *=* USP.SNUM;
```

Результат цього запиту такий:

```
SPRIZ     SNUM      PNUM      SNUM
Поляченко 3412      2001      3412
Старченко 3413      2003      3413
Грищенко  3414      2005      3414
Котенко   3415      NULL      NULL
Поляченко 3412      2003      3412
Нагірний  3416      2004      3416
```

У цей вивід потрапили рядки із значеннями NULL, тобто запис для студента Котенко був як би прив'язаний до неіснуючого рядка таблиці USP, що містить тільки значення NULL, і доданий в результат запиту. Як видно з цього прикладу, зовнішнє об'єднання є таким, що зберігає інформацію і кожен рядок таблиці-операнда представлений у виводі результатів запиту. Отже, повне зовнішнє об'єднання таблиць виходить з внутрішнього об'єднання двох таблиць звичайним способом, при цьому кожен запис першої таблиці, який не має зв'язку ні з одним рядком в другій таблиці, доповнюється в результатах запиту значеннями NULL. В той же час, кожен рядок другої таблиці, який не має зв'язку ні з одним рядком першої таблиці, доповнюється в результатах запиту значеннями NULL.

Існує ще так звані ліве і праве зовнішні об'єднання. Якщо повне зовнішнє об'єднання симетричне по відношенню до обох таблиць-операндів, то ці два типи зовнішніх об'єднань не симетричні щодо двох таблиць.

Ліве зовнішнє об'єднання двох таблиць записується в команді WHERE у вигляді *= і отримується в результаті виконання внутрішнього об'єднання таблиць. При цьому відбуваються дії, аналогічні повному об'єднанню, проте без заміни NULL значеннями інформації, взятої з другої таблиці. Приклад лівого зовнішнього об'єднання фактично нами вже був розглянутий вище в прикладі для викладачів і предметів.

Праве зовнішнє об'єднання двох таблиць записується в команді WHERE у вигляді =* і отримується із звичайного внутрішнього об'єднання таблиць в послідовності реалізації повного об'єднання, при цьому заміна на NULL значення в першій таблиці не проводиться.

На практиці ліве і праве зовнішні об'єднання корисніші, ніж повне зовнішнє об'єднання, особливо при зв'язку таблиць через ключі. Система запису зовнішнього об'єднання дозволяє використовувати не тільки знак рівності, але і інші знаки відношень. Наприклад, допустимий наступний запит, що представляє, по суті, ліве зовнішнє об'єднання:

```
SELECT STUDENTS.SPRIZ, STUDENTS.SNUM, USP.PNUM, USP.SNUM
     FROM STUDENTS, USP
```

```
WHERE STUDENTS.SNUM *> USP.SNUM;
```

Результат цього запиту, приведений не повністю через його значні розміри, буде наступний:

```
SPRIZ      SNUM      PNUM      SNUM
Старченко 3413       2001      3412
Грищенко  3414       2001      3412
Котенко   3415       2001      3412
Нагірний  3416       2001      3412
Поляченко 3417       2001      3412
Грищенко  3414       2003      3413
…
```

Даний запит виводить інформацію про студентів і про складені навчальні предмети, при цьому дані про предмети вибираються для студентів, що мають номер менше, ніж поточний. У випадку, якщо таких не виявляється, SQL формує рядок з першої таблиці, заповнюючи NULL значеннями дані, які повинні були бути отримані з другої.

Хоча система запису зовнішнього об'єднання достатньо зручна, вона має свої недоліки. Справа в тому, що проблеми із записом зовнішніх об'єднань виникають при розширенні об'єднання до трьох або більше таблиць. Наприклад, при записі

```
TABLE1 *=* TABLE2 *=* TABLE3
```

фактично виконується зовнішнє об'єднання

```
(TABLE1 *=* TABLE2) *=* TABLE3
```

що, взагалі кажучи, приведе до різних результатів.

Для творців стандарту SQL2 зовнішні об'єднання були серйозною проблемою, оскільки зовнішні об'єднання є інколи єдиним способом представлення результатів вкрай необхідних запитів. Тому в стандарті SQL2 був означений абсолютно новий метод підтримки зовнішніх об'єднань: у специфікації стандарту SQL2 підтримка зовнішніх об'єднань здійснюється в пропозиції FROM із спеціальним синтаксисом, що дозволяє користувачу точно визначити, як початкові таблиці повинні бути об'єднані в запиті.

Наприклад, вже розглянутий нами запит внутрішнього об'єднання в стандарті SQL1 виглядає так:

```
SELECT STUDENTS.SPRIZ, STUDENTS.SNUM, USP.PNUM, USP.SNUM
     FROM STUDENTS, USP
     WHERE STUDENTS.SNUM = USP.SNUM;
```

а в стандарті SQL2 – так (хоча використання першого варіанту допускається):

```
SELECT STUDENTS.SPRIZ, STUDENTS.SNUM, USP.PNUM, USP.SNUM
     FROM STUDENTS INNER JOIN USP ON STUDENTS.SNUM = USP.SNUM;
```

Зверніть увагу на те, що в останньому випадку дві об'єднувані таблиці з'єднуються явно за допомогою операції JOIN, а умова пошуку, що описує об'єднання, знаходиться тепер в пропозиції ON всередині пропозиції FROM. У умові пошуку, що слідує за ключовим словом ON, можуть бути задані будь-які критерії порівняння рядків двох об'єднуваних таблиць, зокрема

з використанням булевих операторів.

Нарешті, приведемо приклад повного зовнішнього об'єднання (аналогічний запит був вже розглянутий):

```
SELECT STUDENTS.SPRIZ, STUDENTS.SNUM, USP.PNUM, USP.SNUM
     FROM STUDENTS FULL OUTER JOIN USP
          ON STUDENTS.SNUM = USP.SNUM;
```

Його результати приведені нижче:

```
SPRIZ      SNUM       PNUM       SNUM
Поляченко  3412       2003       3412
Поляченко  3412       2001       3412
Старченко  3413       2003       3413
Грищенко   3414       2005       3414
Котенко    3415       NULL       NULL
Нагірний   3416       2004       3416
Поляченко  3417       NULL       NULL
```

Таким чином, в цих двотабличних об'єднаннях весь вміст пропозиції WHERE просто перейшов в пропозицію ON, отже, нічого принципово нового ON не додає в мову SQL. Проте така структура дозволяє точніше означити умову об'єднання.

2.2.8. Використання вкладених запитів

У найзагальнішому випадку, запити можуть управляти іншими запитами – це робиться шляхом розміщення запиту всередину іншого предиката, який використовує вивід внутрішнього запиту для встановлення вірного або невірного значення предиката. Наприклад, потрібна інформація про успішність студента з прізвищем Поляченко, проте через які-небудь причини невідомий номер цього студента. Тоді необхідно витягнути цей номер з таблиці із даними про студентів, і після цього застосовувати результат до таблиці успішності. Це можна реалізувати шляхом наступної конструкції:

```
SELECT * FROM USP
     WHERE SNUM = (SELECT SNUM FROM STUDENTS
          WHERE SPRIZ = 'Грищенко');
```

Результат цього запиту буде наступний:

```
UNUM       OCINKA     UDATE           SNUM       PNUM
1003       3          2005-11-06      3414       2005
```

Щоб виконати основний запит, SQL спочатку повинна оцінити внутрішній запит (його називають підзапитом) всередині пропозиції WHERE. Відбувається це традиційним чином, тобто виконується вкладений запит, що витягує необхідні для визначення значення предиката дані, а тільки потім – основной. Зрозуміло, підзапит повинен вибрати тільки одне поле, а тип даних цього поля повинен співпадати з тим значенням, з яким він порівнюватиметься в предикаті.

З іншого боку, можлива ситуація, коли підзапит видає як результат декілька різних значень, що може зробити нездійсненним оцінку предиката основного запиту і команда видасть помилку. При використанні підзапитів в предикатах, заснованих на реляційних операторах, обов'язково потрібно переконатися, що використаний підзапит, який відображатиме тільки один

рядок виводу. Крім того, при використанні підзапиту, який взагалі не виводить ніяких значень, основний запит не виведе ніяких значень: його предикат матиме невідоме значення.

В деяких випадках варто використовувати DISTINCT для того, щоб в підзапиті набути одиночного значення. Припустимо, що викладачі можуть вести заняття з різних дисциплін. Тоді для отримання відповіді на питання про те, які дисципліни веде викладач Вікулін, можна скористатися запитом:

```
SELECT * FROM PREDMET
     WHERE TNUM =
     (SELECT DISTINCT TNUM FROM TEACHERS
          WHERE TPRIZ = 'Вікулін');
```

В результаті отримаємо:

```
PNUM      PNAME     TNUM      HOURS     COURS
2001      Фізика    4001      34        1
```

Підзапит встановив, що значення поля TNUM співпало з прізвищем Вікулін при значенні 4001, а потім основний запит виділив всі записи з цим значенням TNUM з таблиці предметів. Оскільки, взагалі кажучи, могло вийти, що викладач веде декілька предметів, то фраза DISTINCT в даному випадку обов'язкова – якщо підзапит повернув би більше одного значення, то це викликало б помилку.

Слід мати на увазі, що предикати з підзапитами є необоротними, тобто предикати, що включають підзапити, використовують конструкцію в наступному порядку:

```
<ВИРАЗ> <ОПЕРАТОР> <ПІДЗАПИТ>,
```

і у жодному випадку не

```
<ПІДЗАПИТ> <ОПЕРАТОР> <ВИРАЗ>,
```

або

```
<ПІДЗАПИТ> <ОПЕРАТОР> <ПІДЗАПИТ>,
```

Інакше кажучи, попередній приклад, записаний таким чином:

```
SELECT * FROM PREDMET
     WHERE (SELECT DISTINCT TNUM FROM TEACHERS
          WHERE TPRIZ = 'Вікулін') = TNUM;
```

є невірним.

Зрозуміло, в підзапитах допускається використання агрегатних функцій – це зручно і з тієї причини, що вони автоматично проводять одиночне значення для будь-якого числа рядків, яке може бути використане в основному предикаті. Наприклад, якщо виникає необхідність у виведенні всіх оцінок по навчальних дисциплінах, значення яких вище за середнє, то для цього скористаємося запитом:

```
SELECT * FROM USP
     WHERE OCINKA > (SELECT AVG (OCINKA) FROM USP);
```

Вивід такого запиту наступний:

```
UNUM       OCINKA     UDATE           SNUM       PNUM
1001       5          2005-10-06      3412       2001
1005       5          2005-12-06      3416       2004
```

Середня оцінка за наявними даними складає 4,2, отже, основний запит вибирає тільки ті записи, в яких значення поля OCINKA більше, ніж 4,2.

Не варто забувати, що згруповані агрегатні функції, означені в термінах пропозиції GROUP BY, можуть видавати численні значення, а значить, не допускаються в підзапитах такого характеру. Навіть якщо GROUP BY або HAVING використовуються так, що тільки одна група значень виводиться за допомогою підзапиту, все одно команда буде відхилена. До речі кажучи, можна використовувати підзапити, які проводять будь-яке число рядків, якщо використовується спеціальний оператор IN (оператори BETWEEN, LIKE, і IS NULL не можуть використовуватися з підзапитами). IN визначає набір значень предиката, одне з яких повинне співпадати з іншим по порядку При використанні IN з підзапитом, SQL просто формує цей набір з виводу підзапиту, а значить, допускається використання IN для того, щоб виконати такий же підзапит. Наприклад, запит

```
SELECT * FROM PREDMET
     WHERE PREDMET.TNUM IN
          (SELECT TEACHERS.TNUM
               FROM TEACHERS
                    WHERE TEACHERS.TPRIZ
                         BETWEEN 'I' AND 'C');
```

Вивід цього запиту такий:

```
PNUM       PNAME           TNUM       HOURS      COURS
2002       Хімія           4002       68         1
2003       Математика      4003       68         1
2005       Економіка       4004       17         3
```

Такий запит набагато зрозуміліший, ніж виконаний за допомогою об'єднання, хоча дає такий же результат:

```
SELECT     PREDMET.PNUM, PREDMET.PNAME,
           TEACHERS.TNUM, PREDMET.HOURS, PREDMET.COURS
     FROM PREDMET, TEACHERS
     WHERE TEACHERS.TNUM = PREDMET.TNUM
     AND TEACHERS.TPRIZ BETWEEN 'I' AND 'C';
```

Взагалі кажучи швидкість виконання того або іншого запиту, наприклад, як в розглянутому випадку, залежить від реалізації тієї СУБД, яка для цього використовується. Вище вже піднімалася проблематика, пов'язана з роботою оптимізатора запитів, який намагається знайти найбільш ефективний спосіб виконання. Традиційно добрий оптимізатор найчастіше перетворить варіант об'єднання в підзапит, який звичайно виконується швидше.

І ще один цікавий момент: у будь-якій ситуації, де застосовується реляційний оператор рівності (=), можна використовувати IN. На відміну від першого, IN не може примусити запит потерпіти невдачу, якщо підзапитом вибрано більше ніж одне значення. Це може бути або перевагою або недоліком. Наприклад, вже розглянутий нами вище запит можна переписати

таким чином:

```
SELECT * FROM PREDMET
     WHERE TNUM IN
          (SELECT TNUM FROM TEACHERS
               WHERE TPRIZ = 'Вікулін');
```

Таким чином, підзапити завжди визначають одиночні стовпці – це обов'язково, оскільки вибраний вивід порівнюється з одиночним значенням. Підтвердженням цьому є те, що команда SELECT * не може використовуватися в підзапиті. У підзапиті допускається використовувати вираз, що базується на полі, а не просто саме поле, в пропозиції SELECT. Це може бути виконано за допомогою реляційних операторів або при використанні IN. Прикладом може служити наступний запит:

```
SELECT * FROM PREDMET
     WHERE PNUM =
          (SELECT PNUM-1 FROM PREDMET
               WHERE PNAME = 'Філософія');
```

Вивід для цього запиту такий:

```
PNUM       PNAME           TNUM       HOURS      COURS
2003       Математика      4003       68         1
```

Цей запит знаходить інформацію про навчальний предмет, код якого на 1 менше коду філософії. Зрозуміло, поле PNUM не повинно містити значень, що повторюються, інакше запит викличе помилку.

Можна також використовувати підзапити всередині пропозиції HAVING. Ці підзапити можуть використовувати свої власні агрегатні функції, якщо вони не призводять до багаточисельних значень, або використовувати GROUP BY або HAVING. Наприклад, розглянемо запит:

```
SELECT OCINKA, COUNT (DISTINCT SNUM) FROM USP
     GROUP BY OCINKA
     HAVING OCINKA >=
          (SELECT AVG (OCINKA) FROM USP
               WHERE PNUM = 2003);
```

Вивід для цього запиту наступний:

```
OCINKA
4          2
5          2
```

Даний запит підраховує кількість студентів з оцінками вище за середню, ніж по дисципліні з PNUM = 2003.

Тепер декілька слів про об'єднання запитів і використання при цьому підзапитів. Операція, яка часто корисна – це об'єднання з двох запитів, в якому другий запит вибирає рядки, виключені першим. Найчастіше це роблять для того, щоб не виключати рядки, які не задовольнили предикату при об'єднанні таблиць. Цей процес прийнято називати зовнішнім об'єднанням.

Припустимо, що деякі із студентів ще не отримали оцінку, проте вже внесені в таблицю USP. Наприклад, в цю таблицю доданий запис {1006, NULL, NULL, 3416, NULL}. Якщо виникає необхідність в прогляданні успішності студентів по дисципліні, не враховуючи тих, хто ще не отримав оцінку. Цього можна досягти, формуючи об'єднання з двох запитів, один з яких виконує об'єднання, а інший вибирає студентів з NULL значеннями поля OCINKA. Цей останній запит повинен вставляти повідомлення в поля, що відповідають полю PNAME, і значення 0 в полі OCINKA в першому запиті. Як було розглянуто раніше, можна вставляти текстові рядки у вивід, щоб ідентифікувати запит, який вивів даний рядок. Використання цієї методики в зовнішньому об'єднанні дає можливість застосовувати предикати для класифікації, а не для виключення. Наступний запит виконує ці дії:

```
SELECT USP.SNUM, STUDENTS.SPRIZ, PREDMET.PNAME, USP.OCINKA
     FROM USP, STUDENTS, PREDMET
          WHERE USP.SNUM = STUDENTS.SNUM
          AND USP.PNUM = PREDMET.PNUM
UNION
SELECT USP.SNUM, STUDENTS.SPRIZ,
'НЕМАЄ', 0
     FROM USP, STUDENTS
          WHERE USP.SNUM = STUDENTS.SNUM
          AND NOT USP.OCINKA = ANY
               (SELECT OCINKA
                    FROM USP)
     ORDER BY 2 ASC;
```

Вивід цього запиту наступний:

```
SNUM      SPRIZ          PNAME          OCINKA
3414      Грищенко       Економіка      3
3416      Нагірний       Філософія      5
3412      Поляченко      Фізика         5
3412      Поляченко      Математика     4
3413      Старченко      Математика     4
```

Декілька слів про використаного оператора ANY, детальніше про яке говоритимемо нижче. Оператор ANY бере всі значення, виведені підзапитом (для нашого випадку – це всі значення, OCINKA в таблиці USP), і оцінює їх як вірні, якщо будь-яке них дорівнює значенню оцінки поточного запису зовнішнього запиту.

На закінчення розмови про вкладені запити, поговоримо про так звані співвіднесені підзапити. Коли використовуються підзапити, в SQL є можливість звернутися до внутрішнього запиту таблиці в пропозиції зовнішнього запиту FROM, за допомогою співвіднесеного підзапиту. При цьому підзапит виконується неодноразово, по одному разу для кожного запису таблиці основного запиту. З використанням співвіднесеного підзапиту можна знайти дані на всіх студентів, які отримували оцінки 10/06/2005:

```
SELECT * FROM STUDENTS FIRST
     WHERE '10/06/2005' IN
          (SELECT UDATE FROM USP SECOND
               WHERE FIRST.SNUM = SECOND.SNUM);
```

Вивід цього запиту такий:

```
SNUM      SPRIZ          SIMA      SBAT           STYP
3412      Поляченко      Анатолій  Олексійович    грн.250,50
3413      Старченко      Любов     Михайлівна     грн.170,00
```

В даному прикладі FIRST і SECOND – псевдоніми таблиць, при цьому виходить, що значення в полі SNUM зовнішнього запиту міняється, а значить, внутрішній запит повинен виконуватися окремо для кожного рядка зовнішнього запиту.

Рядок зовнішнього запиту, для якого внутрішній запит кожного разу виконуватиметься, називатимемо поточним рядком. З врахуванням цього, процедура оцінки, що виконується співвіднесеним підзапитом:

- вибір рядка з таблиці в зовнішньому запиті – це поточний рядок;
- збереження значення поточного рядка в псевдонімі, ім'я якого означене в пропозиції FROM зовнішнього запиту;
- виконання підзапиту, при цьому скрізь, де знайдений псевдонім із зовнішнього запиту, використовується значення з поточного рядка (це прийнято називати зовнішнім посиланням);
- оцінка предиката зовнішнього запиту на основі результатів підзапиту;
- описана вище послідовність повторюється для наступного рядка з таблиці зовнішнього запиту, і так до тих пір, поки всі рядки не будуть перевірені.

Взагалі кажучи, останній приклад міг бути реалізований, використовуючи об'єднання наступного вигляду:

```
SELECT    STUDENTS.SNUM, STUDENTS.SPRIZ, STUDENTS.SIMA,
          STUDENTS.SBAT, STUDENTS.STYP
     FROM STUDENTS, USP
          WHERE STUDENTS.SNUM = USP.SNUM
          AND USP.UDATE = '10/06/2005';
```

Припустимо, що є необхідність у виведенні прізвища і номера всіх студентів, які отримали більше однієї оцінки. Це реалізується наступним запитом:

```
SELECT SNUM, SPRIZ
     FROM STUDENTS FIRST
          WHERE 1<
               (SELECT COUNT(*)
                    FROM USP
                         WHERE SNUM = FIRST.SNUM);
```

Вивід цього запиту приведений нижче:

```
SNUM      SPRIZ
3412      Поляченко
3416      Нагірний
```

Варто звернути увагу на те, що пропозиція FROM підзапиту в даному прикладі не використовує псевдонім. За відсутності імені таблиці або префікса псевдоніма, SQL може спершу прийняти, що будь-яке поле виводиться з таблиці із ім'ям, вказаним в пропозиції FROM поточного запиту. Якщо поле з цим ім'ям відсутнє в цій таблиці, SQL перевірятиме зовнішні запити. Саме з цієї причини префікс імені таблиці звичайно необхідний в співвіднесених підзапитах – для відміни цього припущення. Псевдоніми також часто необхідні для того, щоб

дати можливість посилатися до тієї ж самої таблиці у внутрішньому і зовнішньому запиті без якої-небудь неоднозначності.

Часто співвіднесений підзапит використовують на основі тієї ж самої таблиці, що і основний запит. Це дає можливість витягнути складні форми інформації. Наприклад, за допомогою наступного запиту можна знайти всі оцінки по дисципліні із значеннями, вище середньої по цій же дисципліні:

```
SELECT * FROM USP FIRST
     WHERE OCINKA >
          (SELECT AVG (OCINKA) FROM USP SECOND
               WHERE SECOND.PNUM = FIRST.PNUM);
```

В даному випадку виводу запиту не буде, оскільки в таблиці USP немає записів для студентів, що мають по якому-небудь навчальному предмету оцінку вищу за середню. Якщо ж умову зовнішнього предиката змінити на >=, то як результат будуть виведені всі дані таблиці успішності.

Пропозиція HAVING може також працювати і з співвіднесеними підзапитами. При використанні співвіднесеного підзапиту в пропозиції HAVING необхідно пам'ятати, що предикат оцінюється для кожної групи із зовнішнього запиту, а не для кожного рядка. Отже, підзапит виконуватиметься один раз для кожної групи, виведеної із зовнішнього запиту, а не для кожного рядка. Наприклад, для встановлення середнього значення оцінок за кожен день, причому такого, що це середнє повинно бути більше або рівне хоч би на половину балу, ніж мінімальне значення оцінки цього дня, можна скористатися запитом:

```
SELECT UDATE, AVG (OCINKA) FROM USP FIRST
        GROUP BY UDATE
        HAVING AVG (OCINKA) >=
                (SELECT MIN(OCINKA) +0.5 FROM USP SECOND
                        WHERE FIRST.UDATE = SECOND.UDATE);
```

Результат запиту буде таким:

```
UDATE
2005-12-06     4.5000000000000000
2005-10-06     4.5000000000000000
```

Підзапит обчислює значення MIN для всіх записів з тією ж самою датою, що і у поточної агрегатної групи основного запиту.

Співвіднесені підзапити за своєю суттю близькі до об'єднань – обидві конструкції включають перевірку кожного запису однієї таблиці з кожним записом іншої або з псевдонімом з тієї ж таблиці, при цьому більшість операцій у них схожа. З цієї причини детально зупинятися на цьому матеріалі не будемо.

Таким чином, застосування вкладених запитів в цілях використання їх результату для керування іншим запитом розширює можливості SQL, дозволяючи виконати більшу кількість функцій.

2.2.9. Використання операторів EXISTS, ANY, ALL і SOME

Після ознайомлення з роботою підзапитів, необхідно говорити про деякі спеціальні оператори, які завжди беруть підзапити як аргументи – це EXISTS, ANY, ALL, і SOME.

Оператор EXISTS використовується для вказівки предикату на те, чи здійснювати або не здійснювати вивід в підзапиті, при цьому EXISTS дає як результат значення ІСТИНА або

БРЕХНЯ. Це в свою чергу означає, що він може працювати в предикаті або в комбінації з іншими булевими виразами – AND, OR і NOT. Іншими словами, EXISTS бере підзапит як аргумент і оцінює його як істинний, якщо він здійснює будь-який вивід, або як помилковий, якщо він не робить цього. Наприклад, можна вирішити, чи витягувати дані з таблиці успішності, якщо в ній присутні відмінні оцінки. Це реалізується таким чином:

```
SELECT * FROM USP
     WHERE USP.OCINKA =5
     AND EXISTS
          (SELECT * FROM USP
               WHERE USP.OCINKA = 5);
```

Результати запиту будуть наступні:

```
UNUM      OCINKA    UDATE          SNUM      PNUM
1001      5         2005-10-06     3412      2001
1005      5         2005-12-06     3416      2004
```

При цьому внутрішній запит вибирає всі дані для всіх студентів, що отримали оцінку 5. Оператор EXISTS в зовнішньому предикаті відзначає, що деякий вивід в підзапиті мав місце, значить, це робить предикат вірним. Підзапит був виконаний тільки один раз для всього зовнішнього запиту, і, отже, має одне значення у всіх випадках – з цієї причини EXISTS робить предикат вірним або невірним для всіх рядків відразу. У останньому прикладі, якщо строго говорити, EXISTS повинен бути встановлений так, щоб вибрати тільки один стовпець. Проте він вибирає всі стовпці у вкладеній пропозиції SELECT *. Це не викликає помилки, оскільки при виборі EXISTS як одного стовпця, так і всіх стовпців, він просто помічає виконання виводу з підзапиту, а отримані значення не використовує.

У співвіднесеному підзапиті, пропозиція EXISTS оцінюється окремо для кожного рядка таблиці, ім'я якої вказане в зовнішньому запиті, точно так, як і інші оператори предиката, коли використовується співвіднесений підзапит. Це дає можливість використовувати EXISTS як предикат, який генерує різні значення для кожного запису таблиці, вказаної в основному запиті. Отже, інформація з внутрішнього запиту зберігатиметься. Наприклад, за допомогою наступного запиту виведемо інформацію про студентів, які мають декілька оцінок:

```
SELECT DISTINCT SNUM FROM USP FIRST
     WHERE EXISTS (SELECT * FROM USP SECOND
          WHERE SECOND.SNUM = FIRST.SNUM
          AND SECOND.PNUM <> FIRST.PNUM);
```

Вивід запиту наступний:

```
SNUM
3412
```

Тут для кожного поточного рядка зовнішнього запиту внутрішній запит знаходить записи, які співпадають із значенням поля номера студента SNUM, але не співпадають із значенням коду предмету PNUM. Якщо будь-які такі рядки будуть знайдені підзапитом, це означає, що є, принаймні, дві оцінки, отримані поточним студентом. Якби в зовнішньому запиті DISTINCT не був вказаний, то кожний із студентів, що має декілька оцінок, був би вибраний стільки разів, скільки у нього оцінок.

Для ілюстрації можливості використання комбінації з EXISTS і об'єднань, вдосконалимо

останній приклад так, щоб виводилася детальніша інформація про студентів:

```
SELECT DISTINCT FIRST.SNUM, FIRST.SPRIZ, FIRST.SIMA, FIRST.SBAT
     FROM STUDENTS FIRST, USP SECOND
          WHERE EXISTS (SELECT * FROM USP THIRD
               WHERE SECOND.SNUM = THIRD .SNUM
               AND SECOND.PNUM <> THIRD.PNUM)
          AND FIRST.SNUM = SECOND.SNUM;
```

В результаті отримаємо:

```
SNUM      SPRIZ          SIMA      SBAT
3412      Поляченко      Анатолій  Олексійович
```

В даному прикладі внутрішній запит, як і в попередньому варіанті, повідомляє, що студент отримав декілька оцінок. Зовнішній запит – це об'єднання таблиць успішності і студентів. Нова пропозиція основного предиката, що з'явилася AND FIRST.SNUM = SECOND.SNUM, оцінюється на тому ж самому рівні, що і пропозиція EXISTS, оскільки це елемент предиката самого об'єднання двох таблиць із зовнішнього запиту.

З врахуванням того, що EXISTS може працювати в комбінації з булевими операторами, найбільш очевидним способом його використання є поєднання з оператором NOT. Наприклад, для отримання інформації про студентів, що мають тільки одну оцінку, можна скористатися наступним запитом:

```
SELECT DISTINCT FIRST.SNUM, FIRST.SPRIZ, FIRST.SIMA, FIRST.SBAT
     FROM STUDENTS FIRST, USP SECOND
          WHERE NOT EXISTS
          (SELECT * FROM USP THIRD
               WHERE SECOND.SNUM = THIRD.SNUM
               AND SECOND.PNUM <> THIRD. PNUM)
          AND FIRST.SNUM = SECOND.SNUM;
```

Вивід цього запиту приведений нижче:

```
SNUM      SPRIZ          SIMA           SBAT
3413      Старченко      Любов          Михайлівна
3414      Грищенко       Володимир      Миколайович
3416      Нагірний       Євген          Васильович
```

Важливою властивістю EXISTS є те, що він не може взяти агрегатну функцію в підзапиті, оскільки якщо агрегатна функція знайшла які-небудь записи для операцій з ними, то EXISTS буде вірний у будь-якому випадку. Виходом з такої ситуації є використання вкладеного підзапиту в підзапиті, в предикаті якого присутній EXISTS. Це дозволяє ретельно структурувати запити, роблячи їх зрозумілішими. Повертаючись наприклад до знаходження інформації про студентів, що мають більше однієї оцінки, можна запропонувати такий варіант запиту:

```
SELECT * FROM STUDENTS FIRST
     WHERE EXISTS (SELECT * FROM USP SECOND
          WHERE FIRST.SNUM = SECOND.SNUM
          AND 1 < (SELECT COUNT (*) FROM USP
                    WHERE USP.SNUM = SECOND.SNUM));
```

В результаті отримаємо:

```
SNUM      SPRIZ          SIMA       SBAT            STYP
3412      Поляченко      Анатолій   Олексійович     грн.250,50
3416      Нагірний       Євген      Васильович      грн.250,00
```

Така конструкція працює таким чином: зовнішній запит вибирає з таблиці студентів STUDENTS поточний рядок і виконує підзапити. Для поточного рядка середнім запитом береться у відповідність кожен рядок з таблиці успішності USP. Всякий раз, коли виявляється інформація в середньому запиті, який співпадає по номеру студентського квитка з поточним рядком в зовнішньому запиті, SQL повинна розглядати внутрішній підзапит для того, щоб визначити, чи буде предикат середнього запиту вірний. Внутрішній запит рахує кількість оцінок поточного студента, і, якщо це число більше, ніж 1, робить предикат середнього запиту вірним. Це, у свою чергу робить EXISTS-предикат зовнішнього запиту вірним для поточного рядка таблиці.

Таким чином, не дивлячись на простоту, що здається, EXISTS може виявитися одним з найважчих з погляду використання операторів в SQL. Існують ще три спеціальні оператори, що орієнтуються на підзапити – це ANY, ALL і SOME, що нагадують EXISTS, які сприймають підзапити, як аргументи. Проте ці оператори відрізняються від EXISTS тим, що використовуються спільно з реляційними операторами, аналогічно IN в підзапитах.

Оператори SOME і ANY достатньо часто взаємозамінні, і в наших прикладах працюватимуть однаково. Відмінність в термінології полягає в тому, щоб дозволити користувачам використовувати той термін, який найбільш однозначний. Тому все сказане щодо ANY в рівній мірі відноситься і до SOME.

Розглянемо наступний спосіб знаходження студентів, які отримували оцінки по різних навчальних предметах:

```
SELECT * FROM STUDENTS
     WHERE SNUM = ANY (SELECT SNUM FROM USP);
```

Вивід цього запиту приведений нижче:

```
SNUM      SPRIZ          SIMA       SBAT            STYP
3412      Поляченко      Анатолій   Олексійович     грн.250,50
3413      Старченко      Любов      Михайлівна      грн.170,00
3414      Грищенко       Володимир  Миколайович     грн.0,00
3416      Нагірний       Євген      Васильович      грн.250,00
```

Оператор ANY бере всі значення, виведені підзапитом, тобто в прикладі – всі значення поля SNUM, і оцінює їх як вірні, якщо будь-яке з них дорівнює номеру студентського квитка поточного рядка зовнішнього запиту. Звідси випливає, що підзапит повинен вибирати значення такого ж типу, як і ті, які порівнюються в основному предикаті. У цьому його відмінність від EXISTS, який просто визначає, чи здійснює вивід підзапит чи ні, фактично не використовуючи ці результати.

Взагалі кажучи, у ряді випадків допускається використання операторів IN або EXISTS замість оператора ANY. Наприклад, використовуючи оператора IN, можна створити запит, аналогічний попередньому:

```
SELECT *FROM STUDENTS
     WHERE SNUM IN (SELECT SNUM FROM USP);
```

Вивід цього запиту такий, як в попередньому прикладі, тому наводити його не будемо.

Цікавий той факт, що оператор ANY може використовувати інші реляційні оператори, за винятком дорівнює, і, таким чином, реалізовувати порівняння, які недоступні з IN. Наприклад, можна виконати наступний запит:

```
SELECT PNAME FROM PREDMET
     WHERE HOURS > ANY (SELECT HOURS FROM PREDMET);
```

Результат цього запиту наступний:

```
PNAME
Фізика
Хімія
Математика
```

Тут виводяться назви навчальних предметів, для яких існує хоч би одна навчальна дисципліна з кількістю годин, меншою, ніж у поточної.

В принципі, цей же запит може бути реалізований з використанням EXISTS, таким чином:

```
SELECT PNAME FROM PREDMET FIRST
     WHERE EXISTS (SELECT * FROM PREDMET SECOND
          WHERE FIRST.HOURS > SECOND.HOURS);
```

У будь-якому випадку запит, який може бути реалізований з ANY і ALL, може бути також сформульований з EXISTS, але не навпаки. Правда в ряді випадків, через відмінності в обробці NULL значень, варіант з EXISTS не абсолютно ідентичний, ніж при використанні ANY, тому необхідно в таких випадках застосовувати команду IS NULL.

Основна перевага у використанні ANY і ALL в порівнянні з EXISTS полягає в тому, що останній вимагає співвіднесених підзапитів, інколи складних для розуміння. Крім того, підзапити з ANY або ALL можуть виконуватися один раз і мати вивід, що використовується для визначення предиката для кожного рядка основного запиту, а EXISTS вимагає, щоб весь підзапит повторно виконувався для кожного рядка основного запиту.

Оператор ALL працює таким чином, що предикат є вірним, якщо кожне значення, вибране підзапитом, задовольняє умові в предикаті зовнішнього запиту.

Наприклад, якщо необхідно вивести тільки тих студентів, чиї оцінки вище або рівні отриманим 10/06/2005, то можна скористатися наступним запитом:

```
SELECT * FROM USP
     WHERE OCINKA >= ALL
     (SELECT OCINKA FROM USP
          WHERE UDATE = '10/06/2005 ');
```

Вивід цього запиту такий:

```
UNUM      OCINKA    UDATE          SNUM      PNUM
1001      5         2005-10-06     3412      2001
1005      5         2005-12-06     3416      2004
```

Запит працює так: підзапит перевіряє значення оцінок за 10/06/2005 для всіх студентів.

Потім він знаходить студентів з оцінкою більшою або рівнішою в порівнянні з оцінками за 10/06/2005. Оскільки найвища оцінка цього дня – відмінно, то вибираються тільки записи з оцінкою 5.

Як і у випадку з ANY, допускається використання EXISTS для альтернативного формулювання такого ж запиту:

```
SELECT * FROM USP FIRST
     WHERE EXISTS (SELECT * FROM USP SECOND
          WHERE FIRST.OCINKA >= SECOND.OCINKA
          AND SECOND.UDATE = '10/06/2005');
```

Основне застосування ALL знаходить із знаком рівності, оскільки предикат може бути вірним, якщо порівнюване значення рівне для всіх, тобто всі записи, фактично, ідентичні. Наприклад, запит

```
SELECT * FROM USP
WHERE OCINKA = ALL
(SELECT OCINKA FROM USP
     WHERE UDATE = '10/06/2005');
```

є допустимим, але вивід буде тільки у випадку, якщо всі оцінки за 10/06/2005 виявляться ідентичними. Ймовірно, не дуже добре використовувати запити, які працюють тільки в певних ситуаціях.

Проте ALL набагато ефективніше використовується із знаком нерівності, тобто з оператором <>. При цьому обов'язково треба враховувати, що якщо підзапит повертає багато різних значень, то це означає нерівність будь-якому результату. Інакше кажучи, предикат вірний, якщо дане значення не знайдене серед результатів підзапиту. Розглянемо такий запит:

```
SELECT * FROM USP
     WHERE OCINKA <> ALL (SELECT OCINKA FROM USP
          WHERE UDATE = '10/06/2005');
```

Вивід цього запиту такий:

```
UNUM       OCINKA     UDATE           SNUM       PNUM
1003       3          2005-11-06      3414       2005
```

Даний запит зробить наступне: підзапит вибере всі оцінки за 10/06/2005 – це будуть 5 і 4. Після цього, основний запит виведе всі записи з оцінкою, не співпадаючою ні з однією з них. Аналогічний запит можна сформулювати з використання оператора NOT IN:

```
SELECT * FROM USP
     WHERE OCINKA NOT IN
     (SELECT OCINKA FROM USP
          WHERE UDATE = '10/06/2005');
```

або використовуючи оператора ANY:

```
SELECT * FROM USP
     WHERE NOT OCINKA = ANY
     (SELECT OCINKA FROM USP
```

```
WHERE UDATE = '10/06/2005');
```

Вивід цих запитів такий же, як і першого. Таким чином, в SQL дати запит на порівняння з використанням команди ANY для набору значень – те ж саме, що провести порівняння з будь-яким окремим значенням з набору. Якщо запросити значення за допомогою команди ALL, що не рівні набору значень, то це те ж саме, що визначити факт відсутності цього значення у всьому наборі.

Як вже було сказано, є деякі відмінності між EXISTS і операторами ANY і ALL з погляду роботи з невідомими і відсутніми даними. Крім того, важлива відмінність між ALL і ANY – це спосіб дії за ситуації, коли підзапит не повертає жодних значень. Взагалі кажучи, у всіх випадках, коли допустимий підзапит не робить виводу, ALL автоматично повертає значення ІСТИНА, а ANY – ХИБА. Наприклад, це означає, що запит:

```
SELECT * FROM USP
     WHERE OCINKA >= ANY (SELECT OCINKA FROM USP
          WHERE UDATE = '10/06/2006');
```

не здійснить жодного виводу, тоді як запит

```
SELECT * FROM USP
     WHERE OCINKA >= ALL (SELECT OCINKA FROM USP
          WHERE UDATE = '10/06/2006');
```

виведе всю таблицю USP. Дійсно, якщо немає жодних оцінок за 10/06/2006 то, природно, жодне з цих порівнянь не має значення.

NULL значення мають свої особливості при їх обробці цими операторами. Коли SQL порівнює два значення в предикаті, одне з яких NULL, очевидно, результат цього невідомий. Невідомий предикат, аналогічно невірному, є причиною того, що запис не вибирається, проте працювати він буде трохи інакше, залежно від того, використовується ALL або ANY. Розглянемо попередній приклад з ANY і таким запитом:

```
SELECT * FROM USP FIRST
     WHERE EXISTS (SELECT * FROM USP SECOND
          WHERE FIRST.OCINKA >= SECOND.OCINKA
          AND SECOND.UDATE = '10/06/2005');
```

Поки NULL значень в таблиці немає, обидва запити поводитимуться однаково. Але у разі появи NULL значення в полі OCINKA таблиці успішності, у варіанті з ANY значення предиката при обробці цього рядка стане невідомим і виводу для неї не буде у будь-якому випадку, проте інші рядки оброблятимуться звичайним способом. У варіанті з EXISTS, значення NULL використовується в предикаті підзапиту, роблячи його невідомим у кожному випадку, а це означає, що підзапит не виводити жодних значень, і запит не здійснить виводу взагалі. Це є важливим чинником на користь використання варіанту з ANY.

Таким чином, підзапити не найпростіше, що є в SQL, проте вони є одним з щонайпотужніших засобів цієї мови. Як правило, допускається виконувати той же запит з підзапитами декількома способами, і, зрештою, користувачу вирішувати, яким варіантом скористатися.

2.3 Внесення змін в базу даних

При роботі з SQL виключно важливо не тільки уміти вибирати дані, але і користуватися засобами, які управляють значеннями в таблиці. Значення можуть бути поміщені і видалені з полів трьома командами мови DML (Мова Маніпулювання Даними), а саме:

- INSERT – вставити;
- UPDATE – модифікувати;
- DELETE – видалити.

2.3.1. Додавання інформації в базу даних

Всі записи в SQL вводяться з використанням команди модифікації INSERT. У найпростішій формі ця команда має наступний синтаксис:

```
INSERT INTO <table name>
     VALUES (<value>, <value>, …);
```

Так, наприклад, для додавання запису в таблицю викладачів TEACHERS, можна скористатися наступним виразом:

```
INSERT INTO TEACHERS
   VALUES(4006,'Федченко','Світлана','Генадіївна','01/09/1999');
```

Команда INSERT не здійснює жодного виводу, але бажано, щоб СУБД давала деяке підтвердження того, що дані були успішно внесені. Крім того, слід пам'ятати, що ім'я таблиці, в яку здійснюється вставка, повинно бути заздалегідь означене, а кожне значення в списку даних, що вставляються, повинне співпадати з типом даних стовпця, в який воно вставляється. Значення в цьому списку вводяться в таблицю в тому порядку, в якому вони записані в команді, тому перше значення автоматично потрапляє в перший стовпець, друге – в другий стовпець і т.д.

Якщо потрібно ввести в таблицю NULL значення, то воно вводиться точно так, як і звичайне. Наприклад, наступна команда, що вставляє запис з невідомим значенням коду викладача, цілком допустима:

```
INSERT INTO TEACHERS
   VALUES(NULL,'Федченко','Світлана','Генадіївна','01/09/1999');
```

Оскільки значення NULL – спеціальне службове слово, то брати його в одиночні лапки не потрібно.

Також допускається вказувати стовпці, куди необхідно здійснити вставку значення, що дозволяє робити це у будь-якому порядку. Наприклад, команда:

```
INSERT INTO TEACHERS (TDATE, TPRIZ, TIMA)
     VALUES ('01/09/1999', 'Федченко', 'Світлана');
```

дозволяє вставити значення в поля таблиці у порядку TDATE, TPRIZ, TIMA, причому стовпці TNUM і TBAT відсутні. Це означає, що для цих полів автоматично встановлюється значення за замовчуванням. Значення за замовчуванням може бути введене наперед або, інакше, це буде NULL значення. Якщо обмеження забороняє використання значення NULL в даному полі, то обов'язково треба потурбуватися про забезпечення стовпця змістовним значенням для будь-якої команди INSERT.

Можна використовувати команду INSERT для того, щоб отримувати або вибирати значення з однієї таблиці і поміщати їх в іншу разом із запитом. Для цього пропозиція VALUES замінюється на відповідний запит:

```
INSERT INTO EXCELLENT
     SELECT * FROM USP
          WHERE OCINKA = 5;
```

В результаті буде сформована таблиця з даними, приведеними в табл. 2.5 (нагадаємо, що сама INSERT команда жодного виводу не здійснює).

Таблиця 2.5. Таблиця EXCELLENT.

EXCELLENT				
UNUM	OCINKA	UDATE	SNUM	PNUM
1001	5	10/06/2005	3412	2001
1005	5	12/06/2005	3416	2004

Отже, буде зроблено наступне: всі значення, видані запитом (інформація про студентів, що мають тільки відмінні оцінки), поміщаються в таблицю, названу EXCELLENT. Для того, щоб не відбулося помилки, таблиця EXCELLENT повинна вже бути створена командою CREATE TABLE і мати п'ять стовпців, які співпадають з таблицею USP по типу даних.

Таким чином, буде отримана незалежна таблиця з деякими даними з таблиці успішності USP. При зміні значень в таблиці USP це у жодному випадку не відобразиться на таблиці EXCELLENT.

В принципі, є можливість вказувати стовпці по імені, як це вже було продемонстровано вище, а значить – переупорядковувати інформацію, що додається.

Наприклад, за допомогою нижчеприведеної команди можна вставити інформацію про середній бал кожного студента:

```
INSERT INTO AVGRAITING {SNUM, AVGOCINKA)
     SELECT SNUM, AVG (OCINKA)
          FROM USP
          GROUP BY SNUM;
```

Зверніть увагу на те, що вказані імена стовпців таблиці AVGRAITING, а значить, послідовність даних в списку (тобто порядок проходження полів в пропозиції SELECT), що вставляється, повинна з цим порядком співпадати.

У INSERT можна використовувати підзапити всередині будь-якого запиту, який генерує значення для цієї команди аналогічно тому, як це вже робилося при розгляді відповідного матеріалу. Наприклад, для вставки у вже наявну таблицю STO прізвищ, імен і по батькові студентів, в яких є хоч би одна відмінна оцінка, можна скористатися наступною командою:

```
INSERT INTO STO (SPRIZ, SIMA, SBAT)
     SELECT SPRIZ, SIMA, SBAT FROM STUDENTS
          WHERE SNUM = ANY (SELECT SNUM FROM USP
               WHERE OCINKA = 5);
```

Обидва запити в цій команді функціонують так само, як якби вони не були частиною виразу INSERT. Підзапит знаходить всі рядки для студентів, що мають відмінні оцінки, і формує набір значень SNUM. Зовнішній запит вибирає рядки з таблиці STUDENTS, де ці

значення SNUM знайдені, а INSERT вставляє знайдені дані в таблицю STO.

У команді INSERT допускається використовувати співвіднесені підзапити. Припустимо, що є таблиця MAXOCINKA, в якій зберігається інформація про студента, що має максимальну оцінку за певну дату (скажімо, для нарахування іменної стипендії). Тоді, для відстежування зміни даних в таблиці успішності і модифікації відповідної інформації про претендента на іменну стипендію, необхідно скористатися наступною командою із співвіднесеним підзапитом:

```
INSERT INTO MAXOCINKA (SNUM, OCINKA)
     SELECT SNUM, OCINKA FROM USP FIRST
          WHERE OCINKA = (SELECT MAX (OCINKA) FROM USP SECOND
               WHERE FIRST.UDATE = SECOND.UDATE);
```

При цьому дана команда має підзапит, який базується на тій же самій таблиці, що і зовнішній запит, але не посилається на таблицю MAXOCINKA, на яку впливає команда, тому така конструкція є допустимою.

2.3.2. Видалення даних

Видалення рядків з таблиці можна здійснити командою модифікації DELETE. Варто враховувати, що вона може видаляти тільки цілі записи таблиці, а не індивідуальні значення того або іншого поля. З цієї причини для даного оператора параметр поля є недоступним. Наприклад, для видалення всього вмісту таблиці STUDENTS, можна скористатися наступним:

```
DELETE FROM STUDENTS;
```

В процесі роботи частіше необхідно видаляти не всі дані, а тільки деякі певні рядки з таблиці. Для того, щоб визначити, які рядки будуть видалені, використовують предикат, аналогічно тому, як це робиться для запитів. Наприклад, щоб видалити інформацію про студента Нагірний, можна використовувати наступну команду:

```
DELETE FROM STUDENTS
     WHERE SNUM = 3416;
```

Тут як предикат використаний номер студентського квитка: дійсно, це поле фактично є первинним ключем таблиці, що дає гарантію видалення тільки одного запису. Використання поля SPRIZ, взагалі кажучи, приводить до видалення декількох записів, оскільки в таблиці могла зберігатися інформація про однофамільців.

У команді DELETE допускається використовувати предикат, що вибирає цілу групу рядків. Наприклад, наступна команда видаляє з таблиці USP всі дані, що відносяться до оцінок, отриманих 10/06/2005:

```
DELETE FROM USP
     WHERE UDATE = '10/06/2005';
```

Нарешті, так само, як і у випадку з командою INSERT, допускається в предикаті використання вкладеного запиту. Найчастіше це необхідно, коли критерій, по якому вибираються дані для видалення, базується на іншій таблиці. Наприклад, якщо виникає необхідність у видаленні інформації про студентів з таблиці STUDENTS, причому для таких, у яких є трійки по будь-якому з навчальних предметів, то потрібно виконати наступне:

```
DELETE FROM STUDENTS
     WHERE SNUM = (SELECT SNUM FROM USP
```

```
        WHERE OCINKA = 3);
```

В даному випадку підзапит вибере всіх студентів, що мають трійки, з таблиці успішності, і в предикат основної команди поверне номери їх студентських квитків.

Допускається в предикаті команди DELETE використовувати і підзапити, що дає можливість встановити досить складні критерії того, які рядки віддалятимуться. Крім того, дуже ефективно виконувати спочатку вторинні дії (перевірки і т.п.), після чого виконувати саме видалення. Хоча не можна посилатися на таблицю, з якої віддалятимуться записи, в пропозиції FROM підзапиту, в предикаті допускається посилання на поточний рядок цієї таблиці, тобто можна використовувати співвіднесені підзапити. Наприклад:

```
DELETE FROM STUDENTS
     WHERE EXISTS (SELECT * FROM USP
          WHERE OCINKA = 3
          AND STUDENTS.SNUM = USP.SNUM);
```

Власне вся команда видаляє інформацію, аналогічно попередньому прикладу. Зверніть увагу на те, що частина предиката внутрішнього запиту посилається до таблиці STUDENTS. Це означає, що весь підзапит виконуватиметься окремо для кожного рядка даної таблиці.

В цілому, структура команди DELETE достатньо проста для розуміння, тому детальніше на ній зупинятися не будемо.

2.3.3. Зміна існуючих даних

Набагато серйознішим питанням є можливість зміни деяких або всіх значень в існуючому рядку таблиці, що реалізується за допомогою команди UPDATE.

Ця команда містить ключове слово UPDATE, де вказується ім'я використовуваної таблиці, і пропозиція SET, яка визначає зміну, що вноситься, для необхідного поля таблиці.

Наприклад, щоб змінити оцінки всіх студентів на 5, необхідно використати команду:

```
UPDATE USP
     SET OCINKA = 5;
```

Звичайно, набагато частіше доводиться вказувати не все, а тільки певні рядки таблиці для зміни єдиного значення, і з цією метою разом з UPDATE можна використовувати предикати. Наприклад, змінити оцінки на 5 по предмету з кодом 2003, можна виконавши таку команду:

```
UPDATE USP
     SET OCINKA = 5
     WHERE PNUM = 2003;
```

За допомогою команди UPDATE можна модифікувати дані з декількох полів – пропозиція SET може призначати будь-яке число стовпців, які відокремлюються комами. Всі вказані призначення можуть бути зроблені для будь-якого табличного рядка, але тільки для однієї в кожен момент часу. Припустимо, що викладач Вікулін пішов на пенсію, і замість нього заняття повинен вести викладач Федченко. Це можна реалізувати такою командою:

```
UPDATE TEACHERS
     SET TPRIZ='Федченко', TNAME='Світлана', TBAT='Геннадіївна',
     TDATE = '01/09/1999'
          WHERE TNUM = 4001;
```

Ця команда передасть новому викладачу Федченко всі поточні навчальні предмети з таблиці PREDMET – в нашому прикладі це буде фізика. Проте майте на увазі, що модифікувати відразу багато таблиць в одній команді UPDATE не можна, а отже, не можна і використовувати назву (префікс) таблиці з ім'ям поля для цієї команди. Тобто, наприклад, пропозиція

```
SET TEACHERS.TPRIZ='Федченко'
```

викличе помилку.

У пропозиції SET команди UPDATE можна використовувати вирази, розташовуючи їх в списку для того поля, яке необхідно змінити (нагадаємо, що в пропозиції VALUES команди INSERT виразу використовувати не можна). Наприклад, для того, щоб збільшити стипендію в 2 рази, можна використовувати наступну конструкцію:

```
UPDATE STUDENTS
     SET STYP = STYP*2;
```

При цьому кожного разу, коли команда посилається до вказаного значення поля в пропозиції SET, дія проводиться, зрозуміло, над ще не модифікованими даними поточного запису.

Крім того, можна використовувати складніші предикати вибору запису для модифікації. Наприклад, якщо необхідно подвоїти тільки стипендію розміром 250,50, то команда буде наступною:

```
UPDATE STUDENTS
     SET STYP = STYP*2
          WHERE STYP ='250,50';
```

і будуть змінені тільки задовольняючі предикату значення.

До речі кажучи, команда UPDATE може працювати з NULL значеннями. Оскільки пропозиція SET не є предикатом, то можна вводити NULL значення так само, як вводяться звичайні дані. Отже, якщо необхідно змінити всі оцінки студентів по навчальному предмету з кодом 2003 на NULL, можна скористатися наступною командою:

```
UPDATE USP
     SET OCINKA = NULL
          WHERE PNUM = 2003;
```

Могутнім засобом модифікації даних є використання підзапитів в команді модифікації UPDATE. Важливий принцип, якого треба дотримуватися при роботі з командами модифікації і підзапитами, полягає в тому, що не можна в пропозиції FROM будь-якого підзапиту модифікувати таблицю, до якої посилається основна команда.

Зверніть увагу на наступний важливий момент – в команді модифікації UPDATE (до речі, так само, як і в команді INSERT) може виникнути проблематична ситуація, пов'язана з можливими дублікатами рядків, що отримуються в результаті вкладеного запиту. В цьому випадку, якщо в таблиці, що модифікується, є обмеження, які змушують її значення бути унікальними, команда модифікації або вставки потерпить невдачу. Тому рекомендується яким-небудь чином з'ясувати те, чи ці значення не могли вже бути використані в таблиці, перш ніж намагатися вставити або модифікувати запис. Це можна реалізувати за допомогою додавання вкладеного підзапиту, що використовує в предикаті оператори EXISTS, IN <> або аналогічні.

З іншого боку, не варто забувати про заборону посилання у вкладених запитах до таблиці, яка модифікується командою UPDATE. З цієї причини запити і підзапити в командах

модифікації мають інколи досить складну структуру. Крім того, всередині необов'язкового предиката цієї команди можна використовувати співвіднесені підзапити, аналогічно тому як це робиться для DELETE.

Наприклад, наступний запит збільшує розмір стипендії в 2 рази студентам, у яких є оцінки, принаймні по двох навчальних предметах:

```
UPDATE STUDENTS
     SET STYP = STYP*2
          WHERE 2 <= (SELECT COUNT (SNUM) FROM USP
               WHERE STUDENTS.SNUM = USP.SNUM);
```

Тут внутрішній запит підраховує кількість записів в таблиці успішності для кожного студента, і, якщо воно 2 або більше, предикат основної функції стає істинним, а розмір стипендії модифікується.

Розглянемо ще один, досить складний, приклад із співвіднесеним підзапитом. Тут модифікуватимемо розмір стипендії для студентів, що мають мінімальний бал в той або інший день:

```
UPDATE STUDENTS
     SET STYP = STYP-1
          WHERE SNUM IN
          (SELECT SNUM FROM USP FIRST
          WHERE OCINKA =(SELECT MIN (OCINKA) FROM USP SECOND
               WHERE FIRST.UDATE = SECOND.UDATE));
```

Як вже говорилося, до суттєвого недоліку UPDATE варто віднести неможливість послатися на таблицю, задіяну в будь-якому підзапиті з команди модифікації. Наприклад, неможливо однією командою виконати таку дію, як модифікація оцінок для студентів, у яких оцінки нижче за середню. Для виконання цієї дії спочатку доведеться виконати пошук середньої оцінки

```
SELECT AVG (OCINKA) FROM USP;
```

а потім результат цього запиту (4.2) використовувати для модифікації

```
UPDATE USP
     SET OCINKA = OCINKA-1
     WHERE OCINKA < 4.2;
```

Таким чином, команда UPDATE, що управляє вмістом запису, є однією з ключових в мові SQL. Вона застосовна як до всіх рядків таблиці, якщо не використовується предикат, що визначає записи, які модифікуються, так і до конкретних рядків за наявності предиката, який, у свою чергу, може мати досить складну структуру.

2.4. Способи створення баз даних

2.4.1. Створення баз даних

Кажучи про DML, було встановлено, що ці оператори можуть маніпулювати з даними, що зберігаються в БД, проте не можуть змінювати її структуру. Для зміни структури БД в SQL

передбачена мова визначення даних, або DDL.

Звичайному користувачу вкрай рідко доводиться створювати БД або таблиці всередині неї. Традиційно він працює з вже готовою структурою, яка вже розроблена і реалізована адміністратором БД. Проте, для повного розуміння особливостей роботи SQL на операторах DDL варто зупинитися достатньо детально. За допомогою цих операторів можна:

- створити нову БД;
- визначити структуру нової таблиці і створити цю таблицю;
- видалити існуючу таблицю;
- змінити означення існуючої таблиці;
- означити представлення даних;
- забезпечити умови безпеки БД;
- створити індекси для доступу до таблиць;
- управляти розміщенням даних на пристроях зберігання.

Оператори DDL дозволяють користувачу не вникати в деталі зберігання інформації в БД на фізичному рівні, оскільки оперують, наприклад, такими поняттями, як таблиці або поля. В той же час, оператори DDL володіють можливістю маніпуляції із фізичною пам'яттю.

Власне DDL базується на трьох командах SQL:

- CREATE – створити, що дозволяє означити і створити об'єкт БД;
- DROP – видалити, застосовується для видалення існуючого об'єкту даних;
- ALTER – змінити, за допомогою якого можна змінити означення об'єкту БД.

Використання команд DDL під час роботи дозволяє зробити структуру реляційної БД динамічною. Іншими словами, в СУБД можна створювати, видаляти або змінювати таблиці, одночасно з цим забезпечуючи доступ користувачам до даних. У свою чергу, це означає, що з часом БД може рости і змінюватися, а її експлуатація може продовжуватися в той час, коли в БД додаються нові таблиці і додатки.

Оператори DDL в СУБД можна використовувати як в інтерактивному, так і в програмному SQL. Наприклад, якщо програмі або користувачу потрібна таблиця для тимчасового зберігання результатів, то допускається створити цю таблицю, заповнити її інформацією, виконати необхідні маніпуляції з даними і потім видалити її. У серйозних СУБД за створення нових БД відповідає тільки адміністратор, хоча не виключена можливість того, що і окремим користувачам це може бути дозволено.

Методи створення БД, що використовуються в провідних реляційних СУБД, мають ряд відмінностей. Наприклад, в Microsoft SQL Server існує оператор CREATE DATABASE, який є частиною мови означення даних і служить для створення БД. Відповідно, оператор DROP DATABASE видаляє існуючі БД. Ці оператори можна використовувати як в інтерактивному, так і в програмному SQL.

Більшість розрахованих на багато користувачів БД мають достатньо нескладну організацію фізичної пам'яті, що забезпечує підвищення її продуктивності. Наприклад, в Microsoft SQL Server адміністратор БД може за допомогою оператора CREATE DATABASE задати один або декілька іменованих файлів:

```
CREATE DATABASE <NAME_DATABASE>
     ON <FILE1>, <FILE2>, ...
```

Підхід, що використовується в SQL Server, дозволяє розподіляти вміст БД по декількох дискових томах, про що вже говорилося вище. Наступним кроком, слід за створенням порожньої БД, є заповнення її таблицями.

2.4.2. Створення таблиць

Отже, після створення БД необхідно здійснити створення, зміну, а якщо потрібно – то і

видалення таблиць. Ці дії відносяться до самих таблиць, а не до даних, які в них містяться.

Таблиці створюються командою CREATE TABLE. Ця команда створює порожню таблицю, тобто що не містить записів. Очевидно, що значення в неї можна ввести, наприклад, за допомогою команди INSERT. Головне в команді CREATE TABLE – це означення імені таблиці і опису набору імен полів, які вказуються у відповідному порядку. Крім того, цією командою також задаються типи даних і розміри полів таблиці.

Очевидно, що в кожній таблиці повинно бути, принаймні, одне поле.

Синтаксис команди CREATE TABLE наступний:

```
CREATE TABLE <TABLE_NAME>
     (<COLUMN NAME> <DATA TYPE> [(<SIZE>)],
     <COLUMN NAME> <DATA TYPE> [(<SIZE>)] ...);
```

Майте на увазі: внаслідок того, що пропуски використовуються для розділення елементів команд SQL, вони не можуть бути частиною імені таблиці або будь-якого іншого об'єкту, що створюється. Тому символ підкреслення звичайно використовується для розділення слів в іменах таблиць.

Значення аргументу розміру залежить від типу даних. Якщо його не вказувати, то СУБД сама призначатиме значення автоматично. Треба сказати, що для числових значень це часто буває кращим виходом, оскільки в цьому випадку всі поля такого типа одержать один і той же розмір, і будуть виключені проблеми їх загальної сумісності. Крім того, використання аргументу розміру з деякими числовим даними не зовсім просте питання – якщо потрібно зберігати великі числа, то необхідно переконатися в тому, що поля достатньо великі для розміщення даних.

В той же час, тип даних CHAR вимагає обов'язкової вказівки розміру. Аргумент розміру – це ціле число, що визначає максимальну кількість символів, яку може вміщати поле. Фактично, кількість символів такого поля може бути від нуля (якщо поле має значення NULL) до цього числа. За замовчуванням аргумент розміру рівний 1, а це означає, що поле може містити тільки один символ.

Крім того, таблиці належать користувачу, який їх створив, а імена всіх таблиць, що належать даному користувачу, повинні відрізнятися одне від одного так само, як і імена всіх полів всередині даної таблиці. Проте різні таблиці можуть використовувати однакові імена полів, навіть якщо вони належать одному і тому ж користувачу. Наприклад, поле з ім'ям SNUM присутнє в таблицях STUDENTS і USP, і нітрохи не заважає одне одному.

Як вже було сказано, користувачі, що не є власниками таблиць, можуть посилатися до цих таблиць за допомогою імені власника, що супроводжується крапкою. Наприклад, ім'я таблиці

```
SA.STUDENTS
```

має на увазі, що звернення йде до таблиці STUDENTS, що створена користувачем з ідентифікатором дозволу (ID) SA.

Приведемо приклад команди, яка створить структуру таблиці STUDENTS:

```
CREATE TABLE STUDENTS
     (SNUM INTEGER, SPRIZ CHAR (20), SIMA CHAR (10),
     SBAT CHAR (15), STYP DECIMAL);
```

Порядок розташування полів в таблиці визначається тим, у якій послідовності вони вказані в команді створення таблиці.

Після того, як таблиця була створена, її можна змінити. Команда ALTER TABLE є широко доступним засобом для того, щоб змінити означення існуючої таблиці. Найчастіше з її допомогою додають поля до таблиці, хоча вона може видаляти або змінювати їх розміри. Типовий синтаксис цієї команди для додавання стовпця до таблиці, такий:

```
ALTER TABLE <TABLE NAME>
     ADD <COLUMN NAME> <DATA TYPE> <SIZE>;
```

Варто пам'ятати, що поле буде додано з NULL значеннями для всіх записів таблиці. Крім того, нове поле стане останнім по порядку в таблиці. Допускається додавання відразу декілька нових полів, відокремивши їх комами в одній команді.

Наприклад, для додавання до таблиці STUDENTS двох полів для зберігання інформації про курс і спеціальність студента, можна скористатися наступною командою:

```
ALTER TABLE STUDENTS
     ADD COURS INTEGER,
     ADD SPEC CHAR (10);
```

З використанням цієї команди є можливість видаляти або змінювати поля, причому найчастіше зміною буває просто збільшення його розміру. Обов'язково потрібно переконатися, що будь-які зміни, що вносяться, не суперечать існуючим даним – наприклад, спроба зменшити розмір поля може привести до втрати даних.

ALTER TABLE не діє, коли таблиця повинна бути переозначена, проте при розробці БД не варто виключати необхідність цієї дії. Крім того, зміна структури таблиці в той момент, коли вона знаходиться у використанні, також супроводжується втратою інформації – наприклад, запит може потерпіти невдачу з тієї причини, що деякого поля в таблиці просто вже не існує. З цих причин краще розробляти БД так, щоб використовувати ALTER TABLE тільки в крайньому випадку.

Для того, щоб мати можливість видалити таблицю, користувач повинен бути її власником, тобто творцем. Крім того, перед видаленням, SQL зажадає очищення таблиці від даних, що дозволяє уникнути випадкової і непоправної втрати інформації. Таким чином, таблиця, з рядками, що знаходяться в ній, не може бути видалена.

Синтаксис команди для видалення таблиці (за умови, що вона є порожньою) наступний:

```
DROP TABLE <TABLE NAME>;
```

Після виконання цієї команди, ім'я таблиці більше не розпізнається, і немає таких дій, які могли бути виконані з цим об'єктом. Перед видаленням варто переконатися в тому, що ця таблиця не посилається на іншу таблицю і що вона не використовується в якому-небудь уявленні.

Наприклад, для видалення таблиці STUDENTS, в якій всі записи заздалегідь видалені, просто вводиться наступне:

```
DROP TABLE STUDENTS;
```

Таким чином, використання розглянутих DDL команд дозволяє створювати нові, змінювати структуру тих, що існують і видаляти порожні таблиці БД.

Якщо необхідно видалити конкретну колонку таблиці, то необхідно виконати наступну команду:

```
ALTER TABLE <TABLE NAME>
     DROP COLUMN <COLUMN NAME>;
```

2.4.3. Індекси

Індексом прийнято називати впорядкований список полів або груп полів в таблиці. Таблиці можуть мати величезну кількість записів, при цьому, як було відмічено вище, записи не знаходяться в якому-небудь певному порядку, тому на їх пошук по вказаному критерію може бути потрібний достатньо тривалий час.

Індексна адреса – це спеціальний метод забезпечення об'єднання всіх значень в групи з одним або більше записів, які відрізняються один від одного, оскільки унікальність записів часто необхідна.

Індекси – це корисний інструмент, який широко застосовується у всіх сучасних СУБД. Коли створюється індекс в полі, БД запам'ятовує відповідний порядок всіх значень цього поля у області пам'яті. Про переваги індексів може говорити наступне: припустимо, що таблиця STUDENTS має декілька тисяч записів, і необхідно знайти студента з конкретним номером студентського квитка. Оскільки записи в таблиці не впорядковані, то СУБД буде вимушена проглядати всю таблицю, рядок за рядком, перевіряючи кожного разу значення поля SNUM на рівність шуканому значенню. За наявності індексу в полі SNUM, система могла б знайти шуканий номер прямо в цьому впорядкованому індексі, і дати інформацію про те, як знайти правильний рядок таблиці.

У індексів є і недоліки. Тоді як індекс значно покращує ефективність запитів, використання індексу дещо уповільнює операції модифікації, особливо такі, як INSERT і DELETE. Крім того, сам індекс займає місце на пристрої зберігання інформації.

Звідси випливає, що при створенні таблиці необхідно ухвалити зважене рішення, індексувати її чи ні. Індекси можуть складатися відразу з декількох полів, при цьому перше поле є як би головним, друге впорядковується всередині першого, третє всередині другого, і т.д.

Синтаксис команди для створення індексу наступний:

```
CREATE INDEX <INDEX NAME> ON <TABLE NAME>
     (<COLUMN NAME> [,<COLUMN NAME>]...);
```

Зрозуміло, що таблиця, для якої створюється індекс, повинна вже існувати і містити імена індексованих полів. При цьому ім'я індексу не може бути використано для чого-небудь іншого в БД і SQL сама вирішує, коли він необхідний для роботи і використовує його автоматично.

Приведемо наступний приклад. Очевидно, що в таблиці STUDENTS одним з тих, що найбільш часто використовується може бути індекс по полю, що містить прізвище студента. Тоді команда для створення такого індексу буде наступною:

```
CREATE INDEX SPRIZIDX ON STUDENTS (SPRIZ);
```

Після цього при пошуку інформації про студентів, СУБД знаходитиме її дуже швидко. Проте, при створенні цього індексу, йому не прописана унікальність, не дивлячись на те, що це є одним з його призначень.

Для створення унікальних (що не містять значень, які повторюються) індексів використовують ключове слово UNIQUE в команді CREATE INDEX. Фактично такий індекс буде первинним ключем таблиці. Наприклад, для таблиці STUDENTS поле SNUM підходить як первинний ключ, і він стане першим кандидатом для унікального індексу. Створити його можна командою:

```
CREATE UNIQUE INDEX
     SNUMIDX ON STUDENTS (SNUM);
```

Майте на увазі, що ця команда не буде виконана, якщо в полі SNUM є не унікальні значення. Тому рекомендується створювати індекси відразу після того, як створена таблиця і до введення в неї яких-небудь значень. Цікава і така особливість унікального індексу: якщо в ньому використовується більше одного поля (тобто він є комбінацією значень), то, взагалі кажучи, кожне з цих полів може і не бути унікальним.

Оскільки основною ознакою індексу є його ім'я, то по його імені він може бути ідентифікований і видалений. Звичайно користувачі не знають про існування індексу, а SQL автоматично визначає – чи дозволено користувачу використовувати індекс, і, якщо так, то вирішує його використання.

Проте, для видалення індексу, необхідно знати його ім'я. З врахуванням цього команда для видалення має наступний синтаксис:

```
DROP INDEX <INDEX NAME>;
```

Наприклад, для видалення створеного індексу по прізвищу студента, можна скористатися наступною командою:

```
DROP INDEX SPRIZIDX;
```

Врахуйте, що видалення індексу у жодному випадку не впливає на дані, які містяться в полях.

2.4.4. Означення умов перевірки

Для розширення можливостей роботи з даними, що вводяться в таблицю, цікавий механізм умов перевірки значень, або, як його ще часто називають – обмеження даних.

Обмеження даних – це частина означень таблиці, яка обмежує значення, допустимі до введення в поля таблиці. До цього як обмеження на дані, розглянуті нами, були тільки тип і розмір значень, що вводяться, які повинні бути сумісні з тими полями, в які ці значення поміщаються.

Обмеження дають великі можливості і, наприклад, описавши значення за замовчуванням (це значення, яке автоматично вставляється в будь-яке поле таблиці, коли явне значення відсутнє в команді), команда INSERT для поля вже не вставлятиме NULL значення.

Коли створюється таблиця або змінюється її структура, можна встановлювати обмеження на значення, які можуть бути введені в поля. Якщо це зроблено, SQL відкидатиме будь-які значення, які порушують обумовлені критерії. Як вже було сказано раніше, є два основних типи обмежень – обмеження поля (атрибуту) і обмеження таблиці (відношення). Нагадаємо, що відмінність між ними в тому, що обмеження поля застосовується тільки до певного поля, а обмеження таблиці – до груп з одного і більше полів.

Оголошення обмежень здійснюється таким чином: обмеження поля поміщається в кінець фрагмента команди, що оголошує його ім'я після типу даних; обмеження таблиці поміщається в кінець оголошення імені таблиці після останнього імені поля. Таким чином, повний синтаксис для команди CREATE TABLE такий:

```
CREATE TABLE <TABLE NAME>
     (<COLUMN NAME> <DATA TYPE> [(<SIZE>)] <COLUMN CONSTRAINT>,
     (<COLUMN NAME> <DATA TYPE> [(<SIZE>)] <COLUMN CONSTRAINT>,
     ...
     <TABLE CONSTRAINT> (<COLUMN NAME> [,<COLUMN NAME>])...);
```

Тут обмеження поля (COLUMN CONSTRAINT) застосовується до того поля, після чийого імені це обмеження слідує. Назви полів в круглих дужках після обмеження таблиці (TABLE CONSTRAINT) – це поля, до яких застосоване дане обмеження.

Часто команду CREATE TABLE використовують для того, щоб оберегти поле від попадання в нього NULL значень. З цією метою використовують обмеження NOT NULL, яке може застосовуватися тільки як обмеження поля. Очевидно, що первинні ключі таблиці ніколи не повинні бути порожніми, оскільки це порушуватиме їх функціональні можливості. Більше того, такі поля, як правило, вимагають певних унікальних значень.

Наприклад, якщо при створенні таблиці STUDENTS, помістити ключові слова NOT NULL відразу після типу даних для полів, в яких невизначене значення небажано, будь-яка спроба їх запису в ці поля буде відхилена. Якщо ж при створенні таблиці це не обумовлено, то SQL вважає, що NULL значення дозволені. Отже, розглянемо команду:

```
CREATE TABLE STUDENTS
     (SNUM INTEGER NOT NULL, SPRIZ CHAR (20) NOT NULL,
     SIMA CHAR (10), SBAT CHAR (15), STYP DECIMAL);
```

У такій таблиці невизначене значення не може бути встановлене для полів SNUM і SPRIZ. При цьому важливо не забувати, що тепер для цих стовпців обов'язково повинне бути встановлене явне значення в команді INSERT.

При використанні команди ALTER TABLE для додавання нових стовпців до вже існуючої таблиці, можна поміщати обмеження стовпців типу NOT NULL. Проте якщо новому стовпцю встановлюється обмеження NOT NULL, таблиця, що модифікується, повинна бути порожньою.

Часто в полі потрібно реалізувати обмеження, пов'язане з унікальністю значень – це навіть скоріше властивість даних в таблиці, і тому його логічніше назвати обмеженням даних, а не просто обмеженням поля. Поза сумнівом, унікальні індекси – один з найпростіших і найбільш ефективніших методів пропису унікальності. Проте є можливість встановити унікальність як обмеження стовпця.

Якщо необхідно бути упевненим в тому, що всі значення, введені в поле, відрізняються один від одного, то в обмеження поля при створенні таблиці поміщають ключове слово UNIQUE. При цьому СУБД відхилить будь-яку спробу введення в це поле значення, яке вже має місце в іншому рядку. Це обмеження може застосовуватися тільки до полів, для яких вже було оголошене обмеження NOT NULL, що цілком логічно – не має сенсу дозволити одному запису мати значення NULL, а потім виключати інші рядки з NULL значеннями як дублікати.

Для ілюстрації цих можливостей, удосконалимо попередній приклад і заборонимо значення, що повторюються, в полі SNUM, оскільки в цій таблиці логічне використання його як первинного ключа:

```
CREATE TABLE STUDENTS
     (SNUM INTEGER NOT NULL UNIQUE, SPRIZ CHAR (20) NOT NULL,
     SIMA CHAR (10), SBAT CHAR (15), STYP DECIMAL);
```

Нагадаємо, що поля, значення в яких вимагають унікальності, що проте не є первинними ключами, називаються ключами-кандидатами або унікальними ключами. Наприклад, можна було б передбачити обмеження унікальності для поля SPRIZ, тоді його можна було б вважати ключем-кандидатом, проте це не дозволило б зберігати в таблиці STUDENTS однофамільців.

У SQL можна визначити групу з декількох полів як унікальну за допомогою команди обмеження таблиці UNIQUE. Оголошення групи полів унікальної, принципово відрізняється від

оголошення унікальним одного поля: у першому випадку це – комбінація значень, а в другому – індивідуальне значення кожного запису. При цьому зовсім не обов'язково, щоб в кожному полі, що входить до групи, значення не були такими, що повторюються. З іншого боку, якщо будь-яке з полів, що входять в комбінацію умови унікальності, саме по собі є унікальним, то великого сенсу в такому обмеженні немає.

Наприклад, в таблиці успішності USP така ситуація повинна бути з парами значень номер студентського квитка – код предмету – окремий запис для кожної здачі студентом того або іншого навчального предмету не повинен існувати. Таким чином, при створенні таблиці USP скористаємося наступною командою:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL UNIQUE, OCINKA INTEGER, UDATE DATE,
     SNUM INTEGER NOT NULL, PNUM INTEGER NOT NULL,
     UNIQUE (SNUM, PNUM));
```

Зверніть увагу на те, що обидва поля в обмеженні таблиці UNIQUE ще до того ж використовують обмеження стовпця NOT NULL.

До цього використовувалося обмеження UNIQUE або унікальні індекси в первинних ключах для припису цим полям властивості унікальності. Проте SQL підтримує первинні ключі безпосередньо з обмеженням PRIMARE KEY.

За допомогою PRIMARY KEY можна обмежувати таблицю або окремі його стовпці. Це обмеження працює аналогічно UNIQUE, проте для даної таблиці повинен бути тільки один первинний ключ. Синтаксис і означення його унікальності ті ж, що і для обмеження UNIQUE – первинні ключі не допускають NULL значень, отже, будь-яке поле, що використовується в обмеженні PRIMARY KEY, повинно вже бути оголошено NOT NULL. Розглянемо наступний приклад:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY, OCINKA INTEGER,
     UDATE DATE, SNUM INTEGER NOT NULL, PNUM INTEGER NOT NULL,
     UNIQUE (SNUM, PNUM));
```

Як видно, краще всього поміщати обмеження PRIMARY KEY в полі, яке утворює унікальний ідентифікатор рядка, і зберегти обмеження UNIQUE для поля або групи полів, які повинні бути унікальними логічно, а не використовуватися для ідентифікації рядків.

Обмеження PRIMARY KEY може також бути застосоване для групи полів, що становлять унікальну комбінацію значень.

Спробуємо передбачити для останнього прикладу інший первинний ключ:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL UNIQUE, OCINKA INTEGER, UDATE DATE,
     SNUM INTEGER NOT NULL, PNUM INTEGER NOT NULL,
     PRIMARY KEY (SNUM, PNUM));
```

В цьому випадку первинним ключем є комбінація полів SNUM і PNUM, яка повинна бути унікальною з причин, викладених вище. Очевидно, що ні перше, ні друге поле в цій таблиці не можна примусити не містити значення, що повторюються окремо, проте їх комбінація є унікальна структура. І ще один важливий момент – як первинний ключ краще застосовувати поля числового (переважно – цілого) типу, оскільки використання текстових полів може привести до неоднозначних результатів. Наприклад, не виключена вірогідність того, що при використанні як первинного ключа комбінації полів, що містять прізвище, ім'я і по

батькові студента, в таблиці зможуть з'явитися більше однієї людини, у яких ці дані співпадають. Зрозуміло, що первинний ключ, який в одних випадках працює, а в інших – ні, не є прийнятним.

Зрозуміло, може існувати будь-яке число обмежень, які допускається встановлювати для даних, що вводяться в таблиці БД. Для цього в SQL передбачене обмеження CHECK, що дозволяє поставити умову, відповідно до якої перевіряється значення, що вводиться в таблицю, до того, як його буде збережено. Це обмеження складається з ключового слова CHECK і предиката, який використовує вказане поле. Будь-яка спроба модифікувати або вставити значення поля, яке робить цей предикат невірним, буде відхилена.

Розглянемо для прикладу таблицю USP. Очевидно, що оцінка, одержана студентом, не повинна перевищувати 5 балів, тому можна при створенні цієї таблиці передбачити відповідну перевірку:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER CHECK (OCINKA =< 5), UDATE DATE,
     SNUM INTEGER NOT NULL, PNUM INTEGER NOT NULL,
     UNIQUE (SNUM,  PNUM));
```

Взагалі кажучи, можна використовувати CHECK для того, щоб зумовлювати допустиме значення, що вводиться. Наприклад, для покращення попереднього прикладу, слід передбачити те, що в поле OCINKA можуть бути введені тільки оцінки 1, 2, 3, 4 або 5. Тоді наступна команда дозволить виключити введення інших значень і запобігти можливим друкарським помилкам:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER CHECK IN (1, 2, 3, 4, 5), UDATE DATE,
     SNUM INTEGER NOT NULL, PNUM INTEGER NOT NULL,
     UNIQUE (SNUM, PNUM));
```

Зверніть увагу на те, що команда ALTER TABLE дозволяє змінювати означення таблиці, навіть коли вона знаходиться у використанні. Проте зміна або видалення обмежень не завжди можливе. Надійнішим способом інколи є створення нової таблиці із зміненим обмеженням і заповнення її інформацією із старої таблиці.

У SQL можна також використовувати CHECK як табличне обмеження. Це корисно в тих випадках, коли виникає необхідність включити більше одного поля в умову. Припустимо, що відмінні оцінки можуть бути виставлені тільки до певної дати, скажемо до 15/06/2005. Тоді скористаємося наступною командою:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER,
     UDATE DATE,
     SNUM INTEGER NOT NULL,
     PNUM INTEGER NOT NULL,
     CHECK (OCINKA < 5 AND UDATE > '15/06/2005 '));
```

У таку таблицю тепер можна ввести тільки оцінки нижче 5 балів для дат, пізніше 15/06/2005. При цьому, як можна бачити, два різні поля одного і того ж рядка повинні бути перевірені для того, щоб визначити – вірний предикат чи ні. Крім того, не можна використовувати обмеження CHECK для того, щоб перевірити умову пов'язану відразу з

декількома рядками таблиці, наприклад, що оцінки по дисципліні не нижчі, ніж середня оцінка по всіх навчальних предметах.

Коли відбувається вставка запису в таблицю без вказівки значень для кожного поля, SQL повинна мати значення за замовчуванням для включення його в кожен запис, інакше команда не буде виконана. Найбільш поширеним значенням за замовчуванням є NULL. Це значення за замовчуванням для будь-якого стовпця, якому не було дане обмеження NOT NULL.

У SQL є засіб DEFAULT (за замовчуванням), що вказується в команді CREATE TABLE аналогічно, як і обмеження поля. Майте на увазі, що значення DEFAULT не обмежувальної властивості – воно не обмежує дані, які можна вводити, а просто визначає, що повинно трапитися, якщо не введено будь-яке з явних допустимих значень.

Наприклад, якщо більшість оцінок, яка потрапляє в таблицю USP, це п'ятірки, то логічно вказати це як значення поля OCINKA за замовчуванням, тобто:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER CHECK IN (1, 2, 3, 4, 5) DEFAULT = 5,
     UDATE DATE,
     SNUM INTEGER NOT NULL,
     PNUM INTEGER NOT NULL,
     UNIQUE (SNUM, PNUM));
```

Звичайно, використовувати значення за замовчуванням – це альтернатива для NULL, оскільки невідоме значення невірне при будь-якому порівнянні (окрім IS NULL), і такі записи виключаються за допомогою більшості предикатів. Іноді, коли потрібно бачити порожні значення полів, не обробляючи їх якимсь певним чином, варто встановити значення за замовчуванням нуль для числових або пропуск для текстових полів. Відмінність в даному випадку полягатиме в тому, що тепер такі записи оброблятимуться так само, як і будь-які інші значення. Це означає, що при вибірці даних ці записи будуть включені у виведення запиту, що часто є дуже корисним.

У цьому ж полі допускається використовувати обмеження UNIQUE або PRIMARY KEY, але тоді будь-який рядок, що вставляється, доведеться модифікувати перш, ніж буде доданий який-небудь інший рядок з цією установкою за замовчуванням, що не зовсім зручно.

На практиці для вмісту окремих полів існують і інші обмеження. Наприклад, поле TPRIZ таблиці TEACHERS не повинне насправді містити будь-яке довільне прізвище. Наприклад, прізвище Смирнова є "неправильним", оскільки не відповідає жодному викладачу даного ВНЗу. Формально на мові реляційної БД це означає, що доменом поля TPRIZ є не просто будь-які строкові значення, і навіть не просто прізвища людей, а ряд прізвищ людей, що є викладачами цього ВНЗу.

У стандарті SQL2 означення домена реалізоване як частина означення БД. Згідно цьому стандарту, домен є іменованою сукупністю значень даних і широко використовується у означенні БД, як додатковий тип даних. Домен створюється за допомогою команди CREATE DOMAIN, і, після оголошення, на нього можна посилатися, як на тип даних.

Таким чином, нами розглянуті декілька способів управління значеннями, які можуть бути введені в поля таблиці. Ці засоби дозволять більш повно використовувати можливості SQL і уникати помилок при введенні даних.

2.4.5. Створення синонімів

При кожному зверненні в команді до базової таблиці або уявлення, для якого користувач не є власником, необхідно вказувати префікс імені власника, так щоб SQL знав, де здійснювати її пошук. При виконанні великої кількості запитів і інших команд така вказівка стає незручною,

тому в даному випадку краще скористатися можливістю створення синонімів для таблиць.

Синонім – це альтернативне ім'я для таблиці. Користувач, що створив синонім, стає його власником, а значить, відпадає необхідність того, щоб при зверненні вводився призначений для користувача ідентифікатор доступу.

Якщо для користувача обумовлені правила доступу (привілеї) принаймні, для одного поля таблиці, то він може створити синонім. Наприклад, користувач з ідентифікатором SHER може створити синонім для таблиці STUDENTS, що належить користувачу SA таким чином:

```
CREATE SYNONYM FIRST FOR SA.STUDENTS;
```

Після цього користувач може працювати з таблицею FIRST так само, як це робилося при зверненні до неї на ім'я SA.STUDENTS, оскільки FIRST – це виняткова власність для SHER.

Зверніть увагу на той факт, що префікс (ідентифікатор) користувача – це фактично частина імені будь-якої таблиці. Кожний раз, коли не вказується власне ім'я користувача, SQL має на увазі, що таблиця належить їй і сама заповнює відповідну інформацію. Звідси випливає, що два однакові імена таблиці, пов'язані з різними власниками, є не ідентичними і, значить, не приводять до якого-небудь безладу в SQL. Іншими словами, два користувачі можуть створити дві повністю незв'язані таблиці з однаковими іменами, але це також означатиме, що один користувач може створити уявлення, засноване на імені іншого користувача, що стоїть після імені таблиці. Тому можна створювати власні синоніми користувача, імена яких будуть такими ж, що і первинні імена таблиць. Наприклад, користувач SHER може створити синонім для таблиці STUDENTS, з таким же ім'ям:

```
CREATE SYNONYM STUDENTS FOR SA.STUDENTS;
```

Після цього з погляду SQL тепер є два різні імена однієї таблиці: SHER.STUDENTS і SA.STUDENTS. Проте кожний з цих користувачів може посилатися до цієї таблиці просто як до STUDENTS, а SQL сам розбереться, яка таблиця мається на увазі.

Звичайно, якщо планується мати таблицю студентів STUDENTS, яка використовується великим числом користувачів, то краще так організувати роботу з нею, щоб вони посилалися до неї за допомогою одного і того ж імені. Для того, щоб створити єдине ім'я для всіх користувачів, створюється загальний синонім. Наприклад, якщо всі користувачі викликатимуть таблицю студентів на ім'я STUDENTS, необхідно виконати наступну команду:

```
CREATE PUBLIC SYNONYM STUDENTS FOR STUDENTS;
```

В основному, спільні синоніми створюються власниками об'єктів або привілейованими користувачами, наприклад, адміністратором БД. Користувачам, крім того, повинні ще бути надані привілеї в таблиці STUDENTS для того, щоб вони могли мати до неї доступ. Це пов'язано з тим, що навіть якщо ім'я є спільним, сама таблиця спільною не є.

Спільні та інші синоніми можуть видалятися командою DROP SYNONYM. При цьому вони видаляються їх власниками, за винятком спільних синонімів – для їх видалення, як правило, потрібні привілеї адміністратора БД.

Наприклад, для видалення синоніма FIRST можна скористатися командою:

```
DROP SYNONYM FIRST;
```

Зрозуміло, сама таблиця STUDENTS залишиться без яких-небудь змін.

2.4.6. Архітектура баз даних

Як вже було сказано, в SQL команди CREATE, DROP і ALTER утворюють основу мови означення даних (DDL). Оператори з цими командами використовуються в реляційних СУБД для управління таблицями, індексами і уявленнями. Крім того, ці команди можуть бути використані для маніпуляції іншими додатковими об'єктами БД (див. табл. 2.6).

Відповідно до стандарту SQL1, структура БД повинна бути такою, щоб у кожного користувача була сукупність таблиць, що належить йому. Не дивлячись на те, що в різних СУБД використовується однакова логічна структура окремої БД, спостерігається велика різноманітність в тому, як організована і структурована вся сукупність даних в конкретній операційній системі.

У одних СУБД всі дані зберігаються в одній загальній БД, в інших – в декількох БД, кожній з яких привласнюється ім'я. А в деяких СУБД інформація зберігається в декількох БД, організованих у вигляді системи каталогів. Така різноманітність не робить якого-небудь впливу на структуру мови SQL, що використовується для доступу до інформації, що зберігається в БД, проте вона впливає на спосіб організації даних, а також на спосіб первинного звернення до БД. Тобто при виникненні необхідності звернення до тієї або іншої таблиці, необхідно повідомити СУБД, з якою саме з них потрібно працювати.

Таблиця 2.6 Оператори DDL і об'єкти, керовані ними

Оператори DDL	**Керований об'єкт**
CREATE/DROP/ALTER TABLE	таблиця
CREATE/DROP/ALTER VIEW	уявлення
CREATE/DROP/ALTER INDEX	індекс
CREATE/DROP SYNONYM	синонім
CREATE/DROP/ALTER DATABASE	база даних
CREATE/DROP DEFAULT	значення поля за замовчуванням
CREATE/DROP PROCEDURE	процедура, що зберігається
CREATE/DROP RULE	правило дотримання цілісності поля
CREATE/DROP TRIGGER	тригер, що зберігається

Розглянемо однобазову архітектуру БД, при якій СУБД містить одне спільне сховище даних (див. рис. 2.1).

При такому підході всі дані про студентів, викладачів, навчальні предмети і успішність зберігаються в таблицях всередині однієї БД. Головні таблиці зібрані разом і належать одному користувачу (у нашому випадку – SA). Перевагою такої архітектури є те, що таблиці з різних додатків можуть мати посилання одна на одну. Наприклад, таблиця OTL, що зберігає дані про відмінників, містить зовнішній ключ до таблиці STUDENTS, і СУБД може використовувати цей зв'язок для маніпуляції даними. Маючи відповідні привілеї, користувач може виконувати запити, об'єднуючи дані з різних таблиць.

Таку структуру зберігання використовують, наприклад, в системах DB2 і Oracle. Недоліком такої архітектури є те, що у міру додавання нових елементів, БД набуває величезних розмірів. Проблеми, пов'язані з управлінням БД великих розмірів, очевидні: резервне копіювання, відновлення даних, аналіз ефективності роботи вимагають, як правило, постійної уваги адміністратора БД.

При однобазовій архітектурі доступ до даних здійснюється досить просто, оскільки БД

одна і не потрібно робити який-небудь вибір. При цьому доступ до даних реалізується загалом, а не до будь-якої конкретної БД.

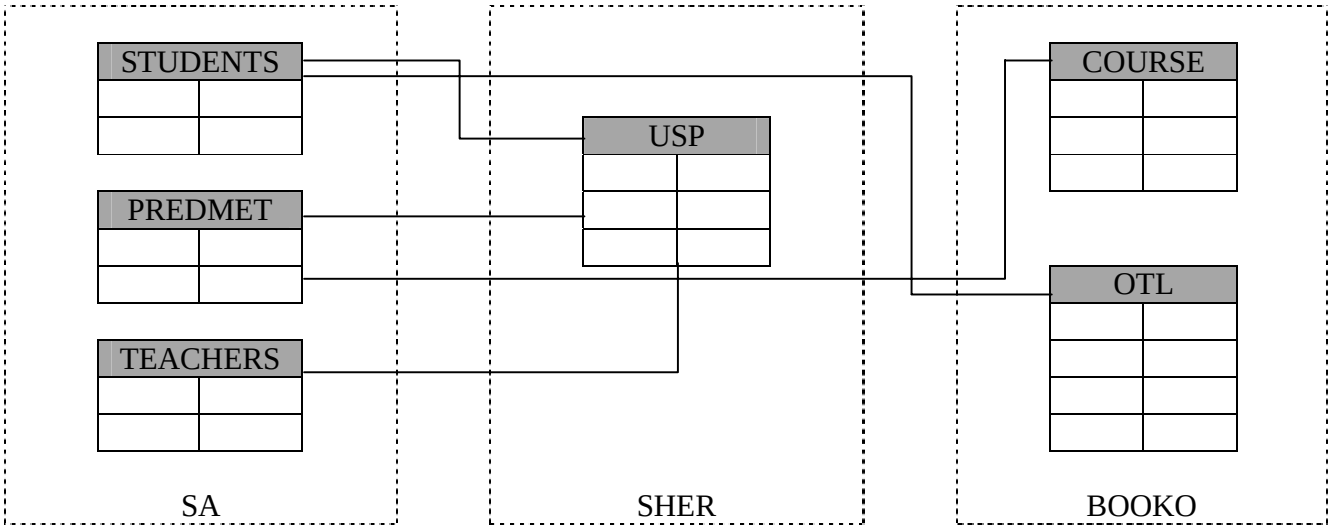

Рис. 2.1. Схема однобазової архітектури БД.

При багатобазовій архітектурі (див. рис. 2.2) кожній БД привласнюється унікальне ім'я. Така архітектура застосовується, наприклад, в SQL Server і Ingres. Як видно з схеми, кожна БД в основному призначена для окремої прикладної задачі. При додаванні нового завдання, створюється нова БД. Основна перевага багатобазової архітектури в порівнянні з однобазовою полягає в тому, що проблема управління розділяється на дрібніші частини, а значить, легше у вирішувані. Користувач, відповідальний за конкретне прикладне завдання, може одночасно бути адміністратором своєї власної БД, менше турбуючись про спільну координацію всієї системи. Якщо з'являється необхідність додати нове завдання, це можна зробити в своїй БД, не порушуючи роботу інших частин системи.

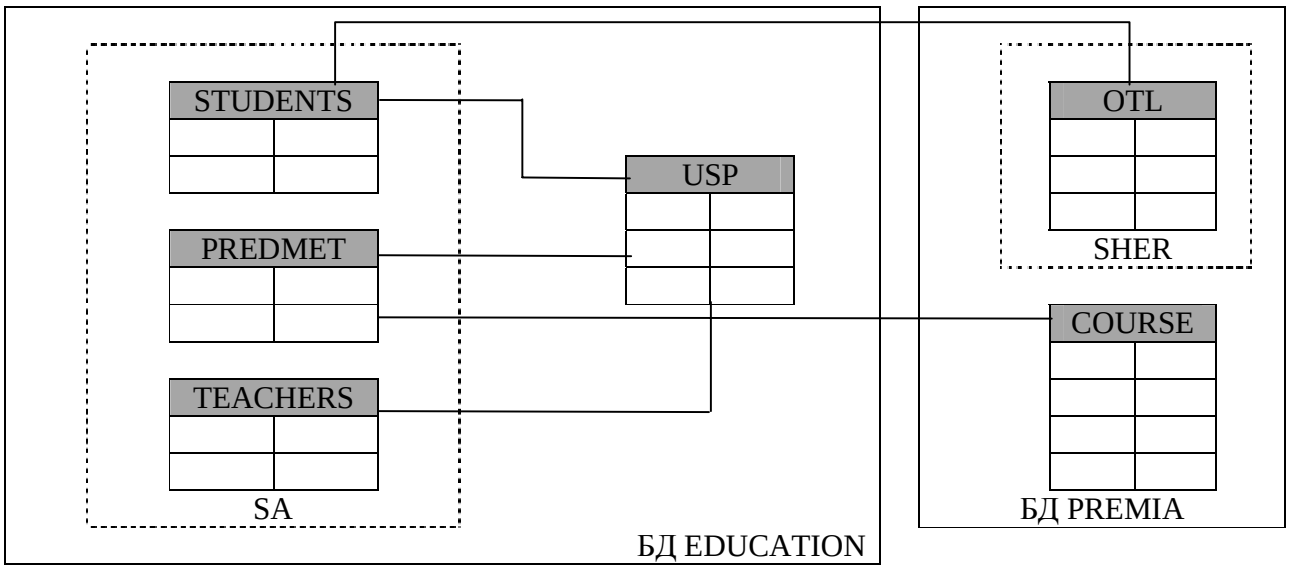

Рис. 2.2. Схема багатобазової архітектури БД.

Головним недоліком багатобазової архітектури є те, що не виключена ймовірність роз'єднаності одна від одної індивідуальних БД. Часто таблиця однієї БД не може містити

зовнішній ключ для таблиці іншої, а значить – СУБД не зможе виконувати міжбазові запити.

Крім того, якщо в СУБД використовується багатобазова архітектура і підтримуються запити між БД, то в ній повинні бути розширені правила іменування полів і таблиць, як це було сказано вище. Нарешті, при багатобазовій архітектурі дещо ускладнюється доступ до даних, оскільки в СУБД необхідно зберігати і передавати інформацію про те, з якою саме БД йде робота. Тому для отримання доступу до даних СУБД просить користувача ввести ім'я БД, ім'я користувача і пароль, і лише після цього доступ до даних дозволяється.

При зберіганні даних за допомогою каталогової архітектури, БД організовується у вигляді деревовидної структури (див. рис. 2.3). Тут, так само, як і в багатобазовій архітектурі, для кожної прикладної задачі створюється власна БД, що має власне ім'я, причому в різних каталогах імена можуть співпадати.

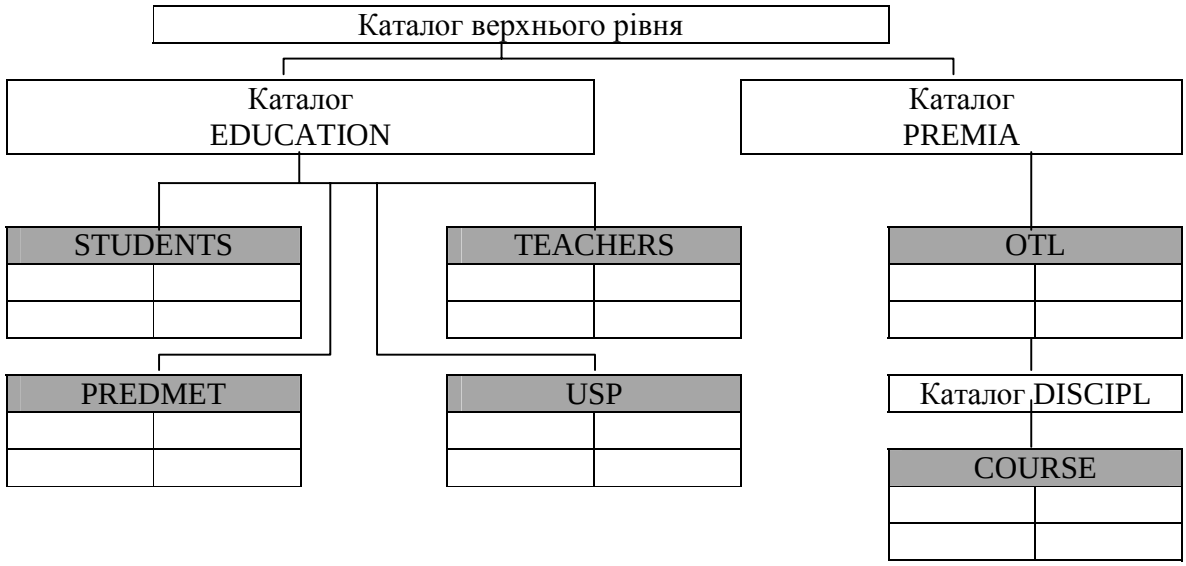

Рис. 2.3. Схема каталогові архітектури БД.

Головне достоїнство каталогової архітектури – це її гнучкість. Така архітектура особливо корисна для інженерних і конструкторських програм, оскільки традиційно з ними працює багато досвідчених користувачів, яким може бути потрібно декілька БД для структурованого зберігання своєї інформації. Недоліки каталогової архітектури аналогічні багатобазовій, а, крім того, СУБД, як правило, не має інформації про всі створені БД по всій структурі каталогів. До того ж тут немає головної БД, що відстежує всі інші, що робить дуже важким централізоване адміністрування цієї системи.

Каталогова архітектура ще більше ускладнює доступ до даних, оскільки для цього необхідно вказати не тільки ім'я БД, але і її місце знаходження в ієрархічній структурі каталогів. Проте СУБД з каталоговою архітектурою, забезпечують доступ до декількох БД одночасно.

З вищесказаного виходить, що існує величезна різноманітність в тому, як в різних СУБД організовані БД і яким чином до даних здійснюється доступ. Ця область мови SQL є однією з самих нестандартних і, іноді, така неузгодженість між різними СУБД робить неможливим або дуже складним перенесення даних з однієї системи в іншу.

Таким чином, найчастіше таблиці і інші об'єкти даних зберігаються в БД і знаходяться там пов'язаними з певними користувачами, які ними володіють. В деякому розумінні, можна сказати, що вони зберігаються в іменній області користувача, хоча це ніяким чином не відображає їх фізичного розташування, та зате забезпечує строгу логічну конструкцію зберігання.

Проте не треба забувати і той факт, що об'єкти даних на пристроях зберігання займають

деякий об'єм фізичної пам'яті і її кількість, яка може використовуватися певним об'єктом або користувачем в даний момент, має свою межу. Дійсно, жоден комп'ютер не може мати прямого доступу до нескінченного числа апаратних засобів для зберігання даних. Тому у великих SQL системах БД розділяється на області, які прийнято називати областями БД (DBS) або розділами.

Це області інформації, що зберігається, які розміщені особливим чином так, щоб дані всередині них перебували поблизу одні до одних для виконання команд, тобто СУБД не повинна розташовувати інформацію десь далеко. Системи, які використовують області БД, дозволяють користувачу обробляти їх за допомогою команд SQL як об'єкти.

Областями БД маніпулюють за допомогою команд CREATE DBSPACE (створити), ACQUIRE DBSPACE (отримати) або CREATE TABLESPACE (створити табличну область), залежно від СУД, що використовується. Одна область може вміщати в себе будь-яке число користувачів, а конкретний користувач може мати доступ до багатьох областей. Привілея створювати таблиці для користувача може бути передана по всій БД, але частіше розповсюджується тільки на конкретну область.

Наприклад, для створення області зберігання БД EDUCATION з нашими прикладами, можна скористатися наступною командою:

```
CREATE DBSPACE EDUCATION
     (PCTINDEX 15,
     PCTFRFE 20);
```

Тут параметр PCTINDEX встановлює, який відсоток області БД повинен бути залишений, щоб зберігати в ньому індекси таблиць, а ключове слово PCTFREE задає відсоток області, який резервується для того, щоб дозволити таблицям розширювати величини своїх записів, наприклад, при збільшенні командою ALTER TABLE розміру полів.

До речі кажучи, більшість СУБД, виходячи з наявного апаратного забезпечення, вдало підбирають параметри області БД автоматично, тому досить часто при створенні використовують параметри за замовчуванням. Корисною властивістю області є можливість встановити певне обмеження розміру БД, або, навпаки, дозволити їй зростати в розмірах необмежено разом зі всіма таблицями.

При розділенні використовуваної БД на області необхідно мати на увазі типи операцій, які часто виконуватимуться. При цьому бажано, щоб таблиці, які часто об'єднуватимуться або які мають зовнішні ключі, знаходилися разом в одній області.

2.5. Спеціальні аспекти роботи з базами даних

2.5.1. Контроль цілісності даних з використанням тригерів

Вище вже не раз піднімалися питання про зв'язки, які існують між деякими полями таблиць в БД. Наприклад, поле SNUM таблиці STUDENTS відповідає полю SNUM в таблиці USP, а поле PNUM таблиці PREDMET – відповідно PNUM в тій же таблиці USP. Такий зв'язок можна назвати довідковою цілісністю – вона необхідна для того, щоб уникнути надмірності інформації в таблицях. Розглянемо використання цієї можливості детальніше.

Нагадаємо, що коли всі значення в одному полі таблиці представлені в полі іншої, звичайно говорять, що перше поле посилається на друге, і це вказує на прямий зв'язок між значеннями двох полів. Наприклад, кожен запис з інформацією про студента в таблиці STUDENTS має поле SNUM, яке вказує на оцінку, що зберігається в таблиці успішності USP.

Коли одне поле в таблиці посилається на інше, воно називається зовнішнім ключем, а поле, на яке воно посилається батьківським ключем. Таким чином, поле SNUM таблиці STUDENTS є батьківським ключем, а поле SNUM в таблиці USP – зовнішнім ключем. Імена зовнішнього ключа і батьківського ключа не обов'язково повинні бути однаковими, проте

дотримання умови ідентичності імен робить з'єднання зрозумілішим.

Крім того, зовнішній ключ не обов'язково повинен складатися тільки з одного поля, хоча ключі, що складаються з одного поля, зустрічаються найчастіше. Подібно до первинного ключа, зовнішній ключ може мати будь-яке число полів, які всі разом обробляються як єдине ціле. Зовнішній ключ і батьківський ключ, на який він посилається, обов'язково повинні мати однаковий тип поля, і, при використанні декількох полів, знаходитися в однаковому порядку.

Коли поле є зовнішнім ключем, очевидно, що воно певним чином пов'язане з таблицею, на яку посилається. Фактично відбувається наступне: кожному значенню в полі, яке є зовнішнім ключем, безпосередньо відповідає значення в іншому полі – батьківському ключі. При цьому кожне значення зовнішнього ключа повинне однозначно посилатися до одного значення батьківського ключа, тоді можна говорити про те, що система перебуває в стані довідкової цілісності.

Якщо це не так, то виникає неоднозначна ситуація. Наприклад, якщо в таблиці STUDENTS в полі SNUM з'являться два однакові значення (скажімо, 3412), то при отриманні результатів запиту про отримані оцінки

```
SELECT STUDENTS.SPRIZ, STUDENTS.SIMA, STUDENTS.SBAT,
          USP.OCINKA
     FROM STUDENTS, USP
     WHERE STUDENTS.SNUM = USP.SNUM
     AND STUDENTS.SNUM = 3412;
```

буде неясно, якому саме студенту ця оцінка належить. З іншого боку, якщо в таблиці успішності USP з'явиться запис із значенням поля SNUM = 3417, яке відсутнє в таблиці STUDENTS, то в результаті запиту

```
SELECT STUDENTS.SPRIZ, STUDENTS.SIMA, STUDENTS.SBAT,
          USP.OCINKA
     FROM USP.STUDENTS
     WHERE USP.SNUM = STUDENTS.SNUM
     AND USP.SNUM = 3417;
```

можна отримати список оцінок, які не належать жодному студенту. Очевидно, що такі ситуації будуть виключені, якщо кожне значення в зовнішньому ключі буде представлено тільки один раз в батьківському ключі, і це значення повинне існувати.

SQL автоматично підтримує довідкову цілісність даних з обмеженням FOREIGN KEY. Ця функція обмежує значення, які можна ввести до БД так, щоб примусити зовнішній ключ і батьківський ключ відповідати принципу довідкової цілісності. Одна з дій цього обмеження – відкидання значень для полів, обмежених як зовнішній ключ, які відсутні в батьківському ключі. До того ж FOREIGN KEY також впливає на можливість користувача змінювати або видаляти значення батьківського ключа.

Обмеження FOREIGN KEY використовується в командах CREATE TABLE або ALTER TABLE, які оголошують зовнішнім ключем відповідне поле таблиці, при цьому дається ім'я батьківському ключу, на яке передбачається посилання всередині цього обмеження. FOREIGN KEY використовується аналогічно тому, як це реалізовувалося для інших типів обмежень, розглянутих вище: воно може бути обмеженням поля або таблиці, причому в останньому випадку – дозволяє використовувати декілька полів як однин зовнішній ключ.

Синтаксис обмеження таблиці FOREIGN KEY наступний:

```
FOREIGN KEY <COLUMN LIST> REFERENCES
     <PKTABLE> [<COLUMN LIST>]
```

Тут перший список полів – це перелік розділених комами з одного або більше полів створюваної або такої, що модифікується таблиці. PKTABLE – це таблиця, що містить батьківський ключ. Другий список полів – це поля, які складатимуть батьківський ключ.

Списки двох полів повинні бути сумісні, тобто повинні виконуватися наступні умови:

- списки повинні містити однакову кількість полів;
- у тій послідовності, в якій йде перелік, повинна дотримуватися ідентичність типу даних і розмір кожного поля.

Для прикладу створимо таблицю успішності USP з полем SNUM, означеним як зовнішній ключ, що посилається на таблицю студентів STUDENTS:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER,
     UDATE DATE,
     SNUM INTEGER NOT NULL,
     PNUM INTEGER NOT NULL,
     FOREIGN KEY (SNUM)
          REFERENCES STUDENTS (SNUM));
```

Зверніть увагу на те, що при використанні ALTER TABLE замість CREATE TABLE для реалізації обмеження FOREIGN KEY значення, які вказуються в зовнішньому і батьківському ключах, повинні бути в стані довідкової цілісності. Інакше команда буде відхилена. Звідси випливає, що використання команди ALTER TABLE приводить до необхідності кожного разу стежити за дотриманням структурних принципів довідкової цілісності.

Варіант обмеження поля FOREIGN KEY часто називають посилальним обмеженням REFERENCES, оскільки воно фактично не містить в собі ключових слів FOREIGN KEY, а просто використовує слово REFERENCES, як це показано в наступному прикладі:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER,
     UDATE DATE,
     SNUM INTEGER NOT NULL
          REFERENCES STUDENTS (SNUM),
     PNUM INTEGER NOT NULL);
```

В даному випадку USP.SNUM визначається як зовнішній ключ, у якого батьківським є STUDENTS.SNUM.

Взагалі кажучи, використовуючи обмеження FOREIGN KEY таблиці або поля, можна не вказувати список полів батьківського ключа, якщо батьківський ключ має обмеження PRIMARY KEY. При цьому, у разі використання ключів з багатьма полями, обов'язкове виконання умови, щоб порядок полів в зовнішніх і первинних ключах співпадав.

Наприклад, якщо в таблиці STUDENTS поле SNUM є PRIMARY KEY, то його можна використовувати як зовнішній ключ в таблиці успішності аналогічно попередньому прикладу таким чином:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER,
     UDATE DATE,
```

```
    SNUM INTEGER NOT NULL
         REFERENCES STUDENTS,
    PNUM INTEGER NOT NULL);
```

Підтримка довідкової цілісності вимагає обмежень і на значення, які можуть бути присутніми в полях, оголошених як зовнішній ключ і батьківський ключ. Це означає, зокрема, що батьківський ключ повинен бути унікальним і не містити NULL значень. З врахуванням цього, при оголошенні FOREIGN KEY, SQL повинна бути впевненою, що подвійні або NULL значення не присутні в батьківському ключі. Отже, користувач повинен переконатися в тому, що всі поля, які використовуються як батьківські ключі, мають обмеження PRIMARY KEY або UNIQUE і NOT NULL.

Рекомендованою стратегією при створенні структури БД є посилання зовнішніх ключів таблиць тільки на первинні ключі. Коли використовуються зовнішні ключі, відбувається зв'язок не просто з батьківськими ключами, на які вони посилаються – зв'язуються певні записи таблиць. Дійсно, сам по собі батьківський ключ не забезпечує жодної інформації, яка б не була б вже представлена в зовнішньому ключі. Наприклад, зміст поля SNUM в ролі зовнішнього ключа полягає в тому, що він забезпечує зв'язок, але не до значень полів SNUM, на які він посилається, а до іншої інформації, що знаходиться в тому ж записі, наприклад, до прізвища студента.

Оскільки основним завданням первинного ключа є ідентифікація унікального запису таблиці, то це – найлогічніший і менш неоднозначний вибір для зовнішнього ключа. Зовнішній ключ, який не має ніякої іншої мети, крім зв'язування рядків таблиць, в цьому змісті нагадує первинний ключ, що використовується виключно для ідентифікації рядків, і є добрим засобом для збереження грамотної і простої структури БД.

З вищевикладеного логічно витікає, що зовнішній ключ, взагалі кажучи, повинен містити тільки ті значення, які фактично є в батьківському ключі, або NULL значення. Спроба ввести інші дані в цей ключ буде відхилена системою.

Якщо оголосити зовнішній ключ, як NOT NULL, то від невизначених значень в ньому можна позбавитися, проте це необов'язково, а іноді небажано. Це може відбутися за ситуації, коли, наприклад, вводиться оцінка, а номер студентського квитка невідомий заздалегідь.

Нагадаємо, що як зовнішній ключ може виступати комбінація з декількох полів. Щоб створити такий зовнішній ключ, необхідно пам'ятати про те, що поєднання цих полів в батьківській таблиці повинне відповідати вимозі обмеження UNIQUE. Створення зовнішнього ключа у такий спосіб підтримує цілісність БД, навіть якщо при цьому користувачу буде заборонене внутрішнє переривання (виключення) цілісності БД.

З погляду реалізації цілісності БД внутрішні переривання украй небажані, і, якщо вони в системі допускаються, то часто цілісність даних доведеться підтримувати вручну.

Розглянуті обмеження впливають на можливість (або неможливість) виконання команд модифікації даних. Для полів, визначених як зовнішні ключі, будь-які значення, що поміщаються командами INSERT або UPDATE, повинні бути вже представлені в їх батьківських ключах. Більше того, допускається вставляти в ці поля NULL значення, не дивлячись на те, що вони не дозволені в батьківських ключах. Крім цього, можна видаляти командою DELETE будь-які записи із зовнішніми ключами, не використовуючи батьківські ключі взагалі.

Відповідно до стандарту, зміна або видалення значень батьківського ключа взагалі не допускається. Це, наприклад, означає, що не можна видалити дані про студента з таблиці STUDENTS до тих пір, поки в таблиці успішності USP для нього є яка-небудь інформація. З одного боку, це є позитивним моментом – неможлива ситуація, коли з'являться оцінки, що не належать жодному студенту. З іншого боку, досить часто виникає необхідність в повному видаленні інформації, наприклад, у разі відрахування студента з вузу. Про ці можливості ми вже згадували вище, говорячи про каскадування і обмеження дій.

Отже, поведінка системи при зміні батьківського ключа може відрізнятися від передбаченої стандартом. Якщо необхідно змінити або видалити поточне посилальне значення батьківського ключа, то є три можливості:

- заборонити зміни, оголосивши, що зміни в батьківському ключі обмежені;
- зробивши зміни в батьківському ключі, провести зміни і в зовнішньому ключі автоматично, що фактично є каскадною зміною;
- зробити зміну в батьківському ключі і встановити зовнішній ключ в NULL значення автоматично, що називається порожньою зміною зовнішнього ключа.

В межах цих можливостей можна обробляти всі команди модифікації. З командою INSERT, правда, в цьому випадку ситуація дещо складніша, оскільки при її використанні поміщаються нові значення батьківського ключа в таблицю. Таким чином, те, що трапиться в батьківському ключі, можна розділити на обмежені (RESTRICTED), каскадні (CASCADES) і порожні (NULL) зміни.

Для прикладу розглянемо те, як ці можливості застосовуються в таблицях. Припустимо, що є необхідність в зміні номера студентського квитка, причому оцінки повинні зберегтися у цього студента з новим номером. Якщо ж дані про студента видаляються, то необхідно, щоб його оцінки в таблиці успішності залишалися, скажімо, для подальшого звіту. Щоб це реалізувати, необхідно вказати умову UPDATE з каскадним, а DELETE з обмеженим ефектом, тобто:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER,
     UDATE DATE,
     SNUM INTEGER NOT NULL
          REFERENCES STUDENTS
     PNUM INTEGER NOT NULL,
     UPDATE OF STUDENTS CASCADES,
     DELETE OF STUDENTS RESTRICTED);
```

Після цього, при видаленні даних про студента з таблиці STUDENTS, команда не буде виконана до тих пір, поки не будуть видалені його оцінки або не зміниться значення поля SNUM. З іншого боку, якщо номер студентського квитка буде змінений, то для відповідних оцінок цього студента значення поля SNUM також автоматично поміняється.

Для демонстрації NULL змін скористаємося таким прикладом:

```
CREATE TABLE USP
     (UNUM INTEGER NOT NULL PRIMARY KEY,
     OCINKA INTEGER,
     UDATE DATE,
     SNUM INTEGER
          REFERENCES STUDENTS,
     PNUM INTEGER NOT NULL,
     DELETE OF STUDENTS NULLS);
```

В цьому випадку видалення інформації про студента з таблиці STUDENTS пройде успішно, а записи, що відповідають його оцінкам, матимуть в полі SNUM NULL значення, що може стати в нагоді, скажімо, для розрахунку середньої оцінки по навчальних предметах за семестр. Зрозуміло, за наявності в таблиці DELETE з NULL ефектом обмеження NOT NULL не повинне бути в полі SNUM.

Говорячи про зовнішні ключі, ми вже згадували про можливість їх посилання на ту ж

таблицю. Інакше кажучи, батьківська і зовнішня таблиця в даному випадку – одне і те ж.

Ця особливість обмеження FOREIGN KEY може стати в нагоді, наприклад, в наступному випадку. Нехай структура таблиці STUDENTS буде дещо змінена (див. табл. 2.7). Назвемо отриману таблицю STUDENTS2.

Таблиця 2.7 Модифікована таблиця STUDENTS.

STUDENTS2					
SNUM	SPRIZ	SIMA	SBAT	STYP	STAR
3412	Поляченко	Анатолій	Олексійович	грн.250,50	3414
3413	Старченко	Любов	Михайлівна	грн.170,00	3414
3414	Грищенко	Володимир	Миколайович	грн.0,00	3416
3415	Котенко	Анатолій	Миколайович	грн.0,00	3414
3416	Нагірний	Євген	Васильович	грн.250,00	NULL

У таблиці STUDENTS2 появилося поле STAR, яке містить номер студентського квитка старости. У нашому випадку студент Нагірний є старостою всього потоку, йому підпорядковується староста Грищенко, якому, у свою чергу, підпорядковуються всі інші. Очевидно, оскільки староста сам є студентом, то він теж представлений в цій таблиці.

Спробуємо створити таку таблицю, причому номер студентського квитка в полі SNUM буде первинним ключем, а на нього посилатиметься зовнішній ключ, у якості якого оголосимо поле STAR:

```
CREATE TABLE STUDENTS2
     (SNUM INTEGER NOT NULL PRIMARY KEY,
     SPRIZ CHAR (20) NOT NULL,
     SIMA CHAR (10),
     SBAT CHAR (15),
     STAR INTEGER REFERENCES STUDENTS2);
```

З цього прикладу видно, що кожний із студентів посилається на іншого як на свого старосту, проте Нагірний, як самий старший, повинен мати в полі STAR NULL значення, тобто це поле повинно допускати неозначені значення. Це змушує зробити принцип довідкової цілісності – в іншому випадку в таку таблицю не можна було б додати жодного запису. Навіть якщо староста посилався б на самого себе, у момент вставки найпершого запису в таблицю студент з таким номером повинен вже існувати, інакше команда вставки була б відхилена. Більше того, якщо зовнішній ключ таблиці посилається на саму себе не безпосередньо, а через посилання до іншої таблиці (наприклад, зовнішнім ключем у STUDENTS2 була б таблиця USP, а у неї, у свою чергу – таблиця STUDENTS2), цей принцип повинен бути збережений.

Такий підхід називається перехресним посиланням. SQL підтримує цю можливість, але практично це може викликати труднощі, оскільки будь-яка таблиця з цих двох, створена попередньою, є таблицею-зсилкою, для ще не існуючої іншої таблиці. Рекомендується бути дуже обережним і акуратним при використанні перехресних посилань, оскільки механізм модифікації і видалення даних може пошкодити інформацію, що зберігається в БД.

2.5.2. Засоби обробки транзакцій

Очевидно, що БД – це структура, що постійно змінюється під впливом користувачів і їх команд. Дотепер передбачалося, що ця система правильно розроблена і функціонує без жодних збоїв. Проте в реальному житті, через помилки людей, що взаємодіють з СУБД, або через нестійку роботу комп'ютерів виникають збої і помилки і в самій БД. Тому в СУБД застосовують спеціальні способи відміни дій, що викликали такі помилки – це, в першу чергу, транзакції, про які вже йшлося. Команда SQL, яка чинить дію на зміст або структуру БД, не обов'язково буде

необоротною – користувач може встановити, що відбудеться після закінчення її дії: чи залишаться внесені зміни в БД, або вони будуть повністю проігноровані.

Транзакція починається всякий раз, коли відбувається сеанс роботи з SQL. Всі команди, які вводяться, будуть частиною цієї транзакції, поки вона не завершиться введенням команди COMMIT WORK або команди ROLLBACK WORK. COMMIT робить всі зміни, проведені транзакцією, постійними, а ROLLBACK може їх відмінити. Нова транзакція починається після кожної команди COMMIT або ROLLBACK.

Синтаксис цих команд дуже простій: щоб залишити всі зміни постійними використовують

```
COMMIT WORK;
```

а для відміни зміни

```
ROLLBACK WORK;
```

В більшості випадків можна встановити параметр, що називається AUTOCOMMIT, який автоматично запам'ятовуватиме всі виконувані команди, причому дії, які привели до помилки, завжди будуть автоматично скасовані. Звичайно цей режим встановлюється за допомогою команди типу:

```
SET AUTOCOMMIT ON;
```

а повернення до звичайної діалогової обробки запитів – за допомогою команди

```
SET AUTOCOMMIT OFF;
```

Крім того, є можливість установки AUTOCOMMIT, яку СУБД виконає автоматично при реєстрації. Якщо сеанс користувача завершується аварійно – наприклад, відбувся збій системи або виконане перезавантаження користувача, – то поточна транзакція виконає автоматичний відкат змін. Це – одна з причин, по якій варто управляти виконанням діалогової обробки запитів, розділивши команди на різні транзакції.

Не рекомендується організовувати роботу так, щоб одиночні транзакції містили багато команд, тим більше не зв'язаних між собою. Це може привести до того, що при відміні змін буде виконано дуже багато дії, у тому числі і тих, які є потрібними і помилки не викликали. Кращим варіантом є робити транзакції такими, що складаються з однієї команди або декількох тісно зв'язаних між собою команд.

Наприклад, виникла необхідність у видаленні даних про студента Поляченко з таблиці STUDENTS. Очевидно, що при цьому потрібно що-небудь зробити з його оцінками за умови, що обмеження зовнішнього ключа відсутнє. Логічне розв'язання полягатиме в тому, щоб встановити поле SNUM для записів з його оцінками, в NULL значення і лише після цього виконувати видалення. Це можна реалізувати таким чином:

```
UPDATE USP
     SET SNUM = NULL
     WHERE SNUM = 3412;

DELETE FROM STUDENTS
     WHERE SNUM = 3412;
```

Якщо при цьому виникають які-небудь проблеми, то у користувача є можливість

відмінити всі зміни командою ROLLBACK. У випадку, якщо все гаразд, і дію цієї групи команд на БД необхідно запам'ятати, то це реалізується командою COMMIT.

У стандарті ANSI/ISO визначена модель транзакцій, а також вказані завдання операторів COMMIT і ROLLBACK. Згідно стандарту вважається, що транзакція автоматично починається з виконання користувачем або програмою першого оператора SQL. Далі відбувається послідовне виконання решти операторів SQL до тих пір, поки транзакція не завершиться одним з чотирьох способів:

- команда COMMIT завершує виконання поточної транзакції, причому зміни, внесені в БД, стають постійними. Нова транзакція починається безпосередньо після COMMIT;
- команда ROLLBACK відміняє виконання поточної транзакції, зроблені зміни відміняються, а нова транзакція починається безпосередньо після ROLLBACK;
- успішне завершення програми обробки даних вважається успішним закінченням транзакції, начебто була виконана команда COMMIT. Нова транзакція не починається, оскільки програма закінчилася;
- неуспішне завершення програми вважається неуспішним закінченням транзакції, начебто була виконана команда ROLLBACK. Нова транзакція не починається, оскільки програма закінчилася.

Зверніть увагу на те, що згідно стандартної моделі транзакцій, користувач або програма можуть виконати транзакцію завжди. Традиційно, при інтерактивному використанні SQL включений режим AUTOCOMMIT, тобто кожна команда стає окремою транзакцією.

У СУБД PostgreSQL та SQL Server використовується дещо розширена модель транзакцій, що надає користувачам додаткові можливості. При цьому використовуються команди:

- BEGIN TRANSACTION – повідомляє систему про початок транзакції. На відміну від стандартної моделі, початок транзакції задається явно за допомогою цієї команди;
- COMMIT TRANSACTION – повідомляє систему про успішне закінчення транзакції. Після виконання цієї команди всі зміни, зроблені в БД протягом транзакції, стають постійними, проте нова транзакція не починається;
- SAVE TRANSACTION – створює всередині транзакції точку збереження. СУБД зберігає стан БД в поточній точці і привласнює збереженому стану ім'я точки збереження, яке вказується в операторі;
- ROLLBACK TO SAVEPOINT – відміняє зміни, зроблені в БД після точки збереження, повертаючи транзакцію до місця, де був виконаний оператор SAVE TRANSACTION;
- ROLLBACK – скасовує всі зміни, зроблені в БД після оператора BEGIN TRANSACTION.

Введені точки збереження особливо корисні в складних транзакціях, що складаються з великої кількості команд. В процесі виконання транзакції програма може періодично зберігати стан БД, створюючи іменовані точки збереження. Якщо під час виконання виникають які-небудь проблеми, програма має можливість відмінити не всю транзакцію цілком, а тільки її частину до точки збереження, відновивши своє виконання транзакції з цього місця. При цьому всі оператори, виконані до точки збереження, залишаються в силі, а виконані після точки збереження відміняються. Проте логічною одиницею роботи все-таки є ціла транзакція, тому, якщо під час виконання транзакції відбувається системний або апаратний збій, то відміняється ціла транзакція.

Реалізація механізму транзакцій в СУБД базується на використанні журналу транзакцій. При виконанні користувачем команди на зміну БД система автоматично вносить в журнал транзакцій інформацію про те, що і яким чином було модифіковано даною командою. І лише після того, як в журналі буде зроблений запис, СУБД змінить фізичний запис на пристрої зберігання. Після цього, при виконанні команди COMMIT, в журналі оголошується кінець транзакції.

Якщо користувач виконує команду ROLLBACK, СУБД звертається до журналу і витягує

з нього копії модифікованих під час транзакції даних. Використовуючи їх, система повертає БД в колишній стан і, таким чином, відміняє внесені зміни. У разі системного збою адміністратор БД відновлює дані по журналу транзакцій, відшукуючи транзакції, які не були завершені до моменту збою.

Використання журналу транзакцій має і недоліки: в результаті його використання тривалість операцій зміни БД збільшується, тому іноді роботу журналу припиняють Очевидно, цю дію не можна віднести до розряду бажаних, оскільки у разі збою або виконання ROLLBACK відновити первинний стан БД не можна.

2.5.3. Методи блокування

Як правило, SQL використовується в розрахованих на багато користувачів системах, де дуже часто виконується одночасно відразу багато маніпуляцій з БД. Це створює потенційну можливість конфлікту між різними діями, що виконуються.

Припустимо, що перший користувач виконує команду:

```
UPDATE USP
     SET OCINKA =3
     WHERE PNUM = 2003;
```

яка модифікує дані в таблиці успішності. В той же час другий користувач робить запит:

```
SELECT SNUM, MIN (OCINKA)
     FROM USP
     GROUP BY SNUM;
```

вибираючи з таблиці мінімальну оцінку по кожному студенту. Очевидно, що в другому випадку для змінних даних важливо, щоб як результат був виведений або кінцевий результат, або не виводилося нічого – адже будь-який проміжний результат є випадковим або непередбачуваним. З іншого боку, якщо мінімальне значення буде виведено з врахуванням модифікації, зробленої першим користувачем, який пізніше свої дії відмінить, то другий користувач отримає неточні результати, оскільки він про відміну може і не знати. Зрозуміло, що рішення подібних задач СУБД повинна забезпечувати на найвищому рівні.

Як було сказано вище, обробка системою транзакцій, що виконуються одночасно, називається паралелізмом. При цьому можуть виникати наступні типи проблем:

- зміна даних може бути здійснена без врахування модифікації, зробленої іншим процесом. Така ситуація може виникнути, якщо в таблиці успішності відбувається коректування оцінок, а паралельно, іншим користувачем, відбувається встановлення розміру стипендії, виходячи з отриманих студентами оцінок;
- зміни даних можуть бути скасовані одним процесом, тоді як модифікованими даними вже скористалися для іншої дії. Цей випадок вже проілюстрований вище, коли мінімальне значення оцінок могло бути вже отримане, після чого перший користувач проведені зміни оцінок відмінив;
- одна дія може частково впливати на результат іншої дії. Наприклад, при отриманні звіту про успішність не виключено, що іншим користувачем проведене видалення із списку одного або декількох студентів, а дані про їх успішність вже можуть увійти до виведення першого запиту.
- тупикова ситуація, яка може відбутися, якщо процеси спробують виконати конфліктуючі одна з одною дії. Це може виникнути, наприклад, якщо один користувач пробує змінити значення зовнішнього ключа, а інший – батьківського ключа, і це відбувається одночасно.

Для виключення подібних ситуацій в SQL присутній засіб управління паралелізмом, який вказує процесам точне місце отримання результату. Основний принцип при цьому наступний: всі одночасно виконувані команди не будуть завершені до тих пір, поки не будуть завершені попередні. Іншими словами, система повинна забезпечувати одночасний доступ до таблиці не більше ніж для однієї транзакції в даний момент часу. Правда, в системах, де потрібно забезпечити доступ до БД дуже великій кількості користувачів, управління паралелізмом здійснюється по дещо видозміненій схемі, дозволяючи адміністратору БД і користувачам самим регулювати ступінь узгодженості і доступності даних.

Механізм, що використовується SQL для управління паралелізмом операцій, як ми вже знаємо, називається блокуванням. Блокування затримують певні операції в БД, поки інші операції або транзакції не завершені. Затримані операції поміщаються в чергу і виконуються тільки тоді, коли блокування зняте, а іноді не відхиляються системою, видаючи при цьому відповідне попередження.

Блокування в розрахованих на багато користувачів системах виконуються по спеціальних схемах, що використовуються до всіх команд в БД, і, крім того, забезпечені засобами виявлення блокувань, блокуючих один одного. В цьому випадку, одна з команд скасовується, а отже, отримує блокування.

Розрізняють два базових типи блокувань:

- розподільні (нежорсткі) блокування – можуть бути встановлені більш ніж одним користувачем в даний момент часу, що дає можливість будь-якому числу процесів звертатися до даних, але не змінювати їх;
- спеціальні (жорсткі) блокування – не дозволяють нікому взагалі, крім власника цього блокування, звертатися до даних. Спеціальні блокування використовуються, наприклад, для команд, які змінюють зміст або структуру таблиці і діють до кінця транзакції.

У механізмі реалізації блокувань використовується поняття рівня ізоляції блокування, що визначає, скільки таблиць буде блоковано. Традиційно використовують три рівні ізоляції, два з яких можна застосувати до розподілених і спеціальних блокувань, а третій, так званий обмежений рівень, служить для того, щоб використовувати ці блокування спільно. Рівні ізоляції управляються командами, поданими самою SQL.

Рівень ізоляції, що називається повторне читання, реалізує таку стратегію, що всередині даної транзакції всі записи, витягнуті за допомогою запитів, не можуть бути змінені. Оскільки записи, що модифікуються в транзакції, піддаються спеціальному блокуванню, вони не можуть бути змінені до тих пір, поки транзакція не завершена. З іншого боку, для запитів повторне читання означає те, що можна вирішити наперед, які записи будуть модифіковані, і заблокувати той процес, якому необхідно їх вибрати. Таким чином, при виконанні запиту гарантується, що жодні зміни не будуть зроблені в цих записах до тих пір, поки не завершиться поточна транзакція. Проте повторне читання, що захищає процес, який встановив блокування, в той же час може значною мірою знизити продуктивність роботи з даними.

Рівень ізоляції, який називають покажчик стабільності, оберігає кожен запис від змін на якийсь час, коли вона прочитується, або від читання на час її зміни. У другому варіанті це є спеціальним блокуванням і застосовується до тих пір, поки зміна не буде завершена або скасована. Отже, при модифікації групи записів, що використовують покажчик стабільності, ці записи будуть заблоковані, доки транзакція не закінчиться, що аналогічно дії, що виробляється рівнем повторного читання. Відмінність між цими двома рівнями в їх дії на запити. У разі рівня покажчик стабільності, записи таблиці, які зараз не використовуються запитом, можуть бути змінені.

Третій рівень ізоляції називається тільки читання. Тільки читання блокує всю таблицю, а отже, не може використовуватися з командами модифікації. Проте будь-яка частина таблиці у момент виконання команди може бути відображена у виводі запиту. Таким чином, тільки читання гарантує, що вивід запиту буде внутрішньо узгоджений з даними таблиці. З цієї

причини цей рівень зручний тоді, коли робляться звіти за даними таблиці, вирішуючи одночасний доступ до більшості або до всіх рядків таблиці відразу декільком процесам, що не проводять модифікації БД.

Деякі СУБД виконують блокування сторінки пам'яті, в якій зберігається фрагмент БД, замість блокування запису. Сторінка може складатися з одного або більше рядків таблиці, причому там же можуть бути індекси і інша службова інформація. Якщо блокуються сторінки замість записів, використовуються ті ж механізми, що описані вище. Основною перевагою такого підходу є його ефективність: SQL не стежить за блокуванням кожного запису індивідуально, тобто вона працює швидше. Проте при цьому можуть бути заблоковані рядки таблиці, для яких це робити в даний момент зовсім необов'язково.

Отже, засоби управління паралелізмом в СУБД визначають те, в якій мірі одночасно подані команди заважатимуть одна одній. У сучасних СУБД вони є засобами, що адаптуються, і автоматично знаходять краще розв'язання з врахуванням забезпечення максимальної продуктивності БД і доступності даних для діючих команд.

2.6 Уявлення

2.6.1. Що таке уявлення

Уявлення, або VIEW, – деяка подоба таблиці, вміст якої вибирається з інших таблиць за допомогою виконання запиту, причому, при зміні значень в цих таблицях, дані автоматично міняються і в уявленні. З цієї причини часто уявлення ефективніші, ніж звичайні запити.

Таблиці, що розглядалися до цього, відносилися до базових, тобто таких, які містять дані і постійно знаходяться на пристроях зберігання інформації. Оскільки дані для створення уявлень вибираються з інших таблиць, то вони не містять ніяких власних значень, проте часто уявлення є гнучкішим засобом, оскільки з їх допомогою інформація може бути показана користувачу в найбільш зручному для нього варіанті.

Отже, уявлення – це фактично той же запит, який виконується всякий раз, коли уявлення бере участь в якій-небудь команді. Вивід цього запиту при цьому в кожен момент часу стає вмістом уявлення.

Коли СУБД зустрічає в команді посилання на уявлення, вона відшукує його означення, що знаходиться в БД. Після цього відбувається перетворення призначеної для користувача команди в її еквівалент з врахуванням того, який запит лежить в основі уявлення, що використовується. Отже, у користувача створюється враження ніби він працює із справжньою, реально існуючою таблицею.

При цьому у СУБД є дві можливості реалізації уявлення. Якщо його означення просте, то система формує кожен запис уявлення в міру необхідності, поступово прочитуючи початкові дані з базових таблиць. У випадку, якщо означення складне, то СУБД доводиться спочатку виконати таку операцію, як матеріалізація уявлення, тобто збереження інформації, з якої складається уявлення в тимчасовій таблиці. Потім система приступає до виконання призначеної для користувача команди і формування її результатів, а після цього тимчасова таблиця видаляється.

2.6.2.Створення, видалення і оновлення уявлень

Уявлення створюється командою CREATE VIEW, після якої вказується його ім'я, а далі слідує запит, що формує тіло уявлення. Наприклад, для створення представлення OTLSTUD, яке містить інформацію про студентів, що отримують стипендію у розмірі 250.50 грн., можна скористатися наступною командою:

```
CREATE VIEW OTLSTUD
```

```
AS   SELECT *
     FROM STUDENTS
     WHERE STYP='250,50';
```

Тепер в БД існує уявлення з ім'ям OTLSTUD, яке можна використовувати в командах так само, як і будь-яку іншу таблицю: вона може бути запрошена, модифікована, в неї можуть бути вставлені записи, вона може бути видалена з БД або з'єднана з іншими таблицями і уявленнями.

При виконанні запиту:

```
SELECT *
     FROM OTLSTUD;
```

буде отриманий наступний вивід:

```
SNUM      SPRIZ          SIMA       SBAT           STYP
3412      Поляченко      Анатолій   Олексійович    грн.250,50
```

Очевидно, що до уявлення можна зробити запит, що містить предикат, що, відповідно, приведе до виводу тільки задовольняючих йому рядків.

Перевага використання уявлень в порівнянні із запитами до основної таблиці, як вже було сказано, в тому, що уявлення буде модифіковано автоматично кожен раз, коли таблиця, що лежить в його основі, змінюється. Вміст уявлення не фіксований, і оновлюється кожного разу. коли на нього посилаються в команді. Якщо, наприклад, в таблиці STUDENTS з'явиться ще один студент із стипендією 250,50 грн., то він автоматично відобразиться в уявленні.

Уявлення значно розширюють можливості управління даними. Зокрема, це чудовий спосіб дати доступ до інформації в таблиці, приховавши частину даних. Наприклад, якщо необхідно приховати розмір стипендії студента від користувача, можна створити уявлення:

```
CREATE VIEW STYPOFF
     AS SELECT SNUM, SPRIZ, SIMA, SBAT
     FROM STUDENTS;
```

При виконанні запиту

```
SELECT *
     FROM STYPOFF;
```

буде виведений весь вміст створеного уявлення:

```
SNUM      SPRIZ          SIMA          SBAT
3412      Поляченко      Анатолій      Олексійович
3413      Старченко      Любов         Михайлівна
3414      Грищенко       Володимир     Миколайович
3415      Котенко        Анатолій      Миколайович
3416      Нагірний       Євген         Васильович
```

Інакше кажучи, уявлення STYPOFF фактично та ж таблиця STUDENTS, але без поля STYP.

Уявлення може тепер змінюватися командами модифікації DML, але модифікація не впливатиме на саме уявлення. Фактично команди будуть перенаправлені до базової таблиці. Наприклад, виконання команди:

```
UPDATE STYPOFF
     SET SIMA = 'Василь'
     WHERE SNUM = 3415;
```

аналогічно виконанню тієї ж команди для таблиці студентів STUDENTS. Проте наступна команда буде знехтувана системою:

```
UPDATE STYPOFF
     SET STYP = '250,50'
     WHERE SNUM = 3415;
```

Це пов'язано з тим, що поле STYP відсутнє в уявленні STYPOFF. І ще одне важливе зауваження: не всі уявлення можуть бути модифіковані, про що піде розмова трохи нижче.

У розглянутих прикладах поля уявлень мають свої імена, отримані безпосередньо з імен полів основної таблиці. Проте іноді виникає необхідність назвати стовпці новими іменами. Це. наприклад, може бути потрібно у випадку, якщо стовпці, що виводяться, не мають імен, або якщо два або більше стовпців в об'єднанні мають ті ж імена, що в їх базовій таблиці.

Імена, які необхідно привласнити полям, записуються в круглих дужках після імені таблиць. Вони можуть не вказуватися, якщо співпадають з іменами полів запрошуваної таблиці.

Коли робиться запит до уявлення, насправді системою здійснюється звернення до базових таблиць. Наприклад, при виконанні команди

```
SELECT *
     FROM OTLSTUD
     WHERE SNUM > 3412;
```

СУБД фактично здійснює наступне:

```
SELECT *
     FROM STUDENTS
     WHERE STYP = '250,50'
     AND SNUM > 3412;
```

В даному випадку СУБД чудово справляється з своїм завданням. Проте іноді виникають ситуації, коли з'являються проблеми з уявленням в результаті комбінації з двох допустимих предикатів, які не працюватимуть.

Як ілюстрацію цього створимо уявлення, яке містить дані про кількість студентів, що отримують ту або іншу стипендію:

```
CREATE VIEW STYPCOUNT (STYP, COL)
     AS SELECT STYP, COUNT (*)
     FROM STUDENTS
     GROUP BY STYP;
```

Тепер зробимо запит до цього уявлення для того, щоб з'ясувати, чи є який-небудь розмір стипендії, призначений для двох студентів.

```
SELECT *
     FROM STYPCOUNT
     WHERE COL = 2;
```

Якщо скомбінувати два предикати, то, швидше за все, отримаємо:

```
SELECT STYP, COUNT (*)
     FROM STUDENTS
     WHERE COUNT (*) = 2
     GROUP BY STYP;
```

Запит, приведений вище, є неприпустимим, оскільки агрегатні функції не можуть використовуватися в предикаті. Виникає закономірне припущення про те, що СУБД в деяких випадках не зможе правильно сформулювати запит, а значить – виконати його. Все залежить від того, яким чином і по яких алгоритмах система інтерпретує призначені для користувача команди. Правильним способом формування вищезазначеного запиту буде наступний:

```
SELECT STYP, COUNT (*)
     FROM STUDENTS
     GROUP BY STYP
     HAVING COUNT (*) = 2;
```

але SQL може не виконати такого перетворення. Краще, що можна зробити в цьому випадку, це самому перевірити, чи справляється СУД, що використовується, з аналогічними запитами і, при необхідності, дотримуватися відповідних обмеженнь при формуванні запитів до уявлень.

У SQL існує поняття групових уявлень – тобто таких, які містять пропозицію GROUP BY або які засновані на інших групових уявленнях. Групові уявлення є чудовим способом обробляти складну інформацію безперервно. Попередній приклад уявлення якраз належить до групових. Тепер кожного разу, коли потрібно визначити кількість студентів, що отримують ту або іншу стипендію, досить просто вибрати всі записи з даного уявлення замість того, щоб створювати досить складний запит.

Уявлення можуть бути засновані відразу на декількох базових таблицях. Прикладом цьому може служити створення уявлення, яке показувало б оцінки студента по навчальному предмету, причому містило б не коди, а повні назви:

```
CREATE VIEW STUDOCIN
     AS SELECT THIRD.UNUM, FIRST.SPRIZ,
               SECOND.PNAME, THIRD.OCINKA
     FROM STUDENTS FIRST, PREDMET SECOND, USP THIRD
     WHERE FIRST.SNUM = THIRD.SNUM
     AND SECOND.PNUM = THIRD.PNUM;
```

Після цього легко орієнтуватися в отриманих оцінках. Наприклад, простий запит

```
SELECT *
     FROM STUDOCIN
     WHERE SPRIZ = 'Поляченко';
```

надасть докладну інформацію про оцінки цього студента:

```
UNUM      SPRIZ          PNAME          OCINKA
1001      Поляченко      Фізика         5
1004      Поляченко      Математика     4
```

Як було сказано вище, допускається об'єднувати уявлення з іншими базовими таблицями або уявленнями. Наприклад, можна вивести інформацію не тільки про оцінки, але і про дату її отримання. Для цього скористаємося запитом:

```
SELECT SPRIZ, PNAME, OCINKA, UDATE
     FROM STUDOCIN FIRST, USP SECOND
     WHERE FIRST.SPRIZ = 'Поляченко'
     AND FIRST.UNUM = SECOND.UNUM;
```

Вивід для цього запиту буде наступний:

```
SPRIZ          PNAME          OCINKA    UDATE
Поляченко      Фізика         5         2005-10-06
Поляченко      Математика     4         2005-12-06"
```

Уявлення можуть також використовувати підзапити, зокрема співвіднесені. Наприклад, за допомогою наступного уявлення можна побачити всі оцінки по дисципліні із значеннями вище за середню по цій же дисципліні:

```
CREATE VIEW AVGOC
     AS SELECT *
          FROM USP FIRST
          WHERE OCINKA >
          (SELECT AVG (OCINKA)
               FROM USP SECOND
               WHERE SECOND.PNUM = FIRST.PNUM);
```

Видобування даних виконується простим запитом:

```
SELECT *
     FROM AVGOC;
```

Їх цього видно, наскільки уявлення SQL спрощують роботу з даними, надаючи можливість користувачу обійтися без складних запитів.

На жаль, досить часто уявлення є об'єктами, доступними тільки для читання. Це означає, що інформацію з них можна запрошувати, але вони не можуть піддаватися діям команд модифікації. Крім того, існують також деякі команди, які не допустимі у означеннях уявлень: уявлення повинне ґрунтуватися на одиночному запиті, тому об'єднання UNION не дозволяється. Крім цього, у означенні уявлення не використовується впорядкування ORDER BY, оскільки, по аналогії з базовою таблицею, в представленні записи є невпорядкованими.

Синтаксис видалення уявлення з БД подібний видаленню базових таблиць:

```
DROP VIEW <VIEW NAME>
```

Слід мати на увазі, що в цьому випадку немає необхідності видалення всіх даних з уявлення, тому що реальні дані в ньому не містяться. Проте при цьому для видалення уявлення користувач повинен бути його власником.

Як вже було відмічено, не всі уявлення здатні до модифікації. Взагалі кажучи, команди модифікації мови DML – INSERT, UPDATE і DELETE – звичайно ж можуть застосовуватися для уявлень. Розглянемо ряд правил, що визначають, чи є уявлення таким, що модифікується. Даний момент є найбільш важким і неоднозначним у використанні уявлень.

Команди модифікації фактично впливають на значення в базовій таблиці уявлення, що є деякою суперечністю внаслідок того, що уявлення складається з результатів запиту. Отже, коли відбувається модифікація уявлення, то редагується набір результатів запиту. Але модифікація не повинна впливати на запит – вона повинна впливати на значення в таблиці, до якої був зроблений запит, і таким чином змінювати вивід запиту.

При цьому може виникнути ряд неоднозначних ситуацій. Пригадаємо розглянуте вище уявлення, що виводить прізвище студента, найменування предмету і оцінку. Припустимо, що це уявлення є таким, що модифікується і необхідно з нього видалити рядок. Якщо яким-небудь спеціальним чином не описати правила видалення, то неясно – видалення рядка з уявлення спричинить за собою видалення оцінки із таблиці успішності USP, студента з таблиці STUDENTS, предмету з PREDMET або якоїсь комбінації цих записів.

Отже, якщо команди модифікації можуть виконуватися в уявленні, то воно вважається таким, що модифікується; інакше воно призначене тільки для читання при запиті. Основна відмінність між ними полягає в тому, що уявлення, що модифікується, – це уявлення, в якому команда модифікації може виконатися так, щоб змінити тільки один запис основної таблиці в кожен момент часу, не впливаючи на інші рядки будь-якої таблиці. Використання цього принципу на практиці, однак ускладнено. Крім того, деякі уявлення, які є такими, що модифікуються в теорії, насправді не є такими, що модифікуються в SQL.

Критерії, по яких визначають, чи є уявлення таким, що модифікується в SQL. наступні:

- уявлення повинне ґрунтуватися тільки на одній базовій таблиці.
- воно повинне містити первинний ключ цієї таблиці;
- воно не повинне мати жодних полів, які б були агрегатними функціями;
- воно не повинне містити DISTINCT в своєму означенні:
- уявлення не повинне використовувати GROUP BY або HAVING в своєму означенні;
- бажано, щоб воно не використовувало в своєму означенні підзапити;
- його можна використовувати в другом уявленні, але це уявлення повинне також бути таким, що модифікується;
- воно не повинне використовувати константи, рядки або вирази значень серед вибраних полів виводу;
- для команди INSERT воно може містити будь-які поля основної таблиці, які мають обмеження NOT NULL, якщо інше обмеження за замовчуванням не означене.

Таким чином, уявлення, що модифікуються, фактично подібні фрагментам базових таблиць, відображаючи певну частину їх вмісту. Вони можуть обмежувати певні рядки шляхом використання предикатів або спеціально іменованих стовпців, деякі поля виключати з виводу, але зрештою, такі уявлення надають користувачу значення безпосередньо, не виводячи інформацію з використанням складених функцій і виразів.

Відмінності між уявленнями, що модифікуються, і уявленнями, призначеними тільки для читання, не випадкові. Цілі, для яких їх використовують, різні: уявлення, що модифікуються, в основному, використовуються так само, як і базові таблиці. Фактично, користувачі не можуть навіть усвідомити, чи є об'єкт, який вони запрошують, базовою таблицею або уявленням, тобто в основному цей засіб захисту для приховання частин таблиці, що є конфіденційними або що не відносяться до потреб даного користувача. Представлення режиму тільки для читання, з іншого боку, дозволяють отримувати і форматувати дані раціональніше. Вони дають цілий арсенал складних запитів, які можна виконати і повторити знову, зберігаючи отриману інформацію. Нарешті, результати цих запитів можуть потім використовуватися в інших запитах самостійно, що дозволяє уникнути складних предикатів і понизити ймовірність помилкових дій.

Приведемо ряд прикладів. Наступне уявлення виводить кількість оцінок, отриманих всіма студентами в той або інший день. Воно є уявленням тільки для читання, оскільки містить у означенні агрегатну функцію і GROUP BY:

```
CREATE VIEW PRCOUNT (UDATE, COL)
     AS SELECT UDATE, COUNT (*)
          FROM USP
          GROUP BY UDATE;
```

Наступне уявлення є таким, що модифікується внаслідок того, що воно задовольняє переліченим вище умовам. Як вивід в ньому містяться дані про успішність по предмету з кодом 2003, тобто математиці:

```
CREATE VIEW MATEMUSP
     AS SELECT *
          FROM USP
          WHERE PNUM = 2003;
```

Приведене нижче уявлення – тільки для читання:

```
CREATE VIEW IDXSTYP (SNUM, SPRIZ, NEWSTYP)
     AS SELECT SNUM, SPRIZ, STYP*2
          FROM STUDENTS
          WHERE STYP='250,50';
```

Це пов'язано з тим, що у означенні присутня функція STYP*2. До речі кажучи, в більшості СУБД останнє уявлення допускатиме видалення рядків з відповідним ключу SNUM видаленням їх в базовій таблиці.

Уявлення, що виводить інформацію про студентів, що отримали оцінки в певний день, буде тільки для читання:

```
CREATE VIEW DATEOC
     AS SELECT *
     FROM STUDENTS
     WHERE SNUM IN
     (SELECT SNUM
          FROM USP
          WHERE UDATE = '10/06/2005');
```

Це відбувається через те, що у означенні уявлення присутній підзапит. В той же час наступне уявлення є таким, що модифікується:

```
CREATE VIEW DATEOC2
     AS SELECT *
          FROM USP
          WHERE UDATE IN
               ('10/06/2005', '11/06/2005');
```

Інший висновок про можливість модифікації уявлення може бути заснований на можливості вводу того або іншого значення. Наприклад, наступне уявлення, що модифікується, виводить дані про студентів, що мають відмінні оцінки:

```
CREATE VIEW ONLY5
     AS SELECT SNUM, OCINKA
          FROM USP
```

```
WHERE OCINKA = 5;
```

В даному випадку уявлення просто обмежує доступ користувача до даних таблиці успішності, дозволяючи бачити тільки частину значень. Виконаємо наступну команду:

```
INSERT INTO ONLY5
     VALUES (3415, 4);
```

Це – допустима команда INSERT в даному уявленні і рядок буде вставлений, з допомогою ONLY5, у таблицю успішності USP. Проте, коли ця інформація буде додана, рядок зникне з уявлення, оскільки значення оцінки не рівне 5.

Іноді це може стати проблемою, оскільки дані вже знаходяться в таблиці успішності, але користувач цього не бачить і не в змозі виконати їх видалення або модифікацію. Для виключення подібних моментів служить фраза WITH CHECK OPTION у означенні уявлення. Тепер можна вдосконалити розглянуте уявлення з врахуванням наявної можливості:

```
CREATE VIEW ONLY5
     AS SELECT SNUM, OCINKA
          FROM USP
          WHERE OCINKA =5
          WITH CHECK OPTION;
```

Після цього вищезазначена вставка значень буде відхилена системою.

Фраза WITH CHECK OPTION проводить дію в режимі "все або нічого", оскільки вона розміщується у означенні уявлення, і всі команди модифікації піддаватимуться перевірці. Таким чином, можна регулювати процес вводу значень, які користувач згодом сам не в змозі коректувати.

Аналогічна проблема може виникнути при вставці записів в уявлення з предикатом, що базується на одному або більше виключених полів. Розглянемо наступне уявлення:

```
CREATE VIEW OTLSTYP
     AS SELECT SNUM, SPRIZ
          FROM STUDENTS
          WHERE STYP='250,50';
```

Його результатом будуть номери студентських квитків і прізвища студентів, що отримують стипендію у розмірі 250,50. Взагалі кажучи, логічно не включити в результати поле STYP – все одно список значень в уявленні для цього поля буде однаковим.

Що ж відбуватиметься кожного разу, коли користувач спробує додати сюди запис? Оскільки він не в змозі ввести значення розміру стипендії, а значення за замовчуванням в більшості випадків буде NULL, то введений запис буде тут же виключений з уявлення. Причому, описана проблема відбуватиметься з будь-яким рядком, який спробують додати в OTLSTYP. Крім того, це ще не означає, що користувач має можливість вставляти значення в базову таблицю. Якщо, наприклад, в таблиці STUDENTS поле STYP обмежене як NOT NULL, то результати вставки взагалі будуть важко передбачені користувачем. Більше того, ця проблема навряд чи буде вирішена, навіть при означенні уявлення таким чином:

```
CREATE VIEW OTLSTYP
     AS SELECT SNUM, SPRIZ
          FROM STUDENTS
          WHERE STYP='250,50'
```

```
WITH CHECK OPTION;
```

Після цього в уявленні можна буде видаляти і модифікувати рядки, проте їх не вдасться вставити. Тому при створенні уявлень, потрібно чітко вирішити – що може відбутися при його використанні. Рекомендація тут така: вкрай бажано, щоб ті поля, які беруть участь в предикаті уявлення, були включені і до складу полів, що виводяться, навіть якщо вони містять однакові значення.

Необхідно згадати про таку властивість фрази WITH CHECK OPTION, як відсутність каскадної зміни даних. Ця фраза застосовується тільки в уявленнях, які засновані на базових таблицях, а не на інших уявленнях. Наприклад, якщо попереднє уявлення вдосконалити з врахуванням висловлених рекомендацій, тобто

```
CREATE VIEW OTLSTYP
     AS SELECT SNUM, SPRIZ, STYP
          FROM STUDENTS
          WHERE STYP='250,50'
          WITH CHECK OPTION;
```

то спроба вставки

```
INSERT INTO OTLSTYP
     VALUES (3417, 'Решітник', '170,00');
```

потерпить невдачу. Проте, якщо буде створене інше уявлення з ідентичним змістом, засноване на першому:

```
CREATE VIEW NEWOTL
     AS SELECT *
          FROM OTLSTYP;
```

то відповідно до стандарту SQL дана вставка запису буде допустима. Річ у тому, що фраза WITH CHECK OPTION просто гарантує, що будь-яке коректування в уявленні відтворить значення, які задовольняють предикату цього уявлення, а модифікація інших уявлень, що базуються на першому, є допустимою, якщо ці уявлення не захищені пропозиціями WITH CHECK OPTION всередині цих же уявлень. Причому перевірці піддаються тільки предикати тих уявлень, в яких здійснюється модифікація. Навіть якщо друге з даних уявлень буде визначене як

```
CREATE VIEW NEWOTL
     AS SELECT *
          FROM OTLSTYP
          WITH CHECK OPTION;
```

це не вирішить проблеми, оскільки рядок, що вставляється, задовольняє предикату представлення NEWOTL. Варто мати на увазі, що цей недолік SQL в деяких сучасних СУБД вирішений і каскадування фрази WITH CHECK OPTION проводитиметься, проте в цьому необхідно переконатися в документації на дані системи.

Одна з сильних сторін SQL – це здатність функціонувати по всіх рядках таблиці: скільки рядків задовольнить предикату, стільки і буде виведено. Проте це може викликати ряд проблем, коли SQL взаємодіє з іншими мовами програмування. Дійсно, важко передати вивід запиту змінним, якщо наперед невідомо, наскільки великі будуть результати виводу. У SQL розв'язання

цього здійснюють за допомогою так званого курсора.

Курсор – це вид змінної, яка пов'язана із запитом. Значенням цієї змінної може бути кожен рядок, який виводиться при запиті і він повинен бути оголошений перш, ніж буде використаний, що робиться командою DECLARE CURSOR таким чином:

```
EXEC SQL DECLARE CURSOR STYPCUR
     FOR SELECT SNUM, SPRIZ, STYP
          FROM STUDENTS
          WHERE STYP= '250,50';
```

Цей запит не виконається негайно, оскільки в даному випадку представлено тільки його означення. Курсор дещо нагадує уявлення, в якому означення містить запит, а змістовною частиною є будь-який вивід запиту, і це відбувається кожного разу, коли курсор стає відкритим. Проте, на відміну від базових таблиць або уявлень, рядки курсора впорядковані і є перший, другий і т.д., а також останній рядок. Цей порядок може бути довільним з явним управлінням за допомогою фрази ORDER BY в запиті.

Коли в програмі необхідно виконати запит, то відкривають курсор за допомогою наступної команди:

```
EXEC SQL OPEN CURSOR STYPCUR;
```

Значення в курсор передаються саме тоді, коли виконується приведена вище команда, а DECLARE і FETCH на зчитування інформації в курсор впливу нечинять.

Команда FETCH використовується для того, щоб витягнути вивід із запиту по одному рядку в кожен момент часу. Наприклад:

```
EXEC SQL FETCH STYPCUR INTO :STUDENTSID,
     :STUDENTSPRIZ, :STUDENTSSTYP;
```

Ця конструкція присвоїть значення з першого вибраного рядка в змінні пам'яті. Наступна команда FETCH виведе черговий набір значень і т.д. Як правило, команду FETCH поміщають всередину циклу так, щоб вибравши рядок з курсора, можна було здійснити переміщення набору значень з цього рядка в змінні, повернутися назад в цикл і перемістити наступний набір значень в ті ж самі змінні.

Зверніть увагу на оператора CLOSE CURSOR, який повинен відповідати OPEN CURSOR. Він звільняє курсор значень, тому після нього запит потрібно буде виконати повторно з оператором OPEN CURSOR, перш ніж перейти до вибору наступних значень. Для нашого випадку цей оператор виглядатиме таким чином:

```
EXEC SQL CLOSE CURSOR STYPCUR;
```

Іншими словами, поки курсор закритий, SQL не стежить за тим, які рядки були вибрані. Якщо курсор відкривається знову, запит повторно виконується з цієї точки, і цикл вибору значень повинен починатися спочатку. І ще одна особливість: коли у FETCH немає більше рядків, які треба витягувати, він просто не міняє значення в змінних пропозиції INTO, тобто якщо дані в курсорі вичерпалися, ці змінні виводитимуться з ідентичними значеннями до тих пір, поки не завершиться цикл.

Таким чином, нами проаналізований матеріал про уявлення і схожі на них об'єкти, що називаються курсорами. Рекомендуємо широко використовувати ці засоби в системах, що розробляються, оскільки вони значно розширюють можливості SQL по виводу і обмеженню

даних залежно від конкретних потреб того або іншого користувача.

2.7. Методи захисту інформації

2.7.1. Безпека баз даних і привілегії

При зберіганні інформації в СУБД одним з основних завдань залишається забезпечення безпеки даних. У мові SQL використовуються наступні основні принципи захисту даних:

- у БД дійовими особами є користувачі. Якщо з даними відбувається яка-небудь маніпуляція, то вона відбувається від імені конкретного користувача. СУБД може відмовитися виконати запрошувані дії залежно від того, який користувач запрошує цю дію.
- об'єкти БД є тими елементами, чий захист може здійснюватися за допомогою SQL. Звичайно забезпечується не тільки захист таблиць і уявлень, але і інших об'єктів БД. Система дозволяє одним користувачам здійснювати дії над одними об'єктами і забороняє це над іншими.
- у SQL використовується система привілеїв, тобто прав користувача на проведення тих або інших дій над певним об'єктом бази даних.

Взагалі кажучи, адміністратор БД сам створює користувачів і дає їм привілеї, але, з іншого боку, користувачі, які створюють таблиці, самі мають права на управління цими таблицями. Існує декілька типів привілеїв, що відповідають цілому ряду типів операцій. У SQL привілеї даються і відміняються двома командами SQL – GRANT (допуск) і REVOKE (відміна).

Кожен користувач в середовищі SQL має спеціальне ідентифікаційне ім'я або номер (ID) доступу. Команда, послана в БД, асоціюється з певним користувачем, тобто спеціальним ідентифікатором доступу. Оскільки це відноситься до SQL БД, ID дозволи – це ім'я користувача, і SQL може використовувати спеціальне ключове слово USER, яке посилається до ідентифікатора доступу, пов'язаного з поточною командою. Команда інтерпретується і вирішується (або забороняється) на основі інформації, пов'язаної з ідентифікатором доступу користувача, що подав команду.

У системах з великою кількістю користувачів, існує спеціальна процедура входу в систему, яку користувач повинен обов'язково виконати для отримання доступу. Ця процедура визначає, який саме ID доступу буде пов'язаний з поточним користувачем. Звичайно кожна людина, що використовує БД, має свій власний ID доступу і при реєстрації перетворюється на дійсного користувача. Проте достатньо часто користувачі, що мають багато задач, можуть реєструватися під різними ID доступу, або навпаки – один ID доступу може використовуватися декількома користувачами. З погляду SQL немає жодної різниці між цими двома випадками – він сприймає користувача просто як його ID доступу.

SQL БД може використовувати власну процедуру входу в систему або може дозволити іншій програмі (наприклад, операційній системі комп'ютера) обробляти файл реєстрації і отримувати ID доступу. Проте, яким би способом СУБД не скористалася, у будь-якому випадку SQL матиме ID доступу для того, щоб пов'язати його з діями користувача, для якого матиме значення ключове слово USER.

Кожен користувач в SQL БД має набір привілеїв. Ці привілеї можуть змінюватися з часом – нові додаватися, старі видалятися. Стандартні SQL привілеї, визначені ANSI – це привілеї об'єкту. Це означає, що користувач має привілей для виконання даної команди тільки на певному об'єкті в БД. Привілеї об'єкту зв'язані одночасно і з користувачами і з таблицями, тобто привілей дається певному користувачу у вказаній базовій таблиці або уявленні. При цьому не варто забувати, що користувач, що створив таблицю будь-якого вигляду, є її власником. Це в свою чергу означає, що такий користувач має всі привілеї в цій таблиці і може передавати привілеї іншим користувачам для цієї таблиці.

Привілеї, які можна призначити користувачу, наступні:

- SELECT – користувач з цим привілеєм може виконувати запити до таблиці;
- INSERT – користувач з цим привілеєм може виконувати вставку записів, тобто команду INSERT в таблиці;
- UPDATE – користувач з цим привілеєм може виконувати коректування даних, тобто команду UPDATE до даної таблиці;
- DELETE – користувач з цим привілеєм може виконувати команду DELETE в таблиці;
- REFERENCES – з цим привілеєм користувач має можливість означити зовнішній ключ, який використовує один або більше стовпців даної таблиці, як батьківський ключ;
- INDEX – дає право користувачу створювати індекс в таблиці;
- SYNONYM – користувач, що володіє цим привілеєм, має право створювати синонім для об'єкту;
- ALTER – дає право користувачу виконувати команду ALTER TABLE в даній таблиці.

Отже, SQL призначає користувачам ці привілеї за допомогою команди GRANT. Наприклад, якщо користувач SA володіє таблицею студентів STUDENTS і бажає дозволити користувачу SHER виконати запит до неї, то SA повинен в цьому випадку виконати наступну команду:

```
GRANT SELECT ON STUDENTS TO SHER;
```

Тепер SHER може виконувати запити до таблиці STUDENTS без інших привілеїв, тобто він може тільки вибрати значення, але не може виконати будь-яку дію, яка б впливала на значення в таблиці, включаючи використання її як батьківську таблицю зовнішнього ключа.

Коли SQL отримує команду GRANT, він перевіряє привілеї користувача, що подав її, і визначає її допустимість. При цьому SHER самостійно не може виконати цю команду або надати право SELECT іншому користувачу, оскільки таблиця йому не належить. Якщо ж для користувача SHER необхідно надати право вставляти в таблицю рядки, то це можна реалізувати за допомогою наступної пропозиції:

```
GRANT INSERT ON STUDENTS TO SHER;
```

Проте часто доводиться передавати декілька привілеїв окремому користувачу командою GRANT. Тоді списки привілеїв або користувачів в команді відокремлюють комами. Наприклад, з метою надання привілеїв для запиту і вставки значень користувачам SHER і MAG в таблиці STUDENTS, можна скористатися командою:

```
GRANT SELECT, INSERT ON STUDENTS TO SHER, MAG;
```

Коли привілеї і користувачі перераховані таким чином, як показано вище, весь список привілеїв надаються всім вказаним користувачам. У деяких СУБД, крім цього, допускається через кому перерахувати список таблиць, до яких застосовуються привілеї для всіх вказаних користувачів.

Всі привілеї об'єкту використовують один і той же синтаксис, окрім команд UPDATE і REFERENCES, в яких можна додатково вказувати імена полів. Наприклад, команда:

```
GRANT UPDATE ON STUDENTS TO SHER;
```

дозволить SHER змінювати значення у всіх полях таблиці STUDENTS. Проте власник може обмежити його в зміні таблиці, наприклад, ввівши таку команду:

```
GRANT UPDATE (STYP) ON STUDENTS TO SHER;
```

дозволяючи модифікувати тільки поле STYP. Іншими словами, можна просто вказати конкретне поле, до якого привілей UPDATE повинен бути застосований. При цьому допускаються списки з декількох, вказаних в довільному порядку, стовпців таблиці, до яких встановлюваний привілей повинен бути застосований, наприклад:

```
GRANT UPDATE (SPRIZ, STYP) ON STUDENTS TO SHER;
```

При використанні привілею REFERENCES слідують тому ж самому правилу, що і для UPDATE. Коли надається привілей REFERENCES іншому користувачу, він зможе створювати зовнішні ключі, що посилаються на стовпці таблиці, як на батьківські ключі.

Подібно до UPDATE, для привілею REFERENCES може бути вказаний список з одного або більш стовпців, для яких призначений цей привілей. Наприклад, можна надати SHER право використовувати STUDENTS, як таблицю батьківського ключа, за допомогою наступної команди:

```
GRANT REFERENCES (SNUM, STYP) ON STUDENTS TO SHER;
```

Ця команда дає SHER право використовувати поля SNUM, STYP як батьківські ключі по відношенню до будь-яких зовнішніх ключів в його таблицях. Як і у випадку з привілеєм UPDATE, тут можна виключати список полів і, таким чином, дозволяти всім без виключення стовпцям бути використовуваними як батьківські ключі, що здійснюється за допомогою команди:

```
GRANT REFERENCES ON STUDENTS TO SHER;
```

Природно, що привілей буде придатний для використання тільки в тих полях, які мають обмеження, необхідні для батьківських ключів.

SQL підтримує два аргументи для команди GRANT, які мають спеціальне значення – це ALL PRIVILEGES (або ALL) і PUBLIC.

ALL використовується замість імен привілеїв в команді GRANT для того, щоб віддати всі привілеї в таблиці. Наприклад, для передачі користувачу SHER всього набору привілеїв для таблиці STUDENTS можна скористатися наступною командою:

```
GRANT ALL ON STUDENTS TO SHER;
```

Привілеї PUBLIC більше схожі на тип аргументу для таблиці – коли вона надається, всі користувачі її отримують автоматично. Найчастіше це застосовується для привілею SELECT в певних базових таблицях або уявленнях, які необхідно зробити доступними для будь-якого користувача. Наприклад, для того, щоб дозволити будь-якому користувачу проглядати таблицю STUDENTS, необхідно виконати наступне:

```
GRANT SELECT ON STUDENTS TO PUBLIC;
```

Звичайно, можна надати будь-які або навіть всі привілеї будь-якому користувачу, але це небажано, оскільки всі привілеї, за винятком SELECT, дозволяють змінювати або обмежувати зміст таблиці, а це може викликати проблеми. Майте на увазі, що будь-який новий користувач, що додається до системи, автоматично отримує всі привілеї, призначені як PUBLIC.

Іноді при створенні таблиці виникає необхідність в тому. щоб інші користувачі могли

отримувати і передавати привілеї в таблиці. SQL дозволяє робити це за допомогою команди WITH GRANT OPTION. Наприклад, якщо SA бажає, щоб SHER мав право надавати привілей SELECT в таблиці STUDENTS іншим користувачам, це можна реалізувати таким чином:

```
GRANT SELECT ON STUDENTS TO SHER WITH GRANT OPTION;
```

Після цього SHER отримує право передавати привілей SELECT третім особам, наприклад, він може використовувати такі команди:

```
GRANT SELECT ON STUDENTS TO MAG;
```

або

```
GRANT SELECT ON STUDENTS TO MAG WITH GRANT OPTION;
```

Таким чином, користувач за допомогою GRANT OPTION знаходиться в особливому привілеї для даної таблиці, і може, у свою чергу, надати цей привілей до тієї ж таблиці іншому користувачу, зокрема використовувати GRANT OPTION для передачі цієї можливості. Проте варто пам'ятати, що це не міняє приналежності самої таблиці, тобто, як і раніше, таблиця належить її творцю. Користувач же за допомогою GRANT OPTION у всіх привілеях для даної таблиці матиме всю повноту влади в цій таблиці.

При виникненні потреби у видаленні привілею, використовується команда REVOKE. Синтаксис команди REVOKE схожий на GRANT, але має зворотний сенс. Наприклад, щоб видалити привілей INSERT для SHER в таблиці STUDENTS, можна ввести:

```
REVOKE INSERT ON STUDENTS FROM SHER;
```

Використання списків привілеїв і користувачів тут допустимі, як і у випадку з GRANT. Тому дозволяється ввести команду, яка відмінить привілеї на видалення і вставку для користувачів SHER і MAG:

```
REVOKE DELETE, INSERT ON STUDENTS TO SHER, MAG;
```

При цьому варто звернути увагу на такий важливий момент – привілеї відміняються користувачем, який їх надав, при цьому відміна каскадуватиме, тобто вона автоматично розповсюджуватиметься на всіх користувачів, що одержали від нього цей привілей.

Дії привілеїв можна зробити точнішими, якщо їх використовувати для уявлень. Всякий раз, коли передаються привілеї в базовій таблиці користувачу, то вона автоматично розповсюджується на всі її рядки, а при використанні можливих виключень UPDATE і REFERENCES – і на всі поля таблиці. Створюючи уявлення, яке посилається на основну таблицю, і потім переносить привілей на уявлення, а не на таблицю, можна обмежувати ці привілеї.

Щоб створювати уявлення, користувач повинен мати привілей SELECT у всіх таблицях, на які посилається означення уявлення. Якщо уявлення модифікується, то будь-який привілей INSERT, UPDATE або DELETE, які має користувач в базовій таблиці, автоматично передаватиметься уявленню. Якщо має місце нестача в привілеях на модифікацію в базових таблицях, на жаль, користувач не зможе отримати їх і в уявленнях, які створив, навіть якщо самі ці уявлення є такими, що модифікуються. Причиною цьому є використання зовнішніх ключів в уявленнях.

Припустимо, що необхідно користувачу SHER надати можливість проглядати тільки

поля SNUM і SPRIZ таблиці STUDENTS. Це реалізується шляхом створення уявлення

```
CREATE VIEW TEST
     AS SELECT SNUM, SPRIZ
          FROM STUDENTS;
```

а потім надання користувачу SHER привілею SELECT в уявленні, а не в самій таблиці, тобто

```
GRANT SELECT ON TEST TO SHER;
```

Можна створити привілеї спеціально для полів команд модифікації. Проте треба пам'ятати, що для інших полів таблиці в команді INSERT це означатиме вставку значень за замовчуванням, а для команди DELETE – обмеження стовпця не матиме значення.

Привілеї застосовують і для певних рядків таблиці знову-таки з використанням механізму уявлень. Це реалізується шляхом використання предиката в уявленні, який і визначить включені рядки.

Наприклад, щоб надати користувачу SHER привілей UPDATE в таблиці STUDENTS, але так, що модифікуватися можуть дані для студентів з нульовою стипендією, створюється таке уявлення:

```
CREATE VIEW NULLSTYP
     AS SELECT *
          FROM STUDENTS
          WHERE STYP = '0'
          WITH CHECK OPTION;
```

а потім – передається привілей UPDATE:

```
GRANT UPDATE ON NULLSTYP TO SHER;
```

У цьому полягає відмінність привілею для певних рядків від привілею UPDATE для деяких стовпців, яка поширена на всі стовпці таблиці. Фраза WITH CHECK OPTION оберігає SHER від заміни значення поля STYP на будь-яке значення, окрім 0.

Інша можливість полягає в тому, щоб пропонувати користувачам доступ до вже витягнутих даних, а не до фактичних значень в таблиці. В даному випадку зручно скористатися агрегатними функціями. Наприклад, можна створити уявлення, яке виводить середнє значення оцінок кожного студента:

```
CREATE VIEW AVGNUM
     AS SELECT SNUM, AVG (OCINKA)
          FROM USP
          GROUP BY SNUM;
```

і потім передати користувачу SHER привілей SELECT в представленні AVGNUM:

```
GRANT SELECT ON AVGNUM TO SHER;
```

До речі, альтернативою обмеження є використання уявлень з WITH CHECK OPTION. Наприклад, можна створити наступне уявлення для введення оцінок в таблицю успішності:

```
CREATE VIEW INPOC
```

```
AS SELECT *
FROM USP
WHERE OCINKA IN (1, 2, 3, 4, 5)
WITH CHECK OPTION;
```

Тоді неправильні оцінки будуть виключені для всіх користувачів, які мають привілеї для введення або модифікації значень в дане уявлення. Це іноді виявляється простішим, ніж змінювати обмеження полів в базовій таблиці.

Тепер декілька слів про привілеї, які не визначаються в термінах об'єктів даних. Такі привілеї називаються привілеями системи або правами БД. Вони найчастіше включають право створювати об'єкти даних, що відрізняються від базових таблиць і уявлень. Привілеї системи для створення уявлень, повинні доповнювати, а не замінювати привілеї об'єктів, про які йшлося вище.

Не варто забувати, що в системі завжди є деякі типи користувачів, які автоматично мають більшість або навіть всі привілеї, а також мають можливість передати свій статус кому-небудь за допомогою привілею або групи привілеїв. Зокрема, таким користувачем є адміністратор БД.

У найзагальнішому випадку є три базові привілеї системи:

- CONNECT – складається з права реєструватися і права створювати уявлення і синоніми, якщо користувачу передані відповідні привілеї об'єкту;
- RESOURCE – складається з права створювати базові таблиці;
- DBA – це привілей користувача, що дає найвищі повноваження в БД.

Деякі системи, крім того, мають спеціального користувача, який іноді називається SYSADM, SYS, SA, а у системі PostgreSQL – postgres, який має найвищі повноваження. Фактично це – спеціальне ім'я, а не просто користувач з особливим DBA привілеєм. Бажано, щоб тільки одна людина мала право реєструватися з таким ім'ям.

Команда GRANT є придатною для використання як з привілеями об'єкту, так і з системними привілеями. Наприклад, адміністратор БД може передати привілей для створення таблиці користувачу SHER таким чином:

```
GRANT RESOURCE TO SHER;
```

Спочатку користувач SHER, в більшості випадків, створюється адміністратором БД, автоматично надаючи йому привілей CONNECT. В цьому випадку звичайно додається пропозиція IDENTIFIED BY, вказуюча пароль. Наприклад, для первинної реєстрації користувача адміністратор бази даних вводить наступне:

```
GRANT CONNECT TO SHER IDENTIFIED BY Roman18;
```

Це приведе до створення користувача з ім'ям SHER, дасть йому право реєструватися і призначить йому пароль Roman18. Після цього SHER і адміністратор БД мають можливість при необхідності змінити пароль.

Якщо новий користувач створюватиме базові таблиці, для нього потрібно буде також надати привілей RESOURCE. Дана дія породжує іншу проблему – якщо буде зроблена спроба видалити привілей CONNECT цього користувача, і в БД є таблиці, які ним створені, команда буде відхилена, тому що ця дія залишить таблицю без власника, що не допускається. Отже, перед видаленням користувача спочатку необхідно видалити всі створені ним таблиці.

Таким чином, привілеї дають можливість SQL виконувати дії тільки через певних користувачів в спеціальній системі БД відповідно до встановлених для них дозволів і обмежень.

2.7.2. Використання системного каталога (на прикладі SQL Server)

Щоб правильно функціонувати, СУБД повинна стежити за багатьма різними об'єктами і елементами БД: таблицями, уявленнями, індексами, синонімами, привілеями, користувачами і т.д. Є різні способи реалізації цього, проте найчастіше це відбувається шляхом збереження службової інформації в спеціальних таблицях. Це дає можливість СУБД розміщувати і управляти інформацією, якої вона потребує, використовуючи ті ж самі процедури, що і для управління даними предметної області. Набір SQL таблиць, що зберігають службову інформацію для своїх внутрішніх потреб, називають системним каталогом, словником даних або системними таблицями.

Таблиці системного каталога, по суті, нагадують звичайні SQL таблиці, що містять поля і записи даних. Таблиці каталога створюються і модифікуються за допомогою самої БД, і ідентифікуються за допомогою спеціальних імен, наприклад SYSTEM. БД створює ці таблиці і модифікує їх автоматично, причому користувачами таблиці каталога не можуть бути безпосередньо піддані дії команди модифікації – якщо це відбудеться, то система може заплутатися і стати непрацездатною. Проте в більшості систем, системний каталог може бути запрошений користувачем, що дає можливість дізнатися додаткову інформацію про БД. Звичайно, ця інформація не доступна всім користувачам – подібно до інших таблиць, доступ до системного каталога обмежений для користувачів без відповідних привілеїв.

Оскільки каталог належить самій системі, то традиційно привілеї системного каталога надає адміністратор БД. Крім того, деякі привілеї можуть надаватися користувачам автоматично.

Типовий системний каталог складається з наступних таблиць:

- SYSTEMCATALOG – тут зберігається інформація про базові таблиці і уявлення;
- SYSTEMCOLUMNS – дані про поля таблиць;
- SYSTEMTABLES – дані про системні таблиці системного каталога;
- SYSTEMINDEXES – інформація про індекси в таблиці;
- SYSTEMUSERAUTH – дані про користувачів БД;
- SYSTEMTABAUTH – дані про об'єктні привілеї користувачів;
- SYSTEMCOLAUTH – дані про привілеї користувачів в полях таблиць;
- SYSTEMSYNONS – синоніми для таблиць.

Адміністратор БД може надати користувачу право переглядати SYSTEMCATALOG наступною командою:

```
GRANT SELECT ON SYSTEMCATALOG TO SHER;
```

Після цього SHER зможе побачити деяку інформацію про всі таблиці в БД. Наприклад, інформація про таблиці БД зберігається у вигляді окремого запису. Як правило, її структура така, що присутні поля для імені таблиці, імені користувача-власника, кількість полів в таблиці і ознаки того, чи є вона уявленням або базовою таблицею.

Оскільки SYSTEMCATALOG – це таблиця, її можна використовувати в уявленні. Припустимо, що тільки таблиці каталога є власністю користувача SA. Тоді спробуємо дозволити користувачам бачити тільки їх власні об'єкти. Для цього спочатку необхідно створити наступне уявлення:

```
CREATE VIEW OWNDB
     AS SELECT *
          FROM SYSTEMCATALOG
          WHERE OWNER = USER;
```

Тут виходитимемо з того, що поле OWNER містить ідентифікатор доступу USER

користувача. Після цього адміністратор БД може надати всім користувачам доступ до цього уявлення:

```
GRANT SELECT ON OWDB TO PUBLIC;
```

Після цієї команди кожен користувач, здатний вибирати дані тільки про ті об'єкти з SYSTEMCATALOG, власником яких він сам є. Аналогічним чином можна надати можливість користувачам проглядати таблицю SYSTEMCOLUMNS для стовпців з його власних таблиць, оскільки традиційна її структура містить найменування поля таблиці в БД і імені власника. Таким чином, це реалізується шляхом виконання наступних команд:

```
CREATE VIEW OWNCOL
     AS SELECT *
          FROM SYSTEMCOLUMNS
          WHERE OWNER = USER;
```

і

```
GRANT SELECT ON OWNCOL TO PUBLIC;
```

Майте на увазі, що в різних СУБД кількість даних і назви полів в таблицях системного каталога можуть відрізнятися, проте основна ідея при цьому зберігається.

Більшість версій SQL дозволяють поміщати коментарі в спеціальні стовпці пояснень таблиць каталогів SYSTEMCATALOG і SYSTEMCOLUMNS, що зручно, так як ці таблиці не завжди можуть пояснити свій вміст. Для цього можна використовувати команду COMMENT ON з рядком тексту коментаря, наприклад:

```
COMMENT ON TABLE SHER.TEACHERS IS 'Це список викладачів';
```

Цей текст буде поміщений в стовпець пояснень SYSTEMCATALOG. Сам коментар поміщається в службову таблицю, в даному випадку, що містить інформацію про таблицю TEACHERS, власником якої є SHER.

Приведемо типову структуру таблиці SYSTEMINDEXES, що містить дані про індекси в БД.

- iname – ім'я індексу:
- iowner – ім'я користувача, який створив індекс;
- tname – ім'я таблиці, яка містить індекс;
- cnumber – номер стовпця в таблиці, по якому створений індекс;
- tabowner – користувач, який володіє таблицею, що містить індекс;
- numcolumns – кількість стовпців в індексі;
- cposition – позиція поточного стовпця серед набору індексів;
- isunique – чи є індекс унікальним.

За допомогою даних з цієї таблиці можна визначити, наприклад, інформацію про індекс з ім'ям SNUMIDX:

```
SELECT INAME, IOWNER, TNAME, CNUMBER, ISUNIQUE
     FROM SYSTEMINDEXES
     WHERE INAME = 'SNUMIDX';
```

Типова структура системної таблиці SYSTEMUSERAUTH, що містить інформацію про користувачів, така:

- username – ідентифікатор доступу користувача;
- password – пароль користувача, що вводиться при реєстрації;
- resource - об'єкти, де користувач має права RESOURCE;
- dba – об'єкти, де користувач має права адміністратора БД.

Оскільки всі користувачі отримують привілей CONNECT за замовчуванням при реєстрації, то він не описаний в таблиці вище.

Даними цієї таблиці можна скористатися, наприклад, для знаходження всіх користувачів, які мають привілей RESOURCE:

```
SELECT USERNAME
     FROM SYSTEMUSERAUTH
     WHERE RESOURCE;
```

Таблиця SYSTEMTABAUTH містить привілеї об'єкту, за винятком привілеїв стовпців. Традиційна структура цієї таблиці наступна:

- username – користувач, який має привілеї;
- grantor – користувач, який передав привілеї;
- tname – ім'я таблиці, в якій існують привілеї;
- owner – ім'я власника таблиці tname;
- selauth – чи має користувач привілей SELECT;
- insauth – чи має користувач привілей INSERT;
- delauth – чи має користувач привілей DELETE.

Можливі значення для кожного з перерахованих привілеїв об'єкту (три останні поля) – Y, N, і G, причому G вказує на те, що користувач має привілей з можливістю її передачі.

Перші чотири стовпці цієї таблиці складають первинний ключ. Оскільки UPDATE і REFERENCES є привілеями, які можуть бути певними стовпцями, відомості про них знаходяться в інших таблицях каталога. Крім того, якщо користувач отримує привілеї в таблиці більш ніж від одного користувача, то вони виділяються окремими рядками в цій таблиці, що необхідне для каскадного відстежування при виклику привілеїв. Як приклад приведемо запит, що визначає, які привілеї були передані SA користувачам таблиці STUDENTS:

```
SELECT USERNAME, SELAUTH, INSAUTH, DELAUTH
     FROM SYSTEMTABAUTH
     WHERE GRANTOR = 'SA'
     AND TNAME = 'STUDENTS';
```

SYSTEMCOLAUTH містить дані про привілеї користувачів в полях таблиць. Типова структура таблиці наступна:

- username – користувач, який має привілеї;
- grantor – користувач, який надав привілеї;
- tname – ім'я таблиці, в якій існують привілеї;
- cname – ім'я поля, в якому існують привілеї;
- owner – власник tname;
- updauth – чи має користувач привілей UPDATE в цьому стовпці;
- refauth – чи має користувач привілей REFERENCES в цьому стовпці.

Два останні поля можуть бути в стані Y, N або G, а окремий рядок тут може існувати для кожного поля будь-якої таблиці. Як і у випадку з SYSTEMTABAUTH, той же привілей може бути описаний в більше ніж одному рядку цієї таблиці, якщо вона була передана більше ніж одним користувачем. Для прикладу приведемо запит на визначення стовпців, в яких користувач має привілей REFERENCES:

```
SELECT OWNER, TNAME, CNAME
     FROM SYSTEMCOLAUTH
     WHERE REFAUTH IN ('Y', 'G')
     AND USERNAME = USER;
```

Системна таблиця SYSTEMSYNONS містить інформацію про синоніми для таблиць в БД. Типова структура така:

- synonym – ім'я синоніма;
- synowner – користувач, який є власником синоніма;
- tname – ім'я таблиці, що використовується власником;
- tabowner – ім'я користувача, який є власником таблиці.

Для виводу, наприклад, всіх синонімів таблиці STUDENTS можна скористатися запитом:

```
SELECT *
     FROM SYSTEMSYNONS
     WHERE TNAME = 'STUDENTS';
```

Як і для звичайних таблиць, в системному каталозі допускається використовувати складні варіанти запитів. Оскільки принципово їх використання нічим не відрізняється від запитів до звичайних таблиць, розглянемо всього один приклад. За допомогою приведеного запиту можна відобразити поля таблиць і базові індекси, встановлені для них:

```
SELECT FIRST.TNAME, FIRST.CNAME, INAME, CPOSITION
     FROM SYSTEMCOLUMNS FIRST, SYSTEMINDEXES SECOND
     WHERE FIRST.TABOWNER = SECOND.TABOWNER
     AND FIRST.TNAME = SECOND.TNAME
     AND FIRST.CNUMBER = SECOND.CNUMBER
     ORDER BY 3 DESC, 2;
```

Таким чином, система SQL використовує набір таблиць, що називаються системним каталогом в структурі БД. Дані цих таблиць можуть запрошуватися користувачем, але не модифікуватися, оскільки це може внести безлад в роботу СУБД.