МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

А. В. Боярчук, А. В. Шостак

ОРГАНІЗАЦІЯ БАЗ ДАНИХ

Конспект лекцій

Харків «ХАІ» 2020

УДК 004.65(075.8)
Б86

Рецензенти: д-р техн. наук, проф. Т. О. Говорущенко,
канд. техн. наук, доц. О. О. Гордєєв

Боярчук, А. В.

Б86 Організація баз даних [Текст] : консп. лекцій / А. В. Боярчук, А. В. Шостак. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2020. – 160 с.

ISBN 978-966-662-777-6

Стисло викладено теоретичні питання, пов’язані з організацією і проектуванням баз даних. Подано опис лабораторних робіт з курсу «Організація баз даних», метою проведення яких є набуття практичних навичок з проектування й створення баз даних.

Для студентів очної та заочної форм навчання зі спеціальності «Комп’ютерна інженерія».

Іл. 19. Табл. 25. Бібліогр.: 14 назв

УДК 004.65(075.8)

ISBN 978-966-662-777-6

Лабораторна робота № 1

ПРОЕКТУВАННЯ РОЗПОДІЛЕНОЇ БАЗИ ДАНИХ

Постановка завдання

Спроектувати структуру бази даних (БД):

1) вибрати предметну галузь для проектування БД;

2) розробити інфологічну модель даних з використанням нотації Баркера;

3) спроектувати фізичну модель даних (проектування реалізації);

4) розробити таблиці фізичної моделі даних та описати їх;

5) розробити бізнес-правила.

Письмовий звіт про виконання лабораторної роботи має містити:

1) титульну сторінку, на якій наведено назву лабораторної роботи, прізвище, ім'я, по-батькові, номер групи виконавця, дату складання;

2) постановку задачі та основні вихідні дані для проектування БД;

3) інфологічну модель даних з використанням нотації Баркера;

4) фізичну модель даних;

5) таблиці фізичної моделі даних і їх опис;

6) опис бізнес-правил;

7) висновки (слід відобразити особливості побудови моделей і шляхи подальшої модернізації БД).

Приклад опису таблиці STUDENT

Таблиця STUDENT

Поле	Назва	Тип даних	PRIMARY KEY	FOREIGN KEY	NOT NULL	UNIQUE
ID	Код	tinyint	*		*	*
FAM	Прізвище	nvarchar(10)			*	
OCENKA	Оцінка	tinyint				
ID_GR	Код групи	tinyint		*	*	
YEAR_B	Рік народження	smalldatetime				

Контрольні запитання

1. Які існують моделі БД?
2. У чому полягають переваги й недоліки реляційного підходу?
3. Якими є основні поняття реляційних БД?
4. Якими є властивості відношень?
5. Як задається відношення?

6. Які види ключів існують у відношенні?
7. Які види цілісності існують у реляційних БД?
8. Які типи зв'язків існують у БД?
9. Дайте означення функціональної залежності в БД.
10. Якими є основні цілі нормалізації відношень?
11. Які існують види нормальних форм?
12. Якими є основні характеристики нормальних форм?
13. Назвіть недоліки 1НФ, які усуваються після переходу до 2НФ.
14. Перелічіть недоліки 2НФ, які усуваються після переходу до 3НФ.
15. Укажіть недоліки 3НФ, які усуваються після переходу до форми Бойса – Кодда (BCNF).
16. Назвіть недоліки BCNF, які усуваються після переходу до 4НФ.
17. Якою є мета денормалізації таблиці? Назвіть основні рекомендації щодо денормалізації таблиць.
18. Які типи бінарних зв'язків підтримуються (не підтримуються) у базах даних?
19. Які існують типи обов'язковості зв'язків?

Лекція 1. ОСНОВИ ВИКОРИСТАННЯ БАЗ ДАНИХ

1.1. Означення та основні поняття

Один із провідних фахівців К. Дж. Дейт назвав систему баз даних «комп'ютеризованою системою зберігання записів». Саму ж базу даних можна розглядати як подобу електронної бібліотеки, що складається із величезної кількості карток або записів, якщо користуватися відповідним терміном.

Основні завдання, що виникають під час роботи з БД:

1) зберігання таблиць і записів,

2) керування записами й таблицями.

Отже, БД (або система баз даних) – це набір програмно-апаратних засобів, які вирішують завдання централізованого зберігання й оброблення даних користувача. Система баз даних складається з таких компонент:

1) дані;

2) апаратне забезпечення;

3) програмне забезпечення;

4) користувачі.

Між власне фізичною БД і користувачами БД розташовано рівень програмного забезпечення, так звана система керування БД (СКБД).

СКБД – це сукупність мовних і програмних засобів, призначених для створення, наповнення, оновлення й видалення БД.

Види СКБД:

1) промислові універсального призначення;

2) промислові спеціального призначення;

3) розроблені для конкретного замовника.

За характером використання СКБД поділяють на персональні й такі, що розраховані на багатьох користувачів.

Персональні СКБД забезпечують можливість створення локальних БД, що працюють на одному комп'ютері. До них належать Paradox, dBase, FoxPro, Access та ін.

СКБД, розраховані на багатьох користувачів, дають змогу створювати інформаційні системи, що функціонують в архітектурі клієнт-сервер. До них належать Microsoft SQL Server, Oracle, IBM Db2, MySQL, ElasticSearch та ін. (табл. 1.1).

Таблиця 1.1

Рейтинг популярності СКБД

№ п/п	Виробники СКБД
1	Oracle
2	MySQL
3	Microsoft SQL Server
4	PostgreSQL
5	MongoDB
6	IBM Db2
7	ElasticSearch
8	Redis
9	Microsoft Access
10	Cassandra

Функції СКБД:

1. Безпосереднє керування даними у зовнішній пам'яті.

2. Керування розділами оперативної пам'яті з метою збільшення швидкості роботи БД.

3. Керування транзакціями. Транзакція – це послідовність операцій над БД, розглянутих СКБД як єдине ціле. Поняття транзакції необхідне для підтримки логічної цілісності БД. Механізм транзакцій забезпечує захист БД від апаратних збоїв, можливість доступу багатьох користувачів до даних у дістанційних БД.

4. Журналізація – ведення журналу змін БД з метою підтримки надійності зберігання даних у БД. Журнал та архівна копія БД – основні засоби для відновлення БД.

5. Підтримка мов БД. Мовні засоби сучасних СКБД:

– DDL – Data Definition Language – оператори визначення об’єктів баз даних;

– DML – Data Manipuilation Language – оператори маніпулювання даними;

– DCL – Data Control language – оператори для працювання з правами доступу;

– TCL – Transaction Control Langauge – оператори керування транзакціями.

Користувачів можна поділити на три групи:

1. Прикладні програмісти – відповідають за написання прикладних програм, що використовують базу даних.

2. Кінцеві користувачі – працюють із системами баз даних безпосередньо через робочу станцію або термінал. Кінцевий користувач може отримати доступ до БД з допомогою одного з додатків або скористатися інтегрованим інтерфейсом програмного забезпечення самої системи бази даних, убудованим у БД. У більшості систем є щонайменше один такий додаток: процесор мови запитів, що дає змогу користувачеві вказувати команди або вирази високого рівня для певної СКБД.

3. Адміністратори бази даних – стежать за її роботою і забезпечують безпеку цих даних.

Історичні етапи створення БД

Перший етап – бази даних на великих ЕОМ.

Особливості розвитку цього етапу:

– усі СКБД базуються на потужних мультипрограмних операційних системах (ОС), тому в основному підтримується робота з централізованою БД у режимі розподіленого доступу;

– функції керування розподілом ресурсів в основному здійснюються ОС;

– підтримуються мови низького рівня маніпулювання даними, орієнтовані на навігаційні методи доступу до даних;

– значна роль відводиться адмініструванню даних;

– провадяться роботи з обґрунтування й формалізації реляційної моделі даних і створюється перша система (System R), що реалізує ідеологію реляційної моделі даних;

– виконуються теоретичні роботи з оптимізації запитів і керування розподіленим доступом до централізованої БД, уводиться поняття транзакції;

– обговорюються в друкованих виданнях результати наукових досліджень, що стосуються всіх аспектів теорії і практики БД, результати теоретичних досліджень упроваджуються в комерційні СКБД.

Другий етап – епоха персональних комп'ютерів.

Особливості цього етапу:

– усі СКБД розраховано на створення БД в основному з монопольним доступом. Комп'ютер не підключено до мережі, оскільки БД створювалася для роботи одного користувача (в окремих випадках передбачається послідовна робота декількох користувачів, наприклад, спочатку оператор вносить бухгалтерські документи, а потім головбух визначає проводки, що відповідають первинним документам);

– більшість СКБД мали розвинений і зручний інтерфейс користувача. У багатьох випадках існував інтерактивний режим роботи з БД під час як опису БД, так і проектування запитів. Крім того, більшість СКБД пропонували розвинений і зручний інструментарій для розроблення готових додатків без програмування. Інструментальне середовище складалося з готових елементів програми у вигляді шаблонів екранних форм, звітів, етикеток (Labels), графічних конструкторів запитів, які досить просто можна було зібрати в єдиний комплекс;

– у всіх настільних СКБД підтримувався зовнішній рівень передвідношення реляційної моделі, тобто лише зовнішній табличний вигляд структур даних;

– за наявності високорівневих мов маніпулювання даними типу реляційної алгебри і SQL у настільних СКБД підтримувалися низькорівневі мови маніпулювання даними на рівні окремих рядків таблиць;

– у настільних СКБД не було засобів підтримки посилань і структурної цілісності бази даних. Ці функції мали виконувати додатки, проте неналежний стан засобів розроблення додатків іноді не давав змоги цього зробити. Тому ці функції виконували користувачі, вимагаючи додаткового контролю при введенні й зміненні інформації, що зберігається в БД;

– монопольний режим роботи фактично спричинив виродження функцій адміністрування БД, а в зв'язку з цим і брак інструментальних засобів адміністрування БД;

– порівняно невисокі вимоги до апаратного забезпечення з боку настільних СКБД.

Третій етап – розподілені БД.

Майже всі сучасні СКБД забезпечують підтримку повної реляційної моделі:

– структурної цілісності – допустимими є лише дані, наведені у вигляді відношень реляційної моделі;

– мовної цілісності, тобто мов маніпулювання даними високого рівня (в основному SQL);

– цілісності посилань – контролю за дотриманням цілісності посилань протягом усього часу функціонування системи й гарантій неможливості з боку СКБД порушити ці обмеження;

– більшість сучасних СКБД розраховано на багатоплатформну архітектуру, тобто їх можна використовувати на комп'ютерах з різноманітною архітектурою й операційними системами, при цьому для користувачів доступ до даних на різних платформах майже не різниться;

– необхідність підтримки багатокористувацької роботи з БД і можливість децентралізованого зберігання даних потребували розвитку засобів адміністрування БД з реалізацією загальної концепції засобів захисту даних;

– потреба в нових реалізаціях привела до створення серйозних теоретичних праць з оптимізації реалізації розподілених БД і роботі з розподіленими транзакціями й запитами з упровадженням отриманих результатів у комерційні СКБД;

– щоб не втратити клієнтів, які раніше працювали на настільних СКБД, майже всі сучасні СКБД мають засоби під'єднання клієнтських додатків, розроблених із використанням настільних СКБД, і засоби експорту даних із форматів настільних СКБД другого етапу розвитку.

Четвертий етап – перспективи розвитку систем керування БД:

1. Реляційні системи:

1.1. Стандартизація мови SQL.

1.2. Використання мультипроцесорних організацій.

1.3. Інтеграція й інтероперабільність.

2. Постреляційні системи:

2.1. Бази складних об'єктів, реляційна модель з відмовою від першої нормальної форми.

2.2. Активні бази даних (за означенням БД називають активною, якщо СКБД відносно неї виконує не лише ті дії, які явно вказує користувач, але й додаткові відповідно до правил, закладених у БД).

2.3. Дедуктивні бази даних (за означенням дедуктивна БД складається з двох частин: екстенсіональної, що містить факти, та інтенсіональної, що містить правила для логічного висновку нових фактів на основі екстенсіональної частини й запиту користувача).

2.4. Темпоральні бази даних.

2.5. Інтегровані, або федеративні, системи й мультибази даних.

2.6. СКБД наступного покоління.

2.7. Об'єктно-орієнтовані бази даних.

3. Розподілені СКБД:

3.1. Синхронізація доступу до даних.

3.2. Керуваня транзакціями.

3.3. Підтримка копій даних у кількох вузлах мережі.

3.4. Фрагментація об'єктів БД.

3.5. Алгоритми виконання реляційних операцій.

4. Системи БД з багаторівневим захистом.

1.2. Архітектури БД

Залежно від місця розташування окремих частин СКБД розрізняють локальні (автономні) і дистанційні (розраховані на багато користувачів, або мережні) БД.

До локальних БД належать Paradox, dBase, FoxPro, Access та ін. Локальну архітектуру БД зображено на рис. 1.1.

БД з локальною архітектурою використовують у додатках, які обробляють бухгалтерську або іншу документацію невеликої фірми, кадровий склад малого підприємства. Кожен користувач такого додатка маніпулює власними даними на комп'ютері.

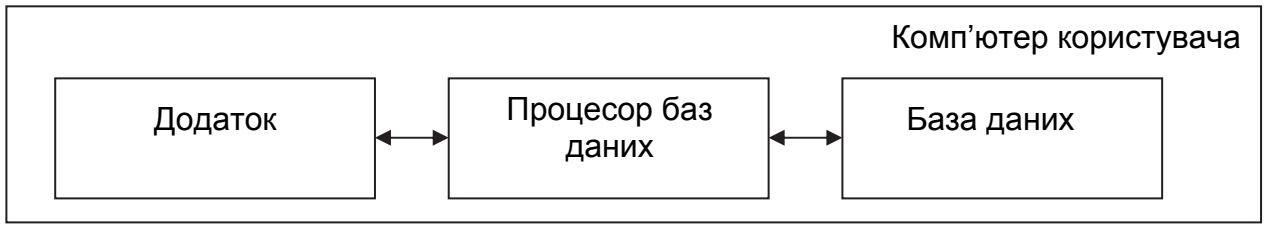


Рис. 1.1. Локальна архітектура бази даних

До віддалених БД належать MySQL, PostgreSQL, Microsoft SQL Server, Oracle, IBM Db2 та ін. (рис. 1.2).

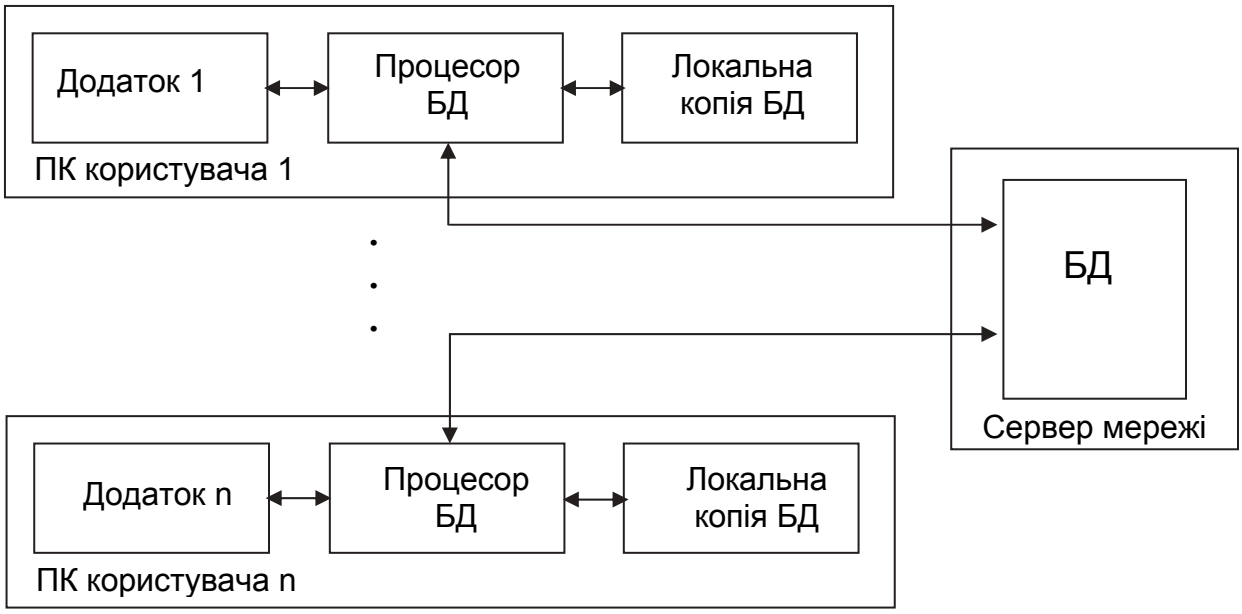


Рис. 1.2. Файл-серверна архітектура бази даних

БД зберігається на мережному файл-сервері в єдиному екземплярі. Для кожного клієнта під час роботи створюється локальна копія БД (що періодично оновлюється), якою користувач маніпулює.

Перевага такої архітектури – можливість одночасної роботи кількох користувачів з однією БД.

Недоліки архітектури:

1) непродуктивне завантаження мережі – оновлення локальної копії БД, існує блокування на рівні таблиці;

2) турбота про цілісність даних покладається на програми користувачів; якщо їх недостатньо ретельно продумано, то в БД можна легко занести помилки;

3) складності з організацією захисту інформації в БД.

Дистанційна база даних у кліент-серверній архітектурі (рис. 1.3) розміщується на потужному віддаленому сервері мережі, а додаток, що працює з цією БД, знаходиться в комп'ютері користувача. Інформаційна система складається з двох неоднорідних частин – віддаленого сервера мережі й клієнтів БД (клієнт – це програма користувача).

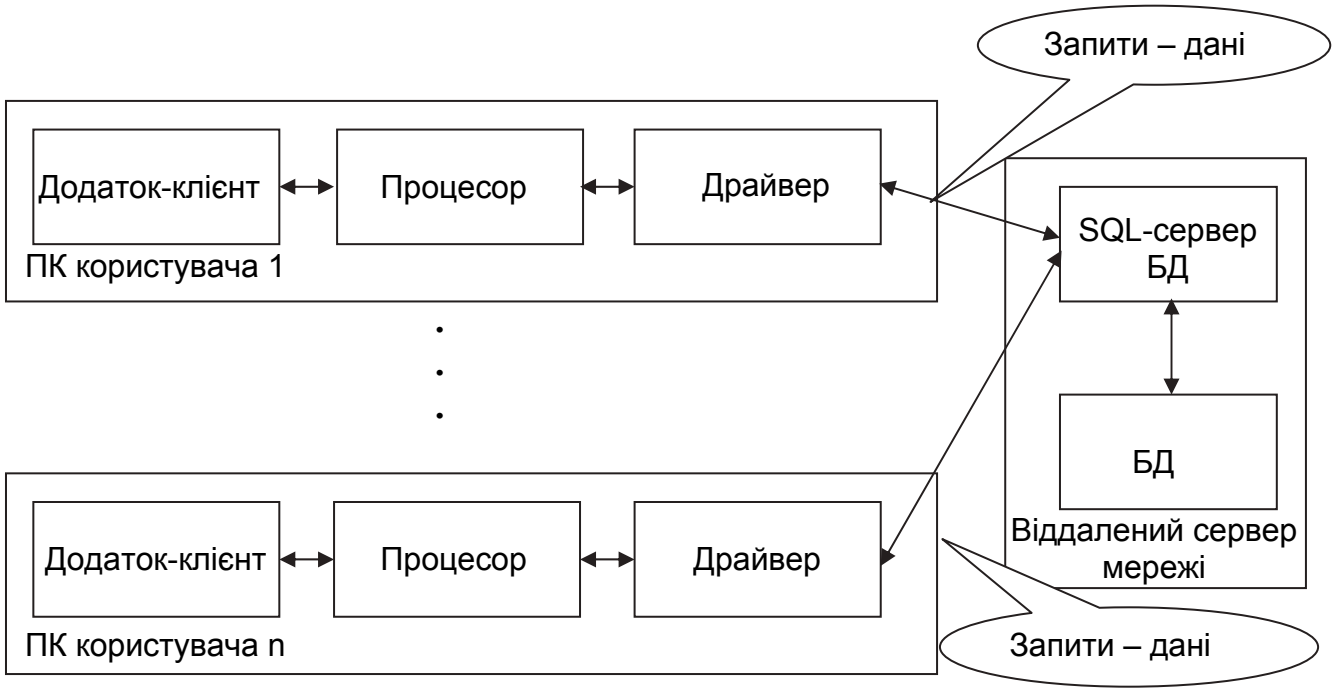


Рис. 1.3. Клієнт-серверна архітектура бази даних

Переваги архітектури:

1) висока ефективність виконання додатка завдяки використанню потужного сервера;

2) низьке навантаження на мережу, у якій циркулюють лише запити й дані;

3) безпека інформації, тому що оброблення запитів усіх клієнтів виконується єдиною програмою, розташованою на сервері;

4) у клієнтських додатках немає коду, що забезпечує керування БД і розмежування доступу в ній;

5) клієнт-серверні СКБД припускають блокування на рівні запису й навіть окремого поля, тобто з таблицею можуть працювати одночасно багато користувачів, але доступ до функції змінення конкретного запису й поля забезпечено лише одному з них.

Для реалізації архітектури клієнт-сервер зазвичай застосовують розраховані на багато користувачів СКБД – Microsoft SQL Server або Oracle.

Розподілена (багатоярусна) архітектура (рис. 1.4). У мережі є кілька віддалених серверів, і таблиці БД розподілено між ними для досягнення підвищеної ефективності. На кожному сервері функціонує своя копія СКБД. Крім того, у такій архітектурі зазвичай застосовують спеціальні програми – сервери додатків, які дають змогу оптимізувати оброблення запитів великої кількості користувачів і рівномірно розподіляти навантаження між ПЕОМ мережі. Найбільш поширеним є триярусний варіант архітектури (триланкова архітектура).

Переваги архітектури:

– розподілена архітектура БД;

– найбільш складна, надійна, живуча й гнучка організація баз даних.

Недоліки:

– складний і дорогий процес створення й супроводження БД;

– високі вимоги до серверних комп'ютерів.

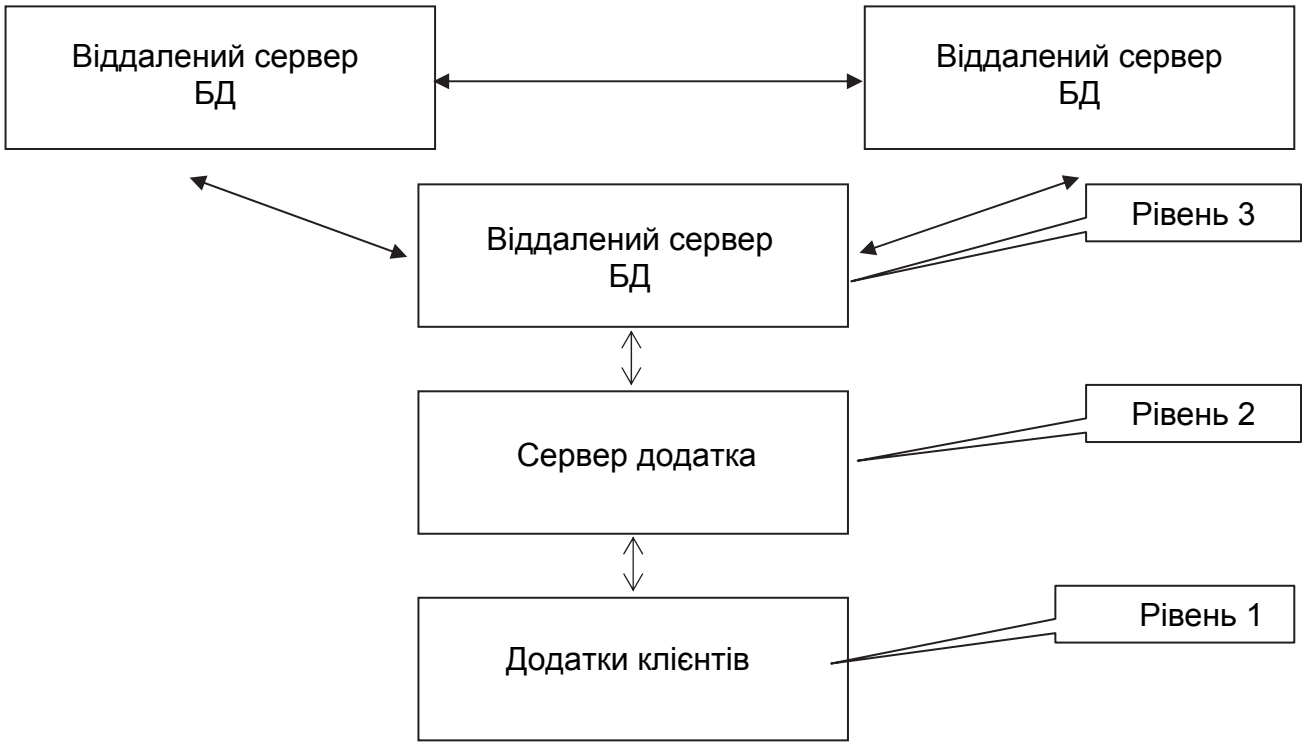


Рис. 1.4. Розподілена архітектура бази даних

На рис. 1.5 показано одно-, дво- і триланкові архітектури з'єднання клієнта з даними.

В одноланковій архітектурі використовують одну ланку (клієнт), що забезпечує необхідну логіку керування даними та їх візуалізацію, у дволанковій – значну частину логіки керування даними бере на себе

сервер БД, тоді як клієнт в основному зайнятий відображенням даних у зручному для користувача вигляді.

Рис. 1.5. Архитектури з’єднання клієнта
з даними: одноланкова, дволанкова, триланкова

У триланкових СКБД є проміжна ланка – сервер додатків, що є посередником між клієнтом і сервером БД. Сервер додатків призначено для повного позбавлення клієнта від будь яких турбот щодо керування даними й забезпечення зв'язку із сервером БД.

Інтернет-архітектура – використання стандартних протоколів інтернету для обміну даними.

Класифікація БД за структурою організації даних

БД містить дані, які використовує деяка прикладна система (додаток). Під моделлю даних розуміють інтегрований набір понять для опису даних, зв'язків між ними й обмежень, що накладаються на дані в деякій їх організації. Залежно від виду організації даних розрізняють ієрархічну, мережну, об'єктно-орієнтовану, реляційну моделі БД.

1. В ієрархічній БД дані наведено у вигляді дерева. Така структура БД є зручною для роботи з даними, упорядкованими ієрархічно. Однак під час оперування даними зі складними логічними зв'язками ця модель є дуже громіздкою.

Переваги ієрархічної моделі:

– наявність СКБД що найкраще виявили себе в роботі;

– зручність і простота подання даних, які мають природну ієрархічну структуру.

Недоліки:

– складність операцій додавання даних й видалення;

– взаємозв'язок М : М реалізується громіздко і з великою надмірністю;

– низький рівень незалежності програмного забезпечення від даних порівнянно з іншими моделями.

2. У мережній моделі кожен вузол (набір) БД взаємодіє з іншими вузлами з допомогою складної структури зв'язків, тобто дані в БД організовуються у вигляді графа.

Переваги мережної моделі:

– безпосередня навігація за зв'язаними даними;

– використання множинних типів даних для опису атрибутів об'єктів (записів);

– дає змогу якнайкраще відображати інфологічні зв'язки складних предметних галузей.

Недоліки:

– складність моделі, жорсткість її структури;

– немає вдалої (універсальної) реалізації мови опису даних.

3. Об'єктно-орієнтовані БД об'єднують мережну й реляційну моделі та використовуються для створення великих БД з даними складної структури. У такій БД зберігаються не лише дані, але й методи їх оброблення у вигляді програмного коду. Ця модель поки не набула активного поширення через складність створення й застосування подібних СКБД.

4. Реляційна БД (relation – відношення, зв'язок) являє собою сукупність таблиць, зв'язаних відношенями.

Переваги реляційної моделі: простота, гнучкість структури, зручність реалізації на ПЕОМ, теоретична основа – теорія відношень реляційної алгебри. Перелічені характеристики забезпечують найвищий ступінь логічної незалежності даних.

Більшість сучасних БД – реляційні.

Недолік реляційної моделі – роботоздатність наявних реляційних СКБД є нижчою, ніж СКБД інших типів.

Контрольні запитання

1. Наведіть означення бази даних.
2. Якими є компоненти системи баз даних?
3. Наведіть означення системи керування базою даних та її основні функції.
4. Наведіть класифікацію архітектур баз даних.
5. Якими є основні особливості, переваги й недоліки клієнт-серверної архітектури бази даних?
6. Якими є основні особливості, переваги й недоліки розподіленої архітектури бази даних?
7. Наведіть класифікацію баз даних за структурою організації даних.
8. Якими є основні особливості, переваги й недоліки реляційної бази даних?

Лекція 2. ОСНОВНІ ПОНЯТТЯ РЕЛЯЦІЙНОЇ МОДЕЛІ ДАНИХ

2.1. Відношення та їх властивості. Домени

У реляційній моделі використовують математичну теорію відношень (від лат. relation – відношення, зв'язок).

Переваги реляційного підходу:

- наявність невеликого набору абстракцій, які дають змогу порівняно просто моделювати більшу частину поширених предметних галузей та мають точні формальні означення, залишаючись інтуїтивно зрозумілими;
- наявність простого й водночас потужного математичного апарату, що спирається переважно на теорію множин і математичну логіку та забезпечує теоретичний базис реляційного підходу до організації баз даних;
- можливість ненавігаційного маніпулювання даними без необхідності знання конкретної фізичної організації баз даних у зовнішній пам'яті;
- простота реляційної БД, гнучкість її структури й зручність реалізації БД на ЕОМ.

Недоліки реляційного підходу:

- недостатня ефективність реляційних СКБД;
- деяка обмеженість (прямий наслідок простоти), влатива цим системам, при використанні у нетрадиційних галузях (найбільш поширені – системи автоматизації проектування), для яких потрібні гранично складні структури даних;
- неможливість адекватного відображення семантики (сенсу) предметної галузі.

Згідно з К. Дейтом, реляційна модель складається з трьох частин, що описують різні аспекти реляційного підходу: структурної, маніпуляційної та цілісної.

Структурна частина моделі має єдину структуру даних, яка використовується у реляційних БД, – нормалізоване n-арне відношення.

Цілісна частина описує обмеження спеціального виду, які мають виконуватися для будь-яких відношень реляційних БД. Цілісність даних – це механізм підтримки відповідності бази даних предметній галузі. У реляційній моделі даних існує цілісність посилань і сутностей.

У маніпуляційній частині моделі затверджуються два фундаментальних механізми маніпулювання реляційними БД: реляційна алгебра (базується на теорії множин) і реляційне обчислення (базується на апараті обчислення предикатів першого порядку).

Структурна частина реляційної моделі

Основні поняття реляційних баз даних – тип даних, домен, атрибут, кортеж, первинний ключ і відношення.

Відношення зручно подавати у вигляді таблиць. На рис. 2.1 показано таблицю T (відношення «Співробітники» ступеня 5), що містить деякі відомости про працівників гіпотетичного підприємства. Рядки таблиці відповідають кортежам. Кожен рядок – це опис одного об'єкта реального світу (у цьому випадку працівника), характеристики якого містяться у стовпцях. Стовпці таблиці називають атрибутами.

Кожен атрибут є визначеним на домені, тому останній можна розглядати як множину допустимих значень певного атрибута.

Відношення T

Ціле	Рядок		Ціле		Типи даних
Номер	Ім'я	Посада	Гроші		Домени
Табельний номер	Ім'я	Посада	Оклад	Премія	Атрибути
24	Іванов	технік	6000	600	Кортежи
25	Петров	інженер	7500	750	
26	Сидоров	бухгалтер	5900	590	

Рис. 2.1. Основні компоненти реляційного відношення «Співробітники»

Домен – це семантичне (смислове) поняття. Його можна розглядати як підмножину значень деякого типу даних, що мають певний сенс.

Властивості домену:

– має унікальне ім'я (у межах БД);

– визначений на деякому простому (атомарному) типі даних або на іншому домені;

– може мати деяку логічну умову, що дає змогу описати підмножини даних, допустимих для певного домену (домен *D*, який має значення «вік співробітника», можна описати як підмножину множини натуральних чисел: $D = \{ n \in N : n \geq 18 \text{ and } n \leq 60 \}$);

– має смислове навантаження.

Кілька атрибутів одного відношення і навіть атрибути різних відношень можна задати на одному й тому самому домені. На рис. 2.1 атрибути «Оклад» і «Премія» визначено на домені «Гроші». Поняття домену має смислове навантаження: дані можна вважати порівняльними лише тоді, коли вони належать до одного домену. У розглянутому прикладі порівняння атрибутів «Табельний номер» і «Оклад» є семантично некоректним, хоча вони й містять дані одного типу.

Означення 1. Атрибутом відношення є пара <ім'я_атрібуту: ім'я_домену>.

Імена атрибутів мають бути унікальними в межах відношення. Часто імена атрибутів відношення збігаються з іменами відповідних доменів.

Означення 2. Відношення R, задане на множині необов'язково різних доменів D_1, D_2, ..., D_n, складається з двох частин – заголовка й тіла:

1. Заголовок містить фіксовану множину атрибутів або, точніше, пар <ім'я_атрибута: ім'я_домену>: {< A_1: D_1 >, ..., < A_n: D_n >}, причому кожен атрибут A_j відповідає одному й лише одному з тих, на якому базується домен D_j (j = 1, 2, ..., n).

2. Тіло відношення містить множину кортежів, а кожний кортеж – множину пар <ім'я_атрибута: значення_атрибута>: {< A_1: V_1 >, ..., <A_n: V_n>}. У кожному кортежі є одна така пара <ім'я_атрибута: значення_атрибута>, тобто < A_j: V_j > для кожного атрибута A_j у заголовку. Для будь-якої пари < A_j: V_j > V_j належить домену D_j, який пов'язаний з атрибутом A_j. Потужність множини кортежів m називають кардинальним числом (або потужністю) відношення R .

3. Відношення зазвичай записується у вигляді $R(A_1, A_2, ..., A_n)$ або ще коротше – R.

Означення 3. Реляційною базою даних називають набір відношень.

Означення 4. Схемою реляційної бази даних називають набір заголовків відношень, що належать до БД.

У табл. 2.1 наведено «табличні» синоніми термінів реляційної моделі даних.

Таблиця 2.1

«Табличні» синоніми термінів реляційної моделі даних

Реляційний термін	«Табличний» термін
База даних	Набір таблиць
Схема бази даних	Набір заголовків таблиць
Відношення, сутність	Таблиця
Схема відношення	Заголовок таблиці
Домен	Загальна сукупність допустимих значень
Заголовок відношення	Заголовок таблиці
Тіло відношення	Тіло таблиці
Атрибут відношення	Найменування стовпця таблиці, поле
Кортеж відношення	Рядок (або запис) таблиці, екземпляр сутності
Ступінь відношення	Кількість стовпців таблиці
Потужність відношення (кардинальне число)	Кількість рядків таблиці
Первинний ключ	Унікальний ідентифікатор

Фундаментальні властивості відношень

1. Немає кортежів-дублікатів**.** З цієї властивості випливає, що кожний перший кортеж має первинний ключ. Для кожного відношення принаймні повний набір його атрибутів є первинним ключем. При визначенні первинного ключа необхідно дотримуватися вимоги мінімальності кількості атрибутів у ключі.

Відношення не містять кортежів-дублікатів. Ця властивість випливає з означення відношення як множини кортежів.

2. Немає впорядкованості кортежів. Ця властивість випливає з означення відношення-екземпляра як множини кортежів.

3. Немає впорядкованості атрибутів. Атрибути відношень не впорядковано, оскільки за означенням схема відношення є множиною пар {ім'я атрибута : ім'я домену}. Для посилання на значення атрибута в кортежі відношення завжди використовується ім'я атрибута. Ця властивість дає змогу, наприклад, модифікувати схеми наявних відношень шляхом не лише додавання нових атрибутів, але й видалення наявних атрибутів.

4. Атомарність значень атрибутів. Значення всіх атрибутів – атомарні, тобто належать до простих типів даних без внутрішньої структури (точніше, внутрішня структура атрибутів не враховується). Це випливає з означення домену як потенційної множини значень простого типу даних, тобто серед значень домену не можуть міститися множини значень (відношень).

Із властивостей відношень випливає, що не кожна таблиця може задавати відношення. Для того щоб таблиця задавала відношення, необхідно дотримуватися таких умов:

1) структура має бути простою (лише рядки й стовпці, причому в кожному рядку має бути однакова кількість полів);

2) не повинно бути однакових рядків;

3) будь-який стовпець має містити дані лише одного типу;

4) усі використовувані типи даних мають бути простими.

Таблицю T (див. рис. 2.1) можна розглядати як зображення відношення (в сенсі визначення відношення), якщо виконуються такі умови:

– є деякі домени, на яких базуються стовпці;

– кожен стовпець відповідає лише одному з цих доменів;

– кожен рядок є кортежем;

– кожне значення атрибута належить до відповідного домену.

Якщо всі правила інтерпретації прийнято, то тоді і лише тоді можна вважати, що таблиця – це прийнятне зображення відношення.

Приклад таблиці, що є відношенням:

25	Іванов	Іван
28	Петров	Семен
30	Сєров	Петро

Приклад таблиць, що не є відношеннями:

25	Іван	Іванов
28	Петров	Семен
Петро	Сєров	30

25	Іванов	Іван
28	Семен	Петров
30	Сєров Петро	

2.2. Поняття ключа у відношенні

Первинний ключ (PRIMARY KEY, PK) – це унікальний ідентифікатор для деякого відношення. Однак первинний ключ є окремим випадком більш загального поняття потенційного ключа.

Означення 5. Нехай R – деяке відношення. Тоді потенційний ключ, наприклад *K*, для *R* – це підмножина множини атрибутів *R*, що має такі властивості:

1. Унікальність. Немає двох різних кортежів у відношенні *R* з однаковим значенням *K*.

2. Ненадмірність. Ніяка з підмножин *K* не має властивості унікальності.

Отже, кожне відношення має хоча б один потенційний ключ, оскільки не містить однакових кортежів, тобто оскільки кортежі є унікальними, то принаймні комбінація всіх атрибутів має властивість унікальності й тому можливими є два варіанти:

1) ця комбінація має властивість ненадмірності, тобто буде потенційним ключем;

2) існує принаймні одна відповідна підмножина цієї комбінації, яка безпреречно має властивості унікальності й ненадмірності.

На практиці як первинний ключ використовують не всі атрибути, а лише декілька. Якщо кортежі ідентифікуються лише зчепленням значень декількох атрибутів, то відношення має складний ключ.

Як первинний ключ також часто використовують сурогатний (штучний) ключ, який не є властивістю сутності, але є унікальним і його просто генерувати.

Причина важливості потенційних ключів полягає в тому, що вони забезпечують основний механізм адресації на рівні кортежів у реляційній

системі. Отже, єдиний гарантований спосіб точно вказати на будь-який кортеж – це обчислити значення потенційного ключа.

Із цього означення можна зробити такі висновки:

1. Відношення, які не мають потенційних ключів (тобто дають змогу дублювати кортежі), обмежують відображення порушень або відхилень від нормального режиму роботи за наявності конкретних обставин.

2. Система, де не використовуються потенційні ключі, іноді є обмеженою в можливостях відображення стану, який не є дійсно реляційним.

Первинні й альтернативні ключі

Базове відношення може мати більше одного потенційного ключа.

Базове відношення – це таке відношення, що реально існує в БД, а не виникає внаслідок виконання запиту. У цьому випадку один з потенційних ключів необхідно вибрати як первинний у базовому відношенні, а інші, якщо вони є, будуть альтернативними (можливими) ключами. Як первинний ключ (див. рис. 2.1) можна вибрати атрибут «Табельний номер», оскільки його значення є унікальним для кожного працівника підприємства. Тоді атрибут «Ім'я» буде альтернативним ключем.

Отже, кожне базове відношення завжди повинно мати первинний ключ.

Приклад. На рис. 2.2 зображено множину потенційних ключів деякого відношення. Очевидно, що множина потенційних ключів відношення {*K*} дорівнює множині альтернативних ключів {*AltK*}, об'єднаній з первинним ключем {*PK*}, тобто {*K*} = {*AltK*}∪{*PK*}.

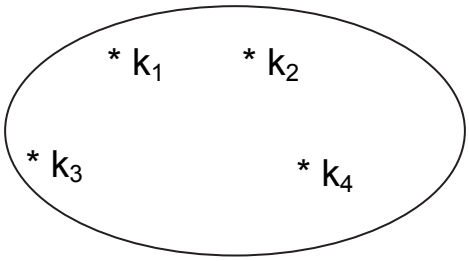


Рис. 2.2. Множина потенційних ключів відношення {*K*}

Зовнішні ключі

Для відображення зв'язків між кортежами різних відношень використовують дублювання їх ключів. Наприклад, відомості про підрозділи підприємства та їх співробітників стосовно реляційної моделі показано на рис. 2.3.

Зв'язок між відношенями ВІДДІЛ і СПІВРОБІТНИК створюється шляхом копіювання первинного ключа «Номер_відділу» з першого відношеня в друге. Отже, для того щоб отримати перелік працівників цього відділу, необхідно:

– з таблиці ВІДДІЛ установити значення атрибута «Номер_відділу», що відповідає атрибуту «Найменування_відділу»;

– вибрати з таблиці СПІВРОБІТНИК усі записи, значення атрибута «Номер_відділу» яких дорівнює значенню, отриманому на попередньому кроці.

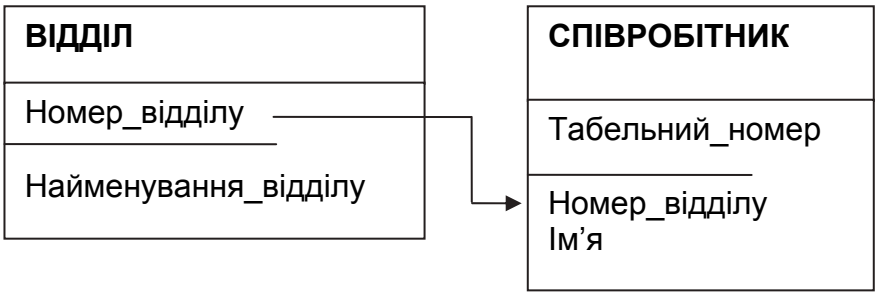


Рис. 2.3. База даних про підрозділи й співробітників підприємства (фрагмент)

Щоб дізнатись, у якому відділі працює співробітник, необхідно виконати зворотну операцію:

– визначити «Номер_відділу» з таблиці СПІВРОБІТНИК;

– знайти за отриманим значенням запис у таблиці ВІДДІЛ.

Атрибути, що являють собою копії ключів (потенційних або зазвичай первинних) інших відношень, називають зовнішніми ключами (FOREIGN KEY, FK).

У реляційній моделі зовнішні ключі можуть посилатися лише на первинні ключі.

Визначення зовнішнього ключа: нехай *R*2 – базове відношення, тоді зовнішній ключ *FK* відносно *R*2 – це підмножина множини атрибутів R2:

1) існує базове відношення *R*1 з потенційним ключем *CK* (вторинним ключем);

2) кожне значення *FK* у поточному значенні *R*2 завжди збігається зі значенням *CK* деякого кортежу в поточному значенні *R*1.

Зовнішній ключ має належати тому ж домену, що й первинний. Непервинний ключ може бути зовнішнім. Ключ *CK* називають вторинним ключем, а ключ *CK* і зовнішній ключ *FK* – полями зв'язку відношень *R*1 і *R*2. Вторинний і первинний ключі в загальному випадку можуть не збігатися.

2.3. Цілісність даних

Цілісність даних – це механізм підтримки відповідності бази даних предметній галузі.

У реляційній моделі даних існує:

1) цілісність сутностей;
2) цілісність посилань.

Вимога до БД полягає в тому, що будь-який кортеж будь-якого відношення БД має відрізнятися від будь-якого іншого кортежу цього відношення, тобто будь-яке відношення повинно мати первинний ключ. Ця вимога автоматично задовольняється, якщо в системі не порушуються базові властивості відношень.

Цілісність сутностей

Вимога цілісності сутностей полягає в такому: кожен кортеж будь-якого відношення має відрізнятися від будь-якого іншого кортежу цього відношення (тобто будь-яке відношення повинно мати первинний ключ).

Цілком очевидно, що якщо кортежі в межах одного відношення не є унікальними, то в БД може зберігатися суперечлива інформація про один і той самий об'єкт. Підтримка цілісності сутностей забезпечується засобами СКБД з допомогою двох обмежень:

– під час додавання записів до таблиці перевіряють унікальність їх первинних ключів;

– не змінюють значення атрибутів, що входять до первинного ключа.

Визначник NULL (логічна величина «невідомо») указує, що значення атрибута на певний час є невідомим або неприйнятним для цього кортежу. Отже, або це значення не входить до деякого кортежу, або ніякого значення ще не задано. Ключове слово NULL є способом оброблення неповних або незвичайних даних.

Цілісність сутностей полягає в тому, що в базовому відношенні жоден атрибут первинного ключа не може містити NULL-значення. Якщо припустити наявність визначника NULL у будь-якій частині первинного ключа, то це рівноцінно твердженню, що не всі його атрибути є необхідними для унікальної ідентифікації кортежів, що суперечить означенню первинного ключа.

Цілісність посилань

Об'єкти реального світу наведено в реляційній БД у вигляді кортежів декількох відношень, пов'язаних між собою:

1. Зв'язки між даними відношення описуються в термінах функціональних залежностей.
2. Для відображення функціональних залежностей між кортежами різних відношень використовують дублювання первинного ключа одного відношення (батьківського) в інше (дочірнє). Атрибути, що являють собою копії ключів батьківських відношень, називають зовнішніми ключами.

Вимога цілісності посилань полягає в такому: для кожного значення зовнішнього ключа, що виникають у дочірньому відношенні, у батьківському відношенні має бути кортеж з таким самим значенням первинного ключа (рис. 2.4).

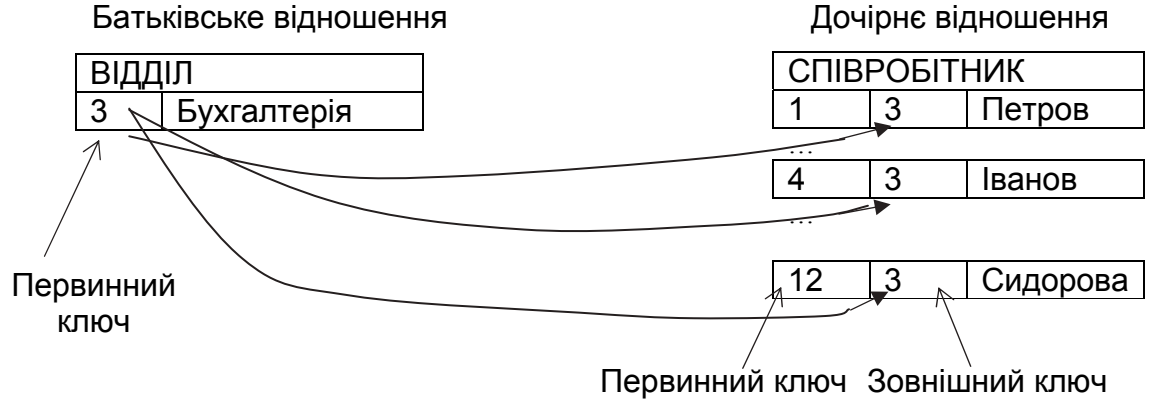


Рис. 2.4. Зв'язок між відношеннями «ВІДДІЛ» і «СПІВРОБІТНИК» типу «один до багатьох» (1 : М)

Вимога цілісності посилань означає, що БД не повинна містити неузгоджених значень зовнішніх ключів. Неузгоджене значення – це значення зовнішнього ключа, для якого не існує відповідного значення потенційного ключа у відповідному батьківському відношенні. Отже, якщо *B* посилається на *A*, то *A* має існувати. Тому підтримка зовнішніх ключів і підтримка цілісності посилань – одне й те саме.

Зазвичай підтримка цілісності посилань також покладається на СКБД. Наприклад, ця система може не дозволити користувачеві додати запис, що містить зовнішній ключ з NULL-значенням.

Вимога цілісності за посиланнями (або узгодженого зовнішнього ключа) полягає в тому, що для кожного значення зовнішнього ключа, який виникає у відношенні, що посилається, у відношенні, на яке вказує посилання, має бути кортеж із таким самим значенням первинного ключа або значення зовнішнього ключа має бути невизначеним (тобто ні на що не вказувати).

Наприклад, якщо для співробітника вказано номер відділу, то цей відділ повинен існувати (див. рис. 2.4).

Під час оновлення відношення, що посилається (додавання нових кортежів або модифікації значення зовнішнього ключа в наявних кортежах) досить стежити за тим, щоб не виникали некоректні значення зовнішнього ключа. Але як бути при видаленні кортежу з відношення, на яке вказує посилання?

Існують три підходи, кожен з яких підтримує цілісність за посиланнями.

1. Забороняється видаляти кортеж, на який існують посилання (тобто спочатку потрібно або видалити кортежі, які посилаються, або змінити значення їх зовнішнього ключа – RESTRICT).

2. При видаленні кортежу, на який є посилання, у всіх кортежах, які посилаються, значення зовнішнього ключа автоматично стає невизначеним – SET NULL.

3. При видаленні кортежу з відношення, на яке вказує посилання, з відношення, яке посилається автоматично видаляються всі кортежі, які посилаються (каскадне видалення – CASCADE).

У реляційних СКБД зазвичай можна вибрати спосіб підтримки цілісності за посиланнями для кожної окремої ситуації визначення зовнішнього ключа.

Корпоративні обмеження цілісності **–** це додаткові правила підтримки цілісності даних, що встановлюють користувачі БД. Наприклад, навчальний відділ відповідно до навчального навантаження визначив кількість викладачів кожної з кафедр. Перевищіти цю кількість не можна, але можна зарахувати двох викладачів на півставки на одну посаду.

Контрольні запитання

1. Обґрунтувати місце реляційних баз даних серед інших їх типів.
2. Переваги реляційних баз даних.
3. Якими є недоліки реляційних баз даних?
4. Дайте означення основних понять реляційних баз даних (відношення, домен, атрибут, кортеж, схема бази даних, арність відношення, потужність відношення, кардинальне число, потенційний ключ, первинний ключ, альтернативний ключ, зовнішній ключ).
5. Що таке реляційна модель? Перелічіть основні її складові.
6. Що таке цілісність сутностей і цілісність посилань?
7. Що таке корпоративні обмеження цілісності?

Лекція 3. НОРМАЛЬНІ ФОРМИ ВІДНОШЕНЬ

3.1. Функціональні залежності

Реляційна база даних містить як структурну, так і семантичну інформацію. Структура бази даних визначається кількістю й типом наявних у них відношень і зв'язків типу «один до багатьох», що існують між кортежами цих відношень. Семантична частина описує множину функціональних залежностей між їх атрибутами.

Концепція функціональної залежності – основа в теорії реляційних баз даних. По суті функціональна залежність – це зв'язок типу «багато до одного» між множинами атрибутів усередині певного відношення. Однак потрібно розрізняти поняття значення відношення і набір усіх можливих

значень. Спочатку визначимо функціональну залежність для значення відношення.

У відношенні *R* атрибут *Y* функціонально залежить від атрибута *X* (*X* і *Y* можуть бути складовими) у тому й лише у тому випадку, якщо кожному значенню *X* відповідає точно одне значення *Y*: *R* **.** *X* (*r*) *R* **.** *Y*.

Отже, якщо задано два атрибути *X* і *Y* деякого відношення і в будь-який момент часу кожному значенню *X* відповідає одне значення *Y,* то *Y* функціонально залежить від *X*.

Функціональна залежність позначається $X \rightarrow Y$ (і читається так: *X* функціонально визначає *Y*, або *X* стрілка *Y*).

Отже, функціональні залежності являють собою зв'язки типу «один до багатьох», що існують усередині відношення.

Деякі функціональні залежності можуть бути небажаними.

Якщо два кортежі відношення *R* збігаються за значенням *X*, то вони так само збігаються й за значенням *Y*.

Розглянемо відношення *SCP*, наведене в табл. 3.1.

Відношення SCP задовольняє наведеній нижче функціональній залежності, оскільки всі кортежі відношення з однаковим значенням атрибута S # мають однакове значення атрибута CITY:

$\{S\ \#\} \rightarrow \{CITY\}$

Таблиця 3.1

Відношення *SCP*

S # – номер постачальника	CITY – місто	P # – товар	QTY – кількість товару в одній поставці
S1	Харків	P1	100
S1	Харків	P2	100
S2	Київ	P1	200
S2	Київ	P2	200
S3	Київ	P2	300
S4	Харків	P2	400
S4	Харків	P4	400
S4	Харків	P5	400

Насправді, це відношення задовольняє декільком функціональним залежностям:

$\{ S\#, P\# \} \rightarrow \{ QTY \}$
$\{ S\#, P\# \} \rightarrow \{ CITY \}$
$\{ S\#, P\# \} \rightarrow \{ CITY, QTY \}$
$\{ S\#, P\# \} \rightarrow \{ S\# \}$
$\{ S\#, P\# \} \rightarrow \{ S\#, P\#, CITY, QTY \}$

Ліву й праву сторони символічного запису функціональної залежності іноді називають детермінантою і залежною частиною. З означення видно, що детермінант і залежна частина – множини атрибутів. Коли множина містить один атрибут, її називають одноелементною, дужки опускають і символічний запис набуває вигляду S # → CITY.

Як було зазначено вище, ці функціональні залежності належать до значень відношення. Однак при розгляді змінних відношення, наприклад базових, інтерес становлять не лише функціональні залежності для певного значення в певний момент часу, а й такі, що виконуються для всіх можливих значень певної змінної. Наприклад, у відношенні SCP функціональна залежність S # → CITY виконується для всіх можливих значень *SCP*, оскільки в будь-який момент часу певному постачальнику відповідає точно одне місто. Таким чином, будь-які два кортежі відношення *SCP* в один і той самий момент часу і з одним і тим самим номером постачальника мають відповідати одному й тому самому місту. Твердження, що ця функціональна залежність виконується завжди, є обмеженням цілісності для відношення *SCP*, тому що при цьому накладаються певні обмеження на всі допустимі значення.

Означення. Функціональна залежність – це залежність для набору всіх можливих значень відношень.

Нехай *R* – змінна відношення, а *X* і *Y* – довільні підмножини множини атрибутів відношення *R*. Тоді *Y* функціонально залежить від *X* (що в символічному вигляді записується як *X* → *Y*) тоді й лише тоді, коли для будь-якого допустимого значення відношення *R* кожне значення *X* пов'язане точно з одним значенням *Y*.

Інакше, якщо два кортежі відношення *R* для будь-якого допустимого значення *R* збігаються за значенням *X*, то вони будуть збігатися й за значенням *Y*.

Отже, правильніше використовувати термін «функціональна залежність» в останньому безвідносному до часу сенсі.

Наведемо деякі безвідносні залежності для змінної відношення *SCP*:

{S#, P#} → QTY
{S#, P#} → CITY
{S#, P#} → {CITY, QTY}
{S#, P#} → S#
{S#, P#} → {S#, P#, CITY, QTY}
S# → CITY

Слід звернути увагу на те, що частина функціональних залежностей у цьому випадку не виконується:

S# → QTY
QTY → S#

Наприклад, таке твердження, як кількість деталей (товару) для кожної поставки певного постачальника є однаковою, справджується для

значень табл. 3.1, але не справджується для всіх можливих допустимих значень відношення *SCP*.

Слід зазначити, що якщо *X* – потенційний ключ відношення *R*, наприклад, *X* є первинним ключом, то всі атрибути *Y* відношень *R мають* бути функціонально залежними від *X*.

Одна з важливих цілей – зменшення до мінімуму кількості функціональних залежностей усередині відношення. Причина цього полягає в тому, що функціональні залежності є обмеженнями цілісності, тому при кожному оновленні даних у СКБД усі їх необхідно перевірити. Отже, для заданої множини функціональних залежностей *S* бажано знайти таку множину *T*, яка би була набагато меншого розміру, ніж множина *S*, причому кожну функціональну залежність множини *S* можна замінити функціональною залежністю множини *T*. Тому завдання пошуку відповідної множини *T* становить практичний інтерес.

Означення. Два або більше атрибутів є взаємно незалежними, якщо жоден з них функціонально не залежить від інших.

Означення. Надмірна функціональна залежність – це така залежність, що містить інформацію, яку можна отримати на основі інших залежностей, наявних у базі даних.

3.2. Перша, друга й третя нормальні форми та нормальна форма Бойса – Кодда

Очевидно, що для однієї й тієї самої предметної галузі реляційні відношення можна спроектувати різними способами. Наприклад, можна реалізувати кілька відношень за великою кількістю атрибутів або, навпаки, рознести всі атрибути за великою кількістю невеликих відношень. Як установити, за якими ознаками потрібно поміщати атрибути в ті чи інші відношення? Які відношення є «хорошими», а які – «поганими»?

Поняття нормалізації увів Е. Ф. Кодд на початку 70-х років. Сьогодні під нормалізацією реляційної БД розуміють формальний процес видалення надмірних даних.

Коректною вважається така схема бази даних, у якій немає надмірних функціональних залежностей. В іншому випадку слід застосовувати процедури декомпозиції наявної множини відношень. Суть декомпозиції полягає в тому, що відношення поділяється, наприклад, на два відношення за значенням певного атрибута. При цьому виникає така множина, що містить більшу кількість відношень, які є проекціями відношення вихідної множини.

Зворотний покроковий процес заміни цієї сукупності відношень іншою схемою відношень із усуненням надмірних функціональних залежностей називають нормалізацією.

Умова зворотності потребує, щоб декомпозиція зберігала еквівалентність схем при заміні однієї схеми іншою, тобто:

– не повинні виникати кортежі, яких раніше не було;

– на відношеннях нової схеми має виконуватися вихідна множина функціональних залежностей.

Нормалізація – це розбиття таблиці на дві або більше, що мають кращі властивості при додаванні, зміненні й видаленні даних. Остаточна мета нормалізації – отримання такого проекту бази даних, де кожен факт виникає лише в одному місці, тобто вилучається надмірність інформації.

Мета нормалізації:

– уникнення можливої суперечливості збережених даних;

– мінімізація пам'яті, зайнятої даними;

Кожна таблиця в реляційній БД задовольняє умову, відповідно до якої в позиції на перетині кожного рядка і стовпця таблиці завжди знаходиться єдине атомарне значення і ніколи не може бути множини таких значень. Будь-яка таблиця, яка задовольняє цю умову, має назву нормалізованої.

Існує вісім нормальних форм, у яких можуть перебувати відношення всередині БД (рис. 3.1). Саме в такому відношенні й перебувають між собою всі форми, тобто нормальна форма нижчого порядку є підмножиною нормальної форми вищого порядку. Більшість сучасних баз даних перебувають у третій нормальній формі.

Основні властивості нормальних форм:

– кожна наступна нормальна форма є кращою за попередню;

– при переході до наступної нормальної форми властивості попередніх нормальних форм зберігаються.

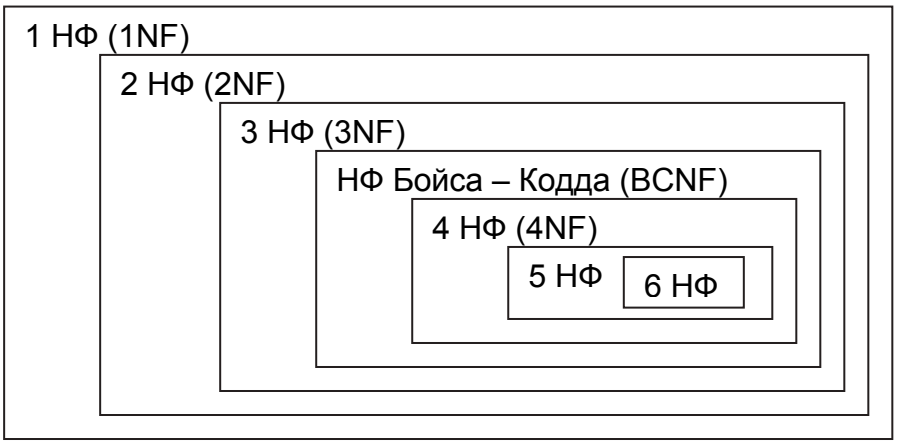


Рис. 3.1. Взаємозв'язок нормальних форм

Основою проектування БД є метод нормалізації: декомпозиція відношення, що перебуває в попередній нормальній формі, відповідає вимогам наступної нормальної форми. Декомпозиція – важливий інструмент при нормалізації і має проводитися без утрати інформації.

Приклади декомпозиції з утратами й без утрат

Відношення постачальників S має вигляд

S		
S#	STATUS	CITY
S3	30	Київ
S5	30	Полтава

Над цим відношенням можна виконати такі декомпозиції:
1.

SST	
S#	STATUS
S3	30
S5	30

SC	
S#	CITY
S3	Київ
S5	Полтава

2.

SST	
S#	STATUS
S3	30
S5	30

STC	
STATUS	CITY
30	Київ
30	Полтава

Суть декомпозиції полягає в тому, що відношення поділяється на два відношення за значенням певного атрибута. У першому випадку це атрибут S #, у другому – STATUS.

У першому випадку інформація не втрачається, оскільки відношення SST і SC усе ще містять дані про те, що постачальник S3 має статус 30 і знаходиться в Києві, а постачальник S5 має статус 30 і знаходиться в Полтаві, тобто перша декомпозиція – це декомпозиція без утрат.

У другому випадку, навпаки, деяка інформація втрачається, оскільки обидва постачальники мають статус 30, але при цьому не можна стверджувати, який із них і в якому місті знаходиться, тобто друга декомпозиція не є декомпозицією без утрат.

Означення. Відношення перебуває в першій нормальній формі, якщо значення всіх його атрибутів є атомарними (неподільними).

Перша нормальна форма (1НФ) – це звичайне відношення. У реляційній БД будь-яке відношення автоматично перебуває в 1НФ.

Властивості першої нормальної форми:

– у відношенні немає однакових кортежів;

– кортежі є невпорядкованими;

– атрибути є невпорядкованими й відрізняються за найменуванням;

– усі значення атрибутів є атомарними.

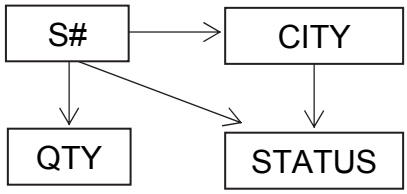
Приклад. Розглянемо відношення *First* (табл. 3.2).

Таблиця 3.2

Відношення *First*

S# – номер постачальника	STATUS – статус постачальника	CITY – місто	P# – товар	QTY – кількість товару в поставці
S1	20	Харків	P1	300
S1	20	Харків	P2	200
S1	20	Харків	P3	400
S1	20	Харків	P4	200
S1	20	Харків	P5	100
S1	20	Харків	P6	100
S2	10	Київ	P1	300
S2	10	Київ	P2	400
S3	10	Київ	P2	200
S4	20	Харків	P2	200
S4	20	Харків	P4	300
S5	3 0	Суми	P5	400

Відношення *First* має такі функціональні залежності:



У цьому прикладі є функціональна залежність CITY→STATUS – статус постачальника визначає його місце розташування.

Недоліки відношення *First*:

1. Надмірність. Дані зберігаються з великою надмірністю – повторюються міста, статуси постачальників, види товару тощо.

2. Аномалії оновлень (Update). Назви, наприклад, міст (статусів, товарів) повторюються в багатьох кортежах, тому якщо постачальник змінює місто, то таке змінення необхідно одночасно виконати в багатьох місцях, інакше відношення стане некоректним. Отже, під час оновлення потрібно переглядати всю таблицю для знаходження й змінення всіх відповідних кортежів.

3. Аномалії включення (Insert). У БД не можна записати нового постачальника (S6, 30, Полтава, P2), якщо продукт (P2) йому не постачається. Отже, до бази можна занести, наприклад, інформацію про постачальника, не зазначивши, який товар і в якій кількості він поставив.

4. Аномалії видалення (Delete). Під час видалення деяких даних може втратитися інша важлива інформація. Наприклад, проблема виникає, якщо необхідно видалити товар (P5). При такому видаленні буде втрачено відомості про постачальника цього товару (S5).

Отже, необхідно перейти до другої нормальної форми.

Означення. Повна функціональна залежність – неключовий атрибут, який функціонально повно залежить від складного ключа, якщо він функціонально залежить від усього ключа в цілому, але не перебуває у функціональній залежності від будь-якого атрибута, який належить до нього. (Функціональну залежність $R.X(r)R.Y$ називають повною залежністю, якщо атрибут Y не залежить функціонально від будь-якої точної підмножини X.)

У відношенні *First* неключовий атрибут CITY залежить від частини ключа S#.

Означення. Відношення *R* перебуває в другій нормальній формі (2НФ) у тому й лише у тому випадку, коли воно перебуває у 1НФ і кожен неключовий атрибут повністю залежить від первинного ключа.

Для того щоб перевести відношення в другу нормальну форму, необхідно провести його декомпозицію на відношення SECOND і SP у такий спосіб:

SECOND {S#, STATUS, CITY},

SP {S#, P#, QTY},

що дає нові відношення:

		SECOND
S#	STATUS	CITY
S1	20	Харків
S2	10	Київ
S3	10	Київ
S4	20	Харків
S5	30	Суми

		SP
S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Переваги декомпозиції:

1. В операціях додавання у відношення SECOND можна додати інформацію про те, що постачальник знаходиться, наприклад, в Ізюмі, навіть якщо він не поставляє ніяких товарів.

2. Операції видалення дають змогу видалити інформацію про поставки, у якій містяться всі відомості про постачальника S3 і товар P2, видаляючи відповідний кортеж із відношення SP, при цьому інформація про знаходження постачальника S3 в Києві не втрачається.

3. У переробленій структурі назва міста для кожного постачальника трапляється всього один раз, оскільки існує лише один кортеж для цього постачальника щодо SECOND, тобто надмірність даних S# – CITY усунуто. Завдяки цьому можна назавжди змінити у відповідному кортежі відношення SECOND назву міста для постачальника S1, наприклад, замість Харкова вказати Полтаву.

Однак незважаючи на те, що надмірність вилучено, відношення SECOND має деякі недоліки:

1. Аномалії включення (Insert) – не можна включити дані про якесь місто, що має деякий статус. Наприклад, не можна вказати, що всі постачальники з Рівного мають статус 50, поки в цьому місті немає деякого конкретного постачальника.

2. Аномалії видалення (Delete) – при видаленні з відношення SECOND кортежу для деякого міста буде видалено інформацію про цього постачальника, а також про те, який статус мало це місто. Наприклад, при видаленні з відношення SECOND кортежу для постачальника S5 буде втрачено інформацію про те, що для Сум було задано статус 30.

3. Аномалії поновлення (Update) – у відношенні SECOND статус для кожного міста повторюється кілька разів. Отже, при зміненні значення статусу Харкова з 20 на 30 виникне проблема або необхідності пошуку відношення SECOND усіх кортежів для Харкова, або отримання несумісного результату. Для вирішення цієї проблеми переходимо до третьої нормальної формі.

Означення. Два або більше атрибутів є взаємно незалежними, якщо жоден з них функціонально не залежить від інших.

Відносно SECOND атрибут STATUS залежить від атрибута CITY.

Означення. Неключовий атрибут – це атрибут, який не входить до складу ключа.

Означення. Відношення *R* перебуває в 3НФ лише в тому випадку, коли воно перебуває в 2НФ і всі неключові атрибути є взаємно незалежними.

Означення. Функціональна залежність R.X → R.Y є транзитивною, якщо існує такій атрібут Z, що мають місце функциональні залежності R.X → R.Z і R.Z → R.Y і немає функціональної залежності R.Z → R.X.

Для того щоб перевести відношення SECOND у 3НФ, необхідно виконати таку декомпозицію:

SC {S #, CITY}, CS {CITY, STATUS}.

Отримаємо такі таблиці:

SC	
S#	CITY
S1	Харків
S2	Київ
S3	Київ
S4	Харків
S5	Суми

CS	
CITY	STATUS
Харків	20
Київ	10
Суми	30
Полтава	40

На практиці 3НФ схем відношень є достатньою в більшості випадків і процес проектування реляційної бази даних зазвичай закінчується зведенням до 3НФ або НФБК. Однак при складних відношеннях бажано продовжити процес нормалізації.

Відношення SC і CS мають такі функціональні залежності:

Усі перелічені форми нормалізації й декомпозиції відношень було виконано, виходячи з означення, що кожне відношення має лише один потенційний ключ (первинний). Однак існують більш складні відношення за таких умов:

– відношення має два або більше потенційних ключів;

– два потенційні ключі – складні;

– ці ключі перекриваються (тобто мають принаймні один спільний атрибут).

Зауважимо, що на практиці відношення з такими умовами трапляються нечасто (для всіх інших відношень 3НФ і BCNF є еквівалентними). Тому оригінальне означення 3НФ було згодом замінено на більш точне означенням Бойса – Кодда, якое набуло окремої назви – нормальна форма Бойса – Кодда (НФБК). Ця форма вносить додаткове обмеження порівняно з 3НФ.

Означення. Відношення *R* перебуває в НФБК тоді й лише тоді, якщо перебуває у 3НФ і детермінанти всіх функціональних залежностей є потенційними ключами. (Відношення перебуває у НФБК, якщо будь-яка функціональна залежність між його полями зводиться до повної функціональної залежності від можливого ключа.)

Означення НФБК є концептуально простішим, ніж означення 3НФ, оскільки в ньому немає явних посилань на 1НФ і 2НФ.

На практиці нормальна форма Бойса – Кодда є основою сучасних методологій аналізу, коли з допомогою деякого абстрактного інструмента, що визначає правила зображення законів роботи й потоків інформації в досліджуваній предметній галузі, у відношення спеціально вносять атрибут, що є первинним ключем, від якого функціонально залежать інші

атрибути відношення. Тому проектувальник не шукає первинного ключа в множині атрибутів, а штучно вводить атрибут, призначаючи його первинним ключем. Такий підхід дуже спрощує роботу з БД і формалізує підхід до їх проектування.

Отже, основна ідея процедури нормалізації полягає в систематичному зведенні відношення *R* (відношення *R* має функціональні залежності, багатозначні й залежні з'єднання) до набору менших відношень, який у деякому заданому значенні є еквівалентним відношенню *R*, але потребує менше часу. Кожен етап процесу зведення складається з розбиття на проекції відношень, отриманих на попередньому етапі. При цьому задані обмеження використовують на кожному кроці процедури нормалізації для вибору проекцій на наступному етапі.

Алгоритм нормалізації:

1. Зведення до 1НФ. Наведено одне або кілька відношень, що відображають поняття предметної галузі. За моделлю предметної галузі (а не за відношеннями) виписують виявлені функціональні залежності. Усі відношення автоматично перебувають у 1НФ.

2. Зведення до 2НФ. Якщо в деяких відношеннях виявлено залежність від частини складного ключа, то провадять декомпозицію цих відношень на кілька відношень у такий спосіб: атрибути, що залежать від частини складного ключа, виносять до окремого відношення разом із цією частиною ключа. У вихідному відношенні залишаються всі ключові атрибути.

3. Зведення до 3НФ. Якщо в деяких відношеннях виявлено залежність деяких неключових атрибутів від інших неключових атрибутів, то виконують декомпозицію цих відношень так: неключові атрибути, що залежать від інших неключових атрибутів, виносять до окремого відношення. У новому відношенні ключем стає детермінант функціональної залежності.

4. Зведення до НФБК. Якщо відношення мають кілька потенційних ключів, то необхідно перевірити, чи є функціональні залежності, детермінанти яких не є потенційними ключами. Якщо такі функціональні залежності є, то потрібно провести подальшу декомпозицію відношень. Атрибути, що залежать від детермінантів, які не є потенційними ключами, виносять до окремого відношення разом з детермінантами.

Правила 1–4 можна сконцентрувати в одному: початкове відношення слід розбити на проекції для вилучення всіх функціональних залежностей, у яких детермінанти не є потенційними ключами.

Після зведення таблиць БД до НФБК можна зробити висновок, що будь-яке змінення інформації приведе до змінення лише одного запису таблиці. Це і є результат нормалізації, тобто множина таблиць, які оновлюються з допомогою одного фрагмента даних.

3.3. Четверта й п’ята нормальні форми

Багатозначні залежності й четверта нормальна форма

Четверта нормальна форма стосується відношень, у яких є багатозначні залежності. У цьому випадку використовують декомпозицію, що базується на багатозначних залежностях.

Означення. Припустимо, що R – відношення, X, Y, Z – деякі з його атрибутів (або множини атрибутів, що не перетинаються). Атрибути Y і Z багатозначно залежать від X (позначається $X \rightarrow\rightarrow Y \mid Z$) тоді й лише тоді, коли з того, що у відношенні R містяться кортежі $r_1 = (x, y, z_1)$ і $r_2 = (x, y_1, z)$, випливає, що у відношенні R є також і кортеж $r_3 = (x, y, z)$.

Багатозначна залежність є узагальненням функціональної залежності і відображає відповідності між множинами значень атрибутів.

Очевидно, що кожна функціональна залежність – багатозначна, але не кожна багатозначна залежність – функціональна.

Означення. Відношення R перебуває у 4НФ лише тоді, коли перебуває у НФБК і якщо в разі існування багатозначної залежності $A \rightarrow\rightarrow B \mid C$ усі інші атрибути R функціонально залежать від A.

Отже, відношення перебуває у 4НФ, якщо перебуває у НФБК і в ньому немає багатозначних залежностей, які не є функціональними залежностями.

Залежності за з’єднанням і п’ята нормальна форма

Раніше було зазначено, що єдина операція, необхідна для усунення надмірності у відношенні, – декомпозиція відношення на дві проекції. Однак існують відношення, коли не можна виконати декомпозицію без утрат на дві проекції, але які можна піддати декомпозиції без утрат на три (або більше) проекції. Це називають залежністю за з’єднанням, а такі відношення – тридекомпозиційними відношеннями.

Відношення, у яких є залежності за з’єднанням, які не є одночасно ні багатозначними, ні функціональними, також характеризуються аномаліями оновлення. Тому вводиться поняття 5НФ.

Означення. Відношення R перебуває у 5НФ (нормальній формі проекції-з'єднання – *PJ*/*NF*) лише в тому випадку, коли будь-яка залежність з'єднання в R випливає з існування деякого можливого ключа в R.

5НФ – це НФ, яку можна отримати шляхом декомпозиції. Її умови є досить нетривіальними й на практиці її не використовують. Зауважимо, що залежність з'єднання є узагальненням як багатозначної, так і функціональної залежностей.

Нормалізація – це процес організації даних у вигляді, який дає змогу вносити зміни до БД без зайвих операцій, а денормалізація – це протилежний процес, який навмисно вносить цю надмірність. За теорією

денормалізація ніколи не повинна виконуватися. Однак на практиці іноді денормалізувати дані потрібно в інтересах продуктивності. Надмірно нормалізована БД уповільнює роботу, оскільки через численні зв'язки між таблицями сервер виконує велику кількість операцій. Причому для сервера прохід за зв'язками між таблицями – одна з найбільш важких операцій.

Рекомендації щодо денормалізації таблиць:

1) якщо модель нормалізованих даних містить таблиці зі складними первинними ключами (у ключі чотири й більше полів), то доцільно денормалізувати дані шляхом уведення довільних сурогатних ключів;

2) якщо в запитах дуже часто використовують обчислювані значення (максимальні, мінімальні тощо), то засобом денормалізації даних може бути додавання цих обчислюваних значень у самі таблиці;

3) розбиття екстремально великих таблиць на кілька надмірних таблиць, до яких винесено або окремі стовпці, або окремі рядки (які нечасто застосовуються) вихідної таблиці;

4) якщо запити до деякої таблиці часто використовують стовпець (фрагмент) з іншої таблиці, то можна внести до цієї таблиці копію цього стовпця (фрагмента).

Дії, наведені в пп. 2–4, можна вирішувати з допомогою створення представлення VIEW.

Для денормалізації таблиць мають існувати вагомі причини, а денормалізацію обов'язково документують і обґрунтовують. Крім того, слід пам’ятати, що в денормалізованих БД операції вставлення, видалення й оновлення виконуються швидше.

3.4. Типи зв'язків

Одна з основних вимог до організації реляційної БД – забезпечення можливості пошуку одних кортежів за значеннями інших, для чого необхідно встановити між відношеннями зв'язок.

Зв'язок (relationship) – пойменована асоціація між двома відношеннями (сутностями), що має значення для аналізованої предметної галузі. Зв'язок – це асоціація між відношеннями (сутностями), за якої зазвичай кожен кортеж одного відношення, яке називають батьківським, асоційований з довільною (у тому числі нульовою) кількістю кортежів іншого відношення, яке називають відношенням-нащадком, а кожен кортеж відношення-нащадка асоційований точно з одним кортежем батьківського відношення.

Бінарний зв'язок – це зв'язок між двома сутностями.

Зв'язок має такі характеристики:

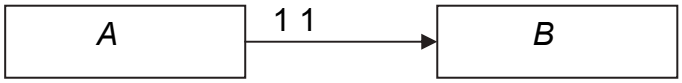
1) ступінь зв'язку (кардинальна кількість зв'язків);

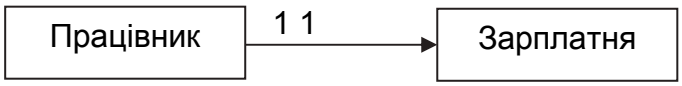
2) обов'язковість (модальність) зв'язку.

Найважливіша властивість зв'язку – кардинальна кількість зв'язків (потужність, або ступінь зв'язку), яка відображає максимально можливу кількість зв'язків для кожного екземпляра сутностей, що беруть участь у зв'язку.

Існують такі основні типи бінарних зв'язків:

1. *Один до одного* – у кожний момент часу кожному кортежу відношення *A* відповідає 0 або 1 кортеж відношення *B*.

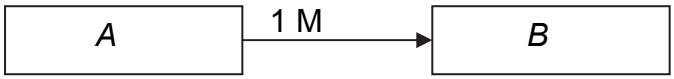


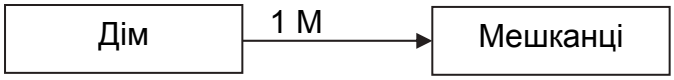


Працівник отримує одну зарплатню

Класичний приклад цього типу зв'язку – зв'язок чоловік – дружина для тих країн, де не дозволено багатоженство.

2. *Один до багатьох* – кожному кортежу відношення *A* відповідає кілька кортежів відношення *B*.





У домі багато мешканців

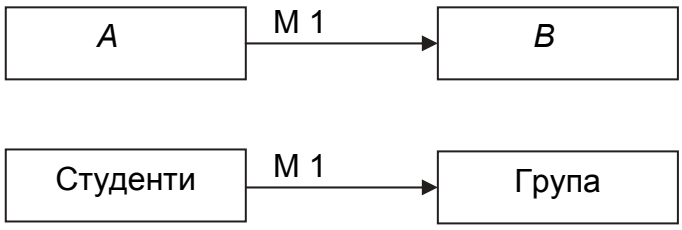
Приклади цього типу зв’язків: група – студент, бригада – працівник, поїзд – вагон, автобус – місце тощо.

Група	
GR#	Ім’я
GR1	Група 1
GR2	Група 2
GR3	Група 3

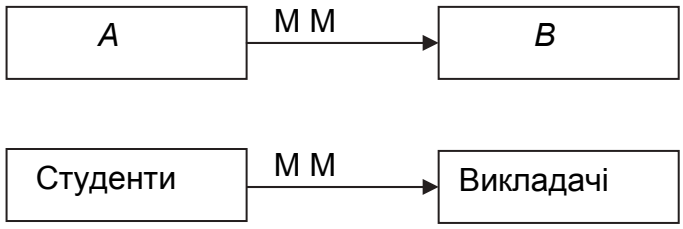
Студент		
ST#	Ім’я	GR#
ST1	Студент 1	GR1
ST2	Студент 2	GR1
ST3	Студент 3	GR2
ST4	Студент 4	GR2
ST5	Студент 5	GR3

Виходячи зі значень атрибутів відношень «Група» і «Студент», можна сказати, що Група 1 складається зі Студент 1, Студент 2, Група 2 – зі Студент 3, Студент 4, а Група 3 – зі Студент 5.

3*. Багато до одного* – множині кортежів відношення *A* відповідає один кортеж відношення *B*.



4. *Багато до багатьох* – множині кортежів відношення *A* відповідає множина кортежів відношення *B* (цей тип зв'язку в реляційних БД безпосередньо не підтримується).



Наприклад, студенти – предмети, книги – читачі тощо.

Реляційна модель не дає змоги описати зв'язок «багато до багатьох» безпосередньо, а лише з допомогою проміжного перетворення. Суть перетворення полягає в поділі зв'язку «багато до багатьох» на два зв'язки «один до багатьох» через проміжне відношення, наприклад зв'язок предмет – студенти. Один предмет може вивчати кілька студентів і, навпаки, кілька студентів можуть вивчати один предмет. Для того щоб перетворити це відношення на два «один до багатьох», необхідно внести додаткове відношення Предмет_Студент, що визначає певний предмет і студента, який його вивчає.

Предмет	
SJ#	Ім’я
SJ1	Предмет 1
SJ2	Предмет 2
SJ3	Предмет 3
SJ4	Предмет 4

Предмет_Студент	
SJ#	ST#
SJ1	ST1
SJ1	ST2
SJ2	ST1
SJ3	ST5
SJ3	ST4
SJ1	ST5
SJ2	ST4

Студент	
ST#	Ім’я
ST1	Студент 1
ST2	Студент 2
ST3	Студент 3
ST4	Студент 4
ST5	Студент 5
ST6	Студент 6
ST7	Студент 7

Виходячи з даних цих відношень зрозуміло, що Предмет1 вивчають Студент1, Студент2 і Студент5. При цьому Студент1 вивчає Предмет1 і Предмет2. Характерною рисою цього перетворення є можливість указати такі дані у відношеннях, які в реальний момент часу не пов'язані між

собою. Наприклад, Предмет4 не вивчає жоден зі студентів, а Студент7, Студент6 і Студент3 не вивчають жодного предмета.

Другим параметром зв'язку може бути його обов'язковість, яку також називають модальністю зв'язку, або класом належності.

Кожен зі зв'язків може мати одну з двох модальностей:

1. «Може» відповідає необов'язковому класу належності – екземпляр сутності може бути пов'язаним з одним або кількома екземплярами іншої сутності, а може бути й не пов'язаним з жодним екземпляром іншої сутності.

2. «Повинен» відповідає обов'язковому класу належності – кожен екземпляр сутності повинен бути пов'язаним з не менш ніж одним екземпляром іншої сутності.

Контрольні запитання

1. Що таке функціональна залежність у значенні відношення? Що таке функціональна залежність для набору всіх можливих значень відношень?
2. Назвіть найбільш поширені в практиці проектування БД нормальні форми відношень і наведіть їх означення, перелічіть кроки переходу від нижчої нормальної форми до вищої.
3. Які недоліки має 1НФ, що усуваються внаслідок переходу до 2НФ?
4. Які недоліки має 2НФ, що усуваються внаслідок переходу до 3НФ?
5. Які недоліки має 3НФ, що усуваються внаслідок переходу до форми Бойса – Кодда?
6. Які недоліки має НФБК, що усуваються внаслідок переходу до 4НФ?
7. Якою є мета денормалізації таблиці? Назвітьі основні рекомендації щодо денормалізації таблиць?
8. Які типи бінарних зв'язків підтримуються (не підтримуються) у базах даних?
9. Які існують типи обов'язковості зв'язків?

Лекція 4. ЛОГІЧНА Й ФИЗИЧНА МОДЕЛІ БАЗИ ДАНИХ

Основні цілі моделювання системи:

1) приховати нецікаві, несуттєві деталі модельованої системи;
2) замінити значні за обсягом компоненти компактними символами;
3) підкреслити важливі (вилучити незначні) символи і зв'язки;
4) сприяти розумінню системи в цілому.

Під час розроблення реляційної БД зазвичай виокремлюють кілька рівнів (етапів) моделювання, з допомогою яких відбувається перехід від

предметної області до конкретної реалізації БД засобами конкретної СКБД.

Наведемо етапи моделювання реляційної БД.

1. *Логічне моделювання*. Логічна модель даних (ЛМД) предметної області описує поняття предметної області, взаємозв'язок цих даних, а також обмеження на дані, що накладаються предметною областю. Основний засіб розроблення ЛМД – різні варіанти ER-діаграм (Entity-Relationshi, діаграми сутність – зв'язок). Чим більше атрибутів мають відношення, отримані під час розроблення ЛМД, тим повільніше будуть виконуватися операції оновлення даних унаслідок витрат часу на перебудову великої кількості індексів, що створюються для таблиць. Збільшення кількості відношень призводить до вповільнення операцій вибірки даних з допомогою запитів SQL, оскільки операція з'єднання таблиць, яка реалізується під час вибірки даних з БД з великою кількістю взаємозалежних відношень, – одна з дорогих у БД.

2. *Фізичне моделювання*. Фізична модель даних описує дані засобами конкретної СКБД. Обмеження, що є в ЛМД, реалізуються різними засобами СКБД, наприклад з допомогою декларативних обмежень цілісності, тригерів, збережених процедур. Рішення, прийняті під час логічного моделювання, впливають на якість фізичної моделі й швидкість роботи БД.

Після розроблення фізичної моделі даних створюється власне база даних і додатки, тобто БД реалізується на конкретній програмно-апаратній основі.

Особливості проектування БД:

– необхідно відобразити предметну область;

– основну увагу приділити нормалізації відношень;

– у відношення може бути додано додаткові атрибути;

– проектувальник ніколи не працює зі значеннями відношень, а лише з відношеннями;

– під час проектування зазвичай не робиться жодних припущень щодо даних, які зберігаються у відношеннях.

Головна мета проектування – виявлення всіх атрибутів і нормалізація всіх відношень, що належать до БД.

4.1. Логічна модель бази даних

Основні компоненти логічної моделі:

– сутність (нормалізоване відношення);

– атрибути сутності;

– зв'язок між сутностями.

Розглянемо приклад проектування в «лоб». Візьмемо предметну область – викладачі, студенти, заняття, аудиторії.

Побудуємо першу нормальну форму БД, що дасть змогу визначити всі необхідні атрибути (табл. 4.1).

Недоліки цього відношення:

– дані зберігаються з великою надмірністю;

– аномалії поновлення, включення й вилучення;

– наявність небажаних функціональних залежностей у БД.

Таблиця 4.1

Таблиця БД у 1НФ

Факультет	Група	ПІБ студента	Номер аудиторії	Предмет	ПІБ викладача	Ім’я кафедри
РКСІ	535	Ст1	118	ОБД	Викл 1	КСС
РКСІ	535 а	Ст2	118	ОБД	Викл 2	КСС
РКСІ	535 б	Ст3	235	ОБД	Викл 3	КСС

Тепер побудуємо другу й третю нормальні форми. Для цього виявимо наявні функціональні залежності й спробуємо їх позбутися.

Функціональні залежності вихідного відношення. Оскільки група вчиться на певному факультеті, то між атрибутами «Група» і «Факультет» існує залежність (Група → Факультет), яка призводить до багаторазового повторення імені факультету. Те ж саме є і в залежності між ім'ям групи й ПІБ студента (Група → ПІБ студента). Таким чином, після першої декомпозиції отримаємо відношення, які показано на рис. 4.1.

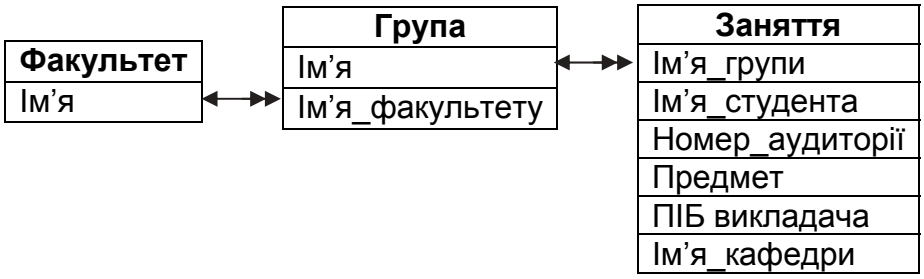


Рис. 4.1. Відношення БД після першої декомпозиції

Перший рядок кожної таблиці визначає її ім'я, інші рядки – атрибути. Лінія, що з'єднує таблиці між собою, відображає зв'язок, а також тип зв'язку й поля, які беруть участь у зв'язку. Подвійна стрілка у кінці лінії позначає сторону «багато», одинарна – «один».

З означень нормальних форм випливає, що таблиці «Факультет» і «Група» перебувають у третій нормальній формі. Таблиця «Заняття» спростилася, але продовжує перебувати у 1НФ.

Розглянемо функціональні залежності, що існують у таблиці «Заняття». Є залежність між атрибутами «Ім’я_кафедри» і «ПІБ викладача» (Ім’я_кафедри ← ПІБ викладача). Атрибут «Ім’я_кафедри»

також функціонально залежить від атрибута «Факультет» (Факультет → Ім'я_кафедри). Ім'я_студента, Номер_Аудіторії і Предмет так само потрібно винести в окремі таблиці. Унаслідок цього після декомпозиції отримаємо структуру, зображену на рис. 4.2.

Тепер усі таблиці перебувають у 3НФ і БД можна вважати нормалізованою.

Ця структура має деякі недоліки.

1. Як первинні ключі вибрано недетермінанти. Ім'я_факультета, ім'я_групи тощо – ідентифікатори з «життя», і незважаючи на те що їх узято атомарними, правила їх складання все ж існують.

Для того щоб перевести таблиці в нормальну форму Бойса – Кодда, потрібно додати до кожної таблиці додатковий атрибут, який буде тільки первинним ключем. Цей атрибут є детермінантою і не змінюється протягом усього «життя»таблиці.

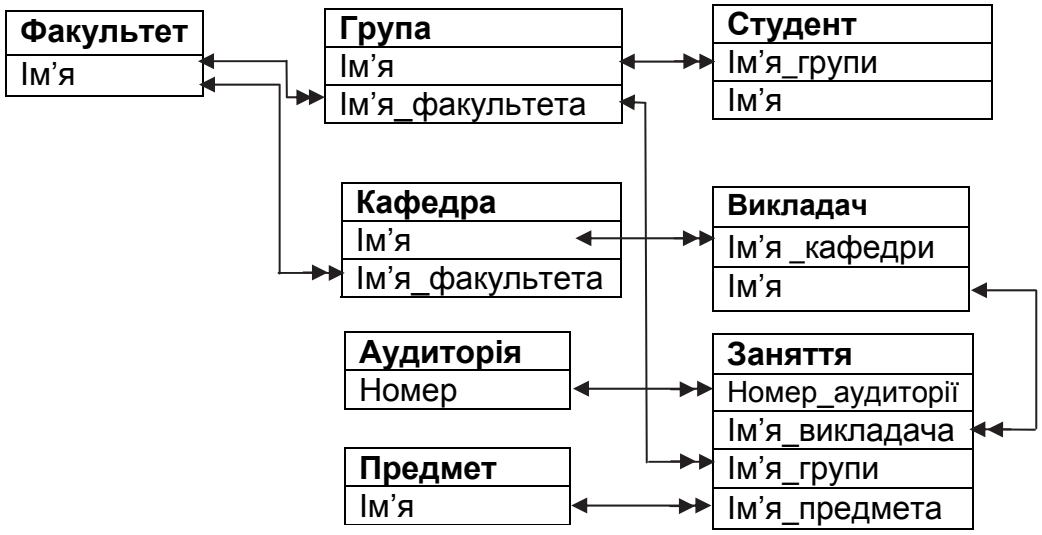


Рис. 4.2. Відношення БД у 3НФ

2. Неповне покриття спроектованої БД предметної області. Наприклад, відомими є аудиторія, де проводяться заняття, предмет і викладач, але не відомо день тижня і час проведення занять. Щоб приклад не вийшов занадто складним, зупинимося на цій функціональності й будемо вважати, що БД створено й нормалізовано.

Існує велика кількість інформаційних моделей:

– побудована з використанням мови Universal Modeling Language (UML) фірми Rational Software,

– побудована з допомогою діаграм «сутність – зв'язок» (ERD – Entity – Relationship) та ін.

Нотація ERD, яку було вперше введено 1976 р. П. Ченом, отримала подальший розвиток у роботах Баркера (case-метод Баркера, нотації Баркера, логічна модель Баркера). Аналогом CASE-методу Баркера є метод IDEF1, розроблений Т. Ремеєм на базі методики П. Чена. IDEF

1X-діаграми використовуються, зокрема, такими CASE-засобами, як ERWin, Design/IDEF.

Модель Баркера має зменшувати кількість помилок у БД і прискорювати процес проектування.

Модель Баркера є логічною і створює сутності, які перебувають у третій нормальній формі. Тому ключовий атрибут, який формує зв'язок, не виникає на жодній сутності, що належить до зв'язку. Наявність самої лінії свідчить про те, що існують ключові атрибути, які формують зв'язок.

Важливий компонент, що гарантує правильне проектування, – постійна перевірка виконаного проекту. Тому необхідно зв'язки підписувати, наприклад: «група складається зі студентів» або, навпаки, «студенти навчаються в групі». Якщо добрати слова не вдається, то це явна ознака того, що сутності не можуть бути зв'язаними між собою безпосередньо.

Логічна модель Баркера для цього завдання має вигляд, показаний на рис. 4.3.

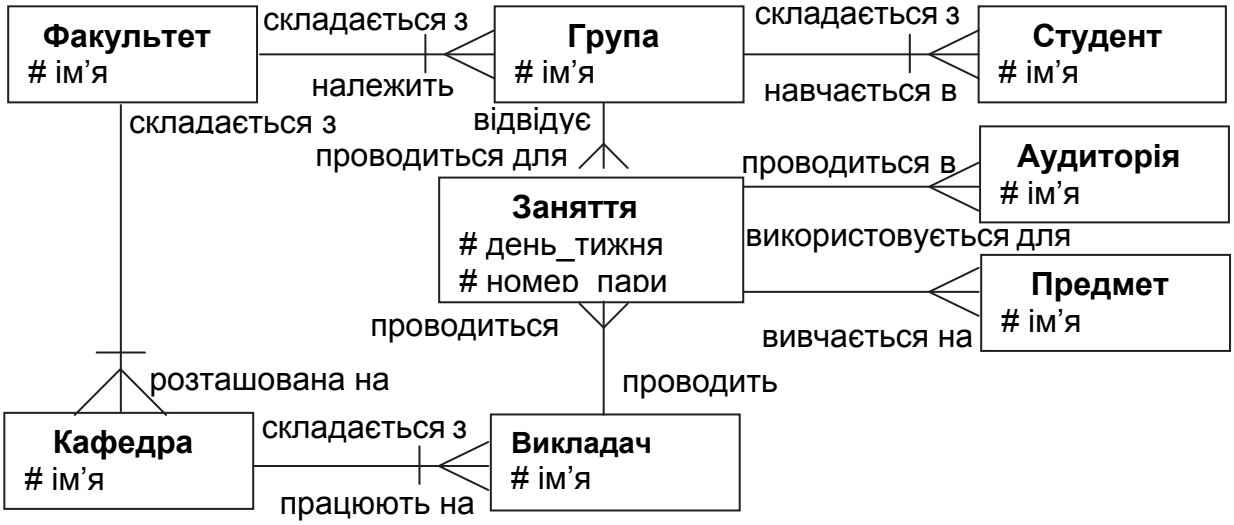


Рис. 4.3. Логічна модель БД

Останній атрибут зв'язку, який зображується як перекреслена в будь-якому місці під кутом 90^0 лінія зв'язку, – ознака обов'язковості. Наприклад, група обов'язково складається зі студентів.

Ця схема має назву логічної моделлї і являє собою логічні сутності й зв'язки між ними. Для розподілу імен сутностей та їх атрибутів використовують символи й слова тієї мови, яка є найбільш зручною.

Ключові атрибути сутностей позначають символом «#».

Ще один важливий момент – виникнення в сутності «Заняття» нових атрибутів «день_тижня» і «номер_пари».

У цій моделі деякі зв'язки є необов'язковими. Наприклад, викладач – заняття. Якщо в поточному семестрі викладач не читає своїх предметів, то це означає, що він не проводить заняття. Тому зв'язок є необов'язковим.

Необов'язкові також зв'язки група – заняття, аудиторія – заняття і предмет – заняття.

4.2. Фізична модель бази даних

Модель Баркера є повністю логічною, тобто в ній немає таких понять, як тип даних, множина значень тощо, які є властивими полям таблиць баз даних як об'єктів керування. Тому наступний крок проектування – це перехід до фізичної моделі. На цьоому етапі логічні сутності перетворюються на конкретні таблиці СКБД. При цьому використовують терміни, властиві цій СКБД.

Основні компоненти фізичної моделі:

– таблиці, поля таблиць, типи даних (з назвами, властивими конкретній СКБД);

– ключі (первинні й зовнішні);

– бізнес-правила;

– SQL-скрипт – набір SQL-оператора для створення таблиць БД.

Алгоритм перетворення моделі Баркера на фізичну модель (гарантується принаймні 3НФ):

1) кожній сутності ставиться у відповідність відношення;

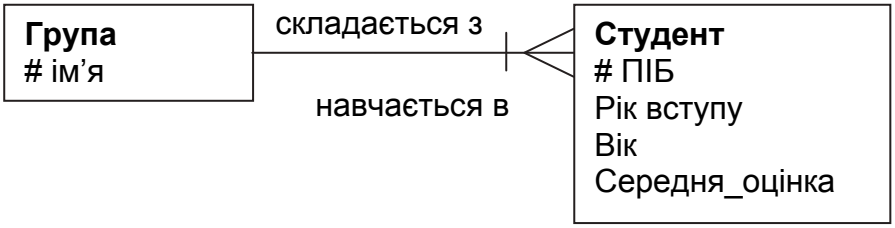
2) кожен атрибут сутності стає атрибутом відношення, якому приписують тип даних і властивість допустимості/недопустимості значення NULL (не визначено);

3) компоненти унікального ідентифікатора сутності (первинний ключ сутності) стають первинним ключем відношення PRIMARY KEY (або як первинний ключ використовують сурогатний ключ), а атрибути, що належать до первинного ключа, отримують властивість обов'язковості (NOT NULL) та унікальності (UNIQUE);

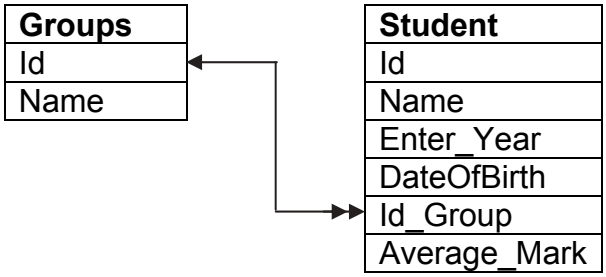
4) атрибути сутності, які утворюють зв'язки «один до багатьох» і «один до одного», стають зовнішніми ключами (для підпорядкованих відношень додають як атрибут зовнішні ключі (FOREIGN KEY), що зазвичай є первинними ключами батьківських відношень);

5) індекси створюються для первинного ключа, зовнішніх ключів і атрибутів, які часто використовують у запитах.

Фрагмент розглянутої раніше структури, що складається з двох сутностей група – студент, має вигляд



і перетворюється на таку фізичну структуру:



Оскільки існує зв'язок між сутностями «Група» і «Студент», то для її реалізації необідно ввести додаткове поле Id_Group у таблиці Student. Поле Id таблиці Groups є первинним ключем.

Поле id_Group таблиці Student – зовнішній ключ, що визначає, які студенти і в якій групі навчаються. Зв'язок між цими полями демонструється з допомогою ліній і стрілок: з боку «один» – стрілка одна, а з боку «багато» – дві.

Фізичну модель бази даних для СКБД MS SQL показано на рис. 4.4. У наведеній структурі вказано імена таблиць, назви полів і зв'язків між таблицями, але вона не описує типи даних і додаткові характеристики полів. Опис можна виконати з допомогою табл. 4.2–4.9.

Таблиця 4.2

Таблиця Facility

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
name	nvarchar(20)			*	*

Стовпець з типом даних tinyint може зберігати значення від 0 до 255 або *Null*. Тип даних займає один байт пам'яті.

Стовпець з типом даних nvarchar призначено для зберігання даних рядків змінної довжини таблиці Unicode. Максимальний розмір стовпця nvarchar – 4000 символів.

Таблиця 4.3

Таблиця Groups

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
name	nvarchar(5)			*	*
id_facility	tinyint		Facility(id)	*	

Формат *numeric* використовують для зберігання дробових чисел. Діапазон значень від ($-10^{38} - 1$) до ($10^{38} - 1$). Ці дані містять два параметри – точність і масштаб.

Точність – це загальна кількість цифр, яка може зберігатися в полі, масштаб – кількість цифр праворуч від точкидесяткової дробу (222.11 – точність дорівнює 5, а масштаб – 2).

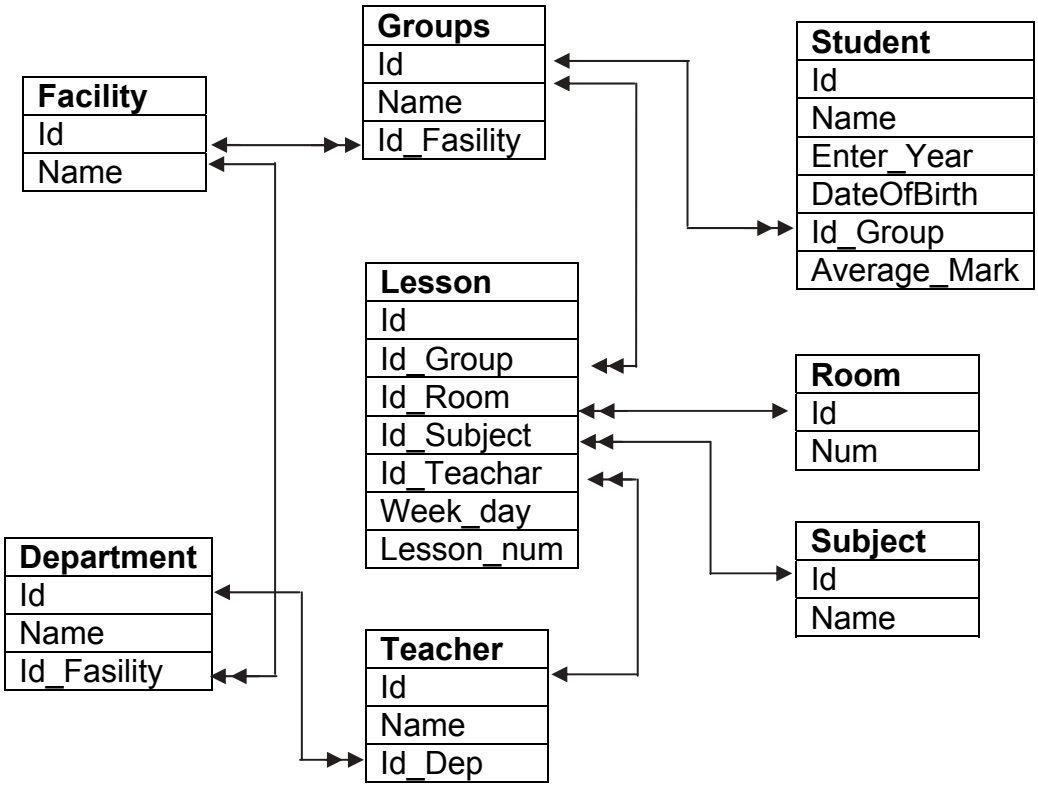


Рис. 4.4. Фізична модель БД

Таблиця 4.4

Таблиця Student

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	int	*		*	*
name	nvarchar(40)			*	
Enter_Year	tinyint				
Age	tinyint				
id_Group	tinyint		Groups(id)	*	
Average_Mark	numeric (5, 2)				

Таблиця 4.5

Таблиця Department

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
name	nvarchar(20)			*	*
id_facility	tinyint		Facility(id)	*	

Таблиця 4.6

Таблиця Teacher

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
name	nvarchar(20)			*	
id_dep	tinyint		Department(id)	*	

Таблиця 4.7

Таблиця Room

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
name	nvarchar(5)			*	*

Таблиця 4.8

Таблиця Lesson

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
id_group	tinyint		Groups(id)	*	
id_room	tinyint		Room(id)	*	
id_subject	tinyint		Subject(id)	*	
id_teacher	tinyint		Teacher(id)	*	
week_day	tinyint			*	
lesson_num	tinyint			*	

Таблиця 4.9

Таблиця Subject

Атрибут	Тип даних	PKEY	FKEY	NOT NULL	UNIQUE
id	tinyint	*		*	*
name	nvarchar(30)			*	*

Розглянемо поля наведених таблиць.

Стовпець «атрибут» задає ім'я атрибута, властивості якого описують.

Стовпець «тип даних» визначає тип даних атрибута. Цей стовпець оперує типами даних СКБД, які використовують для зберігання інформації.

Стовпець PKEY указує, що певий атрибут є первинним ключем, якщо він складається з одного атрибута, або частиною первинного ключа, якщо він містить кілька атрибутів. Для позначення ознаки належності до первинного ключа використовують символ « * ».

Стовпець FKEY описує зовнішні ключі. Наявність інформації в ній свідчить, що певний атрибут – це зовнішній ключ. Під час опису зовнішнього ключа вказують сутність, на яку посилається певний зовнішній ключ, та ім'я її первинного ключа.

Стовпець NOT NULL – поле не може містити значення NULL. Значення NULL указует на те, що певне поле може не містити жодного значення. Наприклад, у студента немає телефону, тоді в користувача є можливість не ставити це значення, а СКБД надає йому спосіб збереження такої інформації.

Стовпець UNIQUE – значення атрибута є унікальним у межах усієї таблиці. Унікальність поля СКБД контролює автоматично. Спроба помістити в базу запис зі значенням поля, яке вже існує, призведе до помилки виконання операції додавання запису. Щоб зрозуміти важливість атрибута UNIQUE, слід звернутися до означення логічної моделі Баркера.

Ключові атрибути сутностей у логічній моделі БД позначають символом «#». Одна з властивостей ключового атрибута – його неповторюваність. Після переходу від логічної моделі до фізичної до бази додаються спеціальні поля для зв'язку сутностей між собою. Отже, виникає первинний ключ і може виникнути зовнішній. Створюється враження, що ключові атрибути сутностей, які були в моделі Баркера, просто загубилися. Це, звичайно, не так. Існування властивості UNIQUE і його підтримка мовою визначення даних дає змогу одночасно зберігати таблиці в нормальній формі Бойса – Кодда і властивість унікальності для полів, що є ключами в «житті» інформаційної системи.

Тепер на основі інформації, що міститься в цих таблицях, можна створювати конкретні таблиці, у яких знаходиться інформація в певній СКБД. Фізична модель таблиць допоможе надалі під час написання серверної частини БД.

Слід звернути особливу увагу на багаторазове повторення однакової інформації. Спочатку описуємо логічну модель, далі переходимо до фізичної, а потім – до кожної таблиці окремо. І кожен раз повторюємося, коли йдеться про атрибути таблиць, зв'язки, первинні й зовнішні ключі. Це зроблено спеціально через велику ціну помилки, яка не виявляється під час проектування структури БД.

Якщо помилку не було виявлено і вона виникла під час кодування програмного забезпечення, то її ціною може бути частковий або повний перегляд структури всього програмного комплексу, а це потребує часу й зусиль конкретних виконавців.

Наприклад, розглянемо набір SQL-операторів, які створюють таблиці певної предметної області. Переміщений у файл такий набір є об'єктом, який має назву SQL-скрипту:

```
CREATE TABLE FACILITY
(
ID tinyint IDENTITY NOT NULL PRIMARY KEY,
NAME nvarchar(20) NOT NULL UNIQUE
)
```

Властивість поля таблиці *IDENTITY* (seed, increment) (ID entity – ідентифікаційний номер сутності) дає змогу автоматично нумерувати поля – у полі таблиці *IDENTITY* автоматично забезпечується додавання унікального значення, що монотонно зростає під час додавання кожного нового рядка. Первинне значення вказується з допомогою аргументу seed (за замовчуванням seed = 1), а крок збільшення – з допомогою аргументу increment (за замовчуванням increment = 1). Властивість *IDENTITY* можна встановити лише для стовпців з типом даних decimal(p, 0), numeric(p, 0), int, smallint і tinyint. У межах однієї таблиці можна створити лише один стовпець з установленою властивістю *IDENTITY*. Обчислити значення за замовчуванням для стовпця зі встановленною властивістю *IDENTITY* неможливо:

```
CREATE TABLE GROUPS
(
ID tinyint IDENTITY NOT NULL,
NAME nvarchar(5) NOT NULL UNIQUE,
ID_FACILITY tinyint NOT NULL,
CONSTRAINT FK_GROUPS /* ім'я обмеження */
FOREIGN KEY (ID_FACILITY) REFERENCES FACILITY(ID) ON
DELETE CASCADE ON UPDATE No Action
)
```

У табл. 4.10 наведено опції цілісності посилань.

Таблиця 4.10

Налаштування опції цілісності посилань

Налаштування	Параметр Delete Rule	Параметр Update Rule
Restrict	Не можна видалити в батьківській таблиці рядок, на який посилається рядок у зовнішній таблиці	Не можна додати до зовнішньої таблиці рядок, для якого немає відповідного запису в батьківській таблиці
Cascade	Якщо в батьківській таблиці буде видалено рядок, на який посилається рядок у зовнішній таблиці, то рядок у зовнішній таблиці також буде видалено	Якщо оновлюється ключове значення в батьківській таблиці, то це значення буде оновлено у всіх рядках зовнішньої таблиці
Set Null	Якщо в батьківській таблиці буде видалено рядок, на який посилається рядок у зовнішній таблиці, то значенням стовпців, які формують зовнішній ключ, буде присвоєно значення Null	Якщо в батьківській таблиці оновлюється рядок, на який посилається рядок у зовнішній таблиці, то значенням стовпців, які формують зовнішній ключ, буде присвоєно значення Null
Set Default	Якщо в батьківській таблиці буде видалено рядок, на який посилається рядок у зовнішній таблиці, то значення в стовпцях зовнішнього ключа набудуть значень за замовчуванням. У цьому випадку всі стовпці зовнішнього ключа повинні мати значення за замовчуванням	Якщо в батьківській таблиці оновлюється рядок, на який посилається рядок у зовнішній таблиці, то значення в стовпцях зовнішнього ключа набудуть значень за замовчуванням. У цьому випадку всі стовпці зовнішнього ключа повинні мати значення за замовчуванням

Якщо не визначено первинний ключ для поля таблиці, то можливими є два шляхи виходу з цієї ситуації:

– видалити наявну таблицю й створити нову з первинним ключем у відповідному полі;

– використати команду для додавання обмеження первинного ключа на полі таблиці.

Приклад. Внесення обмежень до первинного ключа:

```
ALTER TABLE GROUPS
ADD CONSTRAINT PK _ GROUPS -- ім'я обмеження
PRIMARY KEY (ID) -- вид обмеження
```

4.3. Формування бізнес-правил предметної області

Розглянемо табл. 4.2–4.9, які детально відображають структуру відношень БД. Постаає запитання, чому довжина поля Name у таблиці Facility становить 20 символів, а в таблиці Groups – 5 символів? Що є причиною такої різниці?

Відповіді на це запитання не можна отримати, поки не буде створено документ, у якому описано бізнес-правила певної предметної області.

Бізнес-правила (для користувача – цілісність, бізнес-цілісність) – це правила, за якими будується функціонування всієї предметної області і які повністю або частково не може бути задано мовою опису даних (наприклад, обмеження на мінімально можливу заробітну плату співробітника). Для зберігання цієї величини використовують тип даних numeric(6, 2), який є числом із фіксованою точкою. З огляду на зберігання інформації значення (–8345,85 грн) є правільним, але з огляду на успішне функціонування інформаційної системи цей розмір зарплати є абсурдним. Тому бізнес-правила обмежують можливості стандартних типів даних. Це – перший тип бізнес-правил, які важко описати з допомогою мови опису даних, тому потрібні додаткові зусилля програмістів для задання їх логіки роботи.

«Студент належить до певної групи» – другий тип бізнес-правил, які уточнюють властивості зв'язків, реалізується з допомогою мови опису даних та автоматично підтримується СКБД. Існування такого механізму є дуже важливим. По-перше, описане бізнес-правило є наслідком правила цілісності посилань. По-друге, СКБД бере на себе частину роботи, яку має реалізувати програміст. Можливість перекласти частину функціональності на іншу підсистему зміншує кількість помилок програми і час її розроблення.

Третій тип бізнес-правил також належить до уточнення використання стандартних типів даних. Якщо на основі аналізу предметної області максимальна довжина імені групи становить п’ять символів, то цей запис є відправною точкою для визначення максимальної довжини рядка, який зберігає ім'я групи. Відповідно до правил цього типу в табл. 4.2–4.8 було задано довжини всіх символьних атрибутів.

У більшості випадків найкраще підтримувати бізнес-правила на рівні сервера. Якщо помістити бізнес-правила на сторону клієнта, то їх доведеться дублювати в усіх клієнтських додатках.

Бізнес-правила впливають не лише на базу даних, деякі з них реалізуються і в інтерфейсі користувача. Найбільш проста ілюстрація – можливість задання максимальної довжини рядка в стандартному інтерфейсному компоненті «поле внесення символів». Якщо в діалоговому вікні створення нової групи студентів обмежити кількість символів на дисплеї, то це встановить автоматичний контроль бізнес-правил на рівні ресурсу операційної системи.

Для певної предметної області можна визначити такі бізнес-правила:

1) максимальна довжина найменування факультету – 20 символів;
2) максимальна довжина імені групи – 5 символів;
3) група повинна належати до факультету;
4) максимальна довжина імені студента – 40 символів;
5) студент обов'язково вчиться в групі;
6) максимальна довжина назви кафедри – 20 символів;
7) кафедра обов'язково знаходиться на факультеті;
8) максимальна довжина імені викладача – 40 символів;
9) викладач завжди працює на кафедрі;
10) максимальна довжина найменування аудиторії – 5 символів;
11) максимальна довжина найменування предмета – 30 символів;
12) заняття з предмета обов'язково проводить викладач групи в аудиторії;
13) для позначення днів тижня використовують такі константи: 1 – понеділок; 2 – вівторок; 3 – середа; 4 – четвер; 5 – п'ятниця; 6 – субота; 7 – неділя;
14) для позначення номера пари використовують такі константи: 1 – перша; 2 – друга; 3 – третя; 4 – четверта; 5 – п'ята.

Контрольні запитання

1. Якими є цілі моделювання бази даних?
2. Якими є етапи моделювання реляційної бази даних?
3. Що розуміють під концептуальною моделлю бази даних?
4. Якими є особливості логічної моделі бази даних?
5. Якими є особливості фізичної моделі бази даних?
6. Якими є основні компоненти фізичної моделі бази даних?
7. Яким є алгоритм перетворення логічної моделі бази даних на фізичну?
8. Які опції цілісності посилань використовують під час створення таблиць у MS SQL Server?
9. Означення та особливості формування бізнес-правил предметної області.

Лабораторна робота № 2

СТВОРЕННЯ СТРУКТУРИ РОЗПОДІЛУ БАЗИ ДАНИХ

Постановка завдання

На основі таблиць фізичної моделі даних і бізнес-правил предметної області створити:

– скрипт створення таблиць БД;
– скрипт модифікації таблиць БД;
– діаграму БД;
– скрипт видалення таблиць БД;
– скрипт занесення інформації до таблиць БД;
– скрипт змінення інформації в таблицях БД;
– скрипт видалення інформації з таблиць БД.

Для створення наведених скриптів використовують команди мови визначення даних DDL:

Створити об'єкт	Видалити об'єкт	Модифікувати об'єкт
CREATE TABLE	DROP TABLE	ALTER TABLE

і команди мови маніпулювання даними DML:

Змінюють дані таблиць
INSERT
DELETE, TRUNCATE
UPDATE

Обмеження на стовпці й таблиці вводять з допомогою таких пропозицій: NOT NULL, UNIQUE, PRIMARY KEY, CHECK, DEFAULT, CONSTRAINT, FOREING KEY.

Письмовий звіт про виконання лабораторної роботи має містити:

1. Титульний аркуш, на якому наводять назву лабораторної роботи, прізвище, ім'я, по батькові, номер групи виконавця, дату складання.
2. Скрипт створення таблиць БД.
3. Скрипт змінення таблиць БД.
4. Діаграму БД.
5. Скрипт видалення таблиць БД.
6. Скрипт занесення інформації до таблиць БД.
7. Скрипт оновлення інформації в таблиці БД.
8. Скрипт видалення інформації з таблиць БД.
9. Приклади таблиць з даними, таблиць після змінення даних і т. ін.

10. Висновки (відобразити особливості будування моделі, скриптів і шляхи подальшої модернізації БД).

Контрольні запитання

1. До яких мов належить мова SQL?
2. Які форми мови SQL існують?
3. Які компоненти мови SQL існують?
4. Якими є основні пропозиції компонент мови SQL?
5. Які дії можна виконати з допомогою виразів DDL?
6. Якими є основні типи даних СКБД MS SQL Server?
7. Які дії виконує команда CREATE TABLE?
8. Яким є синтаксис команди CREATE TABLE?
9. Яким чином з допомогою команди CREATE TABLE підтримується правило цілісності посилань БД?
10. Як змінюють таблицю БД?
11. Як видаляють таблицю БД?
12. Які існують обмеження на стовпець у БД і як їх уносять?
13. Які існують обмеження на таблицю в БД і як їх уносять?
14. Як здійснюють додавання даних у таблицю БД?
15. Як видаляють дані з таблиці БД?
16. Як змінюють дані в таблиці БД?

Лекція 5. ОПЕРАТОРИ СТВОРЕННЯ, МОДИФІКАЦИЇ ТА ВИДАЛЕННЯ ТАБЛИЦЬ БАЗИ ДАНИХ

5.1. Основні поняття мови SQL

Структуровану мову запитів SQL було розроблено IBM наприкінці 1970-х років, потім до неї вносили зміни й удосконалення, але її принципи залишилися незмінними. Насамперед SQL – непроцедурна мова високого рівня. Прикладна програма не повідомляє машині БД, як виконати завдання, а формулює, що має міститися в результаті (декларативний підхід).

На відміну від процедурних мов, які також можуть використовуватися для роботи з БД, SQL орієнтована не на записи, а на множини. Це означає, що вхідну інформацію для сформульованого на мові SQL запиту до БД застосовують множину записів одного або кількох відношень. Унаслідок цього утворюється множина записів результівної таблиці, тобто в SQL результатом будь-якої операції над відношенням також є відношення. SQL-запит задає не процедуру (послідовність дій, необхідних для отримання результату), а умови, які має задовольняти кортежі

результівного відношення, сформульовані в термінах вхідного (або вхідних) відношення (рис. 5.1).

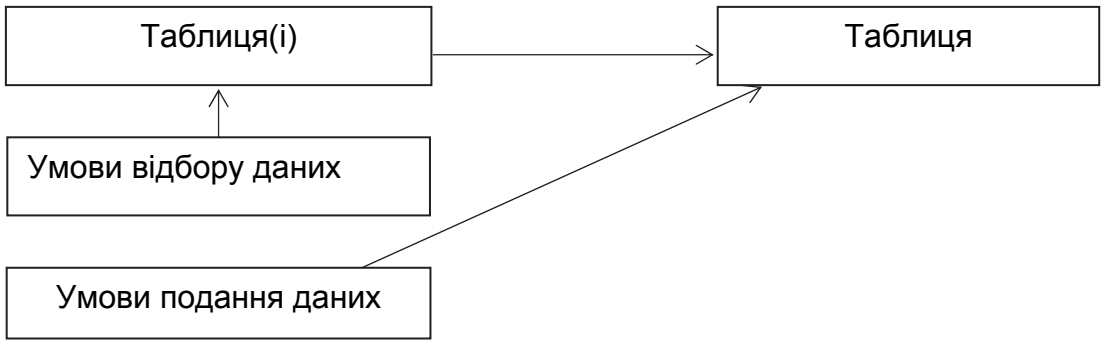


Рис. 5.1. Структура вхідної та вихідної інформації в SQL-запиті

Перший стандарт мови SQL було прийнято 1989 року. Сьогодні всі відомі комерційні продукти підтримують цей продукт. 1992 р. було прийнято другий стандарт SQL, у якому, зокрема, було реалізовано безпосередньо теоретико-множинні операції реляційної алгебри (РА) (до цього їх можна було моделювати на основі аналогії РА з реляційним обчисленням (РО), а РО – на основі SQL). Третій стандарт SQL, спрямований на зближення мови з об'єктно-орієнтованим підходом, було введено 1998 року. Різні виробники БД вносять у синтаксис SQL невеликі зміни. Однак стандартний запит SQL вільно переноситься на різні платформи.

З допомогою SQL можна не лише обробляти інформацію запитами, але й керувати даними (додавання, модифікація, видалення записів, сортування), а також здійснювати супроводження БД (опис типів даних і структури таблиць, видалення і змінення таблиць, індексування, керування правами доступу до даних).

Є дві форми SQL – інтерактивна і вкладена. Обидві форми SQL працюють майже однаково, але використовуються по-різному.

Інтерактивний SQL застосовують для функціонування безпосередньо в БД. У цій формі мови введена команда виконується відразу і результат (якщо він існує) негайно відображається.

Вкладений SQL складається з команд мови, переміщених усередині програм, написаних зазвичай іншою мовою (C#, C++, PHP тощо; часто СКБД надає своє процедурне розширення – мову Transact SQL від MS SQL Server або PL/SQL від ORACLE). Це робить програми більш потужними й ефективними. Однак, допускаючи інші мови, доводиться мати справу зі структурою SQL і стилем керування даними, який потребує деяких розширень до інтерактивного SQL.

Команди SQL поділяють на такі категорії (компоненти SQL):

– вирази мови визначення даних – Data Definition Language (DDL). DDL (в ANSI його називають мовою опису схем – Shema Definition

Language) складаються з команд, що створюють об'єкти у БД (таблиці, представлення, індекси). DDL-пропозиції використовують також для визначення характеристик БД, їх зміни і видалення БД;

– вирази мови маніпулювання даними – Data Manipulation Language (DML). DML складаються з пропозицій (команд), що визначають, які дані знаходяться в таблицях у будь-який момент часу. DML-пропозиції призначено для маніпулювання даними в структурах даних, створених DDL-виразми;

– вирази мови керування даними – Data Control Language (DCL). DCL складаються з пропозицій, які визначають, чи може користувач виконати окрему дію. Згідно з ANSI DCL – частина DDL.

DDL, DML і DCL – це розділи команд однієї мови, згруповані відповідно до їх функціонального призначення.

Основні команди SQL

Вирази DDL:

Створити об'єкт	Видалити об'єкт	Модифікувати об'єкт
CREATE DATABASE	DROP DATABASE	ALTER DATABASE
CREATE INDEX	DROP INDEX	ALTER INDEX
CREATE PROCEDURE	DROP PROCEDURE	ALTER PROCEDURE
CREATE TABLE	DROP TABLE	ALTER TABLE
CREATE TRIGGER	DROP TRIGGER	ALTER TRIGGER
CREATE VIEW	DROP VIEW	

DDL базується на трьох командах SQL:
– CREATE (створити) – дає змогу визначати й створювати об'єкти БД;
– DROP (видалити) – дає змогу видаляти наявний об'єкт БД;
– ALTER (змінити) – дає змогу змінювати означення об'єкта БД.
З допомогою операторів DDL:
– можна створити нову БД;
– сформувати структуру нової таблиці;
– змінити означення наявної таблиці;
– видалити наявну таблицю;
– визначити подання даних;
– забезпечити умови безпеки БД;
– створити індекси для доступу до таблиць;
– керувати розміщенням даних на пристроях зберігання.

Вирази DML:

Змінюють дані таблиць	Не змінюють даних таблиць
INSERT	SELECT
DELETE	
UPDATE	

Вирази DCL:

Надати право	Відмінити право
GRANT	REVOKE

У SQL привілеї надаються й відміняються двома командами – GRANT (допуск) і REVOKE (відміна).

Приклад. Користувач SA має таблицю STUD і дозволяє користувачу SB виконати запит (додавання, оновлення) до неї. У цьому випадку SA повинен виконати таку команду:

GRANT SELECT ON STUD TO SB

GRANT I NSERT ON STUD TO SB

GRANT UPDATE (FAM) ON STUD TO SB // дозволено модифікувати лише поле FAM

REVOKE DELETE, INSERT ON STUD TO SB // скасування привілеїв на видалення й додавання.

Діалог із СКБД здійснюється лише з допомогою SQL-операторів незалежно від того, чи потрібно сформувати таблицю, внести до неї дані, визначити користувача СКБД або отримати інформацію про завантаження процесора. Однак цей клас операторів не виокремлюють, оскільки їх реалізація залежить від виробника СКБД і не завжди зрозуміло, що реально робить оператор. Наприклад, для створення користувача необхідно запустити відповідний оператор, який формує об'єкт СКБД. Отже, можна припустити, що цей оператор належить до DDL. Для того щоб подивитися завантаження процесора, потрібно звернутися до певної системної таблиці СКБД з допомогою оператора SQL, тому всю множину операторів роботи з реляційною СКБД називають SQL.

5.2. SQL-оператор створення таблиці CREATE TABLE

У MS SQL Server існує три способи створення об'єктів БД:

1. З допомогою засобів мови SQL.

2. З використанням інструментальних засобів інтегрованого середовища Management Studio (у цьому випадку сервер все одно генерує SQL-скрипт).

3. З використанням першого й другого способів.

Мова DDL, як видно з назви, застосовується для визначення таблиць, полів, індексів та інших об'єктів, які розташовуються в СКБД і зазвичай зберігають дані або полегшують їх оброблення.

Таблиці бази даних створюються з допомогою команди CREATE TABLE. Ця команда:

– створює порожню таблицю, тобто без рядків;

– визначає ім'я таблиці й множину наведених стовпців у зазначеному порядку;

– для кожного стовпця визначає тип даних і розмір;

– для кожного стовпця визначає різні обмеження (PRIMARY KEY, FOREING KEY, Not Null, Unique, Default, Check, Constraint).

Синтаксис оператора CREATE TABLE має такий вигляд:

```
CREATE TABLE ім'я _ таблиці (
поле 1 тип 1 [NOT NULL] [UNIQUE] [PRIMARY KEY],
поле 2 тип 2 [NOT NULL] [UNIQUE],
. . .
поле N тип N [NOT NULL] [UNIQUE],
[FOREI G N KEY ( поле ) R EFERENCES таблиця ( поле )
[ON DELETE {No Action | CASCADE | SET NULL | Set Default}]
[ON UPDATE {No Action | CASCADE | SET NULL | Set Default }]]
)
```

Оскільки проміжки використовують для розподілу частин команди SQL, вони не можуть бути частиною імені стовпця або таблиці (або будь-якого іншого об'єкта БД). В операторі CREATE TABLE визначення полів перелічуються через кому.

Приклад. Створення таблиці STUDENT (ID – номер залікової книжки, FAM – прізвище, STIP – стипендія, OCENKA – оцінка, GR – група, CITY – місто, BIRTHDAY – рік народження):

```
CREATE TABLE STUDENT
(ID SMALLINT IDENTITY NOT NULL PRIMARY KEY,
FAM NVARCHAR(10),
STIP NUMERIC (6, 2),
OCENKA TINYINT,
GR NVARCHAR(5),
CITY NVARCHAR(15),
BIRTHDAY SMALLDATETIME)
```

Порядок розташування полів у табліці визначається послідовністю, у якій їх наведено в команді створення таблиці.

Коли створюється таблиця або змінюється її структура, можна встановлювати обмеження на значення, які може бути внесено в поля. Якщо це виконано, то СКБД буде відкидати будь-які значення, що порушують обумовлені критерії.

Існують такі типи обмежень цілісності БД:

– обмеження поля (атрибута) (застосовується лише до певного поля);

– обмеження таблиці (відношення) (для груп з одного й більше полів);

– обмеження цілісності посилань (для визначення зовнішнього ключа).

Обмеження на рівні поля:

– невизначене значення NOT NULL – заборона невизначених значень (NULL – це спеціальний маркер, який позначає, що поле є порожнім або невизначеним, значення NULL вноситься за замовчуванням);

– унікальність стовпчика UNIQUE (застосовується лише разом з обмеженням NOT NULL);

– первинний ключ PRIMARY KEY, що є еквівалентним NOT NULL UNIQUE (якщо ключ складається з кількох атрибутів, то використовується інше означення);

– зовнішній ключ (обмеження стовпця за посиланням) FOREIGN KEY;

– перевірка на допустимість значення CHECK;

– задане за замовчуванням значення DEFAULT.

Ці обмеження іноді називають обмеженнями семантичної цілісності для можливості смислового опису даних.

Приклад. Невизначене значення не можна встановити для поля FAM:

```
CREATE TABLE STUDENT
(…
FAM NVARCHAR(10) NOT NULL,
…)
```

Приклад. Необхідно гарантувати, що всі значення, внесені в поля FAM, відрізняються одне від одного. При цьому СКБД відхилить будь-яку спробу внесення в це поле значення, яке вже є в іншому рядку. Це обмеження застосовується до полів, де раніше було оголошено обмеження NOT NULL. Якщо таке поле не є первинним ключем, то воно буде альтернативним ключем:

CREATE TABLE STUDENT
(…
FAM NVARCHAR(10) NOT NULL UNIQUE,
…)

Приклад. Оголошення комбінації з двох полів унікальною (обмеження таблиці) з наданням цьому обмеженню таблиці унікального імені:

```
CREATE TABLE STUDENT
(...
FAM NVARCHAR(10) NOT NULL,
NAME NVARCHAR(10) NOT NULL,
...
UNIQUE (FAM, NAME))
```

або так: … CONSTRAINT UN_FAM_NAME UNIQUE (FAM, NAME)… .

Обмеженням таблиць можна надавати унікальні імена. Перевага явного надання імені обмеженню полягає в тому, що при видачі системою повідомлення про порушення встановленого обмеження буде вказано його ім'я, що спрощує виявлення помилок. Для присвоєння імені обмеженню використовується дещо змінений синтаксис команд CREATE TABLE і ALTER TABLE.

У цьому запиті UN _ FAM _ NAME – це ім'я, надане вказаному обмеженню таблиці.

Приклад. Оголошення складного первинного ключа (обмеження таблиці). Будь-яке поле, яке використовується в обмеженні PRIMARY KEY, має бути оголошено NOT NULL. Первинний ключ складається з двох полів:

```
CREATE TABLE STUDENT
(…
FAM NVARCHAR(10) NOT NULL,
NAME NVARCHAR(10) NOT NULL,
…
PRIMARY KEY (FAM, NAME))
```

або так: … CONSTRAINT PK_ STUDENT PRIMARY KEY(FAM, NAME)… .

Умова перевірки на допустимість значення: CHECK (<логічний вираз>) – працює не на всіх СКБД.

Особливості роботи з обмеженням цілісності CHECK:

1) основні конструкції, які використовуються в CHECK: <, >, =, ! =, And, Or, IN, Between, Like;

2) обмеження може стосуватися одного стовпця;

3) обмеження може бути поширено на всю таблицю, тому перевірка значень в одному стовпці здійснюється на базі значень іншого стовпця (за

умови, що стовпці належать одній таблиці й беруться з одного й того самого оновлюваного або доданого рядка);

4) на основі CHECK можна перевірити відповідність деяких поєднань значень стовпців заданому критерію;

5) правила складання CHECK є аналогічними правилам складання конструкції WHERE;

6) перевірка CHECK дає змогу досягти більш високої продуктивності порівняно з перевірками з допомогою правил і тригерів.

Приклад. Уведення обмежень на поле OCENKA (обмеження перевірки на допустимість значення):

```
CREATE TABLE STUDENT
(…
OCENKA TINYINT CHECK (OCENKA <=5),
…)
```

або так: OCENKA TINYINT CHECK IN (2, 3, 4, 5) …,
або так: OCENKA TINYINT CHECK (OCENKA BETWEEN 2 AND 5) …,
або так: CONSTRAINT CH_ OCENKA CHECK (OCENKA <=5)… .

Приклад. Внесення обмежень у поля OCENKA і TDATE (обмеження таблиці):

```
CREATE TABLE STUDENT
(…
OCENKA TINYINT,
TDATE SMALLDATETIME,
…
CHECK (OCENKA < 5 AND TDATE > 31/03/2020))
```

або так: CONSTRAINT CH_STUD CHECK (OCENKA < 5 AND TDATE > 31/03/2020)

До такої таблиці можна внести лише оцінки нижче «5» для дат після 31/03/2020.

Значення за замовчуванням [DEFAULT {<значення> | USER | NULL }] – тут ключове слово USER означає, що при заповненні колонки йому буде призначено символьний рядок, що містить ім'я поточного користувача.

Значення DEFAULT – невизначеної властивості – визначає, що станеться, якщо не внести будь-яке з явних допустимих значень. Використовувати значення за замовчуванням – це альтернатива для NULL.

Приклад. Внесення обмежень у поле OCENKA і задання значення «5» за замовчуванням:

```
CREATE TABLE STUDENT
(…
OCENKA TINYINT CHECK IN (2, 3, 4, 5) DEFAULT 5,
…)
```

Обмеження стовпця за посиланням: оголошення поля зовнішнім ключем:

FOREIGN KEY (<поле>) REFERENCES < ім'я_основної_таблиці> (<ім'я_первинного_ключа_основної_таблиці>)

[ON DELETE {No Action | CASCADE | SET NULL | Set Default}]

[ON UPDATE {No Action | CASCADE | SET NULL | Set Default}]

На рис. 5.2 показано зв'язок «один до багатьох» між полями таблиць GROUPS і STUDENT.

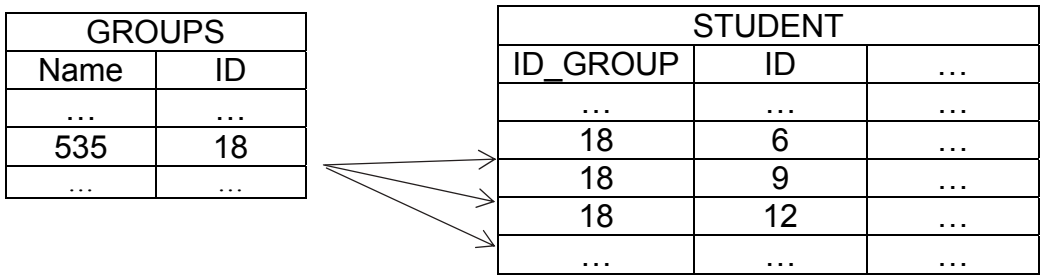


Рис. 5.2. Зв'язок «один до багатьох» між полями таблиць GROUPS і STUDENT

Приклад. Оголошення поля зовнішнього ключа:

```
CREATE TABLE STUDENT ( …
ID_GROUP TINYINT NOT NULL,
…
CONSTRAINT FK_STUD FOREIGN KEY (ID_GROUP) REFERENCES
 GROUPS(ID) ON DELETE CASCADE ON UPDATE CASCADE)
```

Особливості задання обмеження стовпця за посиланням:

1. Основна таблиця має бути описаною до підпорядкованої.
2. Усі поля, які використовуються як зовнішні ключі, повинні в батьківській таблиці мати обмеження PRIMARY KEY або NOT NULL і UNIQUE.
3. Для задання поля, яке є зовнішнім ключем, його спочатку необхідно визначити – описати найменування, тип даних та інші атрибути.
4. Після задання всіх полів може йти одне або кілька означень FOREIGN KEY, які задають поле зовнішнього ключа і визначають таблицю і поле, на яке посилається зовнішній ключ.
5. Кілька означень FOREIGN KEY перелічуються через кому.

Правило цілісності посилань: зовнішній ключ не може посилатися на первинний ключ, якого немає. Цілісність посилань означає підтримку зовнішніх ключів з можливістю вибору одного з принципів видалення зв'язаних кортежів:

– ON DELETE CASCADE – кортежі підпорядкованого відношення знищуються при видаленні зв'язаних з ними кортежів основного відношення (якщо потрібно видалити первинний ключ, то попередньо слід видалити всі записи підпорядкованих таблиць, зовнішні ключі яких посилаються на цей первинний ключ). Означення ON DELETE CASCADE містить механізм каскадного видалення. За наявності ON DELETE CASCADE СКБД автоматично видаляє всі записи, що посилаються на цей первинний ключ;

– ON DELETE SET NULL – кортежі підпорядкованого відношення модифікуються в NULL при видаленні зв'язаних з ними кортежів основного відношення;

– ON DELETE No Action – заборона на видалення кортежу основного відношення за наявності зв'язаних з ним кортежів підпорядкованого відношення.

Розглянемо набір SQL-операторів, що створюють таблиці певної предметної області. Переміщений до файла такий набір є об'єктом, що має назву SQL-скрипту:

```
CREATE TABLE STUDENT (
ID INT IDENTITY NOT NULL PRIMARY KEY,
NAME NVARCHAR (40) NOT NULL,
ENTER_YEAR TINYINT,
AGE TINYINT,
ID_GROUP TINYINT NOT NULL,
AVERAGE_MARK NUMERIC (5, 2),
FOREIGN KEY (ID_GROUP) REFERENCES GROUPS(ID) ON DELETE CASCADE ON UPDATE NO ACTION
)
CREATE TABLE DEPARTMENT (
ID TINYINT IDENTITY NOT NULL PRIMARY KEY,
NAME NVARCHAR (20) NOT NULL UNIQUE,
ID_FACILITY TINYINT NOT NULL,
FOREIGN KEY (ID_FACILITY) REFERENCES FACILITY(ID) ON DELETE CASCADE ON UPDATE NO ACTION
)
CREATE TABLE TEACHER (
ID TINYINT IDENTITY NOT NULL PRIMARY KEY,
NAME NVARCHAR (40) NOT NULL,
ID_DEP TINYINT NOT NULL,
```

```
FOREIGN KEY (ID_DEP) REFERENCES DEPARTMENT(ID) ON DELETE CASCADE  ON UPDATE NO ACTION
)
CREATE TABLE ROOM (
ID TINYINT IDENTITY NOT NULL PRIMARY KEY,
NAME NVARCHAR (5) NOT NULL UNIQUE
)
CREATE TABLE SUBJECT (
ID TINYINT IDENTITY NOT NULL PRIMARY KEY,
NAME NVARCHAR (30) NOT NULL UNIQUE,
)
CREATE TABLE LESSON (
ID TINYINT IDENTITY NOT NULL PRIMARY KEY,
ID_GROUP TINYINT NOT NULL,
ID_ROOM TINYINT NOT NULL,
ID_SUBJECT TINYINT NOT NULL,
ID_TEACHER TINYINT NOT NULL,
WEEK_DAY TINYINT NOT NULL,
LESSON_NUM TINYINT NOT NULL,
FOREIGN KEY (ID_GROUP) REFERENCES GROUPS(ID) ON DELETE CASCADE ON UPDATE NO ACTION,
FOREIGN KEY (ID_ROOM) REFERENCES ROOM(ID) ON DELETE CASCADE ON UPDATE NO ACTION,
FOREIGN KEY (ID_SUBJECT) REFERENCES SUBJECT(ID) ON DELETE CASCADE ON UPDATE NO ACTION,
FOREIGN KEY (ID_TEACHER) REFERENCES TEACHER(ID) ON DELETE CASCADE ON UPDATE NO ACTION
)
```

Дуже важливим є розуміння того, що правило цілісності посилань реалізовано не лише в операціях роботи з даними, але й в операціях створення об'єктів, які ці дані зберігають. Отже, поки не створено таблицю з первинним ключем, можна створити таблицю із зовнішнім ключем, який на нього посилається, тобто порядок створення таблиць існує.

Наведений скрипт ураховує цю властивість. Якщо подивитися на скрипт, то можна побачити явну кореляцію означень полів та інформації, створеної під час опису таблиці. Метою цього є зменшення кількості помилок унаслідок багаторазового визначення сутностей у базі.

5.3. SQL-оператор змінення наявної таблиці ALTER TABLE

Для модифікації структури й параметрів наявної таблиці використовують команду ALTER TABLE. Синтаксис команди ALTER TABLE

для додавання стовпців у таблицю має вигляд ALTER TABLE <ім'я таблиці> ADD (<ім'я стовпця> <тип даних> <розмір>).

За цією командою для наявних у таблиці рядків додають новий стовпець, до якого вносять NULL-значення. Цей стовпець – останній у таблиці. Можна додавати кілька стовпців, тоді їх означення в команді ALTER TABLE розділяють комою.

Оператор ALTER TABLE не діє, якщо таблицю необхідно перевизначити, однак під час розроблення БД не варто виключати необхідність цієї дії.

Приклад. Додавання до таблиці STUDENT двох полів для зберігання інформації про курс і спеціальність студента:

ALTER TABLE STUDENT ADD COURS TINYINT, SPEC NVARCHAR (10).

Приклад. Видалення з таблиці STUDENT поля COURS:

ALTER TABLE STUDENT DROP COLUMN COURS.

Спрощена версія синтаксису оператора ALTER TABLE:

ALTER TABLE < ім'я > {
[ALTER COLUMN < визначення _ стовпця >] |
[ADD < визначення _ стовпця >] |
[DROP COLUMN < ім'я _ стовпця >] |
[ADD [WITH NOCHECK] CONSTRAINT <обмеження_для_таблиці>]}

Ключові слова CHECK (мається на увазі) і NOCHECK перед обмеженням таблиці дають команду SQL Server тестувати або не тестувати наявні в таблиці дані з урахуванням нового обмеження. WITH NOCHECK використовують нечасто.

Наведемо кілька обмежень для фрази ALTER COLUMN. Стовпець не можна змінити:

– якщо він має тип даних text, image, ntext або timestamp;

– є обчислюваним стовпцем або використовується в обчислюваному стовпці;

– є реплікованим;

– використовується в індексі за умови, що стовпець не має типу даних varchar, nvarchar або varbinary; тип даних не змінюється і розмір стовпця не зменшується;

– використовується у статистиці, що генерується оператором CREATE STATISTIC;

– використовується в обмеженнях PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE;

– указується як DEFAULT.

Можна змінити опис стовпців. Часто це пов'язано зі зміненням розмірів стовпців, додаванням або видаленням обмежень, що накладаються на їх значення. Синтаксис команди в цьому випадку має такий вигляд:

ALTER TABLE <ім'я_таблиці> ALTER <ім'я_стовпця> <тип даних> <розмір / точність>

Приклад. Модифікація типу даних поля ID _ NUM у таблиці USP:

ALTER TABLE USP ALTER ID_NUM TINYINT

Слід мати на увазі, що характеристики стовпця можуть змінитися лише з урахуванням таких обмежень:

– змінити тип даних можна лише в тому випадку, якщо стовпець є порожнім;

– для незаповненого стовпця можна змінювати розмір (точність);

– для заповненого стовпця розмір (точність) можна збільшити, але не можна зменшити;

– обмеження NOT NULL можна встановити, якщо жодне значення у стовпці не містить NULL; опцію NOT NULL завжди можна відмінити;

– можна змінювати значення, установлені за замовчуванням.

Змінення структури таблиці під час її роботи може призвести до втрати інформації. Наприклад, запит може зазанати невдачі з тієї причини, що деякого поля в таблиці вже не існує. Отже, краще розробляти БД так, щоб ALTER TABLE застосовувати лише в крайньому разі.

5.4. SQL-оператор видалення таблиць DROP TABLE

SQL-оператор DROP TABLE призначено для видалення таблиць. Загальний вигляд синтаксису оператора DROP TABLE:

DROP TABLE <ім’я_таблиці1>, < ім’я_таблиці2>, ...

Особливості оператора DROP TABLE:

1. Для того щоб мати можливість видалити таблицю, користувач повинен бути її власником, тобто тим, хто її створив.

2. Перед видаленням бажано почистити таблицю від даних, що дасть змогу уникнути випадкової й непоправної втрати інформації.

3. Операція видалення виконується незалежно від існування даних у таблиці.

4. Після видалення таблиці її відновлення стає неможливим. Після виконання цієї команди ім'я таблиці більше не розпізнається.

5. Щоб уникнути багатьох труднощів (наприклад, видалення таблиці було помилкою), рекомендується створити резервну копію БД перед видаленням з неї об'єктів. Згодом, якщо буде необхідно, можна відновити потрібні об'єкти з резервної копії.

6. Правило цілісності посилань реалізовано й тут. Не можна видалити таблицю, яка містить первинний ключ, поки не буде видалено пов’язану з нею таблицю (або її обмеження цілісності) із зовнішнім ключем, що посилається на поле таблиці, яку буде видалено. Це призводить до того, що послідовність видалення таблиць є оберненою до послідовності їх створення.

Розглянемо скрипт видалення таблиць:

```
DROP TABLE lesson, subject;
DROP TABLE room;
DROP TABLE teacher;
DROP TABLE department;
DROP TABLE student;
DROP TABLE groups;
DROP TABLE facility;
```

Під час розроблення програмного продукту іноді потрібно повне видалення бази разом із подальшим її відновленням. Саме тому пишуть скрипти створення й видалення таблиць бази. Крім того, процес створення таблиць БД – це частина інсталяції програмного забезпечення.

5.5. SQL-оператор додавання даних у таблицю INSERT

SQL-оператор INSERT використовують для додавання даних у таблицю БД.

Оператор INSERT має такий синтаксис:

```
INSERT INTO <ім'я_таблиці | подання> [(ім'я_поля1, ім'я_поля2, ...)]
VALUES < (значення1, значення2, ...) > | < Підзапит >
```

Існують дві форми оператора INSERT: перша дає змогу додавати до таблиці нові рядки шляхом безпосереднього задання значення кожного поля, а друга базується на даних, які передаються в оператор INSERT з допомогою запиту SELECT.

Перша форма оператора INSERT має такий синтаксис:

```
INSERT INTO <ім'я_таблиці | подання > [(ім'я_поля1, ім'я_поля2, ...)]
VALUES < (значення1, значення2, ...) >
```

Особливості роботи оператора INSERT:

1. Додають лише один запис до таблиці з ім'ям «ім'я_таблиці», яка попередньо має бути визначеною оператором CREATE TABLE.
2. Якщо поля для додавання взагалі не вказано, то їх послідовність задається оператором CREATE TABLE. Значення в переліку полів для додавання заносяться до таблиці в тому порядку, у якому їх записано в команді.

Приклад. Додати в таблицю Teacher (id (код_викладача), name (ім'я_викладача), id_dep (код_кафедри)) новий запис – 1234, Іванов, 503:

```
INSERT INTO TEACHER VALUES (1234, 'Іванов', 503);
```

3. Якщо в операторі задано не всі поля для додавання, то в порожні поля розміщується значення NULL. При цьому потрібно враховувати можливість наявності умови NOT NULL.

4. Щоб надати значення не всім полям запису або змінити порядок уведення значень, потрібно задати список полів після імені таблиці.

Приклад. Додати в таблицю Teacher новий запис – Петров, код_викладача – 1237:

```
INSERT INTO TEACHER (name, id) VALUES ( 'Петров', 1237);
```

До поля id_dep (код_кафедри) буде автоматично додано значення NULL.

5. Рекомендується чітко вказувати поля, які додаються до таблиці. Це пов'язано з тим, що послідовність полів, яку задає оператор CREATE TABLE, можна просто забути. Існує також можливість додавання нових полів до таблиці після створення БД.

6. Додавання дати/часу в таблицю БД (табл. 5.1).

Таблиця 5.1

Типи даних дати/часу в SQL Server

Ім'я типу даних	Розмір у байтах	Опис
DateTime	8	Дані про дату та (або) час, які стосуються періоду з 1 січня 1753 р. по 31 грудня 9999 р., визначаються з точністю до сотих секунди
SmallDateTime	4	Дані про дату та (або) час, які стосуються періоду з 1 січня 1900 р. по 6 червня 2079 р., визначаються з точністю до однієї хвилини

Перетворити рядок на дату під час додавання можна двома способами:

– неявно – надаючи символьний рядок значенню змінної дата – час (ризикований варіант);

– явно – з допомогою вбудованих функцій Cast і CONVERT (найкращий варіант).

Приклад. Додавання дати в таблицю «Успішність» USP (ID – первинний ключ, ID_NUM – номер залікової книжки, OCENKA – оцінка,

ID_PREDM – код предмета, CDATE – дата виставлення оцінки типу smalldatetime) неявно:

INSERT INTO USP(ID, ID_NUM, OCENKA, ID_PREDM, CDATE) VALUES(1, 3001, 3, 1, '2020/05/19') --(формат ‘YYYY/MM/DD’)
або так: … VALUES(… , '2020-05-19') -- (формат ‘YYYY-MM-DD’)
або так: … VALUES(… , '2020.05.19') -- (формат ‘YYYY.MM.DD’)
або так: … VALUES(… , '05.19.2020') -- (формат ‘MM.DD.YYYY’)
або так: … VALUES(… , '05.19.20') -- (формат ‘MM.DD.YY’)

Додавання дати у форматі VALUES (..., '09.05.20') буде виконано, але дасть результат 2020-09-05 00:00:00, тобто '09.05.20' – це формат 'місяць.день.рік'.

Помилка перетворення (Conversion failed when converting character string to smalldatetime data type) виникне в тому випадку, якщо ... VALUES ('05 19 2020').

У табл. 5.2 наведено основні функції для роботи зі значеннями дати/часу в SQL Server 2005.

Таблиця 5.2

Функції для роботи зі значеннями дати/часу в SQL Server

Функція	Характеристика
DAY	Повертає ціле число – частину дати <date>, відповідає дню місяця. Синтаксис: DAY (<date>)
YEAR	Повертає ціле число, відповідне числовому значенню року. Синтаксис: YEAR (< date >)
MONTH	Повертає ціле число, що відповідає номеру місяця року. Синтаксис: MONTH (<date>) Select id, DAY (tdate) As 'День', MON TH (tdate) As 'Місяць', YEAR (tdate) As 'Рік' From TTime Стовпець tdate має тип smalldatetime
GETDATE	Повертає поточні системні дату й час. INSERT INTO TTime (tdate) VALUES (GETDATE ()) Буде виведено значення 2020-05-17 10:07:00
DATENAME	Повертає рядок зі значенням дати/часу (або фрагмент дати/часу) Синтаксис: DATENAME (<datepart>, <date>) Select id, tdate, DATENAME (DY, tdate) As 'День року' From TTime (DY – шаблон номера дня з початку року)
DATEPART	Повертає ціле число зі значенням дати/часу (або фрагмент дати/часу) Синтаксис: DATEPART (<datepart>, <date>) Select id , tdate , DATEPART (WK, tdate) As 'Тиждень року' From TTime (WK – шаблон номера тижня року)

Закінчення табл. 5.2

DATEADD	Додає значення дати <date> з інтервалом <number> і повертає нове значення дати. Параметр <datepart> визначає одиницю виміру інтервалу Синтаксис: DATEADD (<datepart>, <number>, <date>)
DATEDIFF	Повертає результат обчислення різниці між датами в зазначених одиницях виміру часу. Параметр <datepart> визначає одиницю виміру інтервалу Синтаксис: DATEDIFF (<datepart>, <startdate>, <enddate>)

У табл. 5.3 наведено основні компоненти й шаблони для позначення дати/часу в SQL Server 2005.

Таблиця 5.3

Компоненти й шаблони для позначення дати/часу

Компонент дати	Шаблон дати
Рік	yy , yyyy
Квартал	qq , q
Місяць	mm, m
Номер дня з початку року	dy, y
Доба	dd, d
Тиждень	wk, ww
День тижня	dw
Година	hh
Хвилина	mi, n
Секунда	ss, s
Мілісекунда	ms

Функції Cast і Convert для перетворення типів

Функції Cast і Convert виконують майже однакові дії з перетворення типів, однак Cast відповідає стандарту ANSI, а Convert не передбачена стандартом ANSI.

Синтаксис функцій перетворення:

– Cast (expression As data_type);

– Convert (data_type, expression [, style]),

де expression – вираз для перетворення; data_type – тип, у якому відбувається перетворення; style – опція форматування перетворення (вид результату).

Приклад. Перетворити дату (tdate) на рядок. TTime (id (первинний ключ) smallint IDENTITY (1,1) NOT NULL, tdate (дата) smalldatetime NOT NULL):

Select id, ('дата' + tdate) As 'Date' From TTime

Якщо оператор Select не виконається, то станеться помилка перетворення:

Msg 295, Level 16, State 3, Line 1

Conversion failed when converting character string to smalldatetime data type.

Приклад використання функції Cast:

Select id, ('дата ' + Cast (tdate As varchar)) As 'Date' From TTime

Результат має такий вигляд: дата May 17 2020 10:07 AM.

Select id, ('дата' + Cast (tdate As varchar (12))) As 'Date' From TTime

Результат має такий вигляд: дата May 17 2020.

Приклад використання функції CONVERT:

Select id, ('дата' + CONVERT (VARCHAR (12), tdate, 111)) As 'Date' From TTime,

де style = 111 – стиль японського стандарту виведення дати у вигляді 2020/05/17, style = 100 – стиль стандарту виведення дати у вигляді May 17 2020.

Результат має такий вигляд: дата 2020/05/17.

7. Вираз VALUES задає в круглих дужках значення, які буде додано в таблицю, або ці значення формуються з допомогою підзапиту. Обов’язкова вимога сумісності типів даних полів і типів даних значень, які в них містяться.

8. Значення, які задають у VALUES, – скалярні, і їх не можна обчислити виразами.

9. Команда INSERT не виконує жодного виведення.

Друга форма оператора додавання INSERT має такий синтаксис:

INSERT INTO <ім'я_таблиці | подання> [(ім'я _поля1, ім'я _поля2, ...)] <підзапит>

Застосування оператора INSERT з підзапитом дає змогу завантажувати відразу кілька рядків в одну таблицю, використовуючи інформацію, вибрану SQL-оператором SELECT з іншої таблиці цієї ж або іншої БД. INSERT з підзапитом додає в таблицю стільки рядків, скільки отримує підзапит з іншої таблиці.

Особливості роботи оператора INSERT:

1. Потрібна повна сумісність таблиці, яку поверне SQL-оператор SELECT, з полями, заданими в SQL-операторі INSERT. Отже, кількість полів, порядок їх наступності й типи даних мають збігатися.

2. Вираз FROM підзапиту не повинен містити посилань на таблицю, у яку додаються рядки, тобто не можна використовувати корельовані підзапити.

3. В INSERT можна застосовувати підзапити всередині будь-якого запиту, причому вони можуть бути й співвіднесеними (пов'язаними, корельованими).

Приклад. Додати в таблицю Excellent запис про студентів відмінників, які виокремлюються з таблиці Usp:

```
INSERT INTO EXCELLENT
SELECT * FROM USP WHERE OCENKA = 5
```

Для уникнення помилки таблиця EXCELLENT повинна бути вже створеною і мати таку ж кількість стовпців і такі ж типи даних у стовпцях, як і в таблиці USP.

З допомогою зазначення імен стовпців під час додавання можна перепорядковувати інформацію, яка додається.

Приклад. Додати в таблицю RAITING інформацію про середній бал кожного студента:

```
INSERT INTO RAITING (Id_Stud, AVG_Ocenka)
SELECT Id_Stud, AVG (Ocenka)
FROM USP
GROUP BY Id_Stud
```

Необхідно, щоб імена стовпців таблиці RAITING, а отже, і послідовність даних у переліку, який додається, збігалися з порядком полів у пропозиції SELECT.

Таблиці USP і RAITING ніяк не пов'язані між собою. Якщо в подальшому оцінки студентів, а отже, і середній бал будуть змінюватися, то ці зміни, занесені до таблиці USP, не будуть відображатися в таблиці RAITING. Щоб у таблиці оперативно підтримувалася інформація, потрібно використовувати можливості представлення.

Друга форма оператора часто використовується для вирішення завдань «упакування» таблиць шляхом збереження результівних даних в інших таблицях з виокремленням більш докладної інформації-джерела.

Приклад. Нехай існує БД міської АТС, де реєструються всі дзвінки абонентів. Для кожного дзвінка зберігається дата, тривалість, номер абонента, з яким була розмова. Отже, протягом місяця створюється деяка кількість записів для кожного абонента АТС. Наприкінці місяця на основі цієї інформації абоненту роздруковується й відсилається рахунок за послуги зв'язку, де вказано лише загальний час користування телефоном. Після того як абонент розрахувався за послуги й не вимагає надати йому повну роздруківку, інформація про дзвінки стає непотрібною й потенційно небезпечною. Небезпека полягає в її кількості. Звичайна АТС містить кілька десятків тисяч номерів. Отже, кожні кілька секунд до бази надходить

інформація про зроблені абонентами дзвінки. За рік роботи АТС обсяг інформації буде величезним. Це позначиться на швидкості роботи інформаційної системи, оскільки при складанні звіту використовується вся інформація, яка зберігається за рік, хоча фактично потребується за останній місяць. Складання проміжних звітів дає керівництву змогу бачити картину завантаження АТС протягом заданого інтервалу часу й одночасно відсилати звіти абонентам. Отже, цей приклад демонструє, як можна підвищити продуктивність інформаційної системи завдяки правильному проектуванню її частин.

5.6. SQL-оператори видалення даних з таблиць DELETE І TRUNCATE

SQL Server надає два оператори для видалення рядків з таблиці або представлення – DELETE і TRUNCATE TABLE. Оператор TRUNCATE TABLE без жодних умов видаляє всі рядки в таблиці. Оператор DELETE забезпечує більшу гнучкість і дає змогу видаляти лише вибрані рядки з допомогою фрази WHERE, яка може містити додаткові таблиці й представлення.

Оператор DELETE має такий синтаксис:

```
DELETE [FROM] табліця_або_представлення
[FROM джерела_таблиць]
[WHERE вимога_відбору]
```

Особливості оператора DELETE:

1. Перша фраза FROM є необов'язковою й використовується для зручності читання.
2. Список стовпців в операторі DELETE не вказується, оскільки при видаленні рядка видаляються всі стовпці.
3. Необов'язкова фраза WHERE дає змогу вказувати, які рядки слід видалити.
4. Якщо фразу WHERE не враховано, то видаляються всі рядки в зазначеній таблиці або представленні.
5. Необов'язкова друга фраза FROM дає змогу задавати додаткові джерела (таблиці або представлення), які будуть використовуватися в умові відбору у фразі WHERE.
6. Такий синтаксис оператора може ввести в оману, оскільки рядки не будуть видалятися з таблиць і представлень, зазначених у другій фразі FROM.
7. Якщо в другій фразі FROM указується більше однієї таблиці або одного представлення, то їх імена слід відокремлювати комами.

Приклад. Видалення даних з таблиці STUDENT:

DELETE FROM STUDENT – видалення всього вмісту таблиці STUDENT

DELETE FROM STUDENT WHERE OCENKA = 5

DELETE FROM STUDENT WHERE TDATE <= 31/03/2020

Результат виконання оператора TRUNCATE TABLE є ідентичним результату виконання оператора DELETE, для якого не зазначено умову WHERE, тобто з таблиці видаляються всі рядки.

Приклад. TRUNCATE TABLE STUDENT.

Оператор TRUNCATE TABLE працює більш ефективно, ніж аналогічний оператор DELETE, оскільки останній видаляє по одному рядку за одне проходження і вносить у журнал транзакцій окремі записи для кожної з них. Оператор TRUNCATE TABLE видаляє всі рядки шляхом очищення сторінок, призначених для таблиці, і в журнал транзакцій записуються лише ці видалення.

Особливості оператора TRUNCATE TABLE:

1. Автоінкрементний лічильник обнуляється, під час додавання наступного рядка застосовують вихідне початкове значення для цього стовпця. Інструкція DELETE зберігає значення лічильника.

2. Для таблиці, на яку є посилання, не можна застосовувати зовнішній ключ з інших таблиць. У цьому випадку слід використовувати DELETE.

3. Видалення виконуються з допомогою TRUNCATE TABLE, що не активізують тригери видалення. Це по суті обхід інструкції DELETE.

4. Якщо таблиця є частиною індексованого представлення, то в ній не вдасться використати інструкцію TRUNCATE TABLE.

5.7. SQL-оператор модифікації даних у таблиці UPDATE

SQL-оператор UPDATE змінює вже наявні дані.

Синтаксис оператора модифікації UPDATE:

UPDATE <ім'я_таблиці>

SET <ім'я_поля1> = <значення1 | вираз > [, <ім'я_поля2> =< значення2 |вираз>, ...] [FROM <ім'я_таблиці (ць)_джерела>]

[WHERE <умова обмеження>]

За ключовим словом SET іде перелік відокремлених комами стовпців, які підлягають оновленню, а також їх нові значення. Форма запису є такою: ім'я_стовпця = нове_значення. Нове значення може бути константою, виразом, яке також може посилатися на сам стовпець, DEFAULT або NULL. Наприклад, вираз STIP = STIP * 1.10 буде збільшувати значення в стовпці STIP на 10 %.

Фраза WHERE є необов'язковою. Якщо вона є, то повинна задавати рядки, що підлягають оновленню. Якщо цієї фрази в операторі UPDATE немає, то будуть модифікуватися всі рядки в таблиці.

Особливості оператора UPDATE:

1. Працює лише з однією таблицею.
2. Вираз SET визначає набір полів і значень, які буде їм присвоєно.
3. Якщо умови WHERE немає, то змінення вносяться в усі рядки таблиці.

Приклад. Змінити в таблиці USP оцінки всіх студентів на оцінку «5»:

```
UPDATE USP SET OCENKA = 5.
```

4. Вираз WHERE задає умову, що визначає множини записів, де в зазначені поля буде додано відповідні значення.

Приклад. Змінити в таблиці USP всі оцінки на оцінку «5» з предмета з кодом 503:

```
UPDATE USP SET OCENKA = 5 WHERE Id_Predm = 503.
```

5. У пропозиції можна використовувати вирази, розташовуючи їх у списку для поля, яке необхідно змінити.

Приклад. У таблиці STUDENT збільшити стипендію на 50 % тим студентам, у кого вона не перевищує 450 грн:

```
UPDATE STUDENT SET Stip = Stip * 1.5 WHERE Stip <= 450.
```

6. Оскільки вираз не є предикатом, то можна звичайним способом уводити NULL-значення.

Приклад. Змінити в таблиці USP оцінки на NULL з предмета з кодом 503:

```
UPDATE USP SET OCENKA = NULL WHERE Id_Predm = 503.
```

7. У пропозиції WHERE оператора UPDATE допускається використовувати підзапити. Заборонено посилання у вкладених запитах до таблиці, яка модифікується командою UPDATE.

Приклад. У таблиці STUDENT збільшити стипендію в два рази тим студентам, у кого є оцінки принаймні з двох предметів:

```
UPDATE STUDENT
 SET Stip = Stip * 2
 WHERE 2 <=
 (SELECT COUNT (Id_Predm) FROM USP
WHERE STUDENT.Id_Predm = USP.Id_Predm
AND OCENKA IS NOT NULL)
```

Тут внутрішній запит підраховує кількість записів з оцінками в таблиці успішності USP для кожного студента. Якщо записів два або більше, то предикат основної функції стає справжнім, а розмір стипендії модифікується.

Приклад. Збільшити значення розміру стипендії на 200 грн для студентів, які склали іспити на «4» і«5»:

```
UPDATE STUDENT
SET STIP = STIP + 200
WHERE 4 <=
 (SELECT MIN (OCENKA) FROM USP
WHERE USP.ID_STUD = STUDENT.ID_STUD)
```

8. Істотним недоліком UPDATE є неможливість із команди модифікації UPDATE послатися на таблицю, яка використовується в підзапиті.

Приклад. Одна команда не може виконати таку дію, як модифікація оцінок студентів, що є нижчими за середню. Для цього треба спочатку обчислити середню оцінку:

```
SELECT AVG (OCENKA) FROM USP,
```

а потім результат цього запиту (нехай це буде 4.3) використовувати для модифікації

```
UPDATE USP
SET OCENKA = OCENKA -1 WHERE OCENKA <= 4.3.
```

Приклад. Нехай середній бал студента з прізвищем «Іванов» було обчислено неправільно і потрібно оновити інформацію в полі avg_mark таблиці STUDENT. Послідовність операторів має такий вигляд:

1) SELECT id FROM STUDENT WHERE name = 'Іванов';
2) зберегти поле id;
3) UPDATE STUDENT SET avg_mark = 4.8 WHERE id = збер_поле.

Можна поставити запитання: якщо використовується ім'я студента для отримання його первинного ключа, то чи не можна замість трьох дій скористатися однією й записати оператор оновлення інформації так:

```
UPDATE STUDENT SET avg_mark = 4.8 WHERE name = 'Іванов';
```

Відповідь: звичайно, можна. Однак такий оператор містить потенційну небезпеку, пов'язану з тим, що в базі може зберігатися

інформація про однофамільців або родичів. Тоді оператор змінить зміст у кількох записах. Саме тому правільнішою є перша послідовність, яка складається з трьох операторів. Як і у випадку з SQL-оператором, DELETE-послідовність рознесено в часі. Перші два оператори виконуються для забезпечення користувача необхідною інформацією через інтерфейс, а третій – лише тоді, коли змінює інформацію. Завдання програміста – витягнути первинний ключ разом з інформацією, яка відображається користувачеві на екран, і зберегти його значення для подальшого використання.

Пизначення	Синтаксис оператора SQL
Для модифікації всіх рядків у таблиці	UPDATE табліця_або_представлення SET об'єкти_зміни Об'єкти_зміни являють собою список відокремлюваних комами елементів вигляду стовпець = значення, стовпець = значення, ...
Для модифікації вибраних рядків у таблиці	UPDATE таблиця_або_представлення SET об'єкти_зміни WHERE умова
Для модифікації рядків з використанням фрази FROM	UPDATE таблиця_або_представлення SET об'єкти_зміни FROM таблиця_або_представлення оператор_зв’язування умова_ зв’язування [WHERE (умова_обмеження)]

Контрольні запитання

1. Які існують форми мови SQL?
2. Які існують компоненти мови SQL?
3. Які основні пропозиції входять до складу DDL, DML і DCL?
4. Якими є призначення, синтаксис та особливості оператора CREATE TABLE?
5. Які існують типи обмежень цілісності бази даних?
6. Які існують види обмежень цілісності бази даних на рівні поля? З допомогою яких операторів їх реалізують?
7. Якими є особливості завдання обмеження стовпця за посиланням?
8. Як забезпечується цілісність посилань під час видалення зв'язаних кортежів?
9. Якими є призначення, синтаксис та особливості оператора ALTER TABLE?
10. Якими є призначення, синтаксис та особливості оператора DROP TABLE?

11. Якими є призначення, синтаксис та особливості оператора INSERT? Назвіть дві форми оператора INSERT.

12. Якими є призначення, синтаксис та особливості оператора DELETE?

13. Якими є призначення, синтаксис та особливості оператора TRUNCATE?

14. Якими є призначення, синтаксис та особливості оператора UPDATE?

Лабораторна робота № 3

СТВОРЕННЯ ПРОСТИХ ЗАПИТІВ ДО БАЗИ ДАНИХ

Постановка завдання

На основі таблиць фізичної моделі даних, бізнес-правил предметної області та скриптів формування таблиць БД створити:

1) скрипт створення тестових даних у БД (оператор INSERT) (щонайменше по 5–7 записів у кожній таблиці);

2) оператор SELECT, який отримує дані з однієї таблиці й сортує їх у двох полях у різних напрямках;

3) оператор SELECT із застосуванням умови на основі оператора LIKE;

4) оператори SELECT із застосуванням умови на базі операторів BETWEEN, IN, IS NULL;

5) оператор SELECT, який використовує агрегатні функції й містить групування (GROUP BY і HAVING);

6) скрипт змінення тестових даних у БД (оператор UPDATE) (щонайменше в трьох різних таблицях);

7) скрипт видалення тестових даних у БД (оператор DELETE) (щонайменше в трьох різних таблицях).

Для створення цих скриптів і запитів використовувати команди мови маніпулювання даними DML.

Письмовий звіт із лабораторної роботи повинен містити:

1) титульний аркуш, на якому вказується назва лабораторної роботи, прізвище, ім'я, по батькові, номер групи виконавця, дата складання;

2) діаграму БД;

3) скрипт внесення інформації в таблиці БД;

4) SQL-оператора SELECT для роботи з БД;

5) скрипт змінення інформації в таблицях БД;

6) скрипт видалення інформації з таблиць БД;

7) приклади роботи основних операторів відповідно до поставленого завдання (наприклад, оператор + результат його роботи);

8) висновки (відобразити особливості занесення й модифікації інформації в БД, SQL-оператора SELECT і шляхи подальшої модернізації БД).

Контрольні запитання

1. До яких мов належить SQL?
2. Які форми мови SQL існують?
3. З яких компонентів складається мова SQL?
4. Якими є основні пропозиції мови SQL?
5. Яким є синтаксис оператора SELECT?
6. Якими є основні особливості пропозиції SELECT?
7. Якими є основні особливості пропозиції FROM?
8. Якими є основні особливості пропозиції WHERE?
9. Якими є особливості пропозицій GROUP BY і HAVING?
10. Якими є основні особливості пропозиції ORDER BY?
11. Якими є особливості операторів BETWEEN, IN, IS NULL, LIKE?
12. Якими є основні правила роботи з псевдонімами в SQL?

Лекція 6. ОПЕРАТОР ВИБІРКИ ДАНИХ SELECT (1)

6.1. Загальні відомості про оператора вибірки даних SELECT

SQL-оператор SELECT – один з найважливіших у БД. Основне завдання оператора – вибирання даних з таблиць згідно з умовою, яку поставив користувач. Сучасні інформаційні системи оперують настільки великою кількістю записів, що продуктивність SQL-оператора SELECT стає однією з найважливіших характеристик СКБД.

Оператор SELECT складається з декількох пропозицій, або клауз (від англ. clause – вираз), кожна з яких виконує певні дії. Порядок пропозицій і їх формат є чітко визначеними, але лише кілька пропозицій обов'язкові, решту можна використовувати в разі потреби.

Загальний вигляд SQL-оператора SELECT:

```
SELECT [ ALL | DISTINCT] визначення_полів
FROM визначення таблиць
[WHERE умова]
[GROUP BY умова _групування [HAVING умова]]
[ORDER BY умова _сортування]
```

Необов’язкова частина береться в квадратні дужки.

Оператор SELECT може містити шість пропозицій:

1. SELECT – визначає набір полів, які потрібно отримати внаслідок запиту (ALL – повторювані рядки є, DISTINCT – немає).

2. FROM – установлює набір таблиць, звідки отримується результат.

3. WHERE – визначає умову отримання даних.

4. GROUP BY – задає умову групування даних.

5. HAVING – визначає умову всередині групування.

6. ORDER BY – задає поля й порядок сортування.

Вираз оператора SELECT визначає набір полів, які будуть формувати результівну таблицю. Як поле може бути задано:

– поля таблиць;

– найпростіший вираз або SQL-вираз;

– агрегатна функція, де як аргумент використовується поле або вираз SQL.

Для встановлення поля можна використати такі нотації:

1. * – усі поля таблиць, які беруть участь у запиті;

2. ім'я_поля – позначає поле з ім'ям "ім'я_поля";

3. ім'я_таблиці.ім'я_поля – позначає поле з ім'ям "ім'я_поля" таблиці "ім'я_таблиці" (або псевдонім.ім'я_поля);

4. SQL-вираз – задає вираз, до якого належать поля таблиць і константи.

Кожен виробник СКБД визначає термін «SQL-вираз» по-різному. Відповідно до стандарту ANSI SQL поняття «SQL-вираз» має таке означення:

1. Вираз з арифметичними операціями «плюс», «мінус», «помножити», «розділити», де як аргументи використовуються поля таблиць БД або константи.

2. SQL-функція, де як аргумент використовується поле таблиці, константа або їх комбінація з допомогою арифметичних операцій.

3. Комбінації з використанням арифметичних операцій полів таблиць і викликів SQL-функцій.

Приклади SQL-виразів:

```
a + b
SIN(a + b)
SIN(a) + SIN(b)
SIN(1)
SIN(a) + 1
COS(a) + SIN(b) + a + 1
```

У наведеному прикладі a і b – поля деякої таблиці БД, SIN і COS – функції отримання синуса й косинуса.

Будь-яка сучасна СКБД містить велику кількість стандартних функцій, які можуть застосовуватися в операторі SELECT для отримання даних з БД.

Використання стандартних функцій у деяких випадках є дуже ефективним, тому що економить час розроблення ПЗ і значно прискорює процес отримання кінцевого результату внаслідок того, що SQL-оператор

SELECT виконується на сервері, який має значно більшу потужність, ніж робоча станція користувача.

6.2. Вираз FROM оператора SELECT

Найпростіший SQL-оператор SELECT завжди містить у своєму складі пропозицію FROM, яка задає таблиці, що беруть участь у запиті.

У SQL-операторі SELECT може бути задано стільки таблиць, скільки необхідно для виконання поточного завдання вилучення даних, що дає змогу створювати ефективні оператори SELECT, які спрощують роботу з оброблення даних.

Форма задання однієї таблиці:

FROM ім'я_таблиці [псевдонім].

Якщо необхідно отримати дані з кількох таблиць, то їх перелічують через кому:

FROM ім'я_таблиці1 [псевдонім1], ім'я_таблиці2 [псевдонім2], ...

Для кожної таблиці можна визначити її псевдонім, який діє в межах усього SQL-оператора SELECT. Псевдонім дає змогу зменшити визначення полів, оскільки при проектуванні бази правильним уважається задання імені таблиці, що відображає суть збережених у ній даних. Так, якщо необхідно зберігати інформацію про студентів, то логічно назвати таблицю ім'ям «Student». Однак при подальшому використанні цього імені SQL-оператор SELECT значно збільшується в розмірах, що призводить до труднощів при його прочитанні і, як наслідок, до помилок. Тому для такої таблиці в межах цього оператора вводиться псевдонім, наприклад «S»:

SELECT S.name FROM Student S.

Цей оператор вибирає з таблиці «Student» імена студентів, не сортуючи їх. SQL-оператор INSERT завжди додає дані до БД у порядку їх надходження з метою зменшення витрат, пов'язаних з розташуванням даних у накопичувачі інформації. SQL-оператор SELECT завжди обробляє дані в тому порядку, у якому їх було збережено.

Розглянутий оператор містить деяку кількість надмірних конструкцій. В одній таблиці не може існувати кілька полів з однаковими іменами, отже, явно вказувати ім’я таблиці при заданні поля в пропозиції SELECT не потрібно, так само як і використання псевдоніму імені таблиці. Отже, наведений вище оператор можна записати так:

SELECT name FROM Student.

Обидві версії – повністю правільні.

Розглянемо використання спеціальної нотації «*» при створенні полів таблиць. Символ «*», наведений у клаузі SELECT, позначає всі поля таблиць, указані в пропозиції FROM. Отже, щоб отримати зміст усієї таблиці «Student», достатньо записати:

SELECT * FROM Student – перегляд усіх полів таблиці «Student».

Така форма оператора використовується лише при розробленні програмного забезпечення. Під час написання програми слід указувати всі поля, які формують результівну таблицю SQL-оператора SELECT. Це пов'язано з можливим розширенням надалі набору полів таблиці. Якщо це станеться, то неправильне задання SQL-операторів може призвести до великих труднощів під час доопрацювання програмного забезпечення. Порядок проходження полів у такому операторі визначається порядком проходження полів у SQL-операторі CREATE TABLE.

Вираз SELECT дає змогу задавати кілька полів, перелічуючи їх через кому.

Приклад. Отримати інформацію про середній бал (поле sr_ball) студентів:

SELECT name, sr_ball FROM Student.

Приклад. Переміщення виведених стовпців:

SELECT sr_ball, name FROM Student.

Приклад ілюструє переваги задання псевдонімів для таблиць запиту. Припустимо, що існує таблиця «Groups», яка містить інформацію про групи студентів (рис. 6.1). У таблиці «Student» існує зовнішній ключ, пов'язаний з первинним ключем таблиці «Groups». Якщо поле відображає ім'я групи, то воно має назву «name». Саме таку назву має поле, яке зберігає ім'я студента.

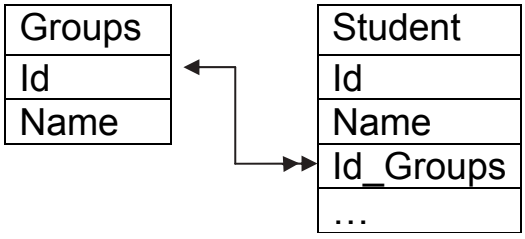


Рис. 6.1. Модель БД з двох таблиць

Тепер необхідно отримати інформацію про те, у якій групі навчається студент. Ось так написати запит не можна:

SELECT name, name FROM Groups, Student,

тому що сервер БД не зможе зрозуміти, поля «name» яких таблиць необхідно отримати в результаті. У цьому випадку використовують форму явного створення імені таблиці при створенні поля в пропозиції SELECT:

SELECT Student.name, Groups.name FROM Groups, Student

Недолік цього оператора – довгий запис. Тут очевидною є перевага псевдонімів при створенні імені таблиці. Цей же оператор можна записати так:

SELECT s.name, g.name FROM Groups g, Student s

Такий оператор є коротшим і при цьому не втрачає свого читабельного вигляду. Не повинно бентежити те, що визначення псевдонімів для таблиць іде після пропозиції SELECT, у якій ці псевдоніми вже використовуються.

Розбір SQL-оператора SELECT СКБД виконує таким чином, що псевдоніми вже є визначеними на етапі перевірки пропозиції SELECT. Тому наведений запис повністю правильний.

Який псевдонім вибрати і для якої таблиці – справа особистого смаку й переваг. Псевдонім може складатися з одного або кількох символів. На практиці при роботі з великою базою застосовують дволітерні позначення, щоб псевдоніми відображали таблиці й оператор не втратив читабельності. У простих операторах достатньо й одного символу, щоб покрити всі імена таблиць запиту.

Розглянутий запит демонструє ще одну важливу властивість оператора SELECT – можливість задання й застосування кількох таблиць. У цьому випадку використано дві таблиці, але можна поставити й більше. Усе залежить від конкретної операції над даними, оскільки ніхто не вимагає, щоб таблиці «Groups» і «Student» містили однакову кількість даних. Тоді яким буде результат попереднього запиту?

Нехай у таблиці «Groups» зберігається інформація про дві групи: у першій – з ім'ям «535» два студенти, а в другій – з ім'ям «535а» також два студенти. Таблиця «Groups» має такий вигляд:

	id	name	id_Facility
▶	2	535	5
	3	535a	5

Вигляд таблиці «Student»:

	id	name	id_groups	sr_ball	stip
▶	1	Иванов	2	3,45	200,80
	2	Петров	3	4,76	250,85
	3	Сидоров	2	4,24	245,25
	4	Семенов	3	4,33	235,76

Під час виконання SQL-оператора Select s.name, g.name From Groups g, Student s база даних «склеїть» дані таблиць так, щоб отримати таблицю в першій нормальній формі. Таке «склеювання» виконується після багаторазового повторення даних таблиці «Groups», у якій записів менше, з таблицею «Student», у якій записів більше. Отже, знайдено результат виконання цього оператора Select:

name	name
Иванов	535
Петров	535
Сидоров	535
Семенов	535
Иванов	535а
Петров	535а
Сидоров	535а
Семенов	535а

Очевидно, що цей результат не відповідає вимозі – отримати інформацію про те, який студент і в якій групі навчається.

Насправді результат роботи цього оператора може бути зовсім іншим. Для вирішення поставленого завдання виведення інформації про те, у якій групі і який студент навчається, потрібно явно з допомогою пропозиції WHERE задати умову зв'язку таблиць «Groups» і «Student». Тоді оператор дійсно виведе ту інформацію, яка є необхідною. Однак як це зробити, розглянемо пізніше.

Оператор SELECT допускає застосування в цільовому списку обчислюваних полів. Якщо потрібно знайти значення будь-якого виразу, навіть не пов'язаного з БД, то можна використовувати оператор SELECT.

Приклад. SELECT exp (3.1) повертає значення 22,1979...

Розглянемо ще одну свідомо пропущену опцію встановлення поля в пропозиції SELECT. Для цього розв’яжемо таку задачу. Якщо таблиця «Student» має поле sr_ball, що містить середній бал студента, то можна вивести для кожного студента не значення середнього бала, а показник

його успішності у відсотках. Нехай «відмінно« – це 100 %. Шляхом нескладних обчислень виведемо формулу, що дає змогу оцінити успішність студента:

усп = sr_ball * 100/5

Ця формула є класичним SQL-виразом. Це означає, що можна написати SQL-оператор SELECT, який надасть інформацію про успішність студента:

SELECT name, sr_ball * 100/5 FROM Student.

Оскільки працюємо з однією таблицею, то при заданні імені поля, вказувати ім'я таблиці не потрібно.

Однак постає таке запитання: яке ім'я буде мати поле, що містить інформацію про успішність студента? Перше поле – «name», а друге – СКБД MS SQL Server згенерує як No column name.

	name	(No column name)
1	Иванов	69.000000
2	Петров	95.200000
3	Сидоров	84.800000
4	Семенов	86.600000

Щоб цього не сталося, потрібно явно вказати в пропозиції SELECT ім'я для поля з допомогою виразу

ім’я_поля_таблиці AS нове_ім’я_поля

Отже, можна записати

SELECT name AS name, sr_ball * 100/5 AS sr_ball FROM student

Тепер імена полів є визначеними, і в подальшому можна на них посилатися під час оброблення результату роботи оператора SELECT у програмі. При заданні імені поля з допомогою AS використовують будь-які символи, що можуть утворювати не лише слова, а й словосполучення. Якщо у найменуванні поля міститься символ «пробіл», то його можна взяти в одинарні лапки. Це дає змогу записати оператор SELECT:

SELECT name AS 'П.І.Б.', sr_ball * 100/5 AS ' Успішність (%)'
FROM Student

	ФИО	Успеваемость(%)
1	Иванов	69.000000
2	Петров	95.200000
3	Сидоров	84.800000
4	Семенов	86.600000

Цей оператор демонструє ще одну техніку запису SQL-оператора SELECT. Імовірність того, що SQL-оператор SELECT не поміститься в один рядок, є досить високою. Згідно з означенням SQL-оператор може бути багаторядковим. Тому, якщо оператор довгий і не поміщається в один рядок, його записують в кілька рядків, використовуючи відступи для підвищення читабельності самого оператора, як це роблять у звичайних програмах. При цьому пропозиції записують одну під одною, а їх аргументи з деяким зсувом уліво, щоб відокремити пропозиції від аргументів. Так само буде виконано й наступний запит, тобто зв'язування AS при роботі з псевдонімом є необов'язковим:

```
SELECT name 'П.І.Б.', Sr_ball * 100/5 ' Успішність (%) ' FROM Student
```

	Ф.И.О.	Успеваемость(%)
1	Иванов	69.000000
2	Петров	95.200000
3	Сидоров	84.800000
4	Семенов	86.600000

Останні пропущені параметри пропозиції SELECT – DISTINCT або ALL. Параметр DISTINCT видаляє з результівної таблиці повторювані записи. Параметр ALL, який установлюється за замовчуванням, залишає результівну таблицю в тому вигляді, у якому вона утворилась після склеювання значень полів:

```
SELECT DISTINCT name AS 'П.І.Б.', sr_ball * 100/5 AS 'Успішність (%)'
FROM Student
```

Якщо припустити, що поле *name* містить лише прізвище, то такий оператор видалить однофамільців з однаковою успішністю.

DISTINCT відстежує, які значення з'явилися в списку вихідних даних, і вилучає з нього значення, що дублюються.

Параметр DISTINCT можна задати лише один раз для пропозиції SELECT. Якщо SELECT вилучає множину полів, то він вилучає і рядки, у

яких усі вибрані поля є ідентичними. Рядки, де деякі значення є однаковими, а інші – різними, додаються до результату. DISTINCT фактично діє на весь вихідний рядок, а не на окреме поле (винятком є його застосування всередині агрегатних функцій), унеможливлює їх повторення.

Текст у запиті розміщується так:

```
SELECT DISTINCT name AS 'ПІБ.', stip AS 'Стипендія', 'грн'
FROM Student
```

Результат цього запиту має такий вигляд:

ПІБ	Стипендія	(No column name)
Іванов	1000.80	грн
Петров	1050.85	грн
...	...	...

Висновки:

1. SQL-оператор SELECT повинен мати у своєму складі пропозицію SELECT, яка задає результівні поля, і пропозицію FROM, що визначає набір таблиць, які беруть участь у запиті.

2. При створенні поля можна використовувати SQL-вирази і визначити для кожного поля його нове ім'я з допомогою виразу AS.

3. Для таблиць з метою скорочення запису оператора й підвищення його читабельності можна задати короткі псевдоніми, шо діють у межах усього SQL-оператора SELECT.

6.3. Вираз WHERE оператора SELECT

Особливості пропозиції WHERE:

1. Створює умову отримання даних з таблиць (і), яка може бути або істинною, або хибною для кожного рядка таблиці.

2. Отримує лише ті рядки з таблиці, де предикат має значення «істина».

3. Створюється зв'язок між таблицями бази (умови зв'язку), а також додаткові умови (умови вибірки), які потрібні для успішної реалізації оператора SELECT.

Розглянемо попередній приклад з таблицями «Groups» і «Student». Таблиця «Student» має у своєму складі зовнішній ключ, який посилається на первинний ключ таблиці «Groups». При створенні таблиць БД це посилання приведе до конкретної пропозиції оператора CREATE TABLE. СКБД не виконує автоматично зв'язку між таблицями, зв'язок потрібно задавати явно при виконанні кожного SQL-оператора SELECT. Лише це

може гарантувати правильність результату. Отже, розглянутий раніше оператор слід записати так:

```
SELECT s.name AS 'ПІБ студента ', g.name AS ' Ім'я групи '
FROM Groups g, Student s WHERE g.id = s.id_group
```

Тут до оператора SELECT додано пропозицію WHERE. В операторі чітко визначено, що ідентифікатор групи в таблиці «Student» має збігатися з ідентифікатором групи в таблиці «Groups». З першого погляду здається, що ця умова не потрібна, але її наявність гарантує правильний результат. Адже відсутність цієї умови дає право СКБД склеїти таблиці в порядку створення записів, а не в порядку належності студентів до групи.

Наступний важливий момент цього оператора – використання полів «id» і «id_group» в умові клаузи WHERE за їх відсутності в пропозиції SELECT. Право звертатися до цих полів дає не їх перелічення в SELECT, а перелічення таблиць, до яких вони належать, у пропозиції FROM. Завжди потрібно пам'ятати про те, що SELECT задає поля, які будуть у результівній таблиці, але не визначає набір полів, який можна використовувати у виразах інших пропозицій SQL-оператора SELECT.

Для задання умов використовують такі операції відношень (реляційні операції порівняння):

1) = (дорівнює);
2) <> (не дорівнює) (або! =);
3) > (більше);
4) < (менше);
5) > = (більше або дорівнює);
6) <= (менше або дорівнює);
7) ! > (не менше);
8) ! < (не більше).

Приклад. Запит на перегляд із БД інформації про успішність студента Іванова:

```
SELECT name AS 'ПІБ', sr_ball * 100/5 AS 'Успішність (%)'
FROM Student WHERE name = 'Іванов'
```

Приклад. Запит на перегляд із БД інформації про студентів, які отримують стипендію:

```
SELECT name AS 'ПІБ',  stip AS 'Стипендія '
FROM Student  WHERE stip > 0
```

Розглянуті умови належать до таких, які часто називають умовами вибірки.

Стандартні булеві оператори SQL – це AND, OR, NOT. Існують і інші, більш складні булеві оператори (як, наприклад, «виключне АБО»), але їх можна побудувати з допомогою трьох наведених булевих операторів. Булева логіка «істина/хибність» являє собою повний базис для роботи цифрового комп'ютера. Тому фактично весь SQL можна звести до булевої логіки. Булеві оператори й основні принципи їх дії:

– AND (операція «логічне І») – використовує два булевих вирази (у вигляді A AND B) як аргументи і дає в результаті істину, якщо вони обидва є істинними;

– OR (операція «логічне АБО») – використовує два булевих вирази (у вигляді A OR B) як аргументи й оцінює результат як істину, якщо хоча б один із них є істинним;

– NOT – використовує єдиний булевий вираз (у вигляді NOT A) як аргумент і змінює його значення з істинного на хибне або навпаки.

Операції AND і OR – бінарні:

– умова1 AND умова2;

– умова1 OR умова2;

– умова1 AND умова2 OR умова3.

Операції AND, OR і NOT можна комбінувати в одну послідовність, створюючи необхідну умову вибірки даних.

Приклад. Вибрати з таблиці «Student» студентів, у яких середній бал не нижче «3» і які належать до групи номер 5:

```
SELECT name AS ' ПІБ ', sr_ball AS 'Середній бал', Id_Group
FROM Student  WHERE sr_ball> = 3 And Id_Group = 5
```

Приклад. Вибрати з таблиці "Student" студентів, у яких середній бал не нижче «3» і які не належать до групи номер 1:

```
SELECT Id, name AS ' ПІБ ', sr_ball AS 'Середній бал', Id_Group
FROM Student WHERE Not (sr_ball < 3 And Id_Group = 1)
```

Булів оператор розміщують перед реляційним оператором, на який він діє, а якщо необхідно розширити дії, то використовують дужки. У цьому випадку SQL сприймає круглі дужки як таки, що означають, що все всередині них буде оцінюватися першим і оброблятися як єдиний вираз з допомогою оператора, розташованого зовні.

Розглянемо типи умов, що існують у пропозиції WHERE. Додамо до прикладу з групами й студентами відношення «факультет». Таблиця буде мати назву «Facility» (рис. 6.2). Група зв'язана з факультетом з допомогою зовнішнього ключа (поле Id_Facility), який посилається на первинний ключ факультету.

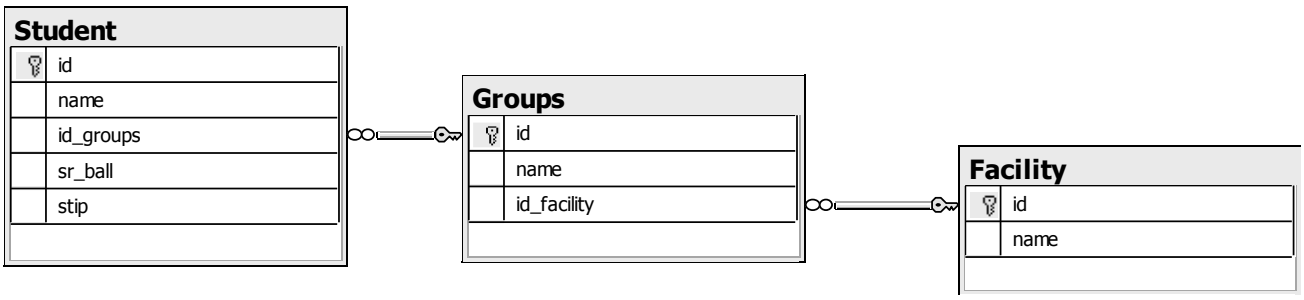


Рис. 6.2. Діаграма таблиць бази даних

Складемо запит, який виводить списки студентів факультету з ім'ям «ФРКСІ». Студенти мають зв'язок із факультетом побічно, через групи, тому SQL-оператор SELECT буде мати вигляд

```
SELECT f.name AS 'Факультет', s.name AS 'ПІБ Студента'
FROM Facility f, Groups g, Student s
WHERE f.id = g.id_facility AND g.id = s.id_groups AND f.name = 'ФРКСІ'
```

Цей оператор містить у пропозиції WHERE дві умови зв'язку й одну умову вибірки. Усі три умови об'єднує оператор AND.

Із рис. 6.2 видно, що ліній, які демонструють зв'язок між групами, дві. Саме тому існує дві умови зв'язку – перша й друга, а третя виконує фільтрацію даних відповідно до поставленого завдання.

Якщо кілька таблиць зв'язуються між собою, то завжди існують умови цього зв'язку, яких на одну менше, ніж кількість таблиць, які беруть участь у запиті. Знання цього правила дає змогу розглядати складні оператори по-іншому. Під час створення таких операторів слід пам'ятати, що необхідно задати умови вибірки й деяку кількість умов зв'язку. Причому умови зв'язку формуються автоматично: підраховують кількість таблиць (можна також підрахувати кількість ліній зв'язку на діаграмі) і створюють відповідні умови, об'єднуючи їх за «І». Потім припускають, якими мають бути умови вибірки. Хоча можна зробити й навпаки – зосередитися на умовах вибірки, а потім автоматично додати до них умови зв'язку. Для групування умов з метою змінення порядку операцій можна використовувати круглі дужки:

```
SELECT f.name AS 'Факультет', s.name AS 'ПІБ Студента'
FROM Facility f, Groups g, Student s
WHERE (f.id = g.id_facility AND g.id = s.id_groups)
AND f.name = 'ФРКСІ'
```

Порядок пріоритету операцій – спочатку AND, а потім OR. У цьому прикладі застосування дужок – надмірність, але їх призначення – продемонструвати синтаксичну конструкцію використання круглих дужок.

Розглянемо таке завдання: вивести список студентів факультету «ФРКСІ». Існує ще кілька способів її розв‘язання. Виходячи з означення зовнішнього ключа й правила посилальної цілісності, у полі id_facility таблиці Groups міститься унікальний ідентифікатор факультету, до якого належить ця група. Отже, якщо ідентифікатор є відомим, то поставлене завдання можна вирішити так:

```
SELECT s.name AS 'ПІБ Студента' FROM Groups g, Student s
WHERE g.id = s.id_groups AND g.id_facility = 5
```

Цей оператор пов'язує лише дві таблиці, але при цьому також вирішує поставлене завдання.

Очевидно, що цей запит буде виконуватися швидше, ніж попередній.

У більшості випадків наведений вище оператор SELECT є ідеологічно неправильним. Чому? Користувач завжди працює з інформаційною системою через інтерфейс. Завдання інтерфейсу – спростити роботу користувача й зробити її максимально наближеною до понять предметної галузі, яку він обслуговує. Студенти ХАІ знають, що факультет ФРКСІ має номер 5, але за всіма офіційними документами студенти вчаться не на факультеті № 5, а на факультеті «ФРКСІ». Тому інтерфейс надає список назв факультетів, а не їх первинних ключів. Ні в якому разі не можна вводити в алгоритм роботи системи правило, що первинний ключ пов'язаний з іншими полями таблиці. Це грубе порушення НФБК. Навпаки, інші поля залежать лише від первинного ключа. Тому користувач вибирає в списку факультет «ФРКСІ» і автоматично отримує оператор, розглянутий першим.

Обмеження на роботу пропозиції WHERE. У WHERE не можна використовувати агрегатні функції (якщо лише не застосовується підзапит), оскільки предикати оцінюються в термінах єдиного рядка, тоді як агрегатні функції – у термінах груп рядків.

Контрольні запитання

1. До яких мов належить SQL?
2. Які форми мови SQL існують?
3. Якими є компоненти мови SQL?
4. Якими є основні пропозиції мови SQL?
5. Яким є синтаксис оператора SELECT?
6. Якими є основні особливості пропозиції SELECT?
7. Якими є основні особливості пропозиції FROM?
8. Якими є основні особливості пропозиції WHERE?
9. Якими є особливості пропозицій GROUP BY і HAVING?
10. Якими є основні особливості пропозиції ORDER BY?
11. Якими є основні правила роботи з псевдонімами у SQL?

Лекція 7. ОПЕРАТОР ВИБІРКИ ДАНИХ SELECT (2)

Під час створення предикатів оператора WHERE використовуються:

– оператори порівняння (=, >, <, > =, <=, <>, ! =,! >, ! <);

– булеві оператори AND, OR, NOT;

– спеціальні оператори IN, BETWEEN, IS NULL, LIKE, ALL, ANY, SOME, EXISTS;

– оператори повнотекстового пошуку:

– CONTAINS і CONTAINSTABLE – для точного й неточного вилучення слів або фраз із текстових стовпців;

– FREETEXT і FREETEXTTABLE – повертають смисловий контекст фраз, заданих у рядку пошуку.

Наведені оператори застосовують для отримання виражених і потужних предикатів.

7.1. Оператор IN (NOT IN)

Оператор IN повністю визначає множину, якій певне значення може належати або не належати.

Приклад. Визначити з таблиці Student прізвища і стипендію студентів, учнів у групах з первинними ключами 2 і 3.

Варіант із застосуванням операції OR («логічне АБО»):

```
SELECT name AS 'ПІБ', stip AS 'Стипендія'
FROM Student WHERE Id_Groups = 2 OR Id_Groups = 3
```

Варіант із використанням оператора IN:

```
SELECT name AS 'ПІБ', stip AS 'Стипендія'
FROM Student WHERE Id_Groups IN (2, 3)
```

Як видно з прикладу, IN являє собою множину, елементи якої точно перелічуються в круглих дужках і розділяються комами. Якщо значення поля, ім'я якого наведено зліва від IN, є одним із перелічених у списку значень (потрібен точний збіг), то предикат уважається дійсним. Якщо елементи множини мають символьний тип, то значення в дужках застосовуються в одиничних лапках:

```
SELECT name AS 'ПІБ', stip AS 'Стипендія' FROM Student WHERE name IN ('Іванов', 'Семенов')
```

Приклад. Визначити з таблиці прізвища студентів, які не проживають в містах Харків і Чугуїв (поле city):

```
SELECT name AS 'ПІБ' FROM Student
WHERE city NOT IN ('Харків', 'Чугуїв')
```

Приклад. Застосування вкладеного підзапиту для формування елементів множини:

```
SELECT * FROM Table 1
WHERE x NOT IN (SELECT x FROM Table2)
```

За означенням, значенням предиката є x NOT IN (S), де (S) – деяка множина, що дорівнює значенню предиката NOT (x IN (S)).

7.2. Оператор BETWEEN (NOT BETWEEN)

Оператор BETWEEN задає межі, у які має потрапити значення, щоб предикат був справжнім.

Синтаксис оператора BETWEEN:

<ім'я_поля> [NOT] BETWEEN <початкове_значення> AND <кінцеве_значення>

Як початкове й кінцеве значення можуть використовуватися вирази й константи.

Синтаксис предиката BETWEEN у більш загальному випадку:

<Value_expression> [NOT] BETWEEN <low_value_expression> AND <high_value_expression>

Фактично цей предикат – це скорочений спосіб запису такого виразу:

((<Low_value_expression> <= <value_expression>) AND (<Value_expression> <= <high_value_expression>))

Особливості оператора BETWEEN:

1. На відміну від оператора IN оператор BETWEEN є чутливим до порядку – low_value_expression у пропозиції має бути першим згідно з алфавітним або числовим порядком.

Приклад. Вилучити з таблиці Student прізвища й середній бал студентів, який знаходиться в діапазоні між «4» і «5».

```
SELECT name AS ' ПІБ ', sr_ball AS 'Середній бал'
FROM Student WHERE sr_ball BETWEEN 4 AND 5
```

2. Оператор BETWEEN – включний, тобто межові значення (у цьому прикладі це «4» і «5») роблять предикат істинним. Мова SQL безпосередньо не підтримує виключне BETWEEN. Слід сформулювати граничні значення так, щоб включена інтерпретація була правдивою або за необхідності вилучити граничні значення. Це можна зробити, наприклад, з допомогою запиту

```
SELECT name AS 'ПІБ', sr _ ball AS 'Середній бал'
FROM Student WHERE (sr_ball BETWEEN 4 AND 5)
AND Not sr_ball IN (4, 5)
```

3. Аналогічно оператор порівняння BETWEEN діє на символьних полях, наведених у двійковому (ASCII) еквіваленті, тобто для вибірки можна скористатися алфавітним порядком. Наступний запит вибирає студентів, значення поля «name» яких потрапляють у заданий алфавітний діапазон:

```
SELECT * FROM Student
WHERE name BETWEEN 'K' AND 'R';
```

Прізвище Romanov буде пропущено, незважаючи на те, що BETWEEN є включним, оскільки порівнює рядки однакової довжини. Рядок ' R ' коротше від рядка 'Romanov', тому 'R' доповнюється проміжками. Останні передують символам латинського алфавіту (у більшості реалізацій), тому Romanov виявився невибраним. Про це необхідно пам'ятати під час використання BETWEEN з алфавітними діапазонами.

Для включення в результат виконання запиту відомостей про студентів, прізвища яких починаються на 'R', слід указати наступну букву алфавіту ('S') або приписати символ 'z' (кілька символів 'z', якщо це необхідно) після другого межового значення.

7.3. Оператор LIKE (NOT LIKE)

Синтаксис оператора LIKE:

<ім'я_поля> [NOT] LIKE <зразок> (<рядок> [NOT] LIKE <підрядок>)

Оператор LIKE застосовується лише до рядкових полів типу CHAR (NCHAR) або VARCHAR (NVARCHAR), оскільки його використовують для пошуку підрядків у предикаті пропозиції WHERE. Отже, він здійснює перегляд рядка для перевірки умови – чи належить заданий підрядок до зазначенго рядка. Для гнучкої реалізації цієї мети використовують шаблони (маски) – спеціальні символи для конструювання зразка рядка.

Існують такі типи шаблонів, які використовуються з LIKE:

1. Символ «підкреслення» (_) – замінює будь-який одиничний символ, наприклад:

– ... LIKE 'б_т' – наприклад, 'б_т' відповідає 'біт' або 'бут', але не 'брат';

– ... LIKE '_рі_' – будь-яке слово з чотирьох літер, що має в середині дві букви 'рі', наприклад 'кріт';

– ... LIKE '____ов' – будь-яке слово з шести літер, які закінчуються на 'ов', наприклад 'Петров'.

2. Символ «відсоток» (%) – замінює послідовність символів довільної довжини, у тому числі й нульовий, наприклад:

– ... LIKE 'І%' – будь-який рядок, що починається з букви І, наприклад 'Іванов';

– ... LIKE '% Е %' – будь-який рядок, що містить букву Е , наприклад 'SELECT';

– ... LIKE '% К% Д' – наприклад, '% К% Д' відповідають 'КБС', 'СКБД', 'РСКБД', але не 'КВДУ';

– ... LIKE '% т_о%' – є істинним для будь-якого слова, що містить шаблон 'т_о', наприклад 'Петров'.

Приклад. Вилучити з таблиці Student інформацію про всіх студентів, прізвища яких починаються з літери І:

```
SELECT * FROM Student  WHERE name LIKE 'І%'
```

3. Символ «квадратні дужки» ([]) – дозволяється застосування в рядку будь-якого окремого символу із зазначених у дужках (перелічення без проміжків та інших роздільників; зазначення інтервалу), наприклад:

– ... LIKE '[cf]%' – будь-який рядок, що починається з літер *c* або *f*;

– ... LIKE '[s - v]% ing' – будь-який рядок, що починається на літери *s*, *t*, *u*, *v* і закінчується на *ing*;

– ... LIKE '[JT] im ' – будь-який рядок з трьох букв, що починається на *J* або *T* і закінчується на '*im*', наприклад *Jim* або *Tim*.

4. Символ «квадратні дужки» ([^]) – будь-який одиничний символ із перелічених у дужках не повинен бути в рядку (допускається перелічення без проміжків та інших роздільників, а також зазначення інтервалу), наприклад:

– ... LIKE ' S [^ f]%' – будь-який рядок, що починається на *S*, за яким не йде *f*.

Дозволено використання конструкції NOT LIKE.

Приклад. Вилучити з таблиці Student інформацію про всіх студентів, прізвища яких не починаються на літеру «І»:

```
SELECT * FROM Student WHERE name NOT LIKE 'І% '
```

LIKE може бути корисним під час пошуку імені або іншого значення, повне написання якого є невідомим. Наприклад, не зовсім зрозуміло, як правильно пишеться прізвище одного зі студентів – 'Коваленко', 'Коленко' або 'Коваленков'. Можна використовувати відому частину і символи шаблону для знаходження всіх можливих варіантів:

```
SELECT * FROM Student WHERE name LIKE 'К%к%'
```

Символ шаблону (%) наприкінці рядка є необхідним у тих реалізаціях SQL, коли довжина поля *name* перевершує кількість букв в імені 'Коваленко'. у такому випадку значення поля *name* реально зберігається як 'Коваленко', а за ним іде кілька проміжків. Отже, символ 'о' не є останнім у рядку. Символ відсотка (%) у шаблоні замінює всі проміжки.

Пошук у рядку символів підкреслення (_) або відсотка (%) виконується з допомогою Escape-символу, який використовується в предикаті безпосередньо перед символом % або _ і означає, що наступний за ним символ інтерпретується як звичайний, а не як символ шаблону.

У SQL рядкі можуть бути однаковими, але не збігатися з точки зору оператора LIKE. Під час перевіркі рівності рядків потрібно більш короткий рядок праворуч заповнити проміжками до довжини довшого, а потім порівняти рядки з допомогою символів. Отже, рядки 'Іванов' і 'Іванов' (з трьома проміжками) будуть однаковими. Предикат LIKE не додає проміжків, тому вираз 'Іванов' LIKE 'Іванов' дасть значення *false*.

Щоб уникнути цієї проблеми, можна скористатися функціями для оброблення рядків LTRIM() та RTRIM() (функції TRIM() у MS SQL Server немає), які видаляють непотрібні проміжки ліворуч і праворуч від одного або обох аргументів ... WHERE RTRIM (name) LIKE 'Іванов'.

Приклад. Скласти запит, який виводить перелік студентів факультету з ім'ям «ФРКСІ».

Наведений нижче спосіб вирішення завдання базується на правилі створення імен груп, що застосовується в ХАІ. Ім'я групи складається з трьох цифр і необов'язкової однієї літери. Перша цифра – номер факультету, друга – номер курсу і третя – номер спеціальності. Літера використовується, якщо груп з цієї спеціальності декілька. Отже, групи першого факультету – 1хх, другого – 2хх, третього – 3хх тощо. Якщо існує таке правило, то це означає, що його можна використовувати для отримання імені групи:

```
SELECT s . name AS 'ПІБ Студента' FROM groups g, student s
WHERE g.id = s.id_group AND g.name LIKE '5%'
```

Однак наступний запит буде неправильним, тому що не вказано умову зв'язку поля s.name з полем g.name у таблицях Student і Groups:

```
SELECT s.name AS 'ПІБ Студента' FROM Groups g, Student s
WHERE g.name LIKE '5%'
```

Причина того, чому попередній запит був неправильним, стане зрозумілою після аналізу такого запиту:

```
SELECT s.name 'ПІБ', g.name Група 'FROM Groups g, Student s
WHERE g.name LIKE '5%'
```

Предикат (g.Name LIKE '5%') із таблиці groups вибере два записи – з іменами груп '535' і '535 а'. Оператор SELECT s.name ... вибере чотири записи з поля s.name – 'Іванов', 'Петров', ... При з'єднанні в одну таблицю спочатку буде вибрано перше значення з таблиці з меншою кількістю записів – це '535', потім до цього значення приєднаються всі записи з таблиці з великою кількістю записів – це 'Іванов', 'Петров', ... Потім буде вибрано групу '535а' і т. д.

Приклад. Нехай стовпець містить дані завдовжки не більше шести символів, а предикат LIKE шукає імена, що починаються з 'Іва'. Для цього можна написати такий код:

```
SELECT * FROM student WHERE name LIKE 'Іва %'
```

Однак цей оператор працює повільніше, ніж код

```
SELECT * FROM student WHERE (name = 'Іва') OR (name LIKE 'Іва_')
OR (name LIKE 'Іва __') OR (name LIKE 'Іва ___')
```

Завершальні проміжки також є символами, які можуть мати точну відповідність.

Предикат із використанням ... name LIKE '%нов' також буде працювати повільніше, ніж предикат вигляду ... WHERE (name LIKE '_нов') OR (name LIKE '__нов') OR (name LIKE '___ нов').

7.4. Оператор IS NULL (IS NOT NULL)

Часто в таблиці трапляються записи з незаданими значеннями будь-якого з полів, тому що значення поля є невідомим або його просто немає. У таких випадках SQL дає змогу вказати в полі NULL-значення:

– коли значення поля є NULL, це означає, що програма БД у спеціальний спосіб позначає поле, яке не містить жодного значення для цього рядка, і, навпаки, у разі простого приписування полю значення «нуль» або «проміжок»;

– оскільки NULL не є значенням як таким, він не має типу даних;

– NULL може розміщуватись у полі будь-якого типу, проте NULL-значення часто використовується в SQL.

Для знаходження невідомого значення використовується оператор IS із ключовим словом NULL.

Приклад. Припустимо, у таблиці є поле OCENKA, а іспиту не було (або частина студентів іспит ще не склала), тоді в цьому полі буде знаходитися значення NULL доти, доки оцінка не стане відомою. Визначити всі записи таблиці, у яких оцінку ще не виставлено:

```
SELECT * FROM Student WHERE OCENKA IS NULL
```

Спільно з операторами IS NULL можут використовуватися булеві оператори AND, OR та NOT.

Приклад. Визначити всі рядки таблиці Student, де розмір стипендії (stip) проставлено:

```
SELECT * FROM Student WHERE stip is not NULL
```

7.5. Використання виразу CASE

Вираз CASE використовується для перевірки списку умов та отримання одного з кількох можливих результатів. Функція CASE має два формати виклику:

– простий вираз CASE – для порівняння заданого виразу з множиною простих для визначення результату;

– пошуковий вираз CASE – для множин булевих (логічних) виразів.

Синтаксис:

Простий вираз CASE:

```
CASE input_expression WHEN when_expression
THEN result_expression [... n ]  [ELSE else_result_expression ] END
```

Пошуковий вираз CASE:

```
CASE WHEN boolean_expression THEN result_expression [... n ]
[ELSE else_result_expression ] END
```

Аргументи:

input_expression

Вираз, отриманий при використанні простого формату функції CASE. Аргумент input_expression є будь-яким припустимим виразом:

WHEN when_expression

Простий вираз, з яким порівнюється аргумент input_expression при використанні простого формату функції CASE. Аргумент when_expression є будь-яким припустимим виразом. Типи даних аргументу input_expression і кожного з виразів when_expression мають бути однаковими або неявно зводитися один до одного:

THEN result_expression

Вираз, що повертається, якщо порівняння виразів input_expression та when_expression дає в результаті TRUE або вираз Boolean_expression обчислюється в TRUE. Аргумент result_expression є будь-яким припустимим виразом:

ELSE else_result_expression

Цей вираз є таким, що повертається, якщо жодна з операцій порівняння не дає в результаті TRUE. Якщо цей аргумент пропущено й жодна з операцій порівняння не дає в результаті TRUE, то функція CASE повертає NULL. Аргумент else_result_expression є будь-яким припустимим виразом. Типи даних аргументу else_result_expression та будь-якого з аргументів result_expression мають бути однаковими або неявно зводитися один до одного:

WHEN Boolean_expression

Цей логічний вираз є таким, який отримано під час використання пошукового формату функції CASE. Аргумент Boolean_expression є будь-яким припустимим логічним виразом.

Вираз CASE може використовуватися в будь-якій інструкції або пропозиції, у якій припускаються допустимі вирази. Наприклад, вираз CASE можна використовувати в таких напрямках, як SELECT, UPDATE, DELETE і SET, а також у виразах IN, WHERE, ORDER BY та HAVING.

Приклад. Оператор SELECT з простим виразом CASE:

```
SELECT fam 'ПІБ', 'Оцінка з математики' =
 CASE math
 WHEN 5 THEN 'відмінно'
```

```
 WHEN 4 THEN 'добре'
 WHEN 3 THEN 'задовільно'
 WHEN 2 THEN 'незадовільно'
 ELSE 'невідомо'
 END FROM Table _1
```

Приклад. Оператор SELECT з пошуковим виразом CASE:

```
SELECT fam 'ПІБ', 'Оцінка з математики' =
 CASE
 WHEN math = 5 THEN 'відмінно'
 WHEN math = 4 THEN 'добре'
 WHEN math = 3 THEN 'задовільно'
 WHEN math = 2 THEN 'незадовільно'
 ELSE 'невідомо'
 END FROM Table_1
```

Контрольні запитання

1. Якими є основні особливості пропозиції WHERE?
2. Якими є особливості оператора BETWEEN?
3. Якими є особливості оператора IN?
4. Якими є особливості оператора IS NULL?
5. Якими є особливості оператора LIKE?
6. Якими є особливості простого виразу CASE?
7. Якими є особливості пошукового виразу CASE?

Лекція 8. ГРУПУВАННЯ Й СОРТУВАННЯ ПОЛІВ В ОПЕРАТОРІ SELECT

8.1. Пропозиції GROUP BY і HAVING

Приклад. Необхідно вирішити таке завдання: нехай у таблиці Student міститься середній бал студентів (поле sr_ball), а ім'я групи (поле name) – у таблиці Groups. Як отримати середній бал групи?

Оператор, який починає вирішувати це завдання, має такий вигляд:

```
SELECT g.name AS 'Група', s.sr_ball AS 'Середній бал'
FROM Groups g, Student s WHERE g.id = s.id_groups
```

Розглянемо послідовно роботу цього оператора. Припустимо, існують дані в таблицях:

Таблиця «Groups»	
id	Name
2	535
3	535a

Таблиця «Student»			
id	name	sr_ball	id_group
1	Іванов	3,45	2
2	Сидоров	4,24	2
3	Петров	4,76	3
4	Семенов	4,33	3

При виконанні цього SQL-оператора SELECT буде склеєно поля *name* таблиці Groups і *sr_ball* таблиці Student та отримано такий результат:

Група	Середній бал
535	3,45
535	4,24
535a	4,76
535a	4,33

Зауважимо, що одні й ті самі значення поля *name* таблиці Groups багаторазово повторюються, даючи змогу для групування даних. Усередині проміжної таблиці виникають групи даних (їх виділено жирними лініями), об'єднані ім'ям групи. Це дає змогу зробити над значеннями поля *sr_ball*, що не є умовою групування, певні арифметичні дії, результатом яких є один рядок. До таких арифметичних дій належать операції обчислення кількості записів, суми значень, середнього арифметичного, мінімуму, максимуму тощо. Функція AVG обчислює середнє арифметичне. Результат роботи SQL-оператора SELECT має такий вигляд:

Група	Середній бал
535	3,845
535a	4,545

Оператор, який вирішує цю задачу, має вигляд

```
SELECT g.name AS 'Група', AVG (s.sr_ball) AS 'Середній бал'
FROM Groups g, Student s WHERE g.id = s.id_groups
GROUP BY g.Name
```

Вираз GROUP BY SQL-оператора SELECT є механізмом групування полів за певною умовою, який дає змогу знаходити підмножини значень окремого поля (*s.sr_ball*) у термінах іншого поля (*g.name*) і застосовувати

функції агрегування до отриманої підмножини. Завдяки цьому можна комбінувати поля й агрегатні функції в одній пропозиції SELECT.

Приклад. Припустимо є таблиця Usp (Id, Num – номер залікової книжки, Ocenka – оцінка, Id_Sub – код предмета, Date – дата). Визначити найменшу оцінку, отриману кожним студентом:

```
SELECT Num, MIN (Ocenka)
FROM Usp GROUP BY Num
```

Висновок для цього запиту має такий вигляд:

Num	
3126	4
3124	5
3125	3
3123	4

Якщо потрібно задати декілька полів, за якими виконується групування, то їх має бути обов'язково задано в пропозиції SELECT і перелічено через кому в пропозиції GROUP BY, інші поля пропозиції SELECT, які не беруть участі в групуванні, обов'язково має бути задано як параметри агрегатних функцій.

Якщо задане в пропозиції SELECT поле не є частиною умови групування й параметром агрегатної функції, то такий оператор уважається помилковим і не виконується.

Приклад. Припустимо, є таблиця Usp. Визначити найменшу оцінку, отриману кожним студентом за кожен день:

```
SELECT Num, Date, MIN (Ocenka) FROM Usp GROUP BY Num, Date
```

При роботі з пропозицією GROUP BY стикаються з певними труднощами. Основна помилка – неправільне виділення поля або полів групування. Щоб правильно їх виділити, потрібно подумки виконати перші дві фази оператора SELECT – склеювання полів та оброблення умови WHERE.

Якщо вираз GROUP BY не задано, то автоматично проводиться групування за всіма виразами, переліченими в пропозиції SELECT. Це дає змогу виконати агрегатні функції над усім вмістом оператора SELECT так, щоб кожен запис був єдиною групою. Це необхідно в тих випадках, коли потрібно, наприклад, підрахувати загальну кількість записів у таблиці:

```
SELECT COUNT (*) FROM Student.
```

COUNT(*) містить записи з NULL-значеннями, а також дублікати, тому DISTINCT у цьому випадку не можна використати.

Вираз HAVING

Приклад. Припустимо, є таблиця Usp (Id, Num, Ocenka, Id_Sub, Date). Визначити найменшу оцінку, отриману кожним студентом, виділити тих, що мають оцінки «2» і «3»:

```
SELECT Num, MIN (Ocenka) FROM Usp
GROUP BY Num
HAVING MIN (Ocenka) <4 (або HAVING MIN (Ocenka) IN (3, 2))
```

Num	Ocenka
3126	4
3124	5
3125	3
3123	4

→

Num	MIN(Ocenka)
3125	3

Вираз HAVING:

1) задає умову всередині групи, визначеної з допомогою пропозиції GROUP BY;

2) має посилатися лише на агрегатні функції й поля, вибрані в GROUP BY;

3) може мати лише аргументи з єдиним значенням для групи вихідних даних;

4) для задання умов використовуються ті ж самі операції, що й у пропозиції WHERE. Відмінність полягає в тому, що вираз WHERE виконується перед групуванням і, отже, з його допомогою неможливо поставити умову, що використовує результати групування.

Приклад. Отримати середній бал груп, вивести перелік груп, для яких середній бал є вищим за 4,5.

Таке завдання потребує групування й відсіювання певної кількості результатів після його виконання:

```
SELECT g.name AS 'Група', AVG (s.sr_ball) AS 'Середній бал'
FROM Groups g, Student s WHERE g.id = s.id_group
GROUP BY g.name HAVING AVG (s.sr_ball)> 4,5
```

В іншому спосіб задання й робота пропозиції HAVING нічим не відрізняються від пропозиції WHERE.

8.2. Вираз ORDER BY

SQL-оператор SELECT автоматично не сортує дані. У SQL-операторі SELECT існує розташований останнім вираз ORDER BY, який дає змогу задати набір полів і порядок їх сортування.

Синтаксис пропозиції ORDER BY:

ORDER BY ім’я_поля1 [ASC | DESC] [, ім’я_поля2 [ASC | DESC], ...]
(to ascend – підніматися, to descend – спускатися)

Особливості пропозиції ORDER BY:

1. Параметр «ім'я_поля» задає поле пропозиції SELECT, за яким виконується сортування.
2. Поля, де відбувається упорядкування, має бути вказано в пропозиції SELECT.
3. Ключові слова ASC (за зростанням) і DESC (за убиванням) визначають порядок (напрямок) сортування. За замовчуванням сортування виконується за зростанням (ASC). Одночасно ASC і DESC не можуть бути заданими.
4. Якщо потрібно сортування не за одним, а за кількома полями, то їх перелічують через кому. Причому для кожного поля можна задати напрямок сортування.
5. Якщо задано кілька полів, то поле, розташоване спочатку, має перевагу перед полем, заданим після нього.
6. Імена полів при сортуванні можна задавати іменами або номерами стовпців.

Приклад. Вивести список студентів, відсортований за зростанням імен:

```
SELECT name AS 'ПІБ' FROM Student ORDER BY name ASC
```

У цьому випадку ASC можна не вказувати.

Приклад. Інформацію з таблиці з даними про студентів упорядкувати за зменшенням розміру стипендії (поле Stip), а для студентів, які мають однаковий її розмір, – в алфавітному порядку їх прізвищ (поле Name):

```
SELECT * FROM Student ORDER BY Stip DESC, Name ASC
```

Для зазначенних полів, за якими впорядковуються вихідні дані, можна використовувати їх порядкові номери – номери стовпців у визначенні вихідних даних оператора SELECT (а не стовпців у базовій таблиці), тобто перше поле в операторі SELECT є для пропозиції ORDER BY полем з номером 1 незалежно від його розташування в базовій таблиці.

Приклад. Вивести перелік студентів, відсортований за іменами:

```
SELECT name AS 'П.І.Б.', stip AS 'Стипендія' FROM Student
ORDER BY 1 ASC
```

Якщо в полі для впорядкування вихідних даних існують NULL-значення, то всі вони знаходяться в кінці або передують всім іншим значенням цього поля. Конкретний варіант не зумовлено стандартом ANSI.

ORDER BY може використовуватися з GROUP BY для впорядкування груп. Цей вираз ORDER BY завжди виконується останнім.

Приклад. Припустимо, що є таблиця Usp (Id, Num, Ocenka, Id_Sub, Date). Знайти найменшу оцінку, отриману кожним студентом, упорядкувати їх за номерами залікових книжок:

```
SELECT Num, MIN (Ocenka) FROM Usp GROUP BY Num
ORDER BY Num;
```

Висновок для цього запиту має такий вигляд:

Num	
3123	4
3124	5
3125	3
3126	4

До цього вихідні дані було згруповано з довільним порядком груп; тепер групи вишиковано в певній послідовності. Оскільки в команді ORDER BY не вказано спосіб упорядкування, за замовчуванням застосовується такий, що зростає.

8.3. Порядок виконання пропозицій у SQL-операторі SELECT

Розуміння порядку оброблення даних у SQL-операторі SELECT значно спрощує його використання. Пропозиції завжди виконуються в тому порядку, у якому їх було задано під час опису оператора SELECT (рис. 8.1). Саме тому користувач має можливість пропустити деякі пропозиції, якщо вони не потрібні, але він не має права змінювати порядок їх визначення.

Першою виконується пропозиція SELECT, що вибирає з таблиці значення всіх полів таблиць і формує з них проміжну таблицю в 1НФ. Існування DISTINCT впливає на вміст цієї таблиці, при цьому видаляються записи, які повторюються.

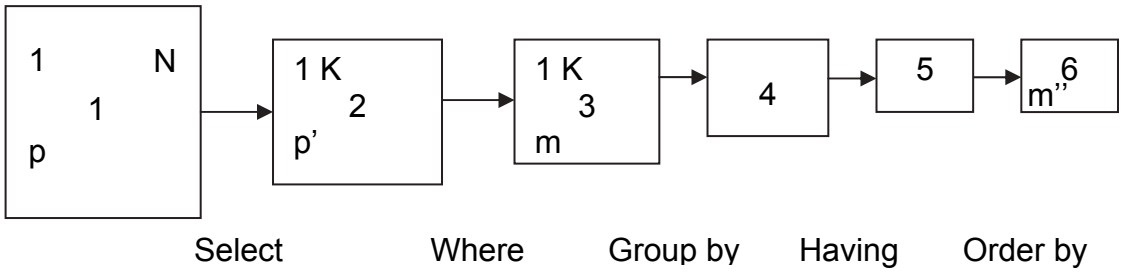


Рис. 8.1. Схема виконання пропозицій SQL-оператора SELECT: N – кількість стовпців у базовій таблиці, K <= N , p – кількість рядків у базовій таблиці, p' <= p, m <= p', m' < = m, m'' <= m'

Потім проміжна таблиця надходить у пропозицію WHERE, де відбираються її записи.

Далі проміжна таблиця потрапляє в пропозицію GROUP BY, де виявляються групи й виконуються задані над ними дії. Після цього результат надходить у пропозицію HAVING, яка може своєю умовою відсікти ще частину даних.

Після цього результат сортується згідно з означеннями пропозиції ORDER BY і надходить до користувача.

Якщо немає однієї з пропозицій WHERE, HAVING або ORDER BY, то це приводить до пропускання цього етапу, що прискорює оброблення запиту користувача. У SQL-операторі SELECT завжди виконується групування даних. Якщо пропозицію GROUP BY не задано, то групування автоматично відбувається за всіма полями.

Контрольні запитання

1. Якими є особливості пропозиції GROUP BY?
2. Якими є особливості пропозиції HAVING?
3. Якими є особливості пропозиції ORDER BY?
4. Яким є порядок виконання пропозицій SQL-оператора SELECT?

Лабораторна робота № 4

СТВОРЕННЯ ЗАПИТІВ ДО ДЕКІЛЬКОХ ТАБЛИЦЬ БАЗИ ДАНИХ

Постановка завдання

На основі таблиць фізичної моделі даних, бізнес-правил предметної області та скриптів формування таблиць БД створити такі оператори:

1. SELECT, який об'єднує дві й більше таблиць з простою умовою.
2. SELECT, який пов'язує дві й більше таблиць з більш складною умовою.

3. SELECT, що використовує агрегатні функції і містить групування (GROUP BY та HAVING).

4. SELECT, що об’єднує кілька запитів разом (UNION).

5. SELECT, що здійснює самооб’єднання.

6. SELECT, що реалізує зовнішнє об'єднання таблиць (ліве, праве й повне).

7. SELECT з використанням вкладених підзапитів.

Для створення зазначених скриптів і запитів використовувати команди мови маніпулювання даними DML.

Письмовий звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш, де наведено назву лабораторної роботи, прізвище, ім'я, по-батькові, номер групи виконавця, дату складання.

2. Діаграму БД.

3. Заповнені таблиці БД.

4. SQL-оператори SELECT для роботи з БД.

5. Приклади роботи основних операторів відповідно до поставленого завдання (у вигляді оператор + результат його роботи).

6. Висновки (відобразити особливості занесення й модифікації інформації в БД, SQL-операторів SELECT і шляхи подальшої модернізації БД).

Контрольні запитання

1. До яких мов належить SQL?
2. Які форми мови SQL існують?
3. Якими є компоненти мови SQL?
4. Якими є основні пропозиції компонент мови SQL?
5. Яким є синтаксис оператора SELECT?
6. Якими є основні особливості пропозиції SELECT?
7. Якими є основні особливості пропозиції FROM?
8. Якими є основні особливості пропозиції WHERE?
9. Якими є особливості пропозицій GROUP BY та HAVING?
10. Якими є основні особливості пропозиції ORDER BY?
11. Якими є особливості операторів BETWEEN, IN, IS NULL, LIKE?
12. Як здійснюється внутрішнє об'єднання таблиць у SQL?
13. Як виконується самооб’єднання таблиць у SQL?
14. Як здійснюється зовнішнє об'єднання таблиць у SQL?
15. Якими є основні правила роботи з псевдонімами в SQL?

Лекція 9. ВКЛАДЕНІ ЗАПИТИ

9.1. Поняття вкладених запитів

Підзапит (вкладений запит) – це запит, який міститься у виразі пропозиції WHERE або HAVING іншого запиту для визначення додаткових обмежень на дані, що виводяться. Підзапит у запиті, у якому він міститься, використовують для накладення умов на виведені дані. Підзапити можуть використовуватися з операторами SELECT, INSERT, UPDATE або DELETE.

Підзапити є необхідними для розміщення в запитах таких умов, точні дані для яких є невідомими, тим самим розширюючи можливості й гнучкість SQL.

Щоб оцінити зовнішній запит, SQL спочатку має оцінити вкладений запит усередині пропозиції WHERE.

Приклад. Нехай існує таблиця USP (Id, Id_Stud, Ocenka, Id_Sub, Date), відомо прізвище студента («Петров»), але невідомо значення поля Id_Stud (зовнішній ключ таблиці USP) для нього. Щоб витягти дані про всі оцінки цього студента, можна записати такий запит:

```
SELECT * FROM U sp WHERE Id_Stud =
(SELECT Id FROM Student WHERE Name = 'Петров')
```

Запит SQL із зв'язаним підзапитом працює в такому порядку:

– вибирається рядок з таблиці USP, ім'я якого зазначено в зовнішньому запиті;

– виконується підзапит, і отримане значення застосовується для аналізу цього рядка в умовах пропозиції WHERE зовнішнього запиту;

– за результатом оцінювання цієї умови приймається рішення про включення або невключення рядка до складу вихідних даних;

– процедура повторюється для наступного рядка таблиці зовнішнього запиту.

Наведений запит є коректним лише в тому випадку, якщо після виконання зазначеного в дужках підзапиту повертається єдине значення. Якщо після виконання підзапиту буде повернуто кілька значень, то підзапит є помилковим. У цьому прикладі це станеться, якщо в таблиці STUDENT буде декілька записів зі значеннями поля Name = 'Петров'.

Отже, необхідно, щоб для операторів відношення (>, <, > =, <=, <>) підзапит повертав одне й лише одне значення. Якщо після виконання підзапиту буде кілька значень, то оцінювати предикат основного запиту щодо істинності або хибності не можна, це призведе до оцінювання запиту як помилкового.

У деяких випадках для гарантії отримання одного значення внаслідок виконання підзапиту застосовується DISTINCT.

Правила для складанні підзапитів:

– підзапит необхідно оформляти в круглих дужках;

– підзапит може посилатися лише на один стовпець у виразі ключового слова SELECT, за винятком випадків, коли в головному запиті використовується порівняння з кількома стовпцями з підзапиту;

– у підзапиті не можна використовувати пропозицію ORDER BY, а лише GROUP BY;

– підзапит, який повертає кілька рядків даних, можна використовувати лише в операторах, що допускають множину значень, наприклад в IN;

– у списку ключового слова SELECT не допускаються посилання на значення типу BLOB, ARRAY, CLOB або NCLOB;

– підзапит не можна безпосередньо використовувати як аргумент, що допускає багато значень функції;

– операцію BETWEEN відносно підзапитів використовувати не можна, але її можна використовувати в самому підзапиті.

Предикати підзапитів – незворотні, тобто умови, що містять підзапити, використовують конструкцію <вираз> <оператор> <підзапит> та в жодному разі не <підзапит> <оператор> <вираз> або <підзапит> <оператор> <підзапит>.

– оператор IN (на відміну від операторів BETWEEN, LIKE, IS NULL) широко застосовується в підзапитах. Цей оператор установлює набір значень, які тестуються на збіг з іншими значеннями для установлення істинності предиката. Коли використовується IN з підзапитом, SQL просто формує цей набір з виведення підзапиту.

Приклад. Вибрати дані про всі оцінки (таблиця USP) студентів з м. Харкова:

```
SELECT * FROM Usp WHERE Id_Stud IN
(SELECT Id FROM Student WHERE City = 'Харків')
```

– у будь-якій ситуації, коли застосовується оператор рівності (=), краще використовувати IN. На відміну від запиту з оператором рівності (=) IN не може змусити запит потерпіти невдачу, якщо підзапит вибрав більше одного значення;

– команда SELECT * **...** не може використовуватися в підзапиті;

– одними із функцій, які автоматично видають в результаті єдине значення для будь-якої кількості рядків, є агрегатні функції.

Приклад. Вибрати дані про всі оцінки студентів, значення яких є вищими від середньої оцінки (це запит із зв'язаним підзапитом):

```
SELECT * FROM Usp WHERE Ocenka >
(SELECT AVG (Ocenka) FROM Usp).
```

– підзапити можна застосовувати при використанні пропозиції GROUP BY усередині пропозиції HAVING.

Приклад. Потрібно визначити кількість предметів навчання, складених студентами з оцінкою, що перевищує середнє значення оцінки студента з номером зовнішнього ключа 301 (запит із зв'язаним підзапитом):

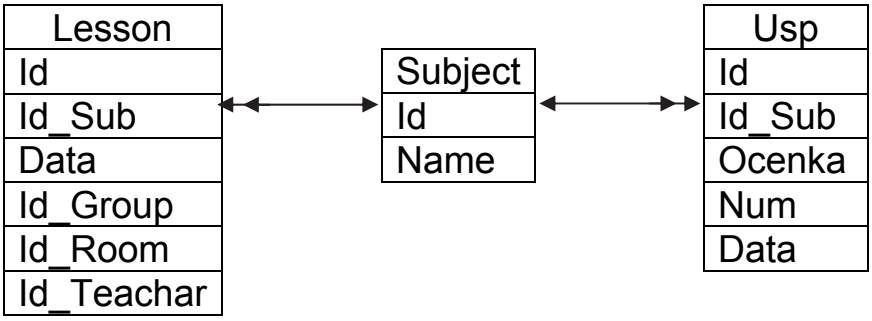
```
SELECT COUNT (DISTINCT I d _ Sub ), Ocenka FROM Usp
GROUP BY Ocenka HAVING Ocenka >
(SELECT AVG (Ocenka) FROM Usp WHERE Id_Stud = 301);
Інший варіант рішення задачі:
SELECT COUNT (DISTINCT Id_Sub), Ocenka FROM Usp
WHERE Ocenka > (SELECT AVG (Ocenka)
FROM Usp WHERE Id_Stud = 301) GROUP BY OCENKA
```

9.2. Формування зв'язаних підзапитів

Зв'язаний (співвіднесений, корельований) підзапит залежить від інформації, яка надається головним запитом.

Коли в SQL використовуються підзапити, у внутрішньому запиті можна посилатися на таблицю, ім'я якої зазначено в пропозиції FROM зовнішнього запиту, тим самим формуючи зв'язаний підзапит (або співвіднесений підзапит – correlated subquery). У цьому випадку підзапит виконується повторно, один раз для кожного рядка таблиці з основного запиту. Зв'язані підзапити належать (через складність їх оцінок) до найбільш тонких понять у SQL. Однак зв'язані підзапити – дуже потужний засіб, що виконує дуже складні функції при досить компактних командах.

Приклад. Вибрати відомості про всі предмети навчання, за якими проводився іспит 20 грудня 2020 р. У таблиці USP є відомості про оцінки й дати іспитів, у таблиці LESSON – відомості про всі заняття й іспити:




```
SELECT * FROM Lesson LE WHERE '20/12/2020' =
(SELECT Date FROM Usp WHERE LE.Id_Sub = Usp.Id_Sub)
```

Процедура, яка виконується зв'язаним запитом:

– вибір поточного рядка з таблиці LESSON у зовнішньому запиті;

– збереження значення поточного рядка у псевдонімі LE, визначеному у FROM зовнішнього запиту;

– виконання підзапиту, при цьому всюди, де знайдено псевдонім із зовнішнього запиту, використовується значення з поточного рядка (це називають зовнішнім посиланням);

– оцінка предиката зовнішнього запиту на базі результатів підзапиту;

– описана послідовність повторюється для наступного рядка з таблиці зовнішнього запиту доти, доки всі рядки з таблиці LESSON не буде перевірено.

У деяких СКБД для виконання цього запиту може знадобитися перетворення значення дати на символьний тип.

Це ж завдання можна вирішити з допомогою операції об'єднання таблиць.

Можна використовувати підзапити, що зв'язують таблицю зі своєю власною копією.

Приклад. Знайти всі оцінки з дисциплін, що є вищими за середню з цих дисциплін:

```
SELECT * FROM Usp U1 WHERE Ocenka >
(SELECT AVG (Ocenka) FROM USP U2
WHERE U2. ID_Sub = U1. ID_Sub)
```

Слід мати на увазі, що реальний час виконання запиту здебільшого залежить від оптимізатора запитів конкретної СКБД.

Вираз GROUP BY дає змогу групувати записи, що виводяться запитом, за значенням деякого поля. Використання пропозиції HAVING дає змогу під час виведення здійснювати фільтрацію таких груп.

Предикат пропозиції HAVING оцінюється не для кожного рядка результату, а для кожної групи вихідних записів, сформованої пропозицією GROUP BY зовнішнього запиту.

Приклад. Необхідно за даними з таблиці USP (Id, Id_Stud, Ocenka, Id_Sub, Date – дата іспиту) визначити суму отриманих студентами оцінок (поле OCENKA), згрупувавши їх за датами іспитів (DATE) і вилучивши ті дні, коли кількість студентів, які склали іспити протягом дня, була меншою від 10:

```
SELECT Date, SUM (Ocenka) FROM Usp U1
GROUP BY Date HAVING 10 <
(SELECT COUNT (Ocenka) FROM Usp U2 WHERE U1.Date=U2.Date)
```

Підзапит обчислює кількість рядків з однією і тією ж датою, що збігається з датою, для якої сформовано чергову групу основного запиту.

Зв'язані підзапити за своєю суттю є близькими до з'єднань (JOIN) – обидві конструкції містять перевірку кожного запису однієї таблиці з кожним записом іншої або з псевдонімом з тієї ж таблиці, при цьому більшість операцій є схожими.

Контрольні запитання

1. Означення підзапиту.
2. Основні правила при складанні підзапитів.
3. Особливості роботи зі зв'язаним підзапитом.

Лекція 10. ФОРМУВАННЯ БАГАТОТАБЛИЧНИХ ЗАПИТІВ

Сучасні БД зазвичай нормалізовано, тому що дані розподіляються за багатьма невеликими таблицями з метою усунення повторення даних, зменшення місткості пам'яті, підвищення продуктивності й забезпечення цілісності даних. Однак майже завжди потрібно вилучати дані з кількох таблиць. Об'єднання (з'єднання, багатотабличний запит) повністю призначено для забезпечення вибірки даних з кількох таблиць і включення цих даних в один результівний набір.

Одна з найбільш важливих характеристик запитів SQL міститься в їх здатності визначати зв'язки між множиною таблиць і відображати інформацію, яку вони містять, у термінах цих зв’язків. Таку операцію називають об'єднанням (з'єднанням), це одна з найбільш значних операцій для реляційних БД. З допомогою з'єднань безпосередньо зв'язується інформація, що міститься в кількох таблицях або між окремими частинами однієї таблиці.

У багатотабличному запиті під час операції об'єднання у пропозиції FROM імена таблиць перелічуються через кому.

Об'єднати таблиці можно трьома способами:

– з використанням пропозиції UNION;

– з допомогою встановлення зв'язків між таблицями в предикаті запиту (за умови, організованої пропозицією WHERE);

– з допомогою оператора об'єднання JOIN.

Другий спосіб називають внутрішнім об'єднанням (з'єднанням), яке повертає лише ті рядки, для яких умова об'єднання набуває значення True. Таблиці, які задіяно у внутрішньому об'єднанні, є рівноправними.

З допомогою конструкції *Join* реалізується внутрішнє й зовнішнє об'єднання.

10.1. Вираз UNION

Вираз UNION використовується для розміщення кількох запитів разом та об'єднання їх виведення в єдиний набір рядків і стовпців.

Приклад. Отримати список усіх студентів і викладачів, прізвища яких знаходяться між буквами «K» і «C»:

```
SELECT SFAM AS 'Прізвище', SIMA AS 'Ім'я' FROM STUDENT
WHERE SFAM BETWEEN 'K' AND 'C'
UNION
SELECT TFAM, TIMA FROM TEARCHER
WHERE TFAM BETWEEN 'K' AND 'C'
```

Особливості виведення:

1. У MS SQL Server заголовок стовпців формується першим запитом.

2. Під час об'єднання стовпці мають бути сумісними для об'єднання, тобто необхідно використовувати в запиті однакову кількість стовпців в однаковому порядку. Має бути наявною сумісність типів, тобто якщо порожні значення NULL є забороненими в будь-якому стовпці об'єднання, то вони мають бути забороненими і в усіх інших стовпцях.

3. Вираз UNION автоматично вилучає дублікати рядків з виведення.

4. Необхідно додавати константи й вирази в пропозицію UNION (має бути сумісність констант і виразів):

```
SELECT 'Студент' AS 'Таблиця', SFAM AS 'Прізвище', SIMA AS 'Ім'я'
FROM STUDENT
UNION
SELECT 'Викладач', TFAM, TIMA FROM TEARCHER
```

Якщо не буде псевдоніма «Таблиця», то SQL Server присвоїть цьому стовпцю ім'я *No column name*.

5. Вираз ORDER BY застосовується й для впорядкування вихідних даних об'єднання. Завдання впорядкування здійснюється за номером стовпця:

```
SELECT 'Студент', SFAM, SIMA FROM STUDENT
UNION
SELECT 'Викладач', TFAM, TIMA
FROM TEARCHER ORDER BY 2 ASC
```

У цьому прикладі виконано впорядкування за зростанням значень полів SFAM і TFAM.

Можна впорядкувати вихідні дані відповідно до значень одного або декількох полів: для кожного з полів незалежно за зростанням чі зменшенням (ASC або DESC) (так само, як і для вихідних даних одного запиту). Число 2 в пропозиції ORDER BY задає номер стовпця в упорядкованому списку пропозиції SELECT.

6. Якщо об'єднання виконується більш ніж для двох запитів, то для впорядкування обчислень потрібно використовувати круглі дужки (круглі дужки можуть визначати порядок усунення дублікатів). Інакше замість того, щоб задати

query X UNION query Y UNION query Z,

необхідно конкретизувати

або (query X UNION query Y) UNION query Z,
або query X UNION (query Y UNION query Z).

Це потрібно тому, що UNION і UNION ALL можна комбінувати для вилучення одних дублікатів без усунення інших. Вираз (query X UNION ALL query Y) UNION query Z не обов'язково генерує ті ж самі вихідні дані, що й вираз query X UNION ALL (query Y UNION query Z), якщо є дублікати рядків, що підлягають вилученню з вихідних даних.

10.2. Внутрішнє об'єднання таблиць

1. Об'єднання двох таблиць з допомогою встановлення зв'язків.

Приклад. Вибрати з таблиць Teacher (id (код_викладача – первинний ключ), name (ім’я_викладача), id_dep (код_кафедри)), Subject (id (код_предмета), name (ім’я_предмета), id_Teach (код_викладача – зовнішній ключ)) інформацію про викладачів (ПІБ) і дисципліни (Предмети), які викладач веде:

```
SELECT T. Name AS 'ПІБ', S. Name AS 'Предмет'
FROM Teacher T, Subject S WHERE T.id = S.id_Teach
```

Припустимо, id – первинний ключ таблиці предка Teacher, а id_Teach – зовнішній ключ таблиці нащадка Subject. Табліці *Teacher* і *Subject* з'єднано з допомогою зв’язку між полями id і id_Teach. Таке з'єднання таблиць називають з'єднанням з допомогою відношення предок – нащадок. У цьому випадку між таблицями створюється відношення «один до багатьох».

SQL не потребує, щоб зв’язані стовпці двох (або більше) таблиць обов'язково являли собою пару первинний ключ – зовнішній ключ. Будь-які два стовпці з двох таблиць можуть бути зв'язаними стовпцями, якщо лише вони мають порівнянні типи даних. Такі стовпці створюють між таблицями відношення «багато до багатьох».

При виконанні багатотабличного запиту SQL досліджує кожну комбінацію рядків двох або більше можливих таблиць і перевіряє ці комбінації за їх предикатами. Якщо комбінація значень рядків дає таке

значення, яке робить предикат істинним, то з комбінації цих рядків буде вибрано значення для виведення.

Об'єднання, що використовує предикати, оновою яких є рівність, називають еквіоб'єднанням (об'єднанням за рівністю). Еквіоб'єдінання є найбільш поширеним але існують і інші типи об'єднань. Фактично в об'єднанні можна використовувати будь-який оператор порівняння.

2. Об'єднання трьох таблиць.

Приклад. Вивести список оцінок, виставлених тим чи іншим викладачем, у вигляді таблиці з полями (ПІБ, Предмет, Оцінка). Припустимо, що існують таблиці Teacher (id (код_викладача), Name (ім’я_викладача), id_dep (код_кафедри)), Subject (id (код_предмета), Name (ім’я_предмета), id_Teach (код_викладача)) і Usp (id, Num (залікова книжка), Ocenka (оцінка), id_Pred (код_предмета), Date):

```
SELECT T.Name 'ПІБ', S.Name 'Предмет', U.Ocenka 'Оцінка'
FROM Teacher T, Subject S, Usp U
WHERE T.id = S.id_Teach AND S.id = U.id_Pred
```

Тут виконано запит, який об'єднує три таблиці: Teacher, Subject і Usp.

Отже, об'єднання дає змогу зробити складні порівняння між будь-якими полями будь-якої кількості таблиць і використовувати отримані результати, для того щоб вирішувати, яку інформацію хотілося би бачити.

3. Об'єднання двох (або більше) копій одиничної таблиці (самооб’єднання, тета-об'єднання або рефлексивне об'єднання).

Об'єднання таблиці з її же копією означає, що будь-який рядок таблиці (один у кожен момент часу) можна комбінувати з її копією та з будь-яким іншим рядком цієї ж таблиці. Кожна така комбінація оцінюється в термінах предиката, як і при об'єднанні кількох різних таблиць. Це дає змогу легко створювати певні види зв'язків між різними записами всередині єдиної таблиці, наприклад, здійснювати пошук пар рядків із загальним значенням поля.

Об'єднання таблиці зі своєю копією можна подати так: реально копія таблиці не створюється, але SQL виконує команду так, ніби-то робилося саме це. Цей тип об'єднання не відрізняється від звичайного об'єднання двох таблиць, за винятком того, що в цьому випадку таблиці є ідентичними.

При об'єднанні таблиці зі своєю копією для посилання до стовпців необхідно мати два різних імені для однієї і тієї ж таблиці, що виконується з допомогою тимчасових імен – псевдонімів, визначених у пропозиції FROM.

Приклад. Визначити студентів, які мають однакові стипендії:

```
SELECT First.Name, Second.Name, First.Stip
FROM Student First, Student Second WHERE First.Stip = Second.Stip
```

Результат такого запиту має вигляд

Name	Name	Stip
Сєров	Сєров	1000
Сєров	Іванов	1000
…	…	...
Іванов	Сєров	1000
Іванов	Іванов	1000

У цьому прикладі SQL поводиться так, якби він об'єднував дві різні таблиці з іменами First і Second, тобто псевдоніми дозволяють одній і тій же таблиці бути обробленою незалежно.

Логіка цього запиту така: з таблиці Student вибирається черговий рядок і запам'ятовується під першим псевдонімом. Після цього SQL почне перевіряти його в комбінації з кожним рядком таблиці Student під іншим псевдонімом. Якщо комбінація рядків задовольняє предикату, то відповідні поля з неї вибираються для виведення.

Особливості роботи з псевдонімами:

- можуть використовуватися в пропозиції SELECT до їх оголошення в пропозиції FROM;

– псевдоніми таблиць можуть збігатися з іменами таблиць;

– допускається використання будь-якої кількості псевдонімів для однієї таблиці в запиті (хоча більше двох псевдонімів у запиті – надмірність);

– SQL відхилить запит, якщо псевдоніми далі не буде визначено в пропозиції FROM;

– у пропозиції SELECT допускається не використовувати кожен псевдонім або таблицю, які наводилися в пропозиції FROM запиту;

– в одному запиті можна змішувати написання імен таблиць і псевдонімів;

– псевдонім існує лише під час виконання команди, а після завершення запиту псевдоніми запиту більше не мають ніякого значення;

– псевдоніми використовуються для спрощення запису довгих і складних імен таблиць і мінімізації синтаксичних помилок.

Недолік цього запиту – висновок має два значення для кожної комбінації прізвищ у різному порядку. Це пов'язано з тим, що поточне значення в першому псевдонімі спочатку вибирається в комбінації зі значенням в іншому, а потім – навпаки, крім того, кожен рядок порівнюється сам з собою, наприклад призвище «Сєров» із призвищем «Сєров».

Спосіб усунення цього недоліку – накладення умови порядку на два значення так, щоб одне значення могло бути меншим за друге або передувало йому в алфавітному порядку.

Це зробить предикат асиметричним відносно зв'язку, тому ті ж самі значення у зворотному порядку не буде вибрано знову.

Тому попередній запит потрібно модифікувати так:

```
SELECT First.Name, Second.Name, First.Stip
FROM Student First, Student Second
WHERE First.Stip = Second.Stip AND First.Name < Second.Name.
```

Результат такого запиту має вигляд

…	…	…
Іванов	Сєров	1000
…	…	…

Зокрема, призвище «Іванов» передує призвищу «Сєров» в алфавітному порядку, тому комбінація задовольняє обидві умови й виникає у висновку. Для комбінації «Сєров – Іванов», як і для «Сєров – Сєров», друга умова запиту не виконується. Якщо ж є необхідність залишити у висновку запиту комбінацію виду «Сєров – Сєров», то в запиті слід використовувати порівняння виду <= замість <.

У SQL дозволяється створення об'єднання, яке містить псевдоніми одиничної таблиці й різні таблиці.

Отже, операція об'єднання з'єднує інформацію з двох (або більше) таблиць (однакових або різних), формуючи пари зв'язаних рядків з них. Об'єднану таблицю утворюють пари тих рядків з різних таблиць, де зв'язані стовпці містять однакові значення.

10.3. Конструкція Join

З допомогою конструкції *Join* реалізується внутрішнє (Inner Join або просто Join) і зовнішнє (Left | Right | Full Outer | Cross Join) об'єднання.

Зовнішнє об'єднання повертає всі рядки з однієї таблиці (головної) і лише ті рядки з другої таблиці (підпорядкованої), для яких умова об'єднання набуває значення True. Рядки другої таблиці, що не задовольняють умову об'єднання (тобто мають значення False), набувають значення Null і в результівному наборі не виводяться.

У стандарті ANSI SQL-92 умови об'єднання (з'єднання) записуються в пропозиції From відповідно до такого синтаксису:

From <таблиця> [Inner | Left | Right | Full Outer | Cross] Join <таблиця> On <умова пошуку>

(у полі 'умови пошуку' може бути задано будь-які критерії порівняння рядків двох таблиць, що об'єднуються, у тому числі з використанням булевих операторів).

Усі різновиди конструкції *Join* мають одну загальну відмінну рису: один їх рядок узгоджується з одним або кількома іншими рядками для отримання результівного рядка, що являє собою надмножину, створену шляхом з'єднання полів із кількох рядків.

У поєднанні розрізняють ліву й праву сторони. Лівою стороною вважається таблиця, зазначена в пропозиції From першою, а правою – така, що зазначена другою.

Конструкція *Inner Join* реалізує внутрішнє об'єднання й повертає рядки, узгоджені за всіма полями, позначені як такі, що використовуються для з'єднання. Внутрішнє з'єднання є винятковим, тобто будь-який рядок, для якого немає відповідності в обох таблицях, неминуче вилучається з остаточного варіанта результівного набору.

Існує чотири види зовнішнього об'єднання:

– ліве (Left Join) – запит повертає всі рядки з лівої таблиці, а з правої – лише рядки, що задовольняють умову з'єднання (рядки правої таблиці, що не задовольняють умову об'єднання, отримують значення *Null* і в результівному наборі не виводяться);

– праве (Right Join) – усе навпаки відносно лівого зовнішнього з'єднання;

– повне (Full Outer Join) – запит повертає всі рядки лівої і правої таблиць, але в особливому порядку;

– перехресне (Cross Join) – запит повертає всі рядки лівої таблиці й перший рядок правої таблиці, усі рядки лівої таблиці й другий рядок правої таблиці і т. д.

Приклад. Припустимо, у БД використовуються таблиці T1 і T2.

T1	
Name	City
Masha	Kharkov
Katja	Kiev
Lena	Kursk
Lena	Kharkov

T2	
Name	City
Petja	Kiev
Lesha	Kharkov
Vitja	Poltava

Необхідно вибрати постачальників, які проживають в одному місті:

1. Запит, який реалізує внутрішнє об'єднання в стандарті SQL-89:

```
Select T1.name as 'Ім'я1', T1.city as 'Місто1', T2.name as 'Ім'я2', T2.city
as 'Місто2' From T1, T2 Where T1.city = T2.city;
```

Результат такого запиту має вигляд

Ім’я1	Місто1	Ім’я2	Місто2
Masha	Kharkov	Lesha	Kharkov
Katja	Kiev	Petja	Kiev
Lena	Kharkov	Lesha	Kharkov

2. Запит, який реалізує внутрішнє об'єднання в стандарті SQL-92 (стандарт SQL-92 допускає використання синтаксису стандарту SQL-89):

```
Select T1.name 'Ім'я1', T1.city 'Місто1', T2.name 'Ім'я2', T2.city 'Місто2'
From T1 Inner Join T2 – аналогічний результат без Inner
On T1.city = T2.city
```

Істотно, що висновок цього запиту є аналогічним висновку до п. 1.

3. Запит, який реалізує внутрішнє об'єднання між частинами однієї таблиці: визначити імена працівників, які живуть в одному місті й назву цього міста:

```
SELECT T1.Name, T1.City, T2.Name, T2.City FROM T1
INNER JOIN T1 T2 ON (T1.City = T2.City) And (T1. Name > T2.Name )
```

4. Запит, який реалізує ліве зовнішнє об'єднання в стандарті SQL-92:

```
Select T1.name as 'Ім'я 1', T1.city as 'Місто 1', T2.name as 'Ім'я 2',
T2.city as 'Місто 2' From T1 Left Join T2 On T1.city = T2.city
```

Результат такого запиту має вигляд

Ім’я1	Місто1	Ім’я2	Місто2
Masha	Kharkov	Lesha	Kharkov
Katja	Kiev	Petja	Kiev
Lena	Kursk	Null	Null
Lena	Kharkov	Lesha	Kharkov

Замість даних, яких немає, підставляється Null-значення.

5. Запит, який реалізує праве зовнішнє об'єднання в стандарті SQL-92:

```
Select T1.name as 'Ім'я1', T1.city as 'Місто1', T2.name as 'Ім'я2', T2.city
as 'Місто2' From T1 Right Join T2 On T1.city = T2.city
```

Результат такого запиту має вигляд

Ім’я1	Місто1	Ім’я2	Місто2
Katja	Kiev	Petja	Kiev
Masha	Kharkov	Lesha	Kharkov
Lena	Kharkov	Lesha	Kharkov
Null	Null	Vitja	Poltava

6. Запит, який реалізує праве зовнішнє об'єднання в стандарті SQL-92 з додатковою умовою:

```
Select T1.name as 'Ім'я1', T1.city as 'Місто1', T2.name as 'Ім'я2', T2.city
as 'Місто2' From T1 Right Join T2 On T1.city = T2.city
And not (T1.city = 'Kiev') Order by 3
```

Результат цього запиту має вигляд

Ім’я1	Місто1	Ім’я2	Місто2
Masha	Kharkov	Lesha	Kharkov
Lena	Kharkov	Lesha	Kharkov
Null	Null	Petja	Kiev
Null	Null	Vitja	Poltava

7. Запит, який реалізує повне зовнішнє об'єднання (до результату має бути включено всі рядки з обох сторін з'єднання) у стандарті SQL-92:

```
Select T1.name as 'Ім'я1', T1.city as 'Місто1', T2.name as 'Ім'я2', T2.city
as 'Місто2' From T1 Full Outer Join T2  On T1.city = T2.city
```

Результат такого запиту має вигляд

Ім’я1	Місто1	Ім’я2	Місто2
Masha	Kharkov	Lesha	Kharkov
Katja	Kiev	Petja	Kiev
Lena	Kursk	Null	Null
Lena	Kharkov	Lesha	Kharkov
Null	Null	Vitja	Poltava

8. Запит, який реалізує перехресне об'єднання:

Select T1.name as 'Ім'я1', T1.city as 'Місто1', T2.name as 'Ім'я2', T2.city as 'Місто2' From T1 Cross Join T2

Результат цього запиту має вигляд

Ім’я1	Місто1	Ім’я2	Місто2
Masha	Kharkov	Petja	Kiev
Katja	Kiev	Petja	Kiev
Lena	Kursk	Petja	Kiev
Lena	Kharkov	Petja	Kiev
Masha	Kharkov	Lesha	Kharkov
Katja	Kiev	Lesha	Kharkov
Lena	Kursk	Lesha	Kharkov
Lena	Kharkov	Lesha	Kharkov
Masha	Kharkov	Vitja	Poltava
Katja	Kiev	Vitja	Poltava

Об'єднання Cross Join характеризується тим, що немає операції On.

10.4. Функції SQL

Сучасні СКБД містять убудовані функції, що дають змогу виконувати різні операції над даними, які вилучаються з таблиць, і перекладати частину роботи з оброблення даних на сервер СКБД. Такі функції можуть бути частиною SQL-виразу.

Припустимо, що необхідний результат формується шляхом обчислення синуса над значенням, яке зберігається. Можна витягти дані з таблиці, а потім виконати на клієнтському комп’ютері операцію «синус». Однак немає необхідності це робити, тому що в SQL-операторі SELECT замість імені поля можна поставити SQL-вираз «SIN (ім'я_поля)» і відразу отримати необхідні значення.

Усі SQL-функції класифікують так:

– скалярні – для оброблення одного даного;

– агрегатні – для оброблення багатьох даних.

Скалярні функції поділяють на такі:

1. Робота з числами.
2. Робота із символами.
3. Робота з датою й часом.
4. Явне перетворення типів даних.
5. Шифрування / дешифрування даних.
6. Інші.

У табл. 10.1 наведено основні категорії функцій MS SQL Server.

Таблиця 10.1

Категорії функцій MS SQL Server

Категорія	Зміст
Математичні функції	Для виконання математичних виразів
Функції дати й часу	Для керування датами й часом
Строкові функції	Для керування текстовими даними
Функції агрегування	Для обчислення однини, ґрунтуючись на всіх значеннях у стовпці (AVG, COUNT, COUNT_BIG, GROUPING, MAX, MIN, STDEV, STDEVP, SUM, VAR, VARP)
Функції курсора	Повертають інформацію про курсори (CURSOR_STATUS)
Функції метаданих	Повертають інформацію про об'єкти БД
Функції конфігурації	Повертають інформацію про поточну конфігурацію сервера
Функції безпеки	Повертають інформацію про користувачів
Функції шифрування / дешифрування	Для шифрування й дешифрування даних
Системні функції	Для керування об'єктами на низькому рівні
Функції для тексту й зображень	Для роботи з великими стовпцями й типами даних

Математичні функції

У табл. 10.2 наведено основні математичні функції MS SQL Server.

Таблиця 10.2

Математичні функції

Функція	Призначення
ABS (n)	Повертає абсолютне значення n
ACOS (n)	Повертає арккосинус n
ATAN (n)	Повертає арктангенс n
ATN2	Арктангенс кута, описаного двома кутами
CEIL ING (n)	Найближче ціле, що перевищує n
COS (n)	Косинус n
COT (n)	Котангенс кута n
FLOOR (n)	Найближче ціле, менше від n
LOG	Логарифм за основою 2
LOG10	Логарифм за основою 10

Закінчення табл. 10.2

Функція	Призначення
POWER (m, n)	Повертає m у степені n
RADIANS	Перетворення градусів на радіани
RAND	Генератор випадкових чисел
ROUND	Округлення чисел
SIGN	Повертає знак виразу
SIN (n)	Синус n
SQRT (n)	Корінь квадратний з n
SQUARE	Піднсенення до другого степеня
TAN (n)	Тангенс n

Функції агрегування

Агрегатні функції SQL (табл. 10.3) призначено для виконання дій над множиною значень, які групуються клаузою GROUP BY SQL-оператора SELECT. Результат роботи функції – значення, отримане під час оброблення множини. Усі функції, за винятком COUNT, не враховують у своїй роботі поля зі значенням NULL і мають таку загальну форму подання: ім’я_функції ([DISTINCT | ALL] вираз).

За замовчуванням перед виразом ставять клаузу ALL, яка стверджує, що для оброблення потрапляють всі значення множини. Клауза DISTINCT дає змогу видаляти з оброблення повторювані значення.

Таблиця 10.3

Функції агрегування (агрегатні функції)

Функція	Призначення
AVG	AVG ([ALL|DISTINCT]<expression>) Функція AVG повертає середнє арифметичне значень, поданих у параметрі <expression>. Параметр expression має містити числові значення. NULL-значення ігноруються
COUNT	COUNT ([ALL|DISTINCT]<expression>|*) Функція повертає дані типу *int* про кількість елементів, поданих у параметрі <expression>. Параметр не може належати до типу даних *uniqueidentifier*, *text*, *ntext* або *image*. При використанні значення параметра «*» відбувається повернення даних про кількість рядків у таблиці, дубльовані значення або NULL-значення не вилучаються
COUNT_BIG	COUNT_BIG ([ALL|DISTINCT]<expression>|*) Функція повертає дані про кількість елементів у групі, є аналогічною функції COUNT, але значення яке повертається, має тип *biqint*

Закінчення табл. 10.3

Функція	Призначення
GROUPING	GROUPING (<column_name>) Ця функція додає додатковий стовпець до виведення оператора SELECT
MAX	MAX ([ALL|DISTINCT]<expression>) Функція повертає максимальне зі значень, поданих у параметрі <expression>. При обчисленні функції всі NULL-значення ігноруються
MIN	MIN ([ALL|DISTINCT]<expression>) Функція повертає мінімальне зі значень, поданих у параметрі <expression>. При обчисленні функції всі NULL-значення ігноруються
STDEV	STDEV (<expression>) Функція повертає результат обчислення середньоквадратичного відхилення за всіма значеннями, поданими в параметрі <expression>. При обчисленні функції всі NULL-значення ігноруються
SUM	SUM ([ALL|DISTINCT]<expression>) Функція повертає суму всіх значень, поданих у параметрі <expression>. При обчисленні функції всі NULL-значення ігноруються
VAR	VAR (<expression>) Функція повертає результат обчислення дисперсії за всіма значеннями, поданими в параметрі <expression>. При обчисленні функції всі NULL-значення ігноруються

Особливості запитів з агрегатними функціями:

1. Із функціями SUM та AVG використовуються тільки числові поля, а з COUNT, MAX і MIN – числові або символьні.

2. Функція COUNT (*) підраховує кількість значень у стовпці або рядків у таблиці.

Приклад. Підрахувати кількість студентів, які склали навчальні предмети:

```
SELECT COUNT (DISTINCT Id_Stud) FROM USP
```

Приклад. Підрахувати загальну кількість рядків у таблиці USP:

```
SELECT COUNT (*) FROM USP
```

COUNT (*) містить записи з NULL-значеннямя, а також дублікати, тому DISTINCT у цьому випадку не може бути використаним.

Приклад. Підрахувати кількість не NULL-значень у полі Id_Predm таблиці USP (аргумент ALL означає – «додавати дублікати»):

```
SELECT COUNT (ALL Id_Predm) FROM USP
```

3. Допускається застосування агрегатних функцій з аргументами, які складаються з виразів, що містять одне або кілька полів, при цьому команда DISTINCT забороняється.

Приклад. Знайти максимальну величину проіндексованої стипендії SELECT MAX (STIP * 2) FROM USP

4. Не можна використовувати агрегатну функцію від агрегатної функції.

У списку стовпців, що повертаються, не можна одночасно вказувати агрегатні функції й прості імена стовпців. Такий запит не має сенсу:

```
SELECT Id_Stud, MIN (OCENKA) FROM USP
```

Оскільки перший стовпець створює таблицю, а другий повертає лише одне значення, такий запит спричинить помилку.

5. Вираз GROUP BY дає змогу визначати підмножину значень у полі в термінах іншого поля й застосовувати функцію агрегата до такої підмножини, а також об'єднувати поля й агрегатні функції в єдиній пропозиції SELECT.

Приклад. Визначити найменшу оцінку, отриману кожним студентом:

```
SELECT Id_Stud, MIN (OCENKA) FROM USP GROUP BY Id_Stud
```

Приклад. Визначити найменшу оцінку, отриману кожним студентом за кожен день:

```
SELECT Id_Stud, DATE, MIN (OCENKA)
 FROM USP GROUP BY Id_St ud, DATE
```

Приклад. Визначити найменшу оцінку, отриману кожним студентом за кожен день, меншу від «4»:

```
SELECT Id_Stud, DATE, MIN (OCENKA) FROM USP
GROUP BY Id_Stud , DATE HAVING MIN (OCENKA) <4
```

Аргументи в пропозиції HAVING повинні мати одне значення на групу виведення. Тому наступну команду буде заборонено:

```
SELECT Id_Stud, MIN (OCENKA) FROM USP
GROUP BY Id_Stud HAVING DATE = 06/03/2020
```

Правильний спосіб виконати розглянутий запит:

```
SELECT Id_Stud, MIN (OCENKA) FROM USP
WHERE DATE = 06/03/2020 GROUP BY Id_Stud
```

Особливості роботи агрегатних функцій з полями зі значенням NULL:

1. Якщо які-небудь зі значень, що містяться в стовпці, дорівнюють NULL, то під час обчислення результату функції їх вилучають.

2. Якщо всі значення в стовпці дорівнюють NULL, то функції AVG(), SUM(), MIN(), MAX() повертають значення NULL. Функція COUNT() повертає нуль.

3. Якщо в стовпці немає значень (тобто стовпець порожній), то функції AVG(), SUM(), MIN(), MAX() повертають значення NULL. Функція COUNT() повертає нуль.

4. Функція COUNT(*) підраховує кількість рядків і не залежить від того, чи є в стовпці значення NULL. Якщо рядків у стовпці немає, то ця функція повертає нуль.

Особливості під час роботи з агрегатними функціями і DISTINCT:

1. Якщо використовується DISTINCT і агрегатна функція, то її аргументом може бути лише ім'я стовпця, а не виразу.

2. У функціях MIN(), MAX() так само немає сенсу використовувати DISTINCT. У функції COUNT() можна використовувати DISTINCT, але це потребується не часто.

3. До функції COUNT(*) узагалі не застосовується DISTINCT, оскільки вона просто підраховує кількість рядків.

4. В одному запиті пропозицію DISTINCT можна вживати лише один раз. Якщо її застосовують з аргументом агрегатної функції, то його вже не можна використовувати з жодним іншим аргументом.

Системні функції і функції метаданих

Системні функції й функції метаданих (табл. 10.4) повертають інформацію про SQL Server і дані, що зберігаються в ньому.

Таблиця 10.4

Системні функції та функції метаданих

Функція	Призначення
CASE	Виконує порівняння
CONVERT і CAST	Перетворює один тип даних на інший (наприклад, цілі числа на символи)
CURRENT_USER	Повертає ім'я поточного користувача, який запустив SQL Server

Закінчення табл. 10.4

Функція	Призначення
ISDATE	Указує, чи являють собою вхідні дані функції реальну дату
ISNULL	Замінює будь-яке порожнє значення, задане для заміни
ISNUMERIC	Указує, чи є вхідні дані функціями числа

Функції дати і часу

У табл. 10.5 наведено основні функції дати й часу MS SQL Server.

Таблиця 10.5

Функції дати і часу

Функція	Призначення
DATEADD	Додає до дати деяку величину
DATEIFF	Виводить кількість одиниць часу, заданих в аргументі datepart, між двома датами
DATENAME	Повертає текстові імена (наприклад, Tuesday), що відповідають заданій даті
DATEPART	Витягує певний фрагмент із заданої дати
DAY	Витягує день із дати
GETDATE	Повертає поточні час і дату
GETUTCDATE	Повертає поточну дату й час за Гринвічем (GMT – Greenwich Mean Time), перетворений на формат універсального синхронізованого часу (UTC – Universal Time Coordinate)
MONTH	Вилучає місяць із дати
YEAR	Вилучає рік із дати

Деякі з функцій дати й часу використовують аргумент *datepart*, що визначає фрагмент, до якого застосовується операція.

Приклад. Функція DATEADD бере як аргументи маркер *datepart*, величину приросту й вихідну дату та повертає результат додавання певної величини в одиницях, зазначених в аргументі *datepart*, до поточної дати. Наприклад:

– додати до поточної дати три дні

```
PRINT DATEADD (d, 3, GETDATE())
```

У табл. 10.6 наведено константи аргументу datepart.

Таблиця 10.6

Константи аргументу datepart

Константа	Призначення
yy або y	Рік
gg або g	Квартал
mm або m	Місяць
wk або ww	Тиждень
dw або w	День тижня
dy або y	День року (1 з 366)
dd або d	День
hh	Година
mi або n	Хвилина
ss або s	Секунда
ms	Мілісекунда

Функції рядків

У табл 10.7 наведено основні функції рядків.

Таблиця 10.7

Деякі функції рядків

Функція	Призначення
LEFT	Вибирає символи з лівого кінця рядка, функція LEFT ('abcdefg', 4) поверне рядок abcd
LEN	Повертає довжину символьного рядка
LOWER	Перетворює рядок на нижній регістр
LTRIM	Видаляє початкові проміжки з рядка
REPLACE	Замінює задані фрагменти рядка іншим рядком, функція REPLACE ('abc', 'b', 'e') поверне рядок aec
RIHGT	Вибирає символи з правого кінця рядка
RTRIM	Видаляє кінцеві проміжки з рядка
UPPER	Перетворює рядок на верхній регістр
+	Операція конкатенації рядків

Приклад. Відбір записів за значеннями символьного поля

```
SELECT * FROM Group WHERE Dol = 'студент'
```

У цьому прикладі складається перелік студентів групи.

В операції порівняння враховуються початкові й кінцеві проміжки, а також регістр символів, отже, слова 'Студент', 'СТУДЕНТ' і 'студент' не дорівнюють один одному. Відмінності в регістрі символів посади, а також

наявність проміжків на початку і в кінці рядка призводять до помилок під час відбору записів. У цій ситуації критерій відбору записують у такому вигляді:

```
WHERE UPPER (TRIM (Dol)) = 'СТУДЕНТ'
```

Якщо перетворення виконати неможливо, то користувач отримує повідомлення про помилку і SQL-оператор не виконується.

Приклад. Завдання обчислюваного поля:

```
SELECT "–" || Fam As Прізвище, Okl, Okl * 2 FROM Group;
```

Виводяться попередні значення окладів співробітників і нові, збільшені на 100 %. До кожного прізвища з допомогою операції конкатенації (||) додається символ «–». Заголовки таблиці: Прізвище, Okl, Okl * 2.

Контрольні запитання

1. Якими є основні способи об'єднання таблиць у запиті?
2. Якими є особливості пропозиції UNION?
3. Якими є особливості внутрішнього об'єднання таблиць у запиті?
4. Якими є особливості самооб’єднання таблиць у запиті?
5. Якими є види об'єднання для реалізації пропозиції JOIN?
6. Якими є види зовнішнього об'єднання для реалізації пропозиції JOIN?
7. Які існують типи SQL-функції?
8. Які існують категорії SQL-функції?
9. Які SQL-функції належать до агрегатних?
10. Якими є особливості запитів з агрегатними SQL-функціями?
11. Які SQL-функції належать до скалярних?

Лабораторна робота № 5

СТВОРЕННЯ ДОДАТКОВИХ ОБ'ЄКТІВ БАЗИ ДАНИХ

Постановка завдання

На основі таблиць фізичної моделі даних, бізнес-правил предметної області та скриптів будування таблиць БД створити:

1. 3–5 різних індексів.
2. 3–4 різних представлення, що модифікуються.
3. 3–4 різних представлення для читання.
4. Запити з використанням представлень.

5. Команди для додавання, змінення й видалення даних базових таблиць на основі представлень.

Для створення наведених скриптів і запитів використовувати команди мови маніпулювання даними DML.

Письмовий звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш з назвою лабораторної роботи, прізвищем, ім'ям, по батькові, номером групи виконавця, датою складання.
2. Діаграму БД.
3. Скрипти для створення індексів БД.
4. Скрипти для створення модифікованих представлень під час роботи з БД.
5. Скрипти для створення представлення для читання для роботи з БД.
6. Скрипти для створення запитів із використанням представлень БД і результати виконання цих запитів.
7. Скрипти для команд модифікації даних базових таблиць на основі представлень.
8. Приклади роботи основних операторів відповідно до постановки завдання (у вигляді оператор + результат його роботи).
9. Висновки (відобразити особливості роботи з індексами й представленнями БД).

Контрольні запитання

1. Означення й призначення представлень.
2. Яким є синтаксис створення (видалення) представлень?
3. Якими є особливості роботи з представленнями бази даних?
4. Які типи представлень існують і якими є їх особливості?
5. Якими є особливості зберігання даних на MS SQL-сервері?
6. Означення й призначення індексу.
7. Які види індексів існують у MS SQL-сервері?
8. Якими є особливості кластерного індексу?
9. Якими є особливості некластерного індексу?
10. На базі яких структур даних будуються індекси?
11. Яким є синтаксис створення (видалення) індексів?
12. Які параметри може мати команда створення індексу?
13. Якими є переваги й недоліки індексів?

Лекція 11. ПРЕДСТАВЛЕННЯ ТА ІНДЕКСИ

11.1. Представлення

Представлення (view) – це об'єкт СКБД, що є іменованим запитом на вибірку, тобто SQL-оператором SELECT з ім’ям. Представлення не зберігає дані, а лише їх представляє. Користувач сприймає представлення як деяке віртуальне відношення (віртуальну таблицю). Представлення – це табличний підзапит, що міститься в таблицях схеми й викликається кожного разу при його застосуванні.

Приклад. Припустимо, є відношення Student (Id, Name, Enter_Year, Age, Id_Group, Average_Mark). Створити представлення, яке приховує інформацію про рік вступу (атрибут Enter_Year) і вік (атрибут Age), а також змінює імена деяких атрибутів:

```
CREATE VIEW SPISOK (Id, Name, Group, Sr_Ball)
AS SELECT Id, Name, Id_Group, Average_Mark FROM STUDENT
```

Тепер у БД існує представлення з ім'ям SPISOK, яке можна використовувати в командах так само, як і будь-яку іншу таблицю БД. Ця таблиця може бути запитаною, модифікованою, з'єднаною з іншими таблицями й представленнями. Запит такого представлення має вигляд

```
SELECT * FROM SPISOK
```

Механізм представлень дає змогу:

1) приховати несуттєві або небажані деталі БД для різних користувачів;
2) забезпечити підвищену продуктивність БД при частому використанні одного й того самого оператора SELECT;
3) модіфікувати реальні структури даних у зручному для додатка вигляді, наприклад, при збільшенні або реструктуризації БД (фактично команди модифікації переспрямовуються до базової таблиці);
4) підтримувати попередню структуру БД (структура БД змінюється, а попереднє програмне забезпечення залишається незмінним): функціональність попередньої БД залишається такою ж самою, а структура БД розширюється новими модулями;
5) розмежовувати права доступу до даних;
6) переглядати одні й ті самі дані в різних варіантах одночасно;
7) обмежувати обсяги даних для зручності роботи з БД;
8) після створення представлення використовувати його в запитах нарівні з таблицями;
9) на основі представлень створювати нові представлення.

Синтаксис команди створення представлення має такий вигляд:

CREATE VIEW <ім'я_ представлення> [(<перелік_стовпців>)]
[WITH [ENCRYPTHION] [, SCHEMABINDING]]
AS <SQL- запит> [WITH CHECK OPTION]

Якщо перелік стовпців під час опису представлення не вказано, то до представлення увійдуть усі стовпці із запиту, на основі яких його створено, з відповідними іменами.

З допомогою опції ENCRYPTHION виконується шифрування представлення, при цьому стають недоступними можливості перегляду й редагування представлення, але його можна виконувати й видаляти.

З допомогою опції SCHEMABINDING виконується зв'язування представлення зі схемою БД, тобто цей зв'язок визначає, від яких об'єктів (таблиць або інших представлень) залежить представлення, що розглядається. Наявність опції SCHEMABINDING дає змогу перешкодити внесенню змін (з допомогою операторів CREATE, ALTER або DROP) в об'єкти, від яких залежить представлення. Створення індексованого представлення є можливим лише за наявності опції SCHEMABINDING.

Конструкція WITH CHECK OPTION визначає, як здійснюється модифікація або додання даних з допомогою такого представлення, якщо рядок, що додається, відповідає критеріям конструкції WHERE представлення оператора SELECT.

Оператор AS у стандарті SQL-92 дає змогу задавати імена результатами виконання виразів підзапитів і використовувати їх. Отже, представлення можна вважати постійним табличним підзапитом, що зберігається у схемі БД і викликається за іменем (табл. 11.1).

Таблиця 11.1

Список об'єктів БД і їх властивості у запиті

Об'єкт і БД	Фізичне існування	Доступ користувача
Таблиця	так	так
Представлення	ні	так
Індекс	так	ні
Домен	ні	ні

Особливості роботи з представленням:

1. Запит, на якому базується представлення, виконується кожного разу, коли представлення бере участь у будь-якій команді. Представлення буде модифіковано автоматично кожного разу, якщо таблиця, що є його основою, змінюється. Представлення «вбирає в себе» вибрану за умовою версію даних.

2. У представленні можна використовувати обчислювані поля, групування, підзапити (у тому числі й співвіднесені), однак при цьому слід ураховувати обмеження, що відображають природу представлення.

Приклад. Якщо змінювати імена атрибутів у представленні не потрібно, то команда зі створення такого представлення матиме вигляд

```
CREATE VIEW SPISOK_1
AS SELECT Id, Name, Id_Group, Average_Mark FROM STUDENT
```

Представлення, що містить інформацію про студентів із середнім балом більше, ніж «4»:

```
CREATE VIEW SPISOK_AVR_4
AS SELECT Id, Id_Group, Name, Average_Mark FROM STUDENT
WHERE Average_Mark > = 4
```

Бажано, щоб поля, які застосовуються у предикаті представлення, було включено й до складу виведених полів, навіть якщо вони містять однакові значення, тобто з допомогою представлення не можна буде організувати додавання значень у поле Average_Mark.

3. З допомогою представлення ніколи не вдасться добитися такої самої швидкодії, як і при безпосередньому виклику на виконання оператора SELECT, що є основою цього представлення.

4. Представлення (з допомогою його імені) можна використовувати в таких командах: запитувати (SELECT); змінювати (UPDATE); додавати до нього записи (INSERT); видаляти з БД (DELETE); з'єднувати з іншими таблицями й представленнями. Існують обмеження на операції модифікації (UPDATE, INSERT, DELETE), які залежать від виду представлення. Фактично команди модифікації представлення перенаправляються до базової таблиці. Унесена до модифікованого представлення зміна має однозначно вноситися й до базової таблиці.

Приклад. Модифікація інформації в БД з допомогою представлення

```
UPDATE SPISOK_1 SET NAME = 'СИДОРОВА' WHERE Id = 32
```

Модифіковане представлення містить первинний ключ таблиці.

Існує два типи представлень:

– таке, що модифікується (оновлюється), – команди модифікації можуть виконуватися в представленні;

– призначене лише для читання при запиті – у представленні може виконуватися лише команда SELECT.

Щоб забезпечити додавання даних з допомогою представлення, необхідно, щоб цим представленням було охоплено всі стовпці без обмежень.

Критерії, за якими визначають модифікованість представлення:

1) необхідно, щоб представлення ґрунтувалося лише на одній базовій таблиці;

2) представлення має містити первинний ключ базової таблиці;

3) представлення не повинно мати жодних полів, які були би агрегатними функціями;

4) представлення не повинно містити запитів із DISTINCT у своєму визначенні;

5) представлення не повинно використовувати GROUP BY або HAVING у своєму означенні;

6) бажано, щоб представлення не використовувало у своєму значенні підзапити;

7) представлення може застосовуватися в іншому представленні, яке має бути також модифікованим;

8) представлення не повинно використовувати константи, рядки або вирази значень серед вибраних полів виведення;

9) для команди INSERT представлення має містити будь-які поля основної таблиці, з обмеженнями NOT NULL, якщо інше обмеження за замовчуванням не визначено.

Отже, представлення, що модифікуються, фактично подібні до фрагментів базових таблиць і відображають певну частину їх вмісту.

Представлення типу «лише для читання», дають змогу отримувати й форматувати дані більш раціонально – уникати складних предикатів і зменшувати ймовірність помилок.

Дані у представленні впорядковуються з допомогою пропозиції ORDER BY запиту, на якому базується представлення.

Представлення може базуватися на кількох базових таблицях.

Представлення обчислюється кожного разу під час його використання, будь-які зміни в даних таблиць БД адекватно відобразяться у представленні – у цьому полягає його відмінність від запиту до БД.

Механізм представлень реалізується безпосередньо на базових відношенях і зберігається у вигляді функції.

Представлення матеріалізується під час реалізації відповідних дій користувача, до цього моменту матеріалізовані дані представлення у БД не зберігаються. Матеріалізація означає, що під час посилання на ім'я представлення СКБД знаходить його означення в таблицях схеми і створює робочу таблицю. Потім нова таблиця заповнюється результатами роботи оператора SELECT, що знаходиться в тілі означення представлення.

Для внесення змін до представлення застосовують два способи:

– з допомогою оператора ALTER VIEW, при цьому передбачається, що представлення, яке розглядається, вже існує (виконується повна заміна представлення, зберігаються всі права на його використання);

– видалення представлення (DROP VIEW <ім’я_представлення>) і його повторне створення з допомогою оператора CREATE VIEW.

При видаленні представлення немає необхідності видаляти всі дані з нього, тому що реальні дані в ньому не містяться. Для видалення представлення користувач повинен бути його власником.

Для отримання інформації про різні об'єкти БД використовують представлення інформаційної схеми й представлення каталогів.

11.2. Індекси

11.2.1. Особливості зберігання даних на MS SQL-сервері

Дані в реляційній БД зберігаються в таблицях, які складаються з рядків певної структури. Послідовність рядків логічно було би подати безперервною, хоча з огляду на реляційну модель самі по собі записи у відношенні не впорядковано.

Для фізичного зберігання даних для MS SQL-сервера рядки таблиць прив'язані до конкретних частин диска фіксованого розміру – сторінок.

Сторінка – це блок фіксованої довжини безперервних віртуальних адрес пам'яті, який бере участь в операціях читання й запису як єдине ціле, тобто потребує лише одного переміщення головки диска для виконання цих операцій. У MS SQL-сервері розмір сторінки становить 8192 байти. Кількість рядків у сторінці може варіюватися залежно від їх розміру. Сторінка може розглядатися як контейнер для зберігання рядків таблиць та індексів. Один рядок не може міститися на двох сторінках.

Сторінка складається з таких компонентів:

– заголовок (page header) (96 байтів);

– дані – покажчики зміщення рядків (row offset), необхідні для визначення на сторінці позиції, з якої починаються дані конкретного рядка;

– два покажчики на попередню й наступну сторінки (сторінки даних утворюють двозв'язний перелік).

Різновиди сторінок:

– сторінки даних (Data Pages) – для зберігання реальних даних таблиць (крім BLOB-даних);

– сторінки індексу (Index Pages) – для зберігання сторінок різних рівнів B-дерева, яке використовується для зберігання даних індексу (індексна сторінка (вузол B-дерева) містить упорядкований перелік усіх значень індексованого стовпця (їх називають ключовими значеннями) разом з покажчиком розташування кожного запису, що містить це значення в таблиці);

– BLOB-сторінки (Binari Large Object) – для зберігання великих бінарних об'єктів;

– карти розміщення сторінок (таблиці розподілу) (Global і Shared Global Allocation Map – GAM і SGAM) – для визначення вільних і використовуваних сторінок та екстентів БД;

– сторінки вільного простору (Page Free Space – PFS);

– карти (таблиці) розміщення індексів (Index Allocation Map – IAM) – містять відомості про екстенти файла БД, що використовуються в таблицях купи або в індексах;

– таблиця масових змін;

– таблиця диференційних змін.

Сторінки об'єднуються в екстенти.

Екстент – це одиниця пам'яті, яку сервер виділяє в міру необхідності як єдине ціле при розміщенні на диску даних таблиці або індексу (рис. 11.1). Екстент означає простір, що використовується для зберігання реальної інформації всередині фізичного простору, виділеного для БД. У MS SQL-сервері 2005 екстент складається з восьми суміжних сторінок даних (64 кбайти). У MS SQL-сервері під час створення таблиці з самого початку не виділяється жодної сторінки, а лише під час додавання до таблиці нових рядків.

Особливості роботи з екстентами:

– після заповнення екстента під час додавання нового запису виділяється новий екстент;

– завдяки попередньому виділенню пам'яті сервер заощаджує час;

– екстенти таблиці не розміщуються фізично один за іншим у файлі БД, їх розкидано, тому доступ до даних купи буде виконуватися повільно;

– загальний обсяг невикористаної частини екстентів зазвичай є невеликим, але він може збільшуватися, особливо при високому ступені дефрагментації даних;

– екстенти блокуються лише під час виділення нового або звільнення попереднього екстента.

Рис. 11.1. Збільшена структура розташування даних у БД (за замовчуванням БД має фізичний файл БД (* .mdf) і файл журналу (* .ldf))

Залежно від способу зберігання розрізняють два типи таблиць – таблиці купи і кластерні таблиці. Тип таблиці залежить від наявності кластерного індексу.

Таблиці купи – це таблиці, розташовані в купі, набір сторінок даних, що виділяються за необхідності для таблиць, які не мають кластерного індексу, а також для самих індексів. Рядки таблиці купи не мають жодного конкретного порядку, сторінки даних, що належать до однієї таблиці, також ніяк не впорядковано й не зв'язано у список. Щоб створити новий рядок у таблиці, в адресному просторі виділяється наступний екстент даних, якщо простір на виділених сторінках вже заповнено. Доступ до конкретного рядка здійснюється шляхом повного сканування таблиці.

Кластерні таблиці – це таблиці, що мають кластерний індекс. Кластерний індекс – це індекс, де логічний порядок ключових значень установлює фізичний порядок відповідних рядків у таблиці. Таблиця може мати лише один кластерний індекс і при його наявності рядки таблиці зберігаються на диску в строго визначеному порядку – їх упорядковано за зростанням (за зменшенням).

Заголовок сторінки даних кластерної таблиці (рис. 11.2) містить ідентифікатор таблиці, до якої належить сторінка, а також покажчики на наступну й попередню сторінки. Іншу частину сторінки заповнюють рядки даних.

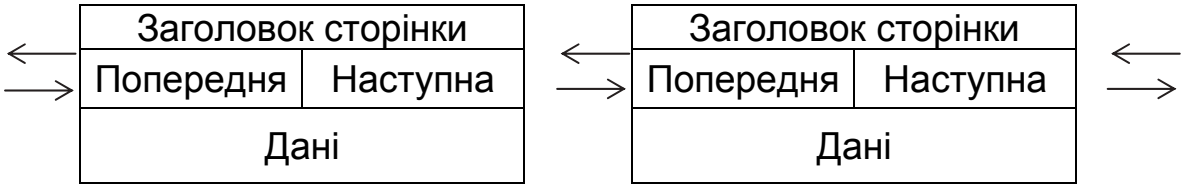


Рис. 11.2. Двозв'язний список для організації сторінок даних кластерної таблиці на SQL-сервері

Під час додавання нового рядка до кластерної таблиці, якщо в потрібній сторінці немає вільного місця, відбувається розщеплення (розбиття) сторінки. Суть процесу розщеплення полягає в переміщенні половини рядків з повністю заповненої сторінки до знову виділеної так, що замість однієї повністю заповненою сторінки виникають дві, заповнені майже наполовину. Так виникає місце для нового рядка зі збереженням фізичного порядку сортування. Доступ до конкретного рядка кластерної таблиці здійснюється за алгоритмом пошуку в збалансованому дереві за значенням ключа, унікального для кожного рядка таблиці.

Існують такі основні методи доступу до даних БД:

1) сканування таблиці (послідовний доступ) – послідовне читання всіх рядків таблиці в тому порядку, у якому їх записано в місці їх фізичного зберігання, відразу зчитується одна сторінка даних;

2) доступ з допомогою індексів (індекс зазвичай має структуру B-дерева або інвертовану структуру файлу) – дає змогу повертати по одному рядку таблиці;

3) доступ з допомогою хешованих індексів – поділяє дані на області з однаковим значенням функції хешування;

4) доступ з використанням бітових векторних індексів – конкретне значення конкретного атрибута (ключ) подається у вигляді одного біта у векторі (масці) або в масиві.

Методи доступу 3 і 4 застосовують лише в Oracle.

11.2.2. Визначення індексу і його структура

Існує два різних способи отримання даних з допомогою запиту з БД у MS SQL-сервері:

– шляхом табличного сканування (послідовного сканування рядків, сторінок та екстентів таблиці (або таблиць) БД);

– з допомогою індексу.

Який із способів автоматично використовується сервером для конкретного запиту? Це буде залежати:

– від доступних на поточний момент індексів;

– запитуваних у запиті стовпців;

– типу об'єднань таблиць, застосованих у запиті;

– фізичного розміру застосованих у запиті таблиць.

Припустимо, що потрібно вивести список студентів, у яких середній бал є вищим від 4,5:

```
SELECT name FROM Student WHERE avg_mark > 4.5
```

Припустимо, що в таблиці Student немає індексів і, отже, таблиця розміщена в купі. Як буде реалізоватися ця вибірка? Єдиний спосіб реалізації цього запиту – табличне сканування, тобто при пошуку даних із діапазону (5.0 > avg_mark > 4.5) сервер повинен звернутися до кожного рядка кожної сторінки кожного екстента таблиці даних Student для пошуку кожного конкретного значення з діапазону пошуку.

Табличне сканування може істотно уповільнити роботу системи, однак не завжди. Якщо таблиця дуже маленька (займає близько одного екстента), то табличне сканування може працювати швидше, ніж індексований доступ. Якщо створюється індекс у маленькій таблиці, то сервер буде вимушений зчитувати індексні сторінки і лише після цього вибирати сторінки даних. У цій ситуації швидше просто сканувати таблицю.

Отже, маленькі таблиці краще робити купами. Однак у великих таблицях слід уникати сканування й використовувати індекси.

Звичайно, простіше виконати цю вибірку з діапазону, якби дані в таблиці Student були відсортованими або була інформація про номери рядків (про первинні ключі рядків) у порядку зростання (зменщення) значень у полі avg_mark. Ця інформація і міститься в об'єктах БД – індексах.

Індекси – це внутрішні об'єкти СКБД, які прискорюють процес оброблення даних (вибірки, додавання, змінення й видалення). Індекс – упорядкований список значень полів або значень груп полів у таблиці. Коли створюється індекс у полі, БД запам'ятовує відповідний порядок усіх значень цього поля в області пам'яті. Оптимізатор сервера для прискорення доступу до даних автоматично застосовує індекси.

Індекс зберігає інформацію про порядок проходження записів відносно поля avg_mark, що може прискорити виконання оператора SELECT. Там, де необхідно здійснювати вибірки з БД із використанням сортування, ця інформація з індексу автоматично додається сервером до процесу вилучення даних.

Приклад. Наведено фрагмент таблиці Student з полями id – ключове поле (створений сервером кластерний індекс), avg_mark – середній бал і таблиця з індексами поля avg_mark за зростанням (некластерний індекс):

id	avg_mark	…
1	4,3	
2	5,0	
3	3,7	
4	4,1	
5	3,7	

Індекси за зростанням поля avg_mark	
3,7	Покажчик на запис з Id=3
3,7	Покажчик на запис з Id=5
4,1	Покажчик на запис з Id=4
4,3	Покажчик на запис з Id=1
5,0	Покажчик на запис з Id=2

У SQL-сервері індекси можуть бути двох типів:

– кластерні (кластеризовані) – аналогом є словник. Якщо кластерний індекс буде створено для стовпця «Прізвище», то фізично запис зі значенням «Іванов» у полі «Прізвище» завжди буде стояти перед записом зі значенням «Петров»;

– некластерні (некластеризовані) – аналогом є предметний покажчик у книзі, у якому, наприклад, є розділ «І» та посилання на сторінки, де згадується прізвище «Іванов».

Кластерний і некластерний індекси реалізуються з допомогою структури даних B-дерева (balanced tree – збалансоване дерево).

Між типами індексів є істотні відмінності:

– з огляду на фізичну реалізацію індексів;

– з огляду на кількість індексів у таблиці БД – кластерний індекс завжди один на базову таблицю, а некластерних індексів може бути декілька;

– з огляду на виконання різних операцій у БД з використанням індексів.

Кластерні індекси

При визначенні для таблиці з даними кластерного індексу сервера фактично віддається команда фізично впорядкувати дані за зростанням (або зменшенням) у порядку індексу (за полем, для якого створено кластерний індекс). Під час додавання (модифікації) даних до таблиц з кластерним індексом запис додається з урахуванням підтримки впорядкованості за полем, для якого побудовано кластерний індекс. Тому в будь-якій таблиці може бути лише один кластерний індекс.

На рівні листа B-дерева кластерного індексу сторінки даних, призначені для зберігання індексної інформації, фізично збігаються зі сторінками даних базової таблиці, для якої цей індекс створено.

Середній розмір кластерного індексу становить близько 5 % від розміру таблиці, але може варіюватися залежно від розміру індексованого стовпця.

У кластерному індексі дані на рівні листа є фізично відсортованими.

Нехай потрібно виконати запит – вибірка діапазону:

```
SELECT name FROM Student WHERE avg_mark > = 4.5;
```

Нехай у таблиці Student за полем avg_mark створено кластерний індекс. Запит буде реалізовано в такий спосіб.

Виконується проходження по B-дереву кластерного індексу (від кореневої сторінки через проміжні до листової сторінки) для пошуку значення ключа кластеризації 4,5. Оскількі сторінками даних листового рівня індексу є двозв'язний список, то далі виконується проходження за списком у порядку зростання значення поля avg_mark до рівності значенням кінця діапазону 5,0 (точніше, поки покажчик не стане дорівнювати Null у цьому запиті).

Кластерний індекс – ідеальний вибір:

– для стовпців, у яких постійно виконується пошук діапазонів даних,

– стовпців з низькою селективністю (низька селективність значень стовпця означає, що в стовпці багато однакових записів, наприклад, значень 4,6 у полі «Середній бал»).

Проблеми виникають під час модифікації даних у полі з кластерним індексом (команди INSERT, UPDATE, DELETE), оскільки сервер має фізично перекомпоновувати дані для їх відповідності параметрам кластерного індексу. Якщо перекомпонування є необхідним, то сервер залишає трохи вільного місця в кінці кожної сторінки даних з кластерним індексом. Це – порожній простір, який регулюється параметром FILLFACTOR – фактором заповнення.

Фактор заповнення задається під час створення кластерного індексу й пізніше його можна змінити. Чим вище фактор заповнення, тим менше вільного місця він виділяє, і навпаки. Якщо фактор заповнення становить 100 %, то це означає, що сторінку даних заповнено на 100 %, але ще є місце для запису.

Якщо потрібно додати дані до повністю заповненої сторінки, то сервер її розбиває (рис. 11.3). Це означає, що сервер переміщує приблизно половину даних із заповненої сторінки на порожню, створюючи дві наполовину заповнені сторінки. Оскільки фізично нова сторінка може знаходитися в будь-якому місці файла бази даних, а сторінки даних таблиці з кластерним індексом мають утворювати двозв'язной список, то далі для трьох сторінок даних має бути оновлено покажчики на попередню й наступну сторінки. Крім того, необхідно додати до B-дерева кластерний індекс, що приведе до перебудови B-дерева.

У БД, де дані використовуються винятково для читання, наприклад у середовищі підтримки прийняття рішень, можна застосовувати високий фактор заповнення (менше вільного місця), завдяки чому дані будуть зчитуватися з меншої кількості сторінок файла БД, що підвищить продуктивність команди SELEC.

Низький фактор заповнення слід використовувати в БД з великими потоками команд INSERT, UPDATE, DELETE, оскільки це забезпечить меншу кількість розбиття сторінок і підвищить продуктивність команд INSERT, UPDATE, DELETE.

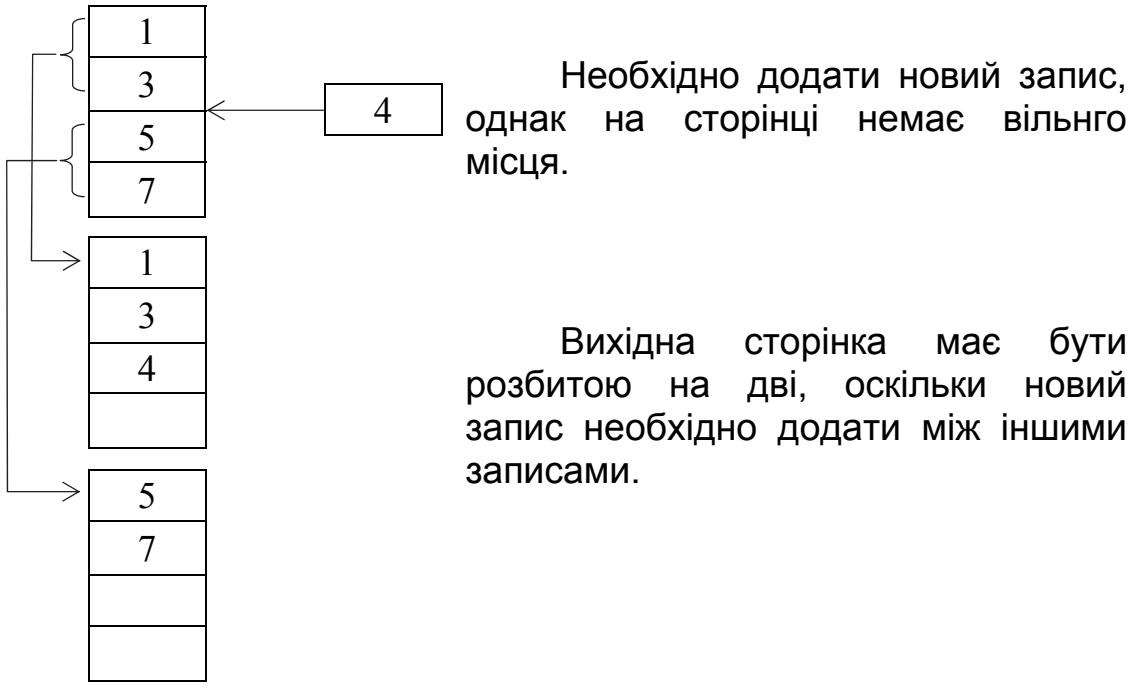


Рис. 11.3. Процес розбиття сторінки кластерного індексу

Некластерні індекси

Некластерний індекс також реалізується з допомогою B-дерева.

Між двома типами індексів існують такі відмінності:

1) листовий рівень некластерного індексу містить не реальні дані, а лише покажчики на сторінки даних;

2) некластерний індекс не переміщує фізично дані таблиці БД (усе це схоже на різницю між словником (кластерний індекс) і покажчиком наприкінці книги (некластерний індекс)).

Існує два види реалізації некластерного індексу:

– такий, що базується на купі (якщо в таблиці спочатку немає кластерного індексу за іншим полем таблиці) – дані додаються до таблиці з некластерним індексом у черговий вільний рядок останньої сторінки, а потім оновлюються покажчики індексу;

– такий, що базується кластерному індексі (якщо в таблиці є кластерний індекс) – під час додавання даних до таблиці сервер фізично поміщає дані відповідно до кластерного індексу, а потім оновлює ключове значення некластерного індексу для указання на ключове значення кластерного індексу.

Якщо в таблиці виконується пошук одного значення із застосуванням некластерного індексу, то сервер звернеться до індексу один раз, оскільки листовий рівень індексу приведе прямо до даних. Якщо виконується пошук діапазону значень, то сервер буде постійно посилатися на індекс у пошуках ключового значення для кожного запису діапазону. Це означає, що слід використовувати некластерний індекс у стовпцях, у яких пошук діапазонів даних виконується не часто, або в стовпцях з високою селективністю (з невеликою кількістю записів, що дублюються).

Існують таки відмінності між кластерним і некластерним індексами.

Кластерний індекс:

– лише один на таблицю;

– фізично перебудовує дані в таблиці згідно з обмеженням індексу;

– застосовується в стовпцях, де часто виконується пошук діапазонів даних;

– використовується в стовпцях з низькою селективністю (висока кількість дубльованих значень).

Некластерний індекс:

– до 249 індексів на таблицю;

– створює окремий список ключових значень з покажчиками на дані сторінки;

– застосовується в стовпцях, де виконується пошук окремих значень;

– використовується в стовпцях з високою селективністю (низька кількість дубльованих значень).

Перевага індексів:

– прискорення операції пошуку;

– прискорення операції сортування;

– прискорення фази пошуку під час видалення, додаванння й модифікації.

Індексація дає змогу знаходити блок даних, що містить індексований рядок, виконуючи невелику кількість звернень до зовнішнього пристрою.

Недоліки індексів:

– для їх зберігання потрібна додаткова пам'ять;

– під час операції додавання, модифікації й видалення потрібна перебудова даних (для кластерних індексів) та індексів.

Керування індексом істотно сповільнює час виконання операцій, пов'язаних з оновленням даних (таких, як INSERT, UPDATE і DELETE), тому що ці операції потребують перебудови індексів.

Індекси можна створювати як за одним, так і за багатьма полями.

SQL Server самостійно вирішує, коли індекс є необхідним для роботи, і використовує його автоматично.

11.2.3. Створення індексу в MS SQL SERVER

Методи створення індексів в MS SQL SERVER:

1) з допомогою утиліти SQL Sever Management Studio – візуальний засіб;

2) з допомогою функції Database Tuning Advisor (ця функція належить до утиліти SQL Profiler, яка виконує моніторинг SQL Sever) – візуальний засіб;

3) шляхом застосування команди Create мови SQL;

4) неявне створення індексу як обов'язкового об'єкта внаслідок уведення в дію деякого обмеження (Primary Key, Unique).

При цьому таблиця має бути вже створеною і містити стовпці, імена яких указано в команді створення індексу. Ім'я індексу, задане в команді, має бути унікальним у базі даних.

Створений одий раз, індекс є невидимим для користувача, усі операції з ним СКБД здійснює автоматично.

Оператор CREATE INDEX створює на зазначеній таблиці або представленні індекс, що базується на заданих стовпцях. Синтаксис команди створення індексу має такий вигляд:

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED]
INDEX <index_name> ON <table or view name>
(<column name> [ASC | DESC] [, … n])
INCLUDE (<column name> [, -n])
[WITH
[PAD_INDEX = {ON | OFF}]
[[,] FILLFACTOR = <коефіцієнт заповнення>]
[[,] IGNORE_DUP_KEY = {ON | OFF}]
```

[[,] DROP_EXISTING = {ON | OFF}]
[[,] STATISTICS_NORECOMPUTE = {ON | OFF}]
[[,] SORT_IN_TEMPDB = {ON | OFF}]
[[,] ONLINE = {ON | OFF}]
[[,] ALLOW_ROW_LOCKS = {ON | OFF}]
[[,] ALLOW_PAGE_LOCKS = {ON | OFF}]
[[,] MAXDOP = <maximum degree of parallelism>
]
[ON {<filegroup>|<partition scheme name>|DEFAULT}]

Ця команда створює індекс з ім'ям index_name для зазначеної таблиці *table* або представлення *view* за переліченими в параметрі *column* стовпцях. Індекс не є автономним об'єктом, він належить до деякого стовпця (стовпціві) таблиці або представлення.

Обов'язкова лише конструкція CREATE INDEX <index_name> ON<table or view name>(<column name>), а всі інші конструкції, які йдуть за нею, не є обов'язковими.

Параметри команди створення індексу мають такий зміст:

UNIQUE – задає індекс, який реалізує обмеження унікальності для своїх стовпців. Це означає, що дані в стовпцях не можуть повторюватися (за замовчуванням індекси не є чимось унікальним).

CLUSTERED – створює кластерний індекс.

NONCLUSTERED – формується некластерний індекс (значення за замовчуванням).

ASC (скорочення від *ascending*) – створює індекс у порядку зростання, і це є стандартним значенням, а DESC – у порядку зменшення.

INCLUDE – для забезпечення кращої підтримки «охоплених» запитів (запит розглядається як «охоплений», якщо всі дані, які має бути отримано після виконання запиту, повністю наведено в індексі). Під час використання INCLUDE сервер переносить вміст указаних стовпців на листовий рівень індексу, що тягне за собою значне зменшення операцій введення/виведення. Цю опцію можна використовувати лише під час створення некластерних індексів, оскільки кластерні індекси вже містять дані на листовому рівні. Негативні наслідки застосування опції INCLUDE – збільшення розмірів рядків індексу листового рівня. Унаслідок цього спроба прискорити виконання одного запиту може призвести до зниження швидкодії під час виконання інших запитів.

WITH – необхідна для передання СКБД вказівки, що за нею йтиме одна або декілька додаткових опцій.

PAD_INDEX – задає ступінь заповнення сторінок індексу на нелистовому рівні (у відсотках) під час первинного створення індексу (за замовчуванням залишається місце для додавання на рівні сторінок індексу двох індексних записів максимальної довжини). Якщо значення PAD_INDEX=ON, то значення FILLFACTOR поширюється і на ступінь

заповнення сторінок на всіх рівнях індексу (якщо значення PAD_INDEX не визначено, то FILLFACTOR застосовується лише в листових сторінках індексу). Використання параметра PAD_INDEX=ON не має сенсу, якщо не задано опції FILLFACTOR.

FILLFACTOR – задає ступінь заповнення даними сторінок на рівні листа (значення за замовчуванням FILLFACTOR=0 означає те ж саме, що й FILLFACTOR=100), це означає повне заповнення сторінок на рівні листа індексу, але є порожнє місце на індексних сторінках верхніх рівнів. Зазвичай значення опції FILLFACTOR – будь-яке число від 1 до 100, яке є відображенням того, наскільки повним має стати наповнення сторінок у відсотках після завершення створення індексу. Під час розбиття сторінок дані перерозподіляються між двома сторінками рівними частинами. Це означає, що під час експлуатації БД неможливо постійно зберігати контроль над тим, у якому відсотковому відношенні заповнюються сторінки індексу, тому під час технічного супроводження індексу необхідно його перебудовувати. Якщо значення FILLFACTOR не задано, то СКБД повністю заповнює сторінки за винятком двох рядків.

Запис FILLFACTOR=100 означає, що кожна індексна сторінка має бути повністю заповненою під час створення індексу. Якщо кластерний індекс FILLFACTOR=100, це означає, що під час додавання запису завжди буде відбуватися процес розбиття сторінки (численні розбиття сторінок можуть зменшити ефективність роботи сервера). Якщо значення FILLFACTOR є занадто малим, то індекс займе невиправдано багато місця в пам'яті.

Ідеальне значення FILLFACTOR=R/W, де R – середня кількість операцій читання, W – середня кількість операцій запису, які виконуються для компонентів індексу. FILLFACTOR добирається експериментально, R і W можна визначити статистично під час роботи з БД з допомогою утиліти Performance Monitor.

Рекомендації щодо вибору фактора заповнення FILLFACTOR:

– нечасто змінювані таблиці (виконується від ста до однієї операції читання на одну операцію запису) FILLFACTOR=100;

– часто змінювані таблиці (кількість операцій читання перевищує кількість операцій запису) FILLFACTOR=50–70;

– проміжна ситуація FILLFACTOR=80–90.

IGNORE_DUP_KEY – значення, що збігаються, у стовпцях індексу ігноруються. У цьому випадку під час спроби додати вже наявне значення видається попередження (а не повідомлення про критичну помилку), додавання рядка не виконується, але відкату транзакції немає.

DROP_EXISTING – вимагає перед створенням нового індексу видалити наявний індекс з ім'ям, що збігається з ім'ям створюваного індексу. Якщо внаслідок перебудови необхідно отримати індекс, що повністю збігається з наявним, то СКБД визначає, що такий некластерний індекс не слід піддавати операції видалення й повторного створення (на

відміну від виконання явних операцій видалення індексу, а потім його створення). Ця опція забезпечує набагато більш ефективне формування індексу порівнянно з тим, коли відбувається просто видалення й повторне створення наявного індексу, якщо він використовується разом з кластерним індексом.

STATISTICS_NORECOMPUTE – наказує не перебудовувати статистику таблиць після створення індексу. За замовчуванням у СКБД робиться спроба автоматизувати процес оновлення статистичних даних, що належать до використовуваних таблиць та індексів. Щоб скасувати цю опцію, необхідно викликати команду UPDATE STATISTICS. Рекомендується не використовувати цю опцію, оскільки статистичні дані, що належать до індексу, використовуються оптимізатором запитів для визначення того, наскільки цей індекс може підвищити продуктивність виконання конкретного запиту. Відмова від автоматичного оновлення статистичних даних призведе до того, що оптимізатор запитів буде вибирати способи їх виконання на основі застарілої інформації.

SORT_IN_TEMPDB – вимагає виконувати сортування під час створення індексу в базі даних TEMPDB (використовується під час створення індексів на великих таблицях), застосування опції дає ефект, якщо поточна БД і БД TEMPDB знаходяться на різних дискових пристроях.

ONLINE – якщо ONLINE=ON, то примусово встановлюється такий режим доступу до індексованої таблиці, що ця таблиця залишається прийнятною й для загального доступу та не створюються будь-які блокування, що не дають змогу звертатися до таблиці і (або) до індексу. Однак виконання операції створення індексу (при ONLINE=ON) істотно знизить продуктивність роботи користувачів. За замовчуванням (при ONLINE=OFF) під час створення індексу таблиця повністю блокується, при цьому індекс формується набагато швидше, ніж при ONLINE=ON.

ALLOW_ROW_LOCKS і ALLOW_PAGE_LOCKS (дозвіл (або недозвіл) блокування рядка або сторінки) – указують на можливість застосовування для створюваного індексу блокування на рівні сторінки або рядка. Блокування – це спосіб резервування ресурсів, що дає змогу запобігти конфліктам між операціями доступу до даних, які могли би призвести до порушення цілісності даних.

MAXDOP – дає змогу переобчислити значення параметра налаштування конфігурації системи, що визначає максимальний ступінь розпаралелювання (Degree Of Parallelism – DOP), який застосовується під час формування індексу. Ступінь розпаралелювання встановлює максимальну кількість процесів, уведених у дію з метою здійснення однієї операції в БД – формування індексу.

ON filegroup – указує, у якій групі файлів потрібно фізично створити й зберігати індекс (за замовчуванням – PRIMARY). Зручніше використовувати для зберігання індексів і сторінок даних різні жорсткі диски.

Приклад. Якщо таблиця EXAM часто застосовується для пошуку оцінки конкретного студента за значенням поля ID_STUDENT, то слід створити індекс за цим полем:

```
CREATE UNIQUE CLUSTERED INDEX IND_STUD
ON EXAM (ID_STUDENT ASC)
WITH PAD_INDEX =ON, FILLFACTOR=90, DROP_EXISTING = ON;
```

Для поля ID_STUDENT таблиці EXAM створено унікальний кластерний індекс за зростанням значень з ім'ям IND_STUD, причому фактор заповнення сторінок індексу всіх рівнів дорівнює 90. Якщо перед створенням цього індексу вже існував індекс з ім'ям IND_STUD, то його буде попередньо видалено.

Індекс видаляється в таких випадках:

– неправільно створені індекси;

– через змінення структури таблиці;

– під час створення нових індексів, що дають змогу краще виконувати запити.

Для видалення індексу використовується команда DROP INDEX, що має такий синтаксис:

```
DROP INDEX <ім'я індексу> [... n ]
```

Приклад. DROP INDEX IND_STUD

З допомогою команди DROP INDEX можна видалити один або кілька індексів лише поточної БД. Під час видалення індексу відбувається звільнення всіх сторінок, які використовувалися для зберігання даних індексу.

З допомогою команди DROP INDEX можна видалити індекси із системних таблиць, а також індекси, створені сервером під час визначення обмежень цілісності Primary Key і Unique.

Команда ALTER INDEX використовується для супроводження індексів і неяк не змінює їх структуру.

11.2.4. Особливості роботи з індексами

1. Після проведення аналізу інтенсивності використання даних і виведення інформації про індексацію стовпців, зазвичай починають створювати індекси. Індекс можна створити безпосередньо або як частину іншої, більш складної операції.

2. Некластерні індекси є найбільш ефективними у швидкодії, коли для них характерним є високий рівень селективності (selectivity). З практики випливає, що якщо рівень унікальності в індексованих стовпцях є меншим

за 90...95 %, то застосовувати некластерні індекси недоцільно через велику кількість різних фізичних операцій читання під час пошуку.

3. Кластерні індекси значно менше залежать від селективності. Після знаходження сервером початку необхідного інтервалу незалежно від відсотка унікальних значень можна безпосередньо отримувати дані, не вдаючись до читання додаткових сторінок індексу.

4. Для стовпця зовнішнього ключа при частому використанні операції об'єднання зі злиттям (merge joins) дуже ефективним є індексування. Зовнішні ключі часто є цільовими об'єктами під час виконання об'єднання з таблицею, на яку вони посилаються. Під час об'єднання зі злиттям з кожної таблиці витягують рядки, а потім їх порівнюють між собою стосовно відповідності критерію об'єднання. Оскільки для обох зв'язаних стовпців існують індекси в обох таблицях, процес видалення обох рядків виконується надзвичайно швидко.

5. За замовчуванням для первинного ключа (обмеження цілісності Primary Key) сервер автоматично створює унікальний кластерний (Clustered) індекс (якщо в таблиці не існує кластерного індексу). Однак поле первинного ключа може бути не найкращим кластерним індексом. Користувач може явно вказати, що для первинного ключа необхідно створити некластерний індекс (Nonclustered):

```
CREATE TABLE MyTable
(   Column1 int IDENTITY PRIMARY KEY NONCLUSTERED,
    Column2 int)
```

6. Унікальний індекс також автоматично створюється під час визначення в таблиці обмежень цілісності *Unique* (визначення альтернативного ключа). Індекс, що використовується для забезпечення унікальності стовпця або групи стовпців, має назву первинного. Індекси, що застосовуються для прискорення роботи запитів у неунікальних стовпцях, називають вторинними.

7. Випадкові індекси (індекси, що створюються з нагоди) – це індекси, які створюються після введення в дію обмежень (Primary Key і Unique). Для цього типу індексів не допускається використання будь-яких опцій, крім [CLUSTERED |NONCLUSTERED] і FILLFACTOR.

8. Індекс будь-якого типу можна створити під час формування таблиці. У команді CREATE TABLE передбачено засоби, що дають змогу виконати організацію індексу як частину процесу створення таблиці. При цьому дозволяється створення індексу як для одного, так і для декількох стовпців. Індекс мона створити після формування таблиці з допомогою команди CREATE INDEX.

9. Після створення кластерного індексу єдиний спосіб його змінити – це видалити його, а потім знову побудувати. При цьому буде потрібна операція пересортування таблиці, яка є дуже тривалою й потребує багато

дискової пам'яті. Для того щоб виконати пересортування таблиці при зміненні кластерного індексу, сумарно знадобиться в 2,2 раза більше дискового простору, ніж зазвичай займає таблиця (для одночасного зберігання попереднього й нового індексів, попередньої таблиці й робочого простору для зберігання тимчасової інформації).

10. Кластерні індекси є найбільш придатними для стовпців, задіяних у запитах:

– з виразами Between або <, > – ранжовані запити;

– з параметрами Group by або агрегатними функціями Min, Max і Count (у цих випадках кластерні індекси добре працюють, оскільки при пошуку даних виконується перехід безпосередньо до фізичних даних, послідовне читання яких продовжується до досягнення кінця інтервалу);

– з параметром Order by на основі кластерного ключа (дані вже фізично відсортовано за полем кластерного індексу).

11. Головне протипоказання для створення кластерного індексу – виконання великої кількості додавань непослідовних даних.

12. Додавання даних у некластерний індекс на основі купи зазвичай виконується швидше, ніж на основі кластерного ключа.

13. Вплив порядку стовпців в індексі. Індекс буде використано, якщо запит починається з першого стовпця, який належить до індексу. Точний збіг кількості й порядку стовпців у запиті й індексі є необов'язковим, але чим більше стовпців збігається (за порядком), тим краще.

14. Індекс буде проігноровано, якщо перший стовпець індексу не наведено в параметрах Join, Order by і Where запиту.

15. Під час супроводження індексу необхідно враховувати:

– розбиття сторінок,

– фрагментацію пам'яті файлу даних БД (висока фрагментація даних означає повільне читання інформації, але, з іншого боку, дає змогу швидко додавати нові дані).

16. Кластерні індекси – більш швидкі, ніж некластерні.

17. Операція SELECT зазвичай виграє від застосування індексів.

18. Якщо стовпець належить до складу виразу, то індекс не використовується. Щоб цього не допустити, до стовпця додають незначущий вираз

```
SELECT * FROM STUDENT WHERE Id = 34 + 0
```

19. Операції INSERT, DELETE і UPDATE (в операції UPDATE використовується такий підхід: спочатку видалення, а потім додавання) сповільнюються за наявності індексів; у цих операціях виконується додаткова робота – вносяться зміни не лише в дані, але й в індекси.

20. Краще створювати індекси на стовпцях, які мають цілочислові, а не рядкові значення. Зовсім невдало використовувати для будування індексів стовпці з типом даних *Float* або *Real* (ці типи абсолютно не

годяться для первинних ключів). Рядкові типи даних є найбільш придатними для індексів, якщо відповідні стовпці є невеликими за розміром, оскільки зі збільшенням ширини острівця збільшуються витрати на підтримку індексу. Не можна будувати індекси за типами даних *Text* і *Image*.

21. Не рекомендується застосовувати для будування індексів стовпці з типом даних *bit*, оскільки SQL-сервер не використовуватиме такий індекс для прискорення запиту.

Контрольні запитання

1. Означення й призначення представлення.
2. Яким є синтаксис створення (видалення) представлення?
3. Якими є особливості роботи з представленнями бази даних?
4. Які типи представлень існують і якими є їх особливості?
5. Якими є особливості зберігання даних на MS SQL-сервері?
6. Означення й призначення індексу.
7. Які види індексів існують у MS SQL-сервері?
8. Якими є особливості кластерного індексу?
9. Якими є особливості некластерного індексу?
10. На основі яких структур даних будуються індекси?
11. Яким є синтаксис створення (видалення) індексів?
12. Які параметри може мати команда створення індексу?
13. Якими є переваги та недоліки індексів?

Лекція 12. ТРАНЗАКЦІЇ

Застосування СКБД для роботи з інтегрованими БД виявило особливу важливість проблеми цілісності БД. Під цілісністю БД розуміють правильність і несуперечність її змісту. Порушення цілісності може бути спричинено, наприклад, помилками й збоями, оскільки в цьому випадку система не може забезпечити нормальне оброблення або видачу правильних даних.

Виокремлюють два аспекти цілісності – на рівні окремих об'єктів та операцій і на рівні бази даних у цілому.

У першому випадку цілісність забезпечується на рівні структур даних та окремих операторів мовних засобів СКБД (згадаємо обмеження цілісності для стовпців і таблиць у мові SQL). При порушеннях такої цілісності (наприклад, уведення значення більше 11 у стовпець *Семестр* таблиці «Навчальний_план» БД «Сесія») відповідний оператор відкидається.

Деякі обмеження цілісності не потрібно наводити в явному вигляді, оскільки їх убудовано в структури даних. Наприклад, у СКБД, що підтримує структури, складені із записів, кожен екземпляр запису в БД має

відображати специфікацію типу запису. Це означає, що всі поля, специфіковані в описі типу, має бути наведено в кожному примірнику запису, а значення, внесене в окреме поле, повинне мати відповідний опис типу даних.

Частіше база даних може мати такі обмеження цілісності, які потребують обов'язкового виконання не однієї, а кількох операцій. Для ілюстрації прикладів цієї глави розширимо функціональні можливості навчальної БД «Сесія», додавши до таблиці «Кадровий_склад» стовпець «Навантаження» для вирішення додаткового завдання – розрахунку загального річного навантаження викладачів (у годинах навчальної роботи). Тоді будь-яка операція зі внесення змін або додавання даних у стовпець ID_Викладач таблиці «Навчальний_план» має супроводжуватися відповідними змінами даних у стовпці «Навантаження». Якщо після внесення змін до стовпця ID_Викладач відбудеться збій, то БД виявиться у стані нецілісності.

Для забезпечення цілісності в разі обмежень на базу даних, а не на будь-які окремі операції, використовується апарат транзакцій.

Транзакція – неподільна з огляду на вплив на БД послідовність операторів маніпулювання даними (читання, видалення, вставки, модифікації), при якій або результати всіх операторів, що входять до транзакції, відображаються у БД, або впливу всіх цих операторів повністю немає.

При цьому для підтримки обмежень цілісності на рівні БД дозволяється їх порушення в транзакції так, щоб до моменту завершення транзакції умови цілісності дотримувалися.

Для забезпечення контролю над цільовими можливостями кожна транзакція починається під час цілісного стану БД і має зберегти цей стан цілісним після свого завершення. Якщо оператори, об'єднані в транзакцію, виконуються, то відбувається нормальне її завершення і БД переходить в обновлений (цілісний) стан. Якщо відбувається збій під час виконання транзакції, то відбувається так званий відкат до початкового стану БД.

12.1. Моделі транзакцій

Розглянемо дві моделі транзакцій, що використовуються у більшості великих комерційних СКБД: модель автоматичного виконання транзакцій і модель керованого виконання транзакцій, що базуються на інструкцях COMMIT і ROLLBACK мови SQL.

12.2. Автоматичне виконання транзакцій

У стандартах ANSI/ISO зафіксовано, що транзакція автоматично починається з виконання користувачем або програмою першої інструкції

SQL. Далі інструкції виконуються послідовно доти, доки транзакція не завершується одним із двох способів:

– інструкцією COMMIT, яка виконує завершення транзакцій (зміни, внесені до БД, стають постійними, а нова транзакція починає працювати після інструкцій COMMIT);

– інструкцією ROLLBACK, яка скасовує виконання поточної транзакції і повертає БД до стану початкової транзакції, нова транзакція починається після інструкції ROLLBACK.

Таку модель створено на основі моделі, прийнятої в СКБД DB2.

12.3. Кероване виконання транзакцій

Модель транзакцій на відмінну від моделі ANSI/ISO використовується в СКБД Sybase, де застосовується діалект Transact-SQL, у якому для оброблення транзакцій використовують чотири інструкції:

– BEGIN TRANSACTION, яка повідомляє про початок транзакції, тобто початок транзакції задається явно;

– COMMIT TRANSACTION, що повідомляє про успішне виконання транзакції, але при цьому нова транзакція не починається автоматично;

– SAVE TRANSACTION, яка дає змогу створити всередині транзакції *точку збереження* та призначити збереженому стану *ім'я точки збереження*, указане в інструкції;

– ROLLBACK, що скасовує виконання поточної транзакції і повертає БД до стану, де було виконано інструкцію SAVE TRANSACTION (якщо в інструкції вказано точку збереження – ROLLBACK TO ім'я_точки_збереження) або на початок транзакції.

12.4. Журнал транзакцій

Можливість відновлення стану бази даних після збоїв забезпечується з допомогою жу*рналу транзакцій*. Журналізація змін, тобто зберігання у зовнішній пам'яті інформації про всіх модифікації БД, тісно пов'язана з керуванням транзакціями.

Основним принципом узгодженої політики запису змін до журналу й безпосередньо до бази даних є те, що запис про змінення об'єкта бази даних має потрапляти до зовнішньої пам'яті журналу раніше, ніж змінений об'єкт потрапляє до зовнішньої пам'яті бази даних. Відповідний протокол журналізації (і керування буферизацією) має назву Write Ahead Log (WAL) – «спочатку запиши до журналу», і полягає в тому, що якщо потрібно зберегти у зовнішній пам'яті змінений об'єкт бази даних, то перед цим потрібно гарантувати збереження запису щодо його змінення у зовнішній пам'яті журналу.

Іншими словами, якщо в зовнішній пам'яті бази даних є деякий об'єкт бази даних, відносно якого виконано операцію модифікації, то в зовнішній

пам’яті журналу обов'язково є запис, що відповідає цій операції. Кожну успішно завершену трансакцію необхідно реально зафіксувати в зовнішній пам'яті. Який би збій не трапився, система повинна мати всі дані для відновлення стану бази даних, що містять результати всіх зафіксованих до моменту збою транзакцій.

Мінімальною вимогою, що гарантує можливість відновлення останнього узгодженого стану бази даних, є збереження у зовнішній пам’яті журналу всіх записів про змінення бази даних цієї транзакції. При цьому останнім записом до журналу, який виконується від імені цієї транзакції, є спеціальний запис про завершення транзакції.

Іноді для відновлення останнього узгодженого стану бази даних після збою журналу змін бази даних явно недостатньо. Основою відновлення в цьому випадку є зазвичай журнал і *архівна копія* бази даних.

Відновлення починається зі зворотного копіювання бази даних із архівної копії. Потім для всіх завершених транзакцій повторно виконуються всі операції. Більш точно відбувається таке: за журналом у прямому порядку виконуються всі операції; для транзакцій, які не завершилися до моменту збою, виконується відкочення.

Хоча ставляться особливі вимоги щодо надійності ведення журналу, реально можлива і його втрата. Тоді єдиним способом відновлення бази даних є повернення до архівної копії. Безумовно, у цьому випадку не вдасться отримати останній узгоджений стан бази даних.

Розглянемо способи виготовлення архівних копій бази даних, найпростіший з яких – архівувати базу даних за умови переповнення журналу. У цьому випадку утворення нових транзакцій тимчасово блокується. Коли всі транзакції завершаться і відповідно база даних набуде узгодженого стану, можна виконати її архівацію, після чого почати заповнювати журнал знову.

Можна виконувати архівацію бази даних рідше, аніж переповнюється журнал. За умови переповнення журналу й закінчення всіх початих транзакцій можна архівувати сам журнал. Оскільки такий архівований журнал потрібен лише для відновлення архівних копій бази даних, журнальну інформацію під час архівування можна значно стиснути.

У висновку формулюємо загальні вимоги до системи відновлення даних у складі СКБД.

1. Користувач не повинен здійснювати рестарт транзакцій або повторне введення даних. Відновлення має відбуватися на базі транзакції з допомогою скасування або змінення окремих транзакцій.

2. Швидке відновлення даних забезпечується генерацією даних, що використовуються для відновлення.

3. Під час виконання процедур автоматизованого відновлення користувач не повинен аналізувати склад даних і вибирати самі процедури.

Для відновлення бази даних СКБД мають у своєму складі сервісні програмні засоби:

1) *програми ведення системного журналу* реєструють операції над БД: опис відповідної транзакції, код користувача, текст вхідного повідомлення, тип змінення БД, адреси даних, що змінюються, з їх значеннями до й після змінення;

2) *програми архівації* використовуються для регулярного отримання копій БД для подальшого її відновлення;

3) *програми відновлення* застосовуються для повернення БД або деяких її частин у стан, що передує виникненню відмови, при цьому використовують резервну копію важливої БД і системний журнал;

4) *програми відкочення* ліквідують наслідки виконання певної транзакції у БД;

5) *програми запису контрольних точок і повторного виконання* дають змогу прискорити відновлення.

Отже, можна резюмувати, що транзакція – це є завершений блок звернень до бази даних і деяких дій над нею, для якого гарантується виконання чотирьох умов, так званих властивостей ACID (Atomicity, Consistency, Isolation, Durability).

Атомарність – операції транзакції утворюють нероздільний атомарний блок з певним початком і закінченням. Цей блок або виконується від початку до кінця, або не виконується взагалі. Якщо під час виконання транзакції стався збій, то відбувається відкочення до початкового стану.

Узгодженість – після завершення транзакції всі задіяні об'єкти знаходяться в узгодженому стані.

Ізольованість – одночасний доступ транзакцій різних додатків до об'єктів, що розділяються, координується так, щоб ці транзакції не впливали одна на одну.

Довгочасність – усі змінення даних, під час виконання транзакції, не втрачаються.

Лекція 13. ПАРАЛЕЛЬНЕ ВИКОРИСТАННЯ ТРАНЗАКЦІЙ

13.1. Паралельне виконання транзакцій

При паралельному обробленні даних (тобто при спільній роботі з БД кількох користувачів) СКБД має гарантувати, що користувачі не будуть заважати один одному. Засоби оброблення транзакцій дають змогу ізолювати користувачів один від одного, щоб у кожного з них було відчуття монопольної роботи з БД.

Транзакції є придатними одиницями ізольованості користувачів завдяки властивості збереження цілісності БД. Дійсно, якщо з кожним сеансом роботи з базою даних асоціюється транзакція, то кожен

користувач починає роботу з узгодженим станом бази даних, тобто з таким станом, у якому база даних могла би знаходитися, навіть якби користувач працював з нею одноосібно.

Щоб зрозуміти, як мають виконуватися паралельні транзакції, розглянемо проблеми, що виникають під час паралельної роботи з даними.

13.2. Зниклі оновлення

Розглянемо приклад роботи двох диспетчерів з модифікованою БД «Сесія». Припустимо, що Диспетчер 1 вносить у поточний навчальний план для кожної дисципліни, яка викладається на третьому курсі, відомості про викладачів, паралельно змінюючи при цьому значення стовпця «Навантаження» у таблиці «Кадровий_склад», а Диспетчер 2 виконує таку ж операцію для дисциплін другого курсу.

Диспетчер 1 починає роботу зі змінення таблиці «Навчальний_план» для Дисципліни 1 з кількістю годин, що дорівнює 50. До стовпця ID_Викладач для цієї дисципліни передбачається внести значення 5. Запит поточного навантаження викладача повертає значення 350, і Диспетчер 1 підтверджує змінення таблиці «Навчальний_план». При цьому виконуються додаткові дії зі змінення стовпця «Навантаження» в таблиці «Кадровий_склад» для рядка з ID_Викладач=5 (до стовпця вноситься значення 400).

До завершення операції Диспетчер 2 починає ті ж дії для Дисципліни 2 з кількістю годин, що дорівнює 32, яку повинен викладати той самий викладач (ID_Викладач=5). Запит поточного навантаження викладача також повертає значення 350, з яким і працює далі Диспетчер 2. Виконавши ті ж самі операції, що й Диспетчер 1 (але після нього), Диспетчер 2 поміщає у стовпець «Навантаження» значення 382, скасовуючи тим самим попередні зміни.

Для уникнення таких ситуацій із СКБД щодо синхронізації транзакцій, які виконуються паралельно, ставиться мінімальна вимога – відсутність утрачених змін.

13.3. Читання «брудних» даних

Диспетчер 1 і Диспетчер 2 знову виконують дії, описані в попередньому прикладі, але Диспетчер 2 починає запитувати дані про навантаження викладача в той момент, коли зміни, зроблені Диспетчером 1, вже зафіксовано у стовпці «Навантаження», а транзакція ще не завершилася. Запит Диспетчера 2 повертає значення 400, і Диспетчер 2 змушений скасувати свої дії, тому що 400 годин – це максимально дозволене значення навантаження. Тим часом транзакція

Диспетчера 1 завершилася поверненням до початкового стану, тобто насправді Диспетчер 2 міг би успішно завершити операцію.

Це теж не відповідає вимозі ізольованості користувачів (кожен користувач починає свою транзакцію при узгодженому стані бази даних і має право на те, щоб побачити узгоджені дані). Щоб уникнути ситуації читання «брудних» даних, необхідно вимагати, щоб до завершення однієї транзакції, що змінює деякий об'єкт, будь яка інша транзакція не могла зчитувати об'єкт під час змін.

13.4. Зчитування неузгоджених даних

Розглянемо ситуацію, яка призводить до отримання неузгоджених даних під час виконання операцій над БД.

Як і раніше Диспетчер 1 виконує операцію із заповнення рядка навчального плану, а Диспетчер 2 під час виконання своєї операції має зробити вибір між двома викладачами відповідно до їх поточного навантаження.

Починаючи роботу майже одночасно з Диспетчером 1, Диспетчер 2 отримує такі відомості: навантаження першого викладача (ID_Викладач=5) становить 350 годин, а навантаження другого викладача (ID_Викладач=7) становить 370 годин. Далі Диспетчер 2 приймає рішення на користь першого викладача, але повторний запит навантаження повертає значення 400, тому що Диспетчер 1 вже зберіг нові дані в таблиці «Кадровий_склад».

У більшості систем забезпечення ізольованості користувачів у таких ситуаціях є максимальною вимогою до синхронізації транзакцій.

13.5. Рядки-привиди

До більш тонких проблем ізольованості транзакцій належить так звана проблема рядків-привидів, що створює ситуації, які також суперечать ізольованості користувачів. Розглянемо такий сценарій.

Диспетчер 1 ініціює Транзакцію 1, яка виконує оператор вибірки рядків таблиці відповідно до заданої умови (наприклад, формування списку студентів, які здали дисципліну з ID_Дисципліна=N, за таблицею «Зведена_відомість»). До завершення Транзакції 1 Транзакція 2, викликана Диспетчером 2, додає до таблиці «Зведена_відомість» новий рядок, що задовольняє умову відбору Транзакції 1 (дані про результат складання дисципліни N ще одним студентом), і успішно завершується. Транзакція 1 повторно виконує оператор вибірки, після чого виникає рядок, якого не було під час першого виконання оператора. Звичайно, така ситуація суперечить ідеї ізольованості транзакцій.

13.6. Серіалізація транзакцій

Щоб добитися ізольованості транзакцій, СКБД має використовувати спеціальні методи регулювання спільного виконання транзакцій.

Метод *серіалізації транзакцій* – це механізм їх виконання за таким планом, коли результат спільного виконання транзакцій є еквівалентним результату деякого послідовного виконання цих же транзакцій. Забезпечення такого механізму є основною функцією керування транзакціями. Система, у якій підтримується метод серіалізації транзакцій, реально забезпечує ізольованість користувачів під час роботі з БД.

Основна реалізаційна проблема методу полягає в забезпеченні такого виконання транзакцій, щоб їх паралельність не дуже обмежувалася. Найпростішим рішенням є дійсно їх послідовне виконання, але існують ситуації, коли можна виконувати оператори різних транзакцій у будь-якому порядку, і це не призведе до проблемних ситуацій. Прикладами можуть бути транзакції, які виконують лише операції зчитування, або працюють з різними об'єктами бази даних.

Насправді, між транзакціями можуть існувати такі види конфліктів:

1) Транзакція 2 намагається змінювати об'єкт, змінений Транзакцією 1, яка ще не завершилася (W-W-конфлікт);

2) Транзакція 2 намагається змінювати об'єкт, зчитаний Транзакцією 1, яка ще не завершилася (R-W-конфлікт);

3) Транзакція 2 намагається зчитати об'єкт, змінений Транзакцією 1, яка ще не завершилася (W-R-конфлікт).

У практичних методах серіалізації транзакцій ці конфлікти враховуються.

13.7. Захоплення і звільнення об'єкта

Для забезпечення серіалізації транзакцій застосовують методи захоплення і звільнення об'єктів, які відбуваються з ініціативи транзакції: транзакція захоплює об'єкт, що призводить до його блокування для інших транзакцій, і звільняє його тільки після свого завершення. При цьому захоплення об'єктів кількома транзакціями на зчитування є сумісними (тобто кільком транзакціям дозволяється зчитувати один і той самий об'єкт), захоплення об'єкта однієї транзакції на зчитування є несумісним із захопленням іншою транзакцією того ж об'єкта на запис, і захоплення одного об'єкта різними транзакціями на запис є несумісними. Таким чином, виокремлюють два основні режими захоплень:

1) спільний режим – S (Shared), що означає роздільне захоплення об'єкта, необхідне для виконання операції зчитування об'єкта;

2) монопольний режим – X (eXclusive), що означає монопольне захоплення об'єкта, необхідне для виконання операцій запису, видалення та модифікації.

Найбільш поширеним у СКБД, що базуються на архітектурі клієнт–сервер, є підхід, який реалізує дотримання *двофазного протоколу* захоплень об'єктів БД. У загальних рисах протокол полягає в тому, що перед виконанням будь-якої операції над об'єктом бази даних від імені транзакції запитується захоплення об'єкта у відповідному режимі (залежно від виду операції – спільної або монопольної). Відповідно до цього протоколу виконання транзакції розбивається на дві фази: *перша* – громадження захоплень; *друга* (фіксація або відкочення) – звільнення захоплень.

При дотриманні двофазного протоколу основна проблема полягає в тому, що саме слід уважати об'єктом для захоплення.

У контексті реляційних баз даних можливими є такі варіанти:

1) файл – фізичний (з огляду на базу даних) об'єкт, місце зберігання кількох відношень і, можливо, індексів;

2) таблиця – логічний об'єкт, що відповідає багатьом записам певного відношення;

3) сторінка даних – фізичний об'єкт, який зберігає записи одного або декількох відношень, індексну або службову інформацію;

4) запис – елементарний фізичний об'єкт бази даних.

Очевидно, що чим більшим є об'єкт захоплення, тим менше захоплень буде підтримуватися в системі, і на це, відповідно, будуть витрачатися менші накладні витрати. Більш того, якщо вибрати як рівень об'єктів для захоплень файл або відношення, то буде вирішено навіть проблему рядків-привидів. Однак під час використання для захоплень великих об'єктів, збільшується ймовірність конфліктів транзакцій і тому зменшується дозволений ступінь їх паралельного виконання. Фактично при збільшенні об'єкта синхронізаційного захоплення ми навмисно погіршуємо ситуацію і бачимо конфлікти в тих ситуаціях, коли насправді їх немає.

БІБЛІОГРАФІЧНИЙ СПИСОК

Браст, Э. Дж. Разработка приложений на основе Microsoft SQL Server 2005. Мастер-класс : пер. с англ. / Э. Дж. Браст, С. Форте. – М. : Русская Редакция, 2007. – 880 с.

Виейра, Р. Программирование баз данных Microsoft SQL Server 2005 для профессионалов : пер. с англ. / Р. Виейра. – М. : Вильямс, 2008. – 1072 с.

Гандерлой, М. Освоение Microsoft SQL Server 2005 : пер. с англ. / М. Гандерлой, Дж. Джорден, Д. Чанц. – М. : Вильямс, 2007. – 1104 с.

Гарсиа-Молина, Г. Системы баз данных. Полный курс : пер. с англ. / Г. Гарсиа-Молина, Дж. Ульман, Дж. Уидом. – М. : Вильямс, 2004. – 1088 с.

Грабер, М. SQL. Справочное руководство : пер. с англ. / М. Грабер. – М. : Лори, 2006. – 354 с.

Дейт, К. Введение в системы баз данных : пер. с англ. / К. Дейт. – М. : Вильямс, 2001. – 1072 с.

Коннолли, Т. Базы данных: проектирование, реализация и сопровождение : пер. с англ. / Т. Коннолли, К. Бег, А. Страчан. – М. : Вильямс, 2000. – 800 с.

Малик, С. Microsoft ADO.NET 2.0 для профессионалов : пер. с англ. / С. Малик. – М. : Вильямс, 2006. — 560 с.

Нейгел, К. C# 2005 для профессионалов : пер. с англ. / К. Нейгел, Б. Ивьен, К. Уотсон. – М. : Вильямс, 2006. – 1376 с.

Нильсен, П. Microsoft SQL Server 2005. Библия пользователя : пер. с англ. / П. Нильсен. – М. : Вильямс, 2008. – 1232 с.

Селко, Д. SQL для профессионалов. Программирование : пер. с англ. / Д. Селко. – М. : Лори, 2004. – 442 с.

Сеппа, Д. Microsoft ADO.NET : пер. с англ. / Д. Сеппа. – М. : Русская Редакция, 2003. – 640 с.

Соколов, А. Ю. Проектирование баз данных / А. Ю. Соколов. – Харьков : ХАИ, 2003. – 238 с.

Троелсен, Э. Язык программирования C # и платформа .NET 2.0 : пер. с англ. / Э. Троелсен. – М. : Вильямс, 2007. – 1168 с.

ЗМІСТ

Навчальне видання

Боярчук Артем Володимирович
Шостак Анатолій Васильович

ОРГАНІЗАЦІЯ БАЗ ДАНИХ

Редактор О. Ф. Серьожкіна

Зв. план, 2020
Підписано до друку 27.11.2020
Формат 60x84 1/16. Папір офс. Офс. друк
Ум. друк. арк. 8,9. Обл.-вид. арк. 10. Наклад 100 пр.
Замовлення 284. Ціна вільна

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001