КИЇВСЬКИЙ СТОЛИЧНИЙ УНІВЕРСИТЕТ ІМЕНІ БОРИСА ГРІНЧЕНКА

Рзаєва С.Л., Машкіна І.В., Складанний П.М., Костюк Ю.В., Рзаєв Д.О., Красюк Ю.М.

ОСНОВИ БАЗ ДАНИХ

Навчальний посібник

Київ 2025

УДК 004.65

Рекомендовано Вченою радою
Факультету інформаційних технологій та математики
(протокол № 10 від 19 листопада 2025 року)

Автори: Рзаєва С.Л., Машкіна І.В., Складанний П.М., Костюк Ю.В., Рзаєв Д.О., Красюк Ю.М.

Рецензенти:
Жебка Вікторія Вікторівна, доктор технічних наук, професор, Державний університет інформаційно-комунікаційних технологій.
Макаренко Анатолій Олександрович, доктор технічних наук, професор, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського"

Рзаєва С.Л.
Основи баз даних : навчальний посібник / Рзаєва С.Л., Машкіна І.В., Складанний П.М., Костюк Ю.В., Рзаєв Д.О., Красюк Ю.М – Київ : Київський столичний університет імені Бориса Грінченка, 2025. – 319 с.

У посібнику систематизовано теоретичні основи та практичні аспекти створення, організації й управління базами даних. Розглянуто архітектуру систем керування базами даних (СКБД), принципи реляційного підходу, а також сучасні технології зберігання та обробки інформації. Описано інструменти Microsoft SQL Server, синтаксис мови SQL, засоби забезпечення цілісності, безпеки й оптимізації запитів. Значну увагу приділено питанням нормалізації, проектування схем даних і розроблення прикладних рішень на основі СКБД.

Посібник призначено для студентів закладів вищої освіти спеціальностей галузі знань 12 «Інформаційні технології», а також для тих, хто цікавляться теорією та практикою побудови сучасних баз даних.

ЗМІСТ

ВСТУП

У сучасних умовах інтенсивного розвитку інформаційного суспільства дані набувають статусу стратегічного ресурсу, від якого безпосередньо залежать ефективність управлінських рішень, конкурентоспроможність підприємств і рівень інноваційного розвитку держави. Швидке зростання обсягів інформації, що накопичується в різних сферах людської діяльності, потребує створення високоефективних технологій її збирання, зберігання, опрацювання та використання. Бази даних стали невід'ємним елементом сучасних інформаційних систем і відіграють провідну роль у забезпеченні доступності, узгодженості та достовірності інформації.

Управління даними є центральною складовою інформаційної інфраструктури будь-якої організації, а система керування базами даних – базовим програмним засобом, що реалізує цю функцію. Формування фахових компетентностей у галузі проєктування, адміністрування та експлуатації баз даних є однією з ключових цілей підготовки сучасного спеціаліста з інформаційних технологій.

Посібник «Основи баз даних» покликаний забезпечити комплексне засвоєння теоретичних положень і практичних навичок, необхідних для створення ефективних інформаційних систем, заснованих на принципах структурованого зберігання даних. Знання концептуальних засад побудови баз даних, моделей подання інформації та механізмів доступу до неї є необхідною передумовою професійної діяльності у сфері інформаційних технологій.

Фахівець у галузі баз даних повинен володіти не лише теоретичними знаннями з моделювання інформаційних структур, але й практичними навичками роботи з конкретними СКБД, зокрема Microsoft SQL Server.

Посібник «Основи баз даних» має на меті забезпечити системне подання навчального матеріалу, що охоплює як фундаментальні теоретичні основи, так і прикладні аспекти проєктування, розроблення та експлуатації баз даних. Зміст посібника структуровано відповідно до логіки навчального процесу: від загальних принципів побудови інформаційних систем до розгляду конкретних типів моделей даних, мов запитів і технологій адміністрування.

У першому розділі викладено базові поняття, принципи організації баз даних, вимоги до сучасних СКБД та їх роль у структурі інформаційних систем.

Другий розділ присвячено реляційному підходу до організації баз даних, розглянуто теоретичні положення, поняття відношення, атрибутів, ключів і нормалізацію даних.

Подальші розділи містять опис сучасних типів моделей даних, особливості розподілених та нереляційних систем, а також практичні аспекти встановлення, конфігурації та адміністрування Microsoft SQL Server.

Матеріал посібника подано з урахуванням сучасних тенденцій розвитку інформаційних технологій і спрямований на формування цілісного уявлення про концепції, принципи та інструменти організації баз даних. Особлива увага приділяється практичній складовій навчання, що включає лабораторні роботи, спрямовані на набуття досвіду встановлення, налаштування, адміністрування та тестування СКБД.

Посібник має на меті забезпечити підготовку компетентного фахівця, здатного проектувати, реалізовувати й підтримувати бази даних різного рівня складності відповідно до вимог сучасного ринку праці. Засвоєння матеріалу сприятиме розвитку аналітичного мислення, здатності працювати з інформаційними моделями, розумінню процесів зберігання, оброблення та захисту даних.

Отже, посібник «Основи баз даних» є комплексним навчальним виданням, що поєднує теоретичну обґрунтованість із практичною спрямованістю та відповідає вимогам підготовки фахівців інформаційно-технологічного профілю. Його використання у навчальному процесі забезпечить формування фундаментальних знань, необхідних для подальшого вивчення дисциплін з проєктування, інтеграції та безпеки інформаційних систем.

У цілому, запропонований навчальний матеріал сприяє становленню професійного світогляду майбутніх фахівців, здатних ефективно працювати з даними, приймати обґрунтовані рішення та впроваджувати сучасні ІТ-рішення в усіх сферах суспільної діяльності.

ТЕМА 1. ЗАГАЛЬНІ ПРИНЦИПИ ТА СУЧАСНІ ПІДХОДИ ДО БАЗ ДАНИХ

ПЛАН:

1.1. Область застосування та основні переваги баз даних (БД).
1.2. Основні поняття бази даних і системи керування базами даних (СКБД).
1.3. Типи моделей даних та баз даних.

1.1. Область застосування баз даних та основні переваги баз даних

Інформаційна система (ІС) - взаємозв'язана сукупність засобів, методів і персоналу, використовуваних для зберігання, обробки і видачі інформації в інтересах досягнення поставленої мети. Сучасні інформаційні технології надають широкий набір способів реалізації ІС, вибір яких здійснюється на основі користувацьких вимог, які, як правило, змінюються в процесі розробки.

Ядром інформаційної системи є ***база даних*** (БД) і ***система керування базою даних*** (СКБД).

Бази даних є в будь-якої компанії. Інформація, що зберігається в них часто є конфіденційною, і її розголошення веде до фінансових і репутаційних ризиків. Як класифікувати інформацію і захистити її від розкрадання.

- Data Mining – збір даних;
- Material Requirement Planning – планування потреб в матеріалах;
- Manufacturing Resource Planning – планування ресурсів виробництва;
- Enterprise Resource Planning – планування ресурсів виробництва + управління фінансами і кадрами підприємства;
- Enterprise Resource and Relationship Processing – планування ресурсів підприємства + взаємини з клієнтами і постачальниками, управління ланцюгами поставок;
- Decision-Support Systems – системи прийняття рішень;
- Data Analysis – аналіз даних;
- Data Warehousing – організація сховищ даних;
- Spatial and Geographic Databases – географічні бази даних;

- Multimedia Databases – мультимедійні бази даних;
- Mobility and Personal Databases – мобільні і персональні БД;
- Information-Retrieval Systems – інформаційні системи;
- Distributed Information Systems – розподілені інформаційні системи;
- The World Wide Web (WWW).

Основні цілі розробки бази даних:

- Забезпечення групам користувачів швидкого доступу до окремих інформаційних елементів бази даних відповідно до їхніх прав та потреб.
- Забезпечення можливості наступного розширення бази даних.
- Запобігання доступу до бази даних користувачів, що не мають відповідних повноважень.

Вимоги до сучасних СКБД:

- масштабованість – відсутність істотного зниження швидкості виконання призначених для користувача запитів при пропорційному зростанні кількості запитів і апаратних ресурсів використовуваних даної СКБД (таких як обсяг оперативної пам'яті, кількість процесорів і серверів);
- доступність – можливість завжди виконати запит;
- надійність – мінімальна ймовірність виникнення збоїв, наявність засобів відновлення даних після збоїв, інструментів резервного копіювання та дублювання даних, що забезпечують можливість здійснювати ці операції не перериваючи роботу користувачів;
- направляемость – простота адміністрування, наявність засобів автоматичної конфігурації (включає: засоби створення баз даних і їх об'єктів, опис правил реплікації даних, утиліти управління користувачами, засоби моніторингу подій, утиліти міграції з інших СКБД тощо);
- наявність засобів захисту даних від втрати і несанкціонованого доступу;
- підтримка доступу до даних за допомогою веб-служб (веб-сервісів);
- підтримка стандартних механізмів доступу до даних: ODBC, JDBC, OLE, DB, ADO .NET та ін.

Переваги застосування баз даних:

- ✓ незалежність даних – скорочення розмірів програмної підтримки (всередині окремих програм);
- ✓ збільшення ефективності розробки додатків;
- ✓ можливість створення і використання стандартів;

- ✓ мінімальна надмірність зберігання даних;
- ✓ збільшення щільності даних і спільного доступу до даних;
- ✓ полегшений доступ до даних і їх відповідність до конкретних професійних завдань.
- ✓ збільшення якості даних;
- ✓ безпека, збереження і відновлення даних.

1.2. Основні поняття бази даних і системи керування базами даних (СКБД).

Базу даних у загальному випадку можна визначити як уніфіковану сукупність збережених і відтворених даних, що використовуються у рамках організації (Engles R.A., 1972 р.). Однак поняття БД не ґрунтується в цей час на єдиній концепції, скоріше це ціле сімейство пов'язаних між собою понять з ПО, програмного й апаратного забезпечення, аналізу й моделювання даних і додатків. Існує кілька визначень БД.

База даних (за Дж. Мартіном) є сукупністю взаємозалежних даних, які спільно використовуються декількома додатками й зберігаються з мінімальною регульованою надлишковістю. Дані запам'ятовуються таким чином, щоб вони у міру можливості не залежали від програм. Для обробки даних застосовується загальний керуючий метод доступу. Якщо БД не перетинаються за структурою, то говорять про систему баз даних.

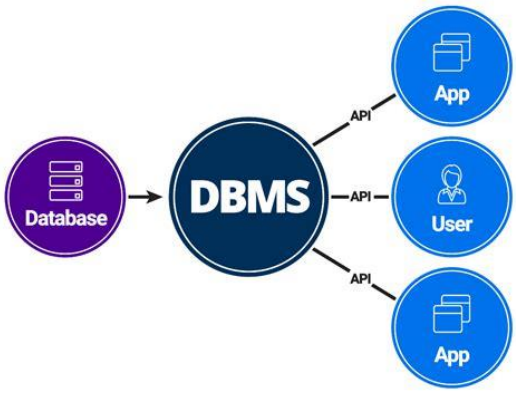
База даних (відповідно до матеріалів комітету КОДАСІЛ) складається зі всіх екземплярів записів, екземплярів наборів записів і областей, які контролюються конкретною схемою. Під схемою можна розуміти карту всієї логічної структури БД.

Для розроблювача ІС істотним моментом при використанні концепції баз даних (БД) є та обставина, що дані стають певним чином організовані, здобувають якусь упорядкованість і внутрішню структуру, а також те, що є деякий набір уніфікованих операцій обробки даних і декларативних засобів подання даних. До таких операцій варто віднести операції «Вставити» (Insert), «Додати» (Add), «Видалити» (Delete) і ряд інших. До декларативних засобів подання даних варто віднести мови визначення даних. Тобто використання даної концепції при створенні ІС припускає наявність мови визначення даних і мови маніпулювання даними, а також правил побудови інтерфейсів програм (додатків) із БД і користувачем.

Такий розподіл засобів маніпулювання даними і їхнього подання є деякою мірою умовним. Мова визначення даних служить для опису логічної структури (схеми) БД, а в деяких випадках і способів зберігання й доступу до даних. Мова маніпулювання даними надає алгоритмічні засоби побудови додатків для обробки елементів даних, які зберігаються в БД.

У разі застосування концепції БД для створення ІС природно виникає питання: хто або що повинне все це підтримувати? Таким чином, постає питання про систему керування базою даних (Database Management System DBMS).

Система керування базами даних (СКБД) – це програмна система, яка надає користувачам інтерфейс для взаємодії з базами даних, дозволяючи їм зберігати, отримувати, оновлювати та ефективно керувати даними. Саме СКБД повинна надати засоби визначення й маніпулювання даними, зробивши дані незалежними від прикладних програм, що їх використовують.



До основних функцій СКБД слід віднести:

- забезпечення мовних засобів опису та маніпуляції даними;
- забезпечення підтримки логічної моделі даних;
- забезпечення взаємодії логічної та фізичної структур даних;
- забезпечення захисту та цілісності даних;
- забезпечення підтримки БД в актуальному стані.

Стосовно комп'ютерної обробки даних під інформацією розуміють деяку послідовність символічних позначень (букв, цифр, закодованих графічних образів і тощо.), яка несе значне навантаження і відображена в зрозумілому комп'ютеру виді.

Зауваження:

Для побудови оптимальної бази даних потрібно правильно організувати збереження й доступ до даних БД.

Необхідно побудувати інформаційну модель предметної області таким чином, щоб вона максимально повно відбивала структуру й взаємозв'язки в ІС. Така модель і буде надалі покладена в основу системи обробки даних.

1.3. Типи моделей даних та баз даних

Подання інформації за допомогою даних вимагає уніфікованого підходу до поняття даних як незалежного об'єкта моделювання. Тому для розробника

ІС вибір відповідної моделі даних є однією з найважливіших проблем. Вибір моделі даних спричиняє вибір засобів аналізу ПО як області реального світу, що підлягає вивченню й обробці. Модель даних обмежує можливість вибору СКБД, тому що, як правило, окремо взята модель підтримує певну модель даних.

***Модель даних* (*Data Model*)** є логічною структурою даних, що становить притаманні цим даним властивості, незалежні від апаратного й програмного забезпечення й не пов'язані з функціонуванням комп'ютера.В залежності від виду логічної структури даних розрізняють декілька типів моделей даних:

Ієрархічна модель (Hierarchical model) подає дані у вигляді дерева і тому є занадто громіздкою при описі даних із складним логічними зв'язками.

Ієрархічна модель являє собою зв'язний неорієнтований граф деревовидної структури, що об'єднує сегменти. Ієрархічна БД складається з упорядкованого набору дерев.

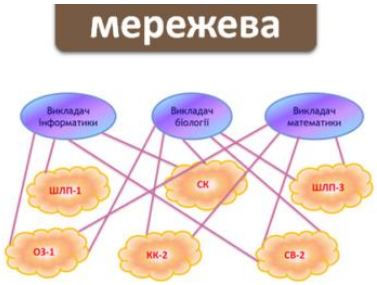
Цікавий факт:

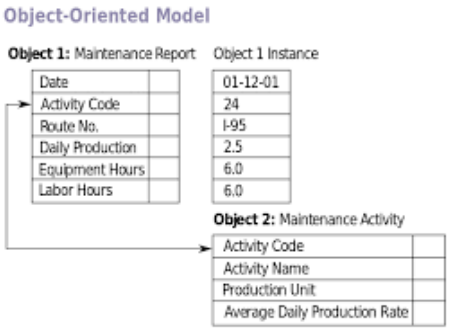
Об'єкти, що мають загального предку, називаються близнюками.

Наприклад, якщо ієрархічна база даних містила інформацію про покупців та їх замовленнях, то буде існувати об'єкт "покупець" (батько) і об'єкт "замовлення" (дочірній). Об'єкт "покупець" матиме покажчики від кожного замовника до фізичного розташування замовлень покупця на об'єкт "замовлення".

У цій моделі запит, направлений вниз по ієрархії, простий (наприклад: які замовлення належать цьому покупцеві), а проте запит, направлений вгору по ієрархії, більш складний (наприклад, який покупець помістив це замовлення). Також, важко уявити не-ієрархічні дані при використанні цієї моделі.

Мережна модель (Network model) використовується для представлення структур даних у вигляді довільного графа. За її допомогою можна описати структуру даних будь-якої складності, проте це пов'язане зі значними труднощами.





Об'єктно-орієнтована модель (Object-oriented model) містить у собі особливості реляційної моделі і об'єктної технології, що полягає в об'єднанні даних і функцій їхньої обробки в єдине ціле.

В основу даної моделі покладена концепція обєктно–орієнтованого програмування, основними признаками якого являється наслідування, інкапсуляція і поліфоризм, а дані представлені у вигляді набору обєктів і класів, звязаних між собою.

Реляційна модель (Relational model) застосовується для представлення структур даних у вигляді сукупності таблиць, пов'язаних відношеннями.

Вона широко поширена завдяки:

1. *Наочності і гнучкості своєї структури.*
2. *Зручності представлення даних.*
3. *Простоті реалізації методів обробки даних.*

Прізвище	І'мя	Адреса	Телефон
Петров	Іван	Суворовский пр., б. 15, кв. 55	345-75-34
Іванова	Інна	Кірочна вул., б.25, кв.13	345-26-96

Реляційна модель даних являє собою сукупність взаємопов’язаних простих двомірних таблиць – відношень.

Зв’язки між двома логічно пов’язаними таблицями в реляційній моделі установлюються по збігу значень однакових атрибутів таблиць – відношень.

Реляційна модель даних (РМД) включає такі компоненти:

Структурний аспект (складова) - дані в базі даних є набором відносин.

Аспект (складова) *цілісності* - відносини (таблиці) відповідають певним умовам цілісності. РМД підтримує декларативні обмеження цілісності рівня домену (типу даних), рівня відносини і рівня бази даних.

Аспект (складова) *обробки* (маніпулювання) - РМД підтримує оператори маніпулювання відносинами (реляційна алгебра, реляційне числення).

Крім того, до складу реляційної моделі даних включають теорію нормалізації.

Для кращого розуміння РМД слід відзначити три важливі обставини:

- модель є логічною, тобто відносини є логічними (абстрактними), а не фізичними (збереженими) структурами;
- для реляційних баз даних вірний інформаційний принцип : все інформаційне наповнення бази даних представлено одним і тільки одним способом, а саме - явним завданням значень атрибутів у кортежі відносин; зокрема, немає ніяких покажчиків (адрес), що зв'язують одне значення з іншим;
- наявність реляційної алгебри дозволяє реалізувати декларативне програмування і декларативне опис обмежень цілісності, на додаток до навігаційного (процедурним) програмування і процедурної перевірці умов.

Зауваження:

Термін «реляційний» означає, що теорія заснована на математичному понятті ставлення (relation). Як неформального синоніма терміну "відношення" часто зустрічається слово таблиця. Необхідно пам'ятати, що "таблиця" є поняття нестроге і неформальне і часто означає не "ставлення" як абстрактне поняття, а візуальне уявлення відносини на папері або екрані. Некоректне і нестрогое використання терміну "таблиця" замість терміна "ставлення" нерідко призводить до нерозуміння. Найбільш часта помилка полягає в міркуваннях про те, що РМД має справу з "плоскими", або "двовимірними" таблицями, тоді як такими можуть бути тільки візуальні представлення таблиць. Відносини ж є абстракціями, і не можуть бути ні "плоскими", ні "неплоским".

В основі реляційної моделі лежать поняття відношення (relations), подане у вигляді таблиці з дотриманням деяких обмежувальних умов. Основні взаємопов’язані поняття фізичного, спеціального прикладного та математичного рівнів побудови реляційної бази даних та їх взаємовідношення показані в таблиці, в рядках якої знаходяться еквівалентні поняття

Стовпці відношення називають *атрибутами* і їм присвоюють *імена*.

Список імен атрибутів відношення називається *схемою відношення*. Таблиці не дозволяють проводити узгодження наступних трьох способів маніпулювання даними: впорядкування, групування по певних ознаках, доступ по дереву параметрів. Це пов'язано з тим, що в таблиці всі три способи маніпулювання жорстко закріплені: дані впорядковані по одному параметру і не впорядковані по іншому.

Цікавий факт:

Принципи реляційної моделі були сформульовані в 1969 - 1970 роках Е. Ф. Коддом (EF Codd). Ідеї Кодда були вперше публічно викладені в статті "A Relational Model of Data for Large Shared Data Banks" [1], що стала класичною.

Суворе викладення теорії реляційних баз даних (реляційної моделі даних) в сучасному розумінні можна знайти в книзі К. Дж. Дейта. "CJ Date. An Introduction to Database Systems" ("Дейт, К. Дж. Введення в системи баз даних").

Концепція реляційної моделі бази даних запропонована Е.Ф.Коддом в 1970 році для розв'язання наступної задачі - забезпечити незалежність представлення та опису даних від прикладних програм.

Постреляційна модель (**Post-shot model**) відрізняється від реляційної наявністю можливості відслідковування зміни даних у часі.

Зауваження:

Постреляційна модель допускає багатозначні поля = поля, значення яких складається із підзначень (наприклад, таблиця, яка вбудована ще в одну таблицю).

Тут процес збереження даних більш ефективний і не потребує виконання операції з'єднання двух таблиць.

В постреляційній моделі на довжину полів не накладаються вимоги постійності, що робить структуру бази даних більш

гнучкою, але дуже сильно ускладнює факт її фізичної реалізації.

Таким чином, постреляційна модель забезпечує високу можливість надання інформації і ефективну обробку даних, але ускладнено забезпечення цілісності даних, які зберігаються в такій моделі.

Структуровані дані – це дані, які мають чіткі, визначені зв'язки між точками даних із попередньо визначеною моделлю, що їх містить.

Структуровані дані часто зберігаються в таких таблицях, як файли Excel або бази даних SQL. У цих випадках рядки та стовпці даних містять різні змінні або функції, і часто можна визначити зв'язок між точками даних, перевіривши, де перетинаються рядки та стовпці даних. Структуровані дані можна легко вмістити в реляційну базу даних SQL, а приклади різних функцій у структурованому наборі даних можуть включати такі елементи, як імена, адреси, дати, статистика погоди, номери кредитних карток тощо. Хоча структуровані дані найчастіше є текстовими даними, це можна також зберігати такі речі, як зображення та аудіо, як структуровані дані.

Загальні джерела структурованих даних включають такі речі, як дані, зібрані з датчиків, веб-журналів, мережеві дані та дані роздрібної торгівлі чи електронної комерції. Структуровані дані також можуть бути створені людьми, які заповнюють електронні таблиці або бази даних даними, зібраними з комп'ютерів та інших пристроїв. Наприклад, дані, зібрані через онлайн-форми, часто відразу вводяться в структуру даних.

Структуровані дані мають довгу історію зберігання в реляційних базах даних та популярні через легкість читання та запису в цих форматах, оскільки більшість платформ і мов можуть інтерпретувати ці формати даних.

Неструктуровані дані – це дані, які не організовані заздалегідь визначеним чином або не мають певної моделі даних або структури. Неструктуровані дані часто називають якісними даними, оскільки їх неможливо проаналізувати або обробити традиційними способами за допомогою стандартних методів, які використовуються для структурованих даних.

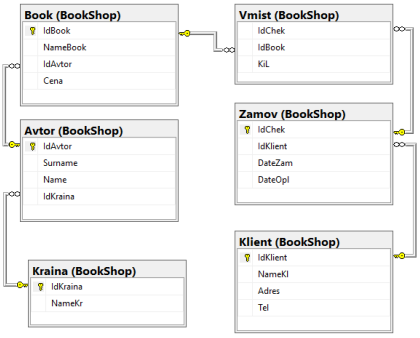
Оскільки неструктуровані дані не мають визначених зв'язків між точками даних, їх не можна організувати в реляційних базах даних. Навпаки, неструктуровані дані зазвичай зберігаються базу даних NoSQL або нереляційну базу даних. Якщо структура бази даних не має особливого значення, для зберігання даних замість бази даних NoSQL можна використовувати озеро даних або великий пул неструктурованих даних.

Зауваження:

Неструктуровані дані важко проаналізувати, і для визначення сенсу неструктурованих даних часто потрібно перевірити окремі фрагменти даних, щоб розпізнати потенційні особливості, а потім перевірити, чи присутні ці особливості в інших фрагментах даних у пулі.

Переважна більшість даних міститься в неструктурованих форматах, за оцінками, неструктуровані дані складають близько 80% усіх даних. Для структурування даних можна використовувати методи інтелектуального аналізу даних.

Напівструктуровані дані можуть містити обидві форми даних. Ми можемо бачити напівструктуровані дані як структуровані, але не визначені, як, наприклад, визначення таблиці в реляційних СКБД. Прикладом напівструктурованих даних є XML-файл.



Реляційна база даних (Relational databases) – це тип бази даних, яка організовує дані в рядки та стовпці, які разом утворюють таблицю, де точки даних пов'язані одна з одною.

Причина, по якій такі базу даних називають реляційною, полягає в тому, що англійський термін «relation» (відношення), по суті, являє собою загальноприйняту математичну назву для таблиці. Тому на практиці терміни відношення і таблиця можна вважати синонімами. Для уточнення терміну відношення виділяються поняття заголовка відношення, значення відношення і змінної відношення. Крім того, для пояснення потрібно допоміжне поняття кортежу.

Системи з підтримкою SQL, або просто системи SQL, в даний час стали переважаючими на ринку баз даних, і однією з важливих причин подібного стану справ є саме те, що такі системи засновані на реляційній моделі даних. Тому не дивно, що в неформальному спілкуванні системи SQL часто називають просто реляційними системами.

Як уже зазначалося, користувач реляційної системи бачить дані у вигляді таблиць і ніяк інакше. Користувач нереляційних системи, навпаки, бачить дані, представлені в інших структурах.

Як було описано в попередніймемі, реляційна модель поширюється на три принципові аспекти організації даних: структура даних, маніпулювання даними та цілісність даних.

Цікавий факт:

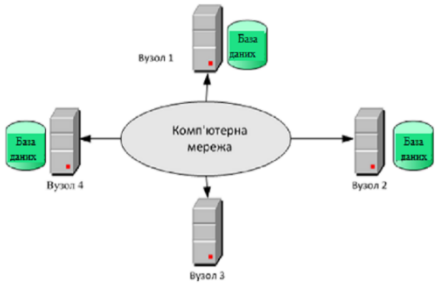
Таблична форма подання відношення була введена з метою популяризації моделі серед непідготовлених користувачів БД. Трактування реляційної теорії на рівні таблиць приховують ряд визначень, важливих для розуміння як теорії реляційних БД, так і мови маніпулювання даними.

Основні характеристики реляційних баз даних:

- Таблична структура – дані зберігаються в таблицях із заздалегідь визначеними схемами, що складаються з рядків і стовпців. Кожна таблиця представляє певну сутність, а зв'язки між таблицями встановлюються за допомогою зовнішніх ключів.
- SQL (мова структурованих запитів) : реляційні бази даних використовують SQL, потужну та стандартизовану мову, для визначення, обробки та запиту даних. SQL забезпечує послідовний і ефективний спосіб взаємодії з базою даних.
- Відповідність ACID : СКБД дотримуються властивостей ACID (атомність, узгодженість, ізоляція, довговічність), забезпечуючи цілісність даних і послідовність під час транзакцій.
- На основі схеми : для реляційних баз даних потрібна заздалегідь визначена схема, яка визначає структуру даних перед тим, як їх можна буде зберегти. Це забезпечує послідовність даних, але може обмежити гнучкість.
- Проблеми масштабованості : у міру зростання обсягів даних вертикальне масштабування (додавання додаткових ресурсів до одного сервера) стає складнішим і дорожчим. Горизонтальне масштабування (розподіл даних на кількох серверах) може бути складним.

Популярні приклади реляційних баз даних включають MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server і IBM Db2.

Розподілені бази даних (***DDB – distribute databases***) сукупність багатьох взаємопов'язаних баз даних, розподілених у комп'ютерній мережі.



Розподілені бази даних можна розділити на дві категорії: однорідні та гетерогенні.

Однорідна система розподіленої бази даних використовує однакове апаратне забезпечення, операційні системи та програми баз даних у всіх місцях. Система представляє себе як єдине ціле для користувача, що спрощує її проектування та керування. Щоб вважатися однорідними, структури даних і додаток бази даних на кожному сайті повинні бути або ідентичними, або сумісними.

І навпаки, неоднорідна система розподіленої бази даних може мати різноманітне апаратне забезпечення, операційні системи або додатки бази даних на кожному сайті. У різних місцях можуть використовуватися різні схеми та програмне забезпечення, що може ускладнити обробку запитів і транзакцій.

У гетерогенних налаштуваннях вузли можуть відрізнятися апаратним забезпеченням, програмним забезпеченням, структурою даних або навіть перебувати в несумісних місцях. Користувачі одного сайту можуть мати доступ для читання даних на іншому сайті, але не мати можливості завантажувати чи змінювати їх. Через їх складність працювати з гетерогенними розподіленими базами даних може бути важко, що робить їх менш економічно життєздатними для багатьох підприємств.

Система керування розподіленою базою даних – програмна система, яка дозволяє керувати базою даних таким чином, щоб її розподіленість була прозора для користувачів. Як правило, системи керування розподіленими базами даних працюють на двох або більше взаємопов’язаних серверах у комп’ютерній мережі. Кожне розташування, де працює версія бази даних, часто називають екземпляром або вузлом.

Основні характеристики реляційних баз даних: розподілення даних між кількома точками для підвищення масштабованості, локальності та надійності. Вони незамінні в таких секторах, як фінанси, телекомунікації, ігри та Інтернет речей, які потребують високої доступності, масштабованості та надійності.

Нереляційні бази даних (Non Relational database) з'явилися у відповідь на зростаючі вимоги сучасних додатків і необхідність ефективної обробки великих обсягів неструктурованих або напівструктурованих даних. Ці бази даних відходять від традиційної табличної структури реляційних баз даних і пропонують більш гнучкі моделі даних.

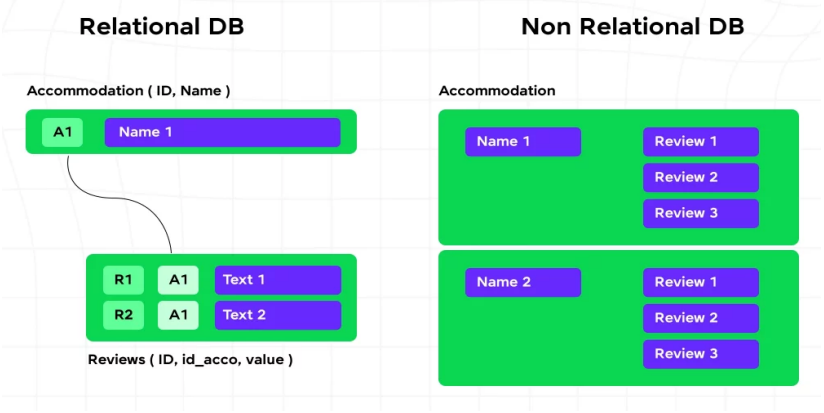
Основні характеристики нереляційних баз даних:

- Гнучкі моделі даних : нереляційні бази даних підтримують різні моделі даних, такі як сховища ключ-значення, бази даних документів, сховища сімейства стовпців і бази даних графів, що забезпечує більш гнучке та масштабоване зберігання даних.
- Безсхемна або динамічна схема : більшість баз даних NoSQL не містять схем або мають динамічну схему, тобто структура даних може змінюватися з часом без необхідності використання попередньо визначеної схеми.
- Горизонтальна масштабованість : нереляційні бази даних призначені для горизонтального масштабування шляхом розподілу даних між декількома серверами або кластерами, що полегшує роботу з великими обсягами даних і великим трафіком.
- Висока доступність і толерантність до розділів: багато баз даних NoSQL надають перевагу високій доступності та допуску до розділів, дотримуючись теореми CAP (узгодженість, доступність і допуск до розділів) і надають перевагу доступності над суворою узгодженістю.
- Мови запитів: хоча бази даних NoSQL не покладаються на SQL, вони часто надають власні мови запитів або API для взаємодії з даними, такі як мова запитів MongoDB або мова запитів Cassandra (CQL).

Популярні приклади нереляційних баз даних включають MongoDB (база даних документів), Cassandra і HBase (сховища сімейства стовпців), Redis і Memcached (сховища ключів і значень) і Neo4j (база даних графів).

✍ Ключові відмінності між реляційними та нереляційними базами даних

Хоча як реляційні, так і нереляційні бази даних служать для зберігання та керування даними, вони суттєво відрізняються принципами проектування, моделями даних, мовами запитів, масштабованістю та гарантіями узгодженості. Розуміння цих відмінностей має вирішальне значення при виборі відповідної системи бази даних для конкретного випадку використання.



У таблиці 1.1 подана порівняльна характеристика реляційних та нереляційних баз даних.

Таблиця 1.1.

Порівняльна характеристика реляційних та нереляційних баз даних

Аспект	***Реляційні бази даних***	***Нереляційні бази даних***
Модель даних	Жорстка схема. Структуру необхідно визначити перед вставкою даних. Зміна схеми може бути складною та трудомісткою.	Використовує різні моделі даних: ключ-значення, документ, стовпчик або графік. Схеми бувають динамічними або безсхемними.
Схема	Основна банківська система банку використовує Oracle або PostgreSQL для керування рахунками, транзакціями та фінансовими записами.	Гнучка схема або без схеми. Може легко вносити зміни в структуру даних.
Масшта-бованість	Вертикальне масштабування (збільшення) зазвичай легше. Горизонтальне масштабування може бути	Призначений для горизонтального масштабування (scaling out). Може легко

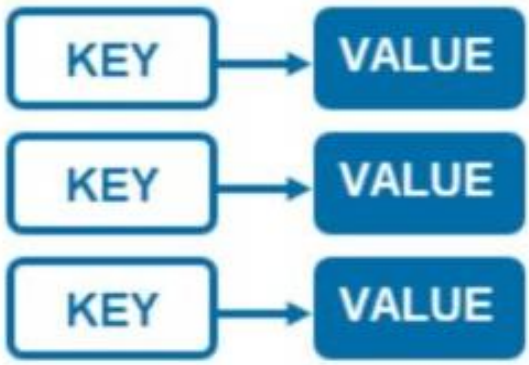
Аспект	*Реляційні бази даних*	*Нереляційні бази даних*
	складним і часто потребує шардингу.	розподіляти дані між декількома серверами.
Мова запитів	Використовує стандартизований SQL (структурована мова запитів) для запитів і обробки даних.	Часто використовують специфічні для бази даних API запитів. Деякі підтримують SQL-подібні запити.
Відповідність ACID	Повністю відповідає ACID (атомарність, консистенція, ізоляція, міцність). Забезпечує цілісність даних і надійність транзакцій.	Багато баз даних NoSQL жертвують повною сумісністю з ACID заради продуктивності та масштабованості. Деякі пропонують регульовану консистенцію.
Відносини даних	Підтримує складні зв’язки між об’єктами через зовнішні ключі та об’єднання.	Зазвичай денормалізує дані, часто вбудовуючи пов’язані дані в документи або використовуючи ідентифікатори посилань.
Продуктивність	Оптимізовано для складних запитів і об’єднань. Може працювати повільніше для дуже великих наборів даних або великих навантажень на запис.	Зазвичай швидше для простих операцій читання/запису, особливо в масштабі. Може мати проблеми зі складними запитами, які потребують об’єднань.
Цілісність даних	Забезпечує цілісність даних через обмеження (наприклад, зовнішні ключі, унікальні обмеження, обмеження перевірки).	Часто не має вбудованих обмежень цілісності даних. Цілісністю необхідно керувати на рівні програми.

Аспект	*Реляційні бази даних*	*Нереляційні бази даних*
Гнучкість	Менш гнучкий для зміни моделей даних. Зміни схеми можуть бути руйнівними.	Висока гнучкість. Може легко адаптуватися до мінливих вимог до даних.
Стандартизація	Високий ступінь стандартизації. SQL широко використовується і зрозумілий.	Менше стандартизації. Кожна база даних NoSQL може мати власну мову запитів та API.
Резервне копіювання та відновлення	Розробка може бути повільнішою через необхідність попереднього проектування та нормалізації схеми.	Резервне копіювання та відновлення можуть бути складнішими, особливо в розподілених налаштуваннях. Може покладатися на реплікацію для довговічності даних.
Узгодженість даних	Забезпечує міцну консистенцію. Усі клієнти бачать однакові дані одночасно.	Часто забезпечує кінцеву послідовність. Може пропонувати регульовані рівні узгодженості.
Обробка великих даних	Може працювати з дуже великими наборами даних (від терабайтів до петабайтів) на одному сервері.	Призначений для обробки величезних обсягів даних, розподілених між кількома вузлами.
Обробка даних у реальному часі	Зазвичай не оптимізовано для масштабної обробки даних у реальному часі.	Багато баз даних NoSQL розроблені для прийому та обробки даних у реальному часі.

Документоорієнтовані БД (Document Oriented) призначені для зберігання і запиту даних у вигляді документів у форматі, подібному до структурованих форматів - XML, JSON, BSON. При цьому зберігається адресний доступ до даних за ключем. Вміст документа може мати різний набір властивостей.

JavaScript Object Notation (JSON) – це відкритий формат обміну даними, який читається як людиною, так і машиною. Розробники можуть використовувати документи JSON у своєму коді та зберігати їх безпосередньо в базі даних документів.

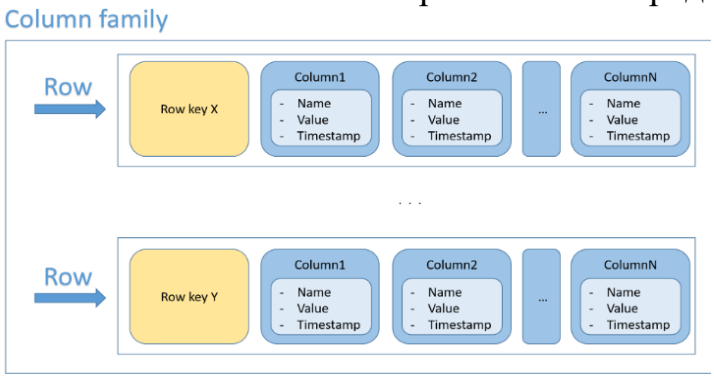
Бази даних «Ключ-значення» (Key-Value) найпростіший різновид нереляційних БД. Дані зберігаються як словник, де покажчиком є ключ. Сховище пар «ключ – значення» по суті являє собою велику хеш-таблицю. Кожне значення зіставляється з унікальним ключем, і сховище ключів використовує цей ключ для зберігання даних, застосовуючи до нього деяку функцію хешування. Вибір функції хешування має забезпечити рівномірний розподіл хешованих ключів по сховищу даних.



Графові (Graph) бази даних призначені для моделювання складних відносин за допомогою теорії графів, де зв’язками виступають ребра графа, а самі об’єкти - це вузли чи вершини.

Вузли і ребра мають властивості, які надають відомості про конкретний вузол або ребро, приблизно як стовпці в реляційній таблиці. Грані можуть мати напрямок, що вказує на характер зв'язку.

Колонкові або стовпцево-орієнтовані (Column-Family) бази даних. Записи в таких базах зберігаються не рядками, а стовпцями (колонками).



Замість таблиць тут використовуються колонкові сімейства. Вони містять ключі, які вказують на формат рядка запису інформації про об'єкт. Кожен рядок має свій набір властивостей, що дозволяє зберігати у межах однієї родини по-різному структуровані дані.

Багатомодельні або Масив-орієнтовані бази (Array-Based) надають гнучкі, масштабовані сервіси для масивних багатовимірних масивів, що складаються з функцій управління зберіганням та обробки багатовимірних масивів, які формують основну структуру даних, дозволяють отримувати дані на основі властивостей і вмісту масиву, використовуючи декларативні мови.

Обробка масивів є основною функціональністю в таких базах даних з великим набором операцій, починаючи від простого налаштування підмножин і закінчуючи статистикою, обробкою сигналів і зображень та загальною лінійною алгеброю.

Питання для самоперевірки

1. *Призначення та основні функції баз даних.*
2. *Назвіть типи моделей даних.*
3. *Перерахуйте основні інформаційні одиниці в ієрархічній моделі баз даних.*
4. *Що означає термін «реляційний»?*
5. *Назвіть основні характеристики реляційних баз даних.*
6. *В чому полягає різниця між реляційними та нереляційними базами даних?*
7. *Особливості розподілених баз даних.*
8. *В чому полягає відмінність між структурованими та неструктурованими даними?*
9. *Назвіть відомі реляційні СКБД.*
10. *Які системи входять до складу нереляційних баз даних?*
11. *Що є основною функціональністю у багатомодельних базах даних*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 1

на тему: *«Встановлення та початкова конфігурація служби MS SQL Server»*

МЕТА: *ознайомитися з процесом встановлення Microsoft SQL Server, вивчити призначення основних компонентів та параметрів конфігурації, сформувати навички правильного налаштування служб, директорій та режимів автентифікації для забезпечення продуктивної та безпечної роботи СКБД*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. Завантажити та запустити інсталяційний пакет Microsoft SQL Server.
2. Виконати перевірку сумісності апаратного та програмного забезпечення із системними вимогами СКБД.
3. Ознайомитися з ліцензійними умовами та обрати відповідну редакцію продукту.
4. Налаштувати тип інсталяції (новий екземпляр або додавання компонентів) та обрати необхідні функції.
5. Виконати конфігурацію екземпляра, служб, облікових записів і параметрів автентифікації.
6. Задати шляхи зберігання даних, журналів, резервних копій та налаштувати TempDB.
7. За потреби виконати інсталяцію додаткових служб (Analysis Services, Reporting Services).
8. Перевірити правильність обраних параметрів і завершити інсталяцію.
9. Провести початкову перевірку працездатності встановленого SQL Server.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Microsoft SQL Server виконується в операційних системах *Microsoft Windows* у вигляді служб.

Служба особливий вид програми, яка виконується в системі у фоновому режимі. Служби зазвичай забезпечують базові функції операційної системи, наприклад, ведення журналу подій, доступ до файлів або вебсторінок.

Служби можуть виконуватися без відображення свого інтерфейсу користувача на робочому столі комп'ютера. Компонент *SQL Server Database Engine*, агент *SQL Server* і деякі інші компоненти *SQL Server* виконуються як служби і зазвичай запускаються при запуску операційної системи. Деякі служби за замовчуванням не запускаються, це залежить від зазначених при установці параметрів.

SQL Server має у своєму склад і компоненти для виконання завдань управління – середовище *SQL Server Management Studio.*

Microsoft SQL Server Management Studio основний засіб адміністрування *Microsoft SQL Server*, це інтегроване середовище, в якому передбачені функції підключення, налаштування, управління, адмініструванняі розробки для всіх компонентів *SQL Server* (рис. 4.1). Середовище *SQL Server Management Studio* об'єднує великий набір графічних засобів і ряд потужних редакторів сценаріїв, а також забезпечує доступ до *SQL Server* для розробників і адміністраторів усіх рівнів.

За допомогою середовища *SQL Server Management Studio* розробники та адміністратори баз даних можуть розробляти і адмініструвати будь-які компоненти *Database Engine*.

Компонент Database Engine ядро СКБД SQL Server є основною службою для зберігання, обробки та захисту даних. Ядро СКБД забезпечує контрольований доступ і швидке опрацювання транзакцій відповідно до вимог найвимогливіших додатків, що використовують дані у вашій організації.

Процес встановлення SQL Server складається з послідовних кроків, кожен із яких має своє функціональне призначення: від перевірки сумісності системи й прийняття ліцензійних умов до вибору компонентів, конфігурації служб і налаштування директорій зберігання даних. Дотримання рекомендованих практик на кожному етапі дозволяє уникнути помилок та забезпечити оптимальні умови для роботи СКБД.

Даний посібник містить покроковий опис інсталяції SQL Server із поясненням значення кожного параметра та рекомендаціями щодо їх налаштування. Для наочності всі етапи супроводжуються відповідними рисунками, які відображають вигляд вікон інсталяційного майстра та його параметри.

Для завантаження ліцензійного програмного продукту *Microsoft SQL Server* необхідно перейти на сайт *Microsoft* за таким покликанням:

https://www.microsoft.com/uk-ua/sql-server/sql-server-downloads

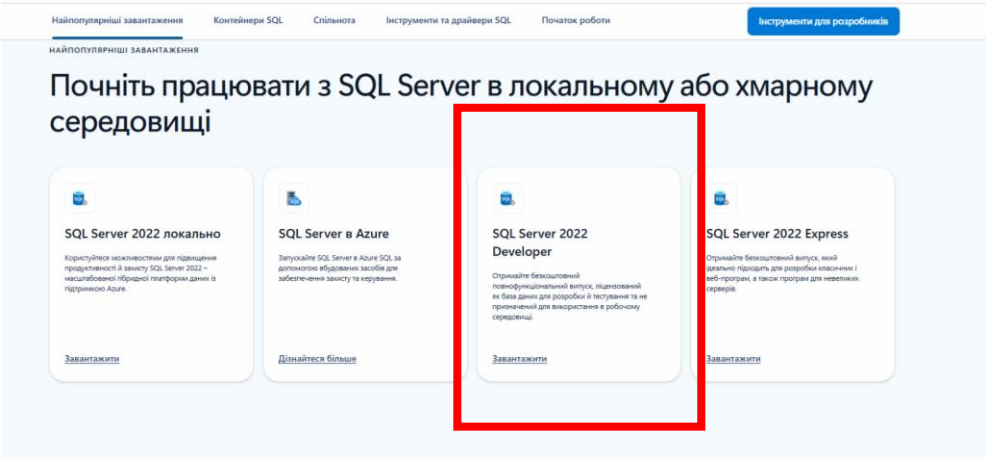
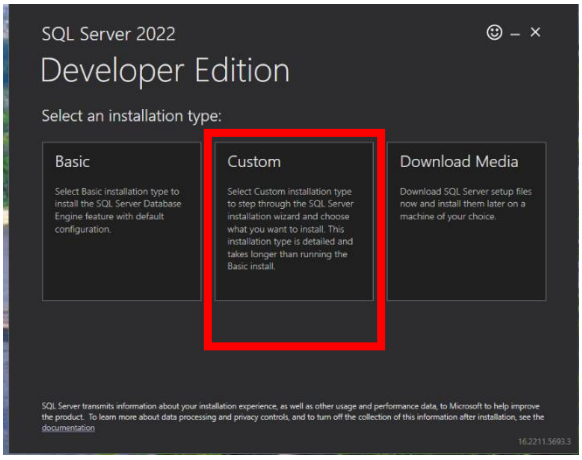


Рис. 1. Вікно завантаження Microsoft SQL Server

У вікні обирається середовище *SQL Server 2022 Developer*. Інсталяційний файл буде завантажено на диск комп'ютера. При його запуску необхідно обрати опцію «Запуск від імені адміністратора».



На початковому етапі установки користувачеві пропонується вибрати тип інсталяції.

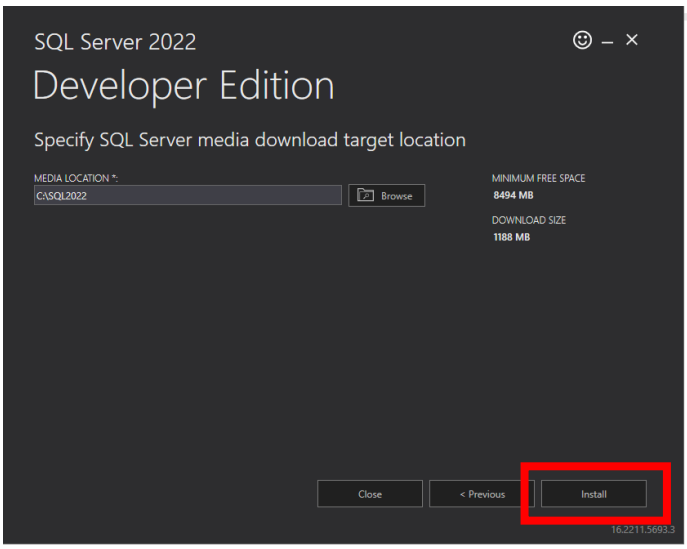
Доступні три варіанти:

Basic – базова інсталяція. Дозволяє швидко встановити *SQL Server* з типовими налаштуваннями без детальної конфігурації.

Custom – користувацька інсталяція. Дає можливість пройти через *SQL Server Installation Wizard*, вибираючи конкретні параметри та компоненти, які будуть встановлені. Це більш детальний та гнучкий варіант порівняно з базовим.

Download Media – завантаження інсталяційних файлів для подальшої офлайн-установки на іншому комп'ютері або в іншому середовищі.

На даному етапі обирається *Custom* (при виборі даного компоненту у нижній частині вікна наведено інформацію від *Microsoft* щодо того, що встановлення *SQL Server Developer Edition* призначене для тестування й розробки, але не для використання у виробничих середовищах, а отже він підходить для використання у навчальних цілях).



Далі на екрані відображаються ліцензійні умови *Microsoft* для відповідної редакції *SQL Server*. Продовжити інсталяцію можна лише після встановлення прапорця про прийняття умов. Після прийняття умов інсталятор розблоковує кнопку *Next*. Відхилення угоди припиняє інсталяцію.

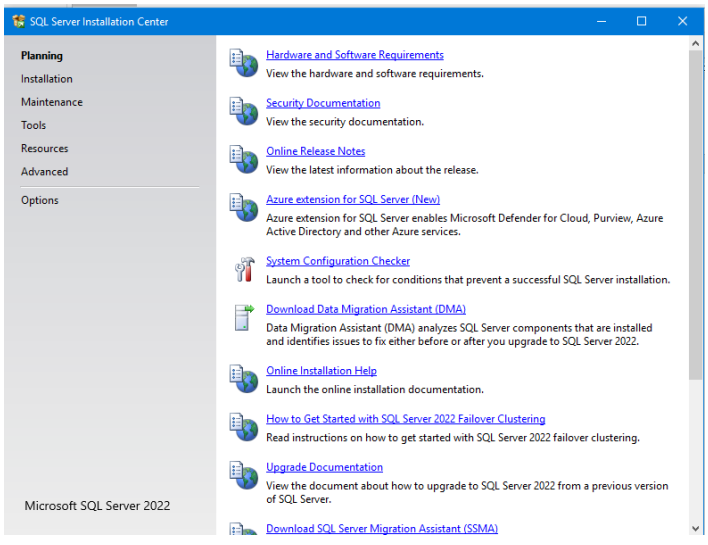
У вікні наступного етапу інсталяції *Microsoft SQL Server 2022 Developer Edition* відображається налаштування шляху для завантаження та розпакування інсталяційних файлів:

MEDIA LOCATION – шлях, куди буде завантажено інсталяційні файли SQL Server (за замовчуванням C:\SQL2022).

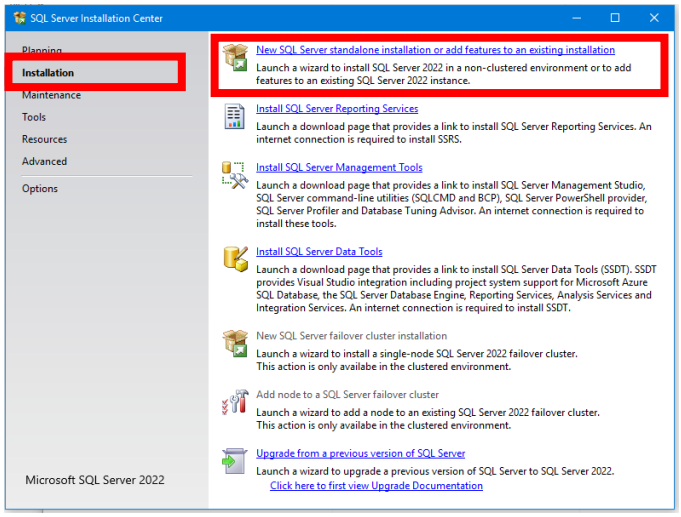
MINIMUM FREE SPACE – мінімально необхідний вільний простір на диску (8494 MB).

DOWNLOAD SIZE – розмір завантажуваних файлів (1188 MB).

Якщо вказані параметри відповідають конфігурації комп'ютера, слід натиснути кнопку *Install***.**

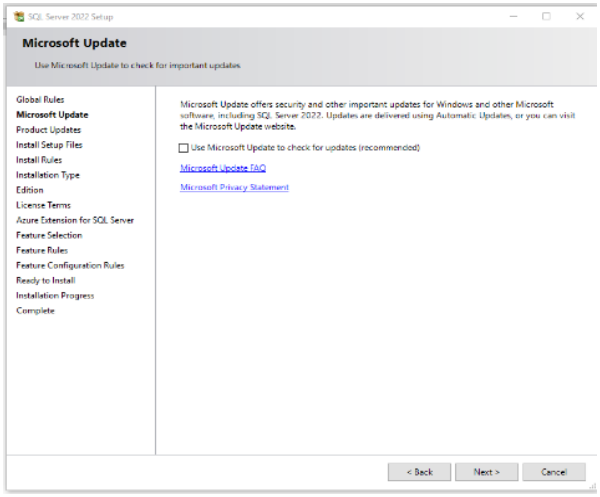


Далі відкривається вікно *SQL Server Installation Center*, яке слугує точкою входу до всіх сценаріїв встановлення та обслуговування. У лівій колонці навігації відображаються розділи на кшталт *Planning, Installation,*



Maintenance, *Tools* тощо. Центральна панель містить кнопки дій, серед яких – встановлення нового екземпляра, додавання компонентів, перевірка конфігурації. Саме тут користувач обирає потрібний сценарій інсталяції.

На цьому екрані користувач обирає між новою інсталяцією *SQL Server* або додаванням функцій до наявного екземпляра. Для нової інсталяції буде створено окремий екземпляр (default або named), із власними службами та параметрами. Додавання функцій дозволяє розширити можливості вже існуючого екземпляра без повного перевстановлення.



Якщо планується кілька екземплярів на одній машині, то зазвичай обирають іменований екземпляр, щоб уникнути конфліктів портів і служб. Для простих сценаріїв достатньо екземпляра за замовчуванням (*MSSQLSERVER*), що спрощує підключення за ім’ям хоста.

У нашому випадку обирається нова інсталяція SQL Server.

Наступний етапі майстра інсталяції *Microsoft SQL Server 2022* – налаштування оновлень.

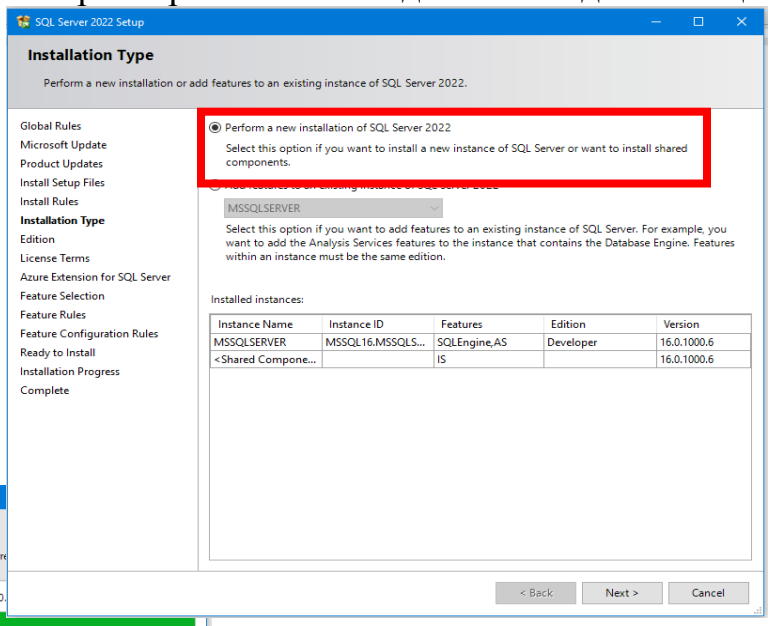
У ньому представлений розділ *Microsoft Update*, який відповідає за перевірку та встановлення важливих оновлень для *Windows* і продуктів *Microsoft*, включаючи *SQL Server 2022*.

Цей крок дозволяє визначити, чи буде *SQL Server* автоматично отримувати та встановлювати важливі оновлення з Microsoft Update під час подальшої роботи.

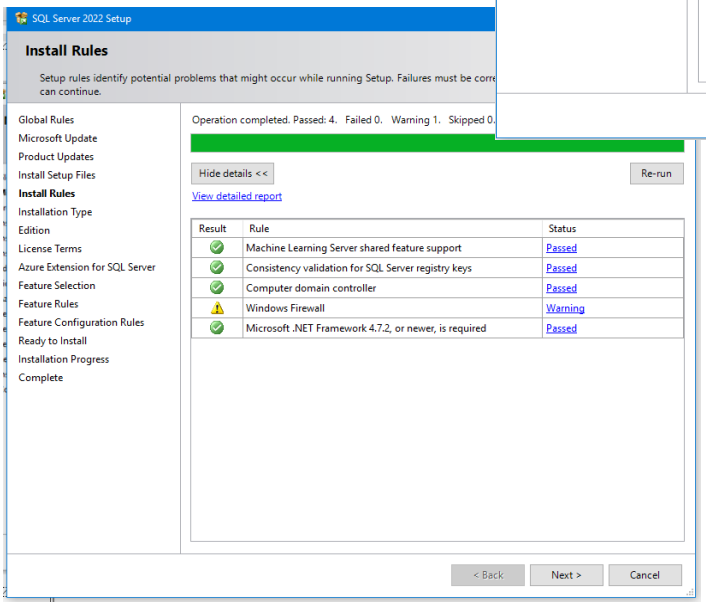
Наступний етап інсталяції називається *Install Rules* і використовується для перевірки системи перед продовженням встановлення. Майстер інсталяції аналізує конфігурацію середовища, щоб визначити можливі проблеми або обмеження, які можуть завадити коректному розгортанню *SQL Server*. На цьому кроці користувач отримує попередню діагностику системи, що дозволяє ще до завершення інсталяції внести необхідні зміни та уникнути критичних помилок.

У центральній частині вікна розміщено *зведений результат перевірки*. Система повідомляє: *Operation completed. Passed: 4. Failed: 0. Warning: 1. Skipped: 0.* Це означає, що чотири правила успішно виконано, жодне не завершилося помилкою, одне має статус "попередження", а перевірок, які було пропущено, немає. Така структура дозволяє користувачеві швидко оцінити загальний стан і зрозуміти, чи можна переходити до наступних етапів. Отже, за позитивного результату перевірки натискається кнопка *Next*.

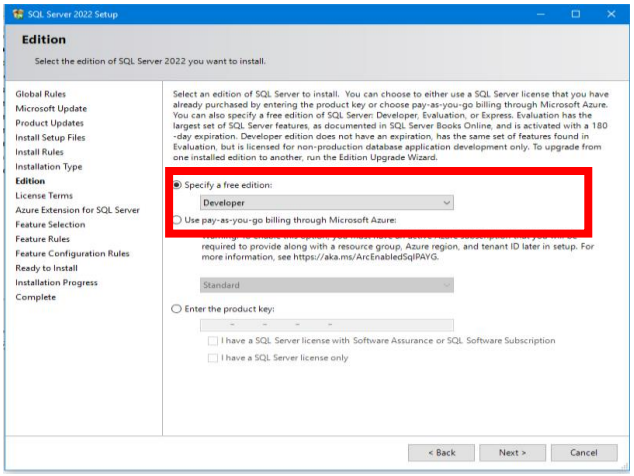
Наступний етап інсталяції *Microsoft SQL Server 2022* має назву *Installation Type* і призначений для вибору типу встановлення. Майстер пропонує два варіанти: створити новий екземпляр SQL Server або додати компоненти до вже існуючої інсталяції. Вибір цього параметра визначає подальший хід інсталяції та конфігурації серверної системи.



У верхній частині вікна відмічено пункт *Perform a new installation of SQL Server 2022*. Цей варіант обирається у випадку, якщо користувач хоче створити абсолютно новий екземпляр SQL Server або інсталювати спільні компоненти. Це типовий сценарій при першій установці продукту на комп'ютер, оскільки дозволяє розгорнути нову інфраструктуру для роботи з базами даних.



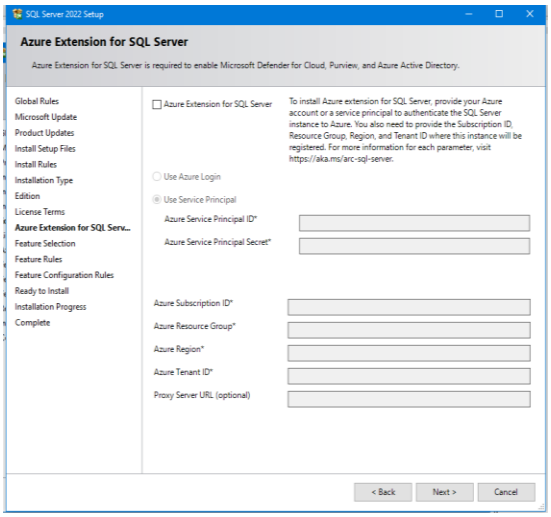
Другий варіант – *Add features to an existing instance of SQL Server*, використовується у



випадках, коли *SQL Server* уже встановлений, але виникає потреба додати нові функції, наприклад, аналітичні служби або інші компоненти, що розширюють функціонал. Для цього користувач повинен вибрати конкретний екземпляр із переліку доступних інсталяцій.

У нижній частині вікна показано таблицю з відомостями про вже встановлені екземпляри. Це дозволяє користувачеві оцінити поточний стан системи перед тим, як приймати рішення про нову інсталяцію або розширення наявної.

У наступному вікні майстра встановлення *Microsoft SQL Server 2022* на етапі вибору редакції продукту. У верхній частині вікна розміщений опис, де пояснюється, що користувач може вибрати безкоштовну редакцію, наприклад *Developer* або *Evaluation*, або ж ввести ключ продукту для комерційних версій. Також вказано можливість використання оплати через Microsoft Azure за моделлю pay-as-you-go.

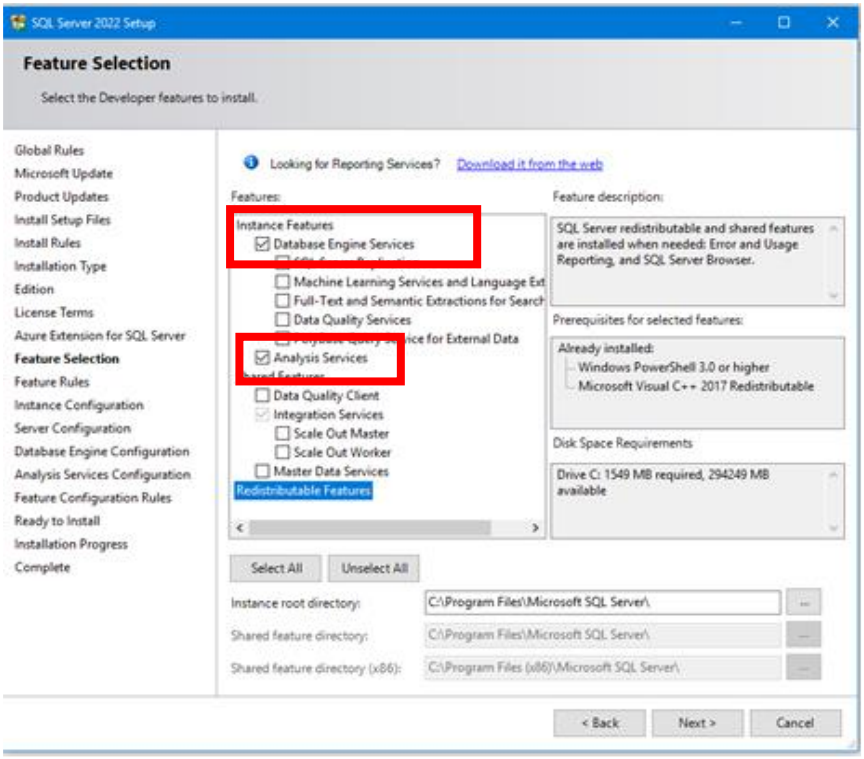


У даному випадку обирається безкоштовна версія *Developer*.

Після натиску по кнопці *Next* з'являється діалогове вікно з параметрами налаштування *Azure Extension for SQL Server*. Цей модуль дає змогу інтегрувати *SQL Server з хмарними* сервісами Microsoft Azure, такими як *Microsoft Defender for Cloud*, *Purview* та *Azure Active Directory*. Тут передбачено два методи автентифікації: через *Azure Login* або використовуючи *Service Principal* із вказанням ідентифікаторів та параметрів доступу до ресурсів у хмарі.

У наведеному прикладі опція *Azure Extension for SQL Server* не була обрана. Це означає, що *SQL Server* встановлюватиметься у локальному режимі без інтеграції з *Azure*, а всі наведені поля залишаються неактивними для користувача. У нижній частині вікна знову доступні кнопки *Back*, *Next* та *Cancel*, які дозволяють повернутися до попереднього кроку, перейти далі або зупинити встановлення.

На етапі вибору компонентів конфігурації *Microsoft SQL Server 2022* обрано лише декілька основних компонентів системи, які позначені галочками. Це ті функції, без яких робота серверної частини бази даних буде неможливою.



Розглянемо їх детальніше.

Перший компонент *Database Engine Services*. Це ядро SQL Server, яке забезпечує збереження, обробку та захист даних. Саме воно відповідає за виконання транзакцій, управління базами даних, підтримку індексів, процедур і тригерів, а також за роботу з T-SQL запитами. Без цього модуля SQL Server не може функціонувати, оскільки всі інші служби лише доповнюють його можливості.

Другій компонент *Analysis Services (SSAS)* – сервер аналітичних моделей і семантичного шару.

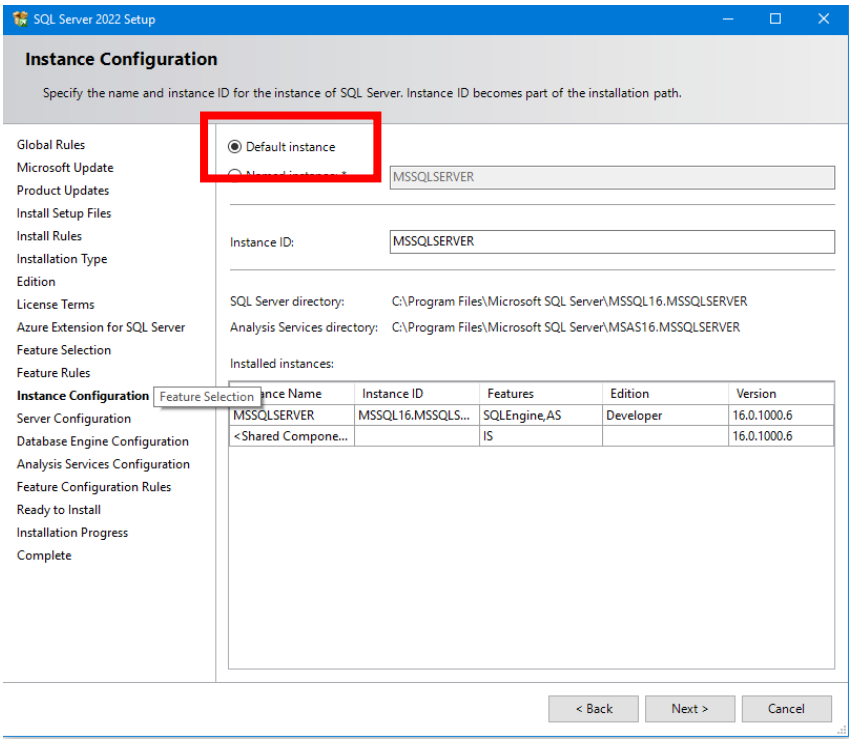
У діалоговому вікні *Instance Configuration* майстра інсталяції *Microsoft SQL Server 2022* користувач визначає конфігурацію екземпляра *SQL Server*:

- *Default instance* – екземпляр за замовчуванням (ім'я *MSSQLSERVER*). У такому випадку до сервера можна підключатися без зазначення імені екземпляра сервера.

Використовується найчастіше в середовищах розробки, тестування або у випадках, коли потрібна одна центральна інсталяція *SQL Server*. Перевага даного з'єднання з сервером – простота підключення (користувачу достатньо вказати лише назву сервера або *localhost*).

Недоліком є те, що на одному комп'ютері може бути лише один екземпляр за замовчуванням, усі інші екземпляри мають бути іменованими (на комп'ютері може бути установлено декілька екземплярів *SQL Server)*.

- *Named instance* – іменований екземпляр, користувач може задати власне ім'я, наприклад SQL2022Dev.

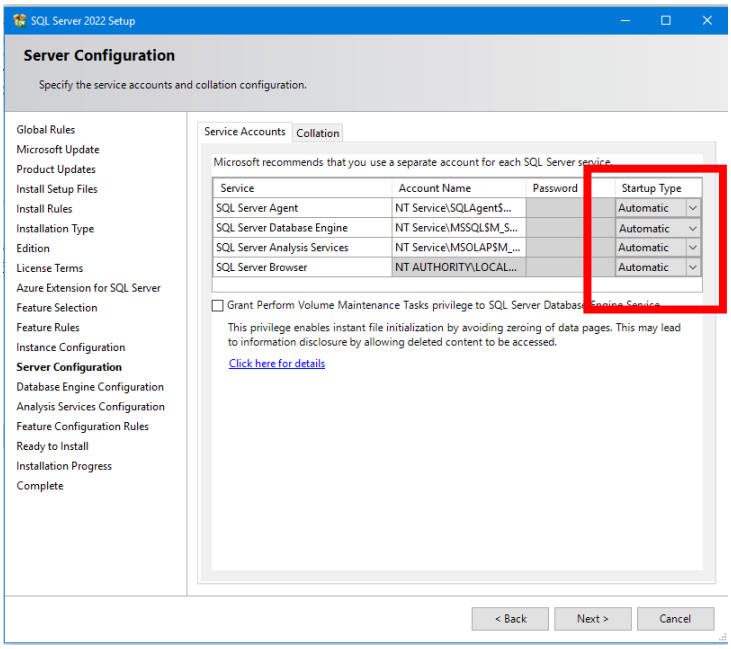


На цьому кроці обрано *Default instance*, тобто сервер буде встановлений із іменем за замовчуванням *MSSQLSERVER*. Це найпростіший варіант, рекомендований у більшості випадків, коли на одному комп'ютері працює лише один екземпляр SQL Server.

У даному вікні наявні додаткові параметри:

- *Instance ID* – ідентифікатор екземпляру серверу (для default instance збігається з *MSSQLSERVER*).
- *SQL Server directory* – шлях до папки, де будуть розміщені файли *SQL Server* (за замовченням це *C:\Program Files\Microsoft SQL Server\MSSQL16.MSSQLSERVER*).
- *Analysis Services directory* – шлях до папки, де будуть розміщені файли служби *Analysis Services* (за замовченням це *C:\Program Files\Microsoft SQL Server\MSAS16.MSSQLSERVER*).
- *Features* – перелік компонентів, які було обрано під час інсталяції. У нашому випадку це: *SQLEngine* – ядро SQL Server (Database Engine), основний компонент для зберігання, обробки та керування даними та служба *Analysis Services*, що використовується для багатовимірного аналізу і роботи з OLAP-кубами або табличними моделями.
- *Edition* – версія SQL Server, в даному випадку встановлюється повнофункціональна версія *Developer Edition*, яка має ті самі можливості, що й *Enterprise Edition*, але призначена лише для навчання, розробки та тестування (не для комерційного використання у продуктивних системах).
- *Version* – версія сервера, вказано 16.0.1000.6, що відповідає *SQL Server 2022* (оскільки 16.x – це версія саме для цієї редакції).

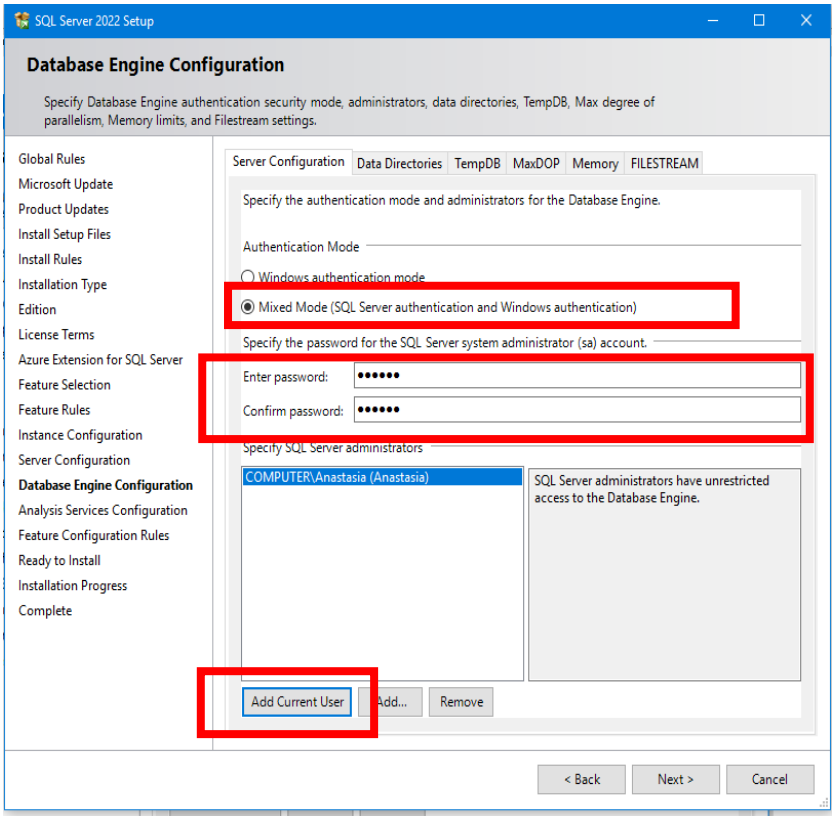
У вікні інсталяції *Server Configuration* відбувається налаштування служб, які відповідають за роботу встановлених компонентів *SQL Server*. У правій



частині відображено перелік служб із призначеними для них обліковими записами Windows та типами запуску. Всі служби мають встановлений режим *Automatic*, що означає їх автоматичний запуск разом із системою для забезпечення безперервної роботи обраних під час інсталяції компонентів.

Таким чином, у цьому вікні виконується призначення служб та способи їхнього запуску, що визначає, як SQL Server буде функціонувати після встановлення.

Вікно інсталяції *Database Engine Configuration* призначене для налаштування основного ядра SQL Server (Database Engine), яке відповідає за збереження, обробку та керування даними. У цьому вікні виконується вибір режиму автентифікації.



У правій частині вікна представлені ключові параметри:

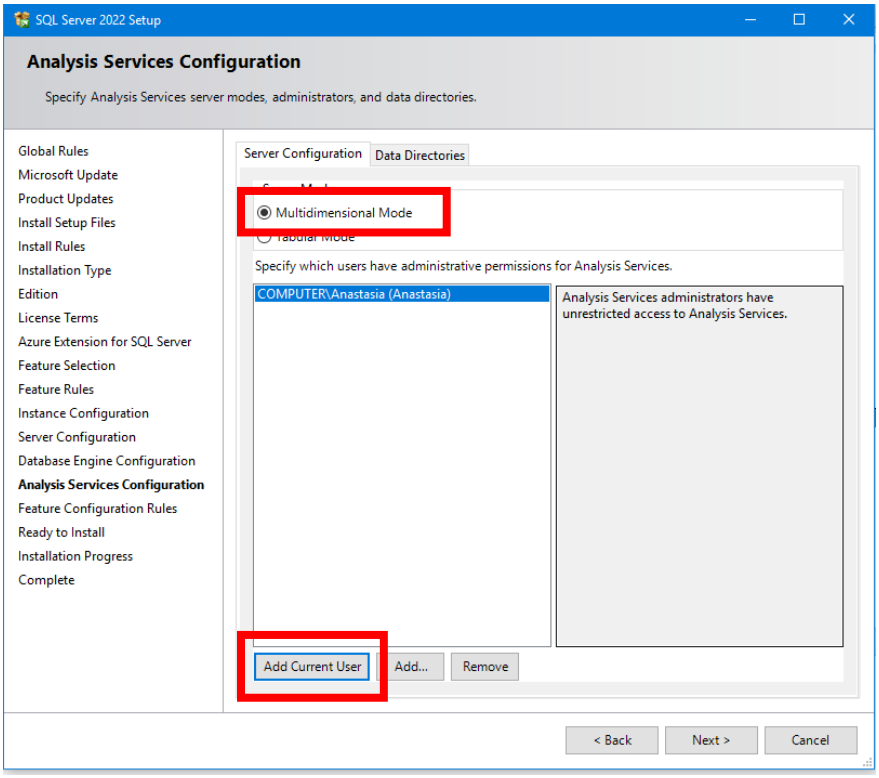
1. *Режим аутентифікації (Authentication Mode)* користувач має одбати один з двох варіантів або *Windows authentication mode* (автентифікація здійснюється лише через облікові записи Windows), або *Mixed Mode (SQL Server authentication and Windows authentication)* – комбінований режим, який дозволяє

використовувати і облікові записи *Windows*, і спеціальні облікові записи *SQL Server*. У даному випадку обрано *Mixed Mode*, що забезпечує більшу гнучкість у підключенні до сервера. При обранні режиму аутентифікації *SQL Server authentication and Windows authentication* користувачеві необхідно ввести пароль для вбудованого адміністратора «sa» (*SQL Server System Administrator Account*). Тобто У полях *Enter password* та *Confirm password* задається пароль для вбудованого адміністративного облікового запису sa, з метою забезпечення безпечного доступу до SQL Server при використанні автентифікації SQL.

2. Кнопки *Add Current User, Add…, Remove* дозволяють відповідно додати поточного користувача, додати інших користувачів чи групи, або видалити зайві записи зі списку адміністраторів. У даному випадку було додано поточного користувача (*Current User).*

Діалогове вікно інсталяції *Analysis Services Configuration* призначений для налаштування служб аналізу, включно з вибором режиму роботи, призначенням адміністраторів та вказанням каталогів даних. Вікно складається з двох вкладок: *Server Configuration* (поточна) та *Data Directories*, де виконується ключове налаштування серверних параметрів.

У верхній частині вкладки обрано параметр *Multidimensional Mode*, який визначає режим роботи служб аналізу. Цей режим дозволяє створювати та обробляти багатовимірні моделі даних (OLAP-куби) для складної аналітики, зберігання агрегованих показників і швидкого виконання запитів. Альтернативним варіантом є *Tabular Mode*, орієнтований на використання табличних моделей.



У центральній частині вікна розташовано список користувачів, які матимуть адміністративні права на службу *Analysis Services*. Тут також виділена кнопка *Add Current User*, яка автоматично додає обліковий запис поточного

користувача Windows до списку адміністраторів. Це забезпечує доступ до конфігурації та керування службою без потреби у додаткових налаштуваннях безпеки після встановлення.

Вкладка *Data Directories* у вікні *Analysis Services Configuration* відповідає за налаштування розташування каталогів, у яких зберігатимуться дані та журнали роботи служб аналізу. Правильне визначення цих шляхів впливає на продуктивність, надійність і масштабованість системи, особливо якщо SQL Server обробляє великі обсяги даних або використовується в середовищах з високим навантаженням.

У цій вкладці передбачено кілька основних параметрів для конфігурації шляхів:

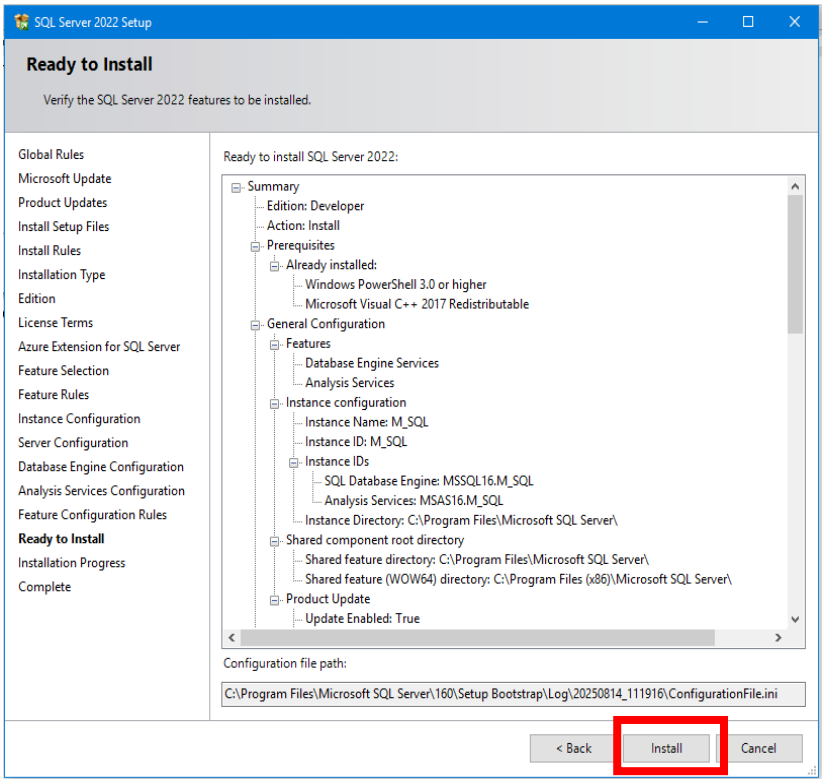
- *Data root directory* – головний каталог, у якому зберігатимуться всі інші підкаталоги з даними та журналами.
- *Data directory* – місце для збереження файлів даних служб аналізу.
- *Log directory* – каталог для файлів журналів, що містять інформацію про роботу сервера та виконання запитів.
- *Temp directory* – використовується для збереження тимчасових файлів під час обробки великих обсягів даних.
- *Backup directory* – призначений для резервних копій, що забезпечують відновлення даних у разі збоїв.

За замовчуванням система пропонує стандартні шляхи в директорії встановлення *SQL Server*, однак адміністратор може змінити їх, наприклад, розташувати дані на окремому високопродуктивному диску, а журнали та резервні копії – на іншому сховищі для підвищення надійності та швидкодії. Після внесення змін користувач натискає *Next* для переходу до наступного кроку інсталяції.

Етап *Ready to Install* є фінальним перед фактичним запуском процесу інсталяції *Microsoft SQL Server 2022.* У цьому вікні майстер надає підсумкову інформацію про всі вибрані параметри, компоненти та конфігураційні опції, що будуть застосовані під час встановлення. Це дозволяє адміністратору ще раз перевірити правильність налаштувань, щоб уникнути помилок перед початком розгортання.

У центральній частині вікна представлено зведений список конфігурацій, в якому відображаються: редакція *SQL Server* (*Developer*), обрані компоненти (*Database Engine Services, Analysis Services*), конфігурація екземпляра з іменами та шляхами розташування файлів, а також параметри оновлення продукту. Додатково вказано, що під час інсталяції використовуватиметься середовище, яке підтримує *Windows PowerShell 3.0* і вище, а також бібліотеку *Microsoft Visual C++ 2017 Redistributable*.

У нижній частині вікна наведено *Configuration file path*, який показує шлях до файлу конфігурації, створеного майстром. Цей файл зберігає всі параметри встановлення та може бути використаний для повторної або автоматизованої інсталяції *SQL Server* на інших системах без необхідності проходити майстера інсталяції вручну. Це особливо зручно для адміністраторів, які розгортають продукт у корпоративних середовищах.



Натиск по головній кнопці Install запускає процес інсталяції з усіма зазначеними параметрами. Після натискання розпочнеться автоматичне копіювання файлів, налаштування служб і розгортання всіх вибраних функцій, після чого з’явиться фінальне вікно з підтвердженням успішного завершення процесу.

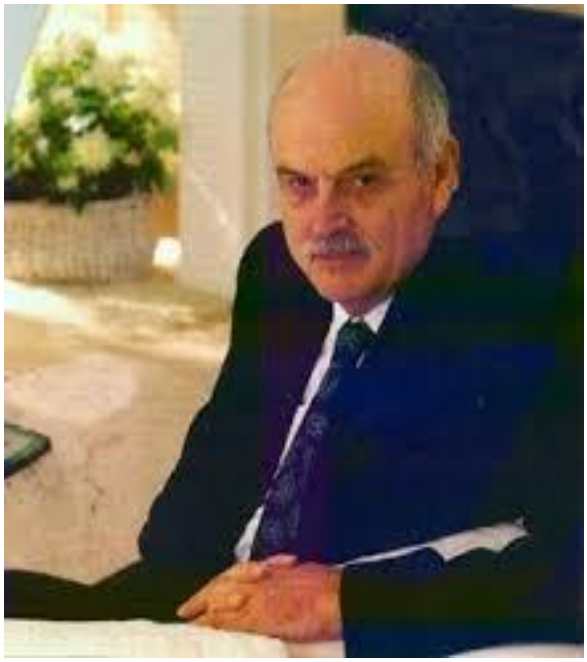
ТЕМА 2. ОСНОВИ РЕЛЯЦІЙНИХ БАЗ ДАНИХ

ПЛАН:

2.1. Реляційний підхід до організації баз даних
2.2. Основні поняття.
2.3. Міжтабличні зв'язки в реляційній базі даних.
2.4. Обмеження цілісності даних.
2.5. Теорія нормалізованих відношень

2.1. Реляційний підхід до організації баз даних

Прийнято вважати, що реляційний підхід до організації баз даних був закладений в кінці 1960-х рр. Едгаром Коддом.



Крім того, переважна більшість наукових досліджень в області баз даних протягом останніх 30 років було присвячено (іноді побічно) саме цієї моделі. Фактично, введення реляційної моделі в 1969 і 1970 роках було, безсумнівно, найбільш важливою подією у всій історії розвитку теорії баз даних. З цих причин, а також з огляду на те, що реляційна модель заснована на певних математичних і логічних принципах і, отже, ідеально підходить для викладу теоретичних концепцій систем баз даних.

Переваги реляційного підходу привели до того, що до кінця 80-х років реляційні системи зайняли на світовому ринку СКБД домінуюче положення. В останні десятиліття цей підхід є найбільш поширеним.

Причина, по якій такі базу даних називають реляційною, полягає в тому, що англійський термін «relation» (відношення), по суті, являє собою загальноприйняту математичну назву для таблиці. Тому на практиці терміни відношення і таблиця можна вважати синонімами. Для уточнення терміну

відношення виділяються поняття заголовка відношення, значення відношення і змінної відношення. Крім того, для пояснення потрібно допоміжне поняття кортежу.

Системи з підтримкою SQL, або просто системи SQL, в даний час стали переважаючими на ринку баз даних, і однією з важливих причин подібного стану справ є саме те, що такі системи засновані на реляційній моделі даних. Тому не дивно, що в неформальному спілкуванні системи SQL часто називають просто реляційними системами.

Як уже зазначалося, користувач реляційної системи бачить дані у вигляді таблиць і ніяк інакше. Користувач нереляційних системи, навпаки, бачить дані, представлені в інших структурах.

Як було описано в попередній темі, реляційна модель поширюється на три принципові аспекти організації даних: структура даних, маніпулювання даними та цілісність даних.

Цікавий факт:

Таблична форма подання відношення була введена з метою популяризації моделі серед непідготовлених користувачів БД. Трактування реляційної теорії на рівні таблиць приховують ряд визначень, важливих для розуміння як теорії реляційних БД, так і мови маніпулювання даними.

У теорії БД розрізняють поняття *таблиці* і *відношення*. Наприклад, у відношенні не обумовлюється порядок стовпців, а тільки їх набір. Ще для відношень вводять певні обмеження, як неможливість мати два зовсім однакові рядки. Можна вважати, що розходження між ними полягає в наступному: відношення – абстрактне представлення інформації про об'єкти предметної області, а таблиця – більш конкретне. Але у першому наближенні у подальшому будемо вважати, що це те саме.

2.2. Основні поняття

Дамо основні означення, які характерні для реляційних баз даних:

Об’єкт елемент інформаційної системи. Об’єкт може бути реальним, наприклад, людина, будь-який предмет або населений пункт, і абстрактною подією, рахунок покупця або курс, що вивчається студентами.

Клас об'єктів	сукупність об'єктів, які мають однаковий набір властивостей. Об'єкти і їхні властивості є поняттями реального світу.
Значення даних	дійсні дані, що містяться в кожному елементі даних. Залежно від того, як елементи даних описують об'єкт, їхні значення можуть бути кількісними, якісними або описовими.

В об'єктному відношенні один (або декілька) з атрибутів однозначно ідентифікують об'єкт. Такий ключовий атрибут називається (одиничним чи множинним) ключем відношень або первинним атрибутом.

Атрибут	інформаційне відображення властивостей об'єкта. Наприклад, студент університету має такі атрибути, як прізвище, ім'я, по батькові, курс, група, факультет, домашня адреса тощо. Атрибут при реалізації інформаційної моделі називають елементом даних, полем даних або просто полем.
Ключовий елемент даних	елемент, за яким можна визначати значення інших елементів даних.
Первинний ключ	атрибут (або група атрибутів), що однозначно ідентифікує кожний рядок у таблиці. Використання первинного ключа є найпростішим засобом запобігання дублювання записів у таблиці. В такому разі він називається складеним ключем.
Складений ключ	первинний ключ, що складається з декількох полів
Зовнішній ключ	поле, значення якого відповідає значенням первинного ключа або частини складеного первинного ключа іншої таблиці, пов'язаної з розглянутою.
Зв'язне відношення	зберігає ключі двох чи більше об'єктних відношень, тобто по ключах встановлюються зв'язки між об'єктами відношень.
Індекс	компактний об'єкт, який містить інформацію про фізичне розташування записів у відповідній таблиці. Цей засіб прискорює пошук та упорядкування (сортування) даних у таблиці.
Запис даних	сукупність значень пов'язаних елементів даних.
Зв'язок	функціональна залежність між об'єктами.

Ключ, як правило, знаходиться у першому стовпці. Інші атрибути функціонально залежать від даного ключа.

Зауваження:

В об'єктному відношенні атрибути не повинні дублюватися. Це основне обмеження в реляційній базі даних для збереження цілісності даних. Зв'язне відношення може мати і інші атрибути, які функціонально залежать від цього зв'язку.

Розглянемо приклад на рис. 2.1. На ньому наведений розклад руху автобусів маршрутом "Київ - Білогородка - Київ". Бачимо певну структуру. Кожен включений у розклад рейс має свій номер, час відправлення й час у дорозі. Розклад може бути наведено у вигляді таблиці. Заголовки колонок таблиці називаються атрибутами. Перелік їх імен має назви схеми відношення. Кожен атрибут визначає тип даних, що разом з областю його значень називається доменом. Уся таблиця цілком називається відношенням, а кожен рядок таблиці називається кортежом відношення. Таким чином, відношення можна представити у вигляді двовимірної таблиці.

Підходи до визначення поняття відношення можуть бути різними. Математично відношення може бути визначене як безліч кортежів, що є підмножиною декартового добутку фіксованого числа областей (доменів). У результаті одержуємо, що у кожному кортежі повинне бути однакове число компонентів (атрибутів) і значення кожного з них вибирається з деякого певного домену.

По-перше, атрибути різних відношень можуть бути визначені на одному домені так само, як і атрибути одного відношення. Це дуже важлива обставина, що дозволяє встановлювати зв'язки за значенням між відношеннями.

По-друге, множина математично за своїм визначенням не може мати елементи, що збігаються, і, отже, кортежі у відношенні можна розрізнити лише за значенням їх компонентів. Це теж є дуже важливим для моделі: ніякі два кортежі не можуть мати компонентів, що повністю збігаються. *Таким чином, у реляційній моделі повністю виключається дублювання даних про сутності реального світу.*

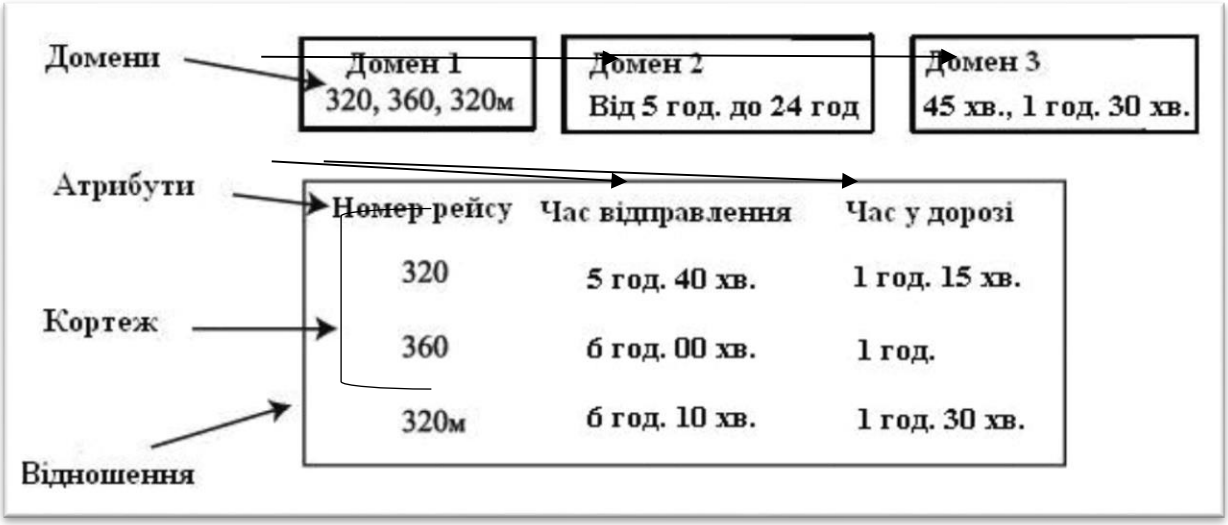


Рис.2.1. Розклад руху автобусів як відношення

По-третє, відзначимо, що схема відношення також є множиною, що дозволяє працювати з ними за допомогою теоретико-множинних операцій. Це є важливим моментом для побудови теорії проектування реляційних схем БД.

Відношення

Звичайним життєвим представленням відношення є таблиця, заголовком якої є схема відношення, а рядками – кортежі відношення-екземпляра; в цьому випадку імена атрибутів відповідають іменам стовпців даної таблиці. Тому іноді говорять про «стовпці таблиці», маючи на увазі «атрибути відношення».

Звичайно, це достатньо груба термінологія, оскільки у звичайних таблиць і рядки, і стовпці впорядковані, тоді як атрибути і кортежі відношень є елементами неврегульованих множин.

ВАЖЛИВО:

Умови і обмеження, які накладаються на відношення реляційних баз даних на табличному рівні представлення, можна сформулювати наступним чином:

- ✓ *не може бути однакових первинних ключів, тобто всі рядки (записи) повинні бути унікальними;*
- ✓ *всі рядки повинні мати однакову типову структуру;*

✓ імена стовпців в таблиці повинні бути різними, а значення стовпців повинні бути однотиповими;

✓ значення стовпців повинні бути атомарними, тобто не можуть бути компонентами інших відношень;

✓ повинна зберігатися цілісність для зовнішніх ключів;

✓ порядок розміщення рядків у таблиці неістотний - він впливає лише на швидкість доступу до потрібного рядка.

Файл бази даних може бути індексованим або ні. Наявність індексації означає, що всі записи в файлі баз даних перевпорядковані у відповідності із наперед заданим принципом (алфавітний порядок, порядок зростання чи спадання тощо) по одному з полів.

✍ Тип даних

Значення даних, що зберігаються в реляційній базі даних, є такими, що типізуються, тобто відомий тип кожного значення, що зберігається. Поняття типу даних в реляційній моделі даних повністю відповідає поняттю типу даних в мовах програмування. В сучасних реляційних БД допускається зберігання символьних, числових даних (точних і приблизних), спеціалізованих числових даних (таких, як «гроші»), а також спеціальних «темпоральних» даних (дата, час, часовий інтервал).

✍ Домен

Поняття домена більш специфічно для баз даних, хоч і є аналогії з підтипами в деяких мовах програмування. У загальному вигляді домен визначається шляхом завдання деякого базового типу даних, до якого відносяться елементи домена. З кожним доменом зв'язується ім'я, унікальне серед імен всіх доменів відповідної бази даних.

Найбільш правильним інтуїтивним трактуванням поняття домена є його сприйняття як допустимої потенційної, обмеженої підмножини значень даного типу.

НАПРИКЛАД *деяка домен ІМЕНА в прикладі визначений на базовому типі символьних рядків, але до числа його значень можуть входити тільки ті рядки, які можуть представляти імена (зокрема, для можливості представлення російських імен такі рядки не можуть починатися з м'якого або твердого знаку і не можуть бути довше, наприклад, 20 символів).*

Якщо деякий атрибут відношення визначається на деякому домені, то надалі обмеження домена грає роль обмеження цілісності, що накладається на значення цього атрибуту.

Слід зазначити також семантичне навантаження поняття домена: дані вважаються порівнянними тільки у тому випадку, коли вони відносяться до одного домена.

2.3. Міжтабличні зв'язки в реляційній базі даних

У реляційних базах даних міжтабличні зв’язки забезпечують логічне поєднання даних між різними таблицями. Вони дозволяють організувати інформацію так, щоб уникнути дублювання даних і забезпечити цілісність інформації. Міжтабличні зв’язки в реляційній базі даних здійснюються за допомогою первинного (PK) та зовнішнього (FK) ключів.

У теорії баз даних відомі три варіанти зв'язків (традиційно називаних відношеннями) між двома таблицями:

Відношення «один-до-одного»	При такому відношенні кожного запису, аналізованої таблиці відповідає не більш одного запису в інший пов'язаній таблиці, і навпаки.

Пов'язані в такий спосіб таблиці легко об'єднати в одну, що містить стовпчики всіх таблиць, що об'єднуються. Зв'язки "один-до-одного" часто використовуються для розбивки дуже широких базових таблиць на декілька більш вузьких. Розбивка великої таблиці на декілька маленьких дозволить зменшити час перегляду полів із найбільше важливою інформацією.

Відношення «один-до-багатьох»	При такому відношенні будь-якому запису аналізованої таблиці може відповідати будь-яка кількість записів у зв'язаній таблиці. Зв'язок при цьому встановлюється між первинним ключем аналізуємої таблиці і відповідного зовнішнього ключа зв'язаної таблиці.

Відношення «багато-до-багатьох» У цьому випадку кожного запису однієї зі зв'язаних таблиць може відповідати будь-яка кількість записів іншої таблиці і навпаки.

ВАЖЛИВО:

Реляційні бази даних не дозволяють створювати зв'язку типу «багато-до-багатьох» напряму. При необхідності створити такий зв'язок її реалізують через допоміжні таблиці, ув'язуючи декілька таблиць зв'язками типу «один-до-багатьох», «багато-до-одного».

Міжтабличні зв’язки також забезпечують референційну цілісність (*Referential Integrity*). Це означає, що база даних не дозволяє створювати «висячі» посилання — записи в підлеглій таблиці завжди мають відповідний запис у головній таблиці. Для забезпечення цілісності використовуються правила каскадного оновлення та видалення:

- *CASCADE UPDATE* – зміни первинного ключа автоматично оновлюють зовнішні ключі у підлеглих таблицях.
- *CASCADE DELETE* – видалення запису в головній таблиці призводить до видалення всіх пов’язаних записів у підлеглих таблицях.

Міжтабличні зв’язки є фундаментом реляційної моделі, оскільки вони дозволяють ефективно організовувати дані, мінімізувати надмірність та забезпечувати узгодженість інформації в базі даних.

2.4. Обмеження цілісності даних

Цілісність даних означає загальну точність, повноту та достовірність даних, що зберігаються в базі даних. Це може бути визначено відсутністю варіацій між двома екземплярами або послідовними оновленнями запису, що вказує на відсутність помилок у вашій інформації. Це також відповідає засобам контролю безпеки та цілісності та методам дотримання нормативних вимог.

Цілісність даних у базі даних зберігається за допомогою низки процедур, правил і принципів перевірки помилок, які базуються на попередньо визначених бізнес-правилах. Це критично важливий аспект управління даними, який гарантує, що інформація, яка вноситься у базу даних, є правильною та достовірною.

Щоб досягти цілісності даних, необхідно впроваджувати різні процеси

та технології контролю, які допомагають підтримувати якість даних протягом усього життєвого циклу. Ці заходи включають перевірку даних, очищення даних, інтеграцію даних і захист даних тощо.

Люди часто плутають цілісність даних із безпекою даних або якістю даних. Однак ці три поняття пов'язані, але різні.

Зауваження:

Безпека даних стосується заходів, вжитих для захисту корпоративних даних від зловживання. Це включає використання методів і прийомів, які роблять ваші дані недоступними для небажаних сторін або роблять доступ до вибраних даних для бажаних сторін. Порушення безпеки даних можуть загрожувати існуванню організації. З іншого боку, цілісність даних стосується точності та повноти даних, присутніх у базі даних.

Кінцева мета безпеки даних — захистити ваші дані від зовнішніх або внутрішніх порушень. Таким чином, це один із багатьох аспектів цілісності даних, але він недостатньо великий, щоб врахувати численні процедури, необхідні для збереження вашої інформації без змін протягом тривалого часу. Подібним чином якість даних є ще одним аспектом цілісності даних, хоча й головним.

Якість даних гарантує, що дані, які зберігаються у вашій базі даних, відповідають стандартам і вимогам організації. Іншими словами, він підтримує цілісність бази даних. При цьому він застосовує набір правил до певного або повного набору даних і зберігає його в цільовій базі даних . Більше того, якість даних — це точність даних, яка явно стосується правильності збережених значень. Цілісність даних і точність даних можна зрозуміти, розглядаючи цілісність даних як загальний термін, де точність даних є однією з багатьох категорій.

Цілісність даних у базі даних охоплює всі аспекти якості даних і вдосконалюється шляхом виконання кількох правил і процедур, які контролюють, як інформація вводиться, зберігається, передається тощо.

Обмеження цілісності являє собою набір певних правил, що встановлюють межі допустимих значень даних і зв'язків між ними. Обмеження цілісності в більшості випадків визначаються особливостями предметної області і можуть відноситися до різних об'єктів БД: атрибутам (полям), записам, таблицям, зв'язкам між ними тощо.

Межі допустимих значень даних для полів таблиці можуть бути задані таким чином:

Типом і форматом поля	можна автоматично обмежити довжину слова, що записується в нього, тому що всі букви алфавіту представляються цифровим кодом однієї і тієї ж довжини.
Довжиною поля	можна автоматично задати діапазон значень чисел, що можуть бути занесені в дане поле. Причому додатковими числовими обмеженнями зверху, знизу або з двох сторін можна додатково обмежити цей діапазон.
Недопустимістю порожнього поля в БД	відсутність «нічийних» записів, відсутність записів у яких пропущені якісь атрибути.
Завданням списку припустимих значень атрибута	можна уникнути зайвої різноманітності даних.
Перевірка на унікальність значення поля	дозволяє уникнути записів дублікатів.
Цілісність усієї бази даних	заснована на цілісності даних на рівні окремої таблиці і реляційній цілісності інформації всієї бази даних на рівні міжтабличних зв'язків.

Підтримка реляційної цілісності припускає, що кожне поле зовнішнього ключа зв'язаної таблиці повинно відповідати полю первинного ключа головної таблиці, що пов'язано із запобіганням виконання наступних операцій:

- Додання записів у таблицю з боку «багато» зв'язку типу «один-до-багатьох» без наявності відповідного запису з боку «один».

- 🕮 Видалення записів із боку «один» зв'язку типу «один-до-багатьох» без попереднього видалення усіх відповідних записів із боку «багато».
- 🕮 Видалення або додання запису в одну таблицю зв'язку «один-до-одного» без видалення або додавання відповідного запису в другу таблицю.
- 🕮 Зміна значення поля первинного ключа головної таблиці, від якого залежать записи пов'язаної таблиці.
- 🕮 Зміна поля зовнішнього ключа пов'язаної таблиці на значення, відсутнє в поле первинного ключа головної таблиці.

У кожний момент часу існування бази даних відомості, що містяться в ній, повинні бути повними, несуперечливими і адекватно відображають предметну область. У цьому й полягає її цілісність.

ВАЖЛИВО:

Цілісність бази даних досягається внаслідок введення обмеження цілісності: вказівка діапазону допустимих значень; співвідношення між значеннями даних; обмеження на видалення інформації тощо. Обмеження реалізуються різними засобами СКБД, наприклад, за допомогою декларативних (повідомлених при розробці бази даних її розробником) обмежень цілісності.

Фактори, що впливають на цілісність бази даних:

- ✍ Людські помилки, адже ручне введення даних збільшує ймовірність помилок, дублювання або видалення. Часто введені дані не відповідають протоколу apt, або помилки в ручному введенні можуть поширюватися на виконання процесів, отже, спотворюючи результати.
- ✍ Помилка передачі даних, яка виникає при передачі даних з одного серверу бази даних на інший. Ці помилки зазвичай виникають, коли елемент даних існує в цільовій таблиці, але відсутній у вихідній таблиці в реляційній базі даних.
- ✍ Цілісність даних також може бути порушена через шпигунське програмне забезпечення, зловмисне програмне забезпечення та віруси, які проникають на комп’ютер і змінюють, видаляють або викрадають дані.

Загальні методи перевірки цілісності даних включають:

- Обмеження доступу до даних і заміна дозволів, щоб заборонити змінювати дані несанкціонованими сторонами.
- Перевірка даних на наявність помилок і невідповідностей перед тим, як вони будуть збережені в базі даних або використані для аналізу.
- Створення резервних копії даних.
- Використання журналів, для відстеження дій, пов'язаних з введенням, модифікацією та видаленням даних.
- Проведення систематичних внутрішніх перевірок.

2.5. Теорія нормалізованих відношень.

Ефективність роботи БД залежить від якості реалізованої в неї моделі. У зв'язку з цим методам створення оптимальних моделей приділяється велика увага зі сторони проектувальників.

Теорія нормалізації заснована на тому, що певний набір таблиць, який називається нормалізованим, має кращі властивості при таких операціях із БД як внесення, модифікація і видалення даних, у порівнянні з іншими можливими наборами таблиць, за допомогою яких можуть бути подані ті ж дані.

Нормалізація відношень	це процес декомпозиції відношення на дві (або більше) проекції з метою вилучення аномалій, які пов'язані з її модифікацією, це ітераційний зворотний процес.
Потенційний ключ	підмножина множини атрибутів відношення, яка має властивості унікальності та ненадлишковості.
Властивість унікальності	означає, що у відношенні не має кортежів (рядків) з однаковим значенням визначеного ключа.
Властивість ненадлишковості	означає, що неяка підмножина визначеного ключа не має властивості унікальності.

Визначимо поняття «*Функціональна залежність*»**:**

R – відношення, *X* та *Y* вільні підмножини множини атрибутів відношення *R***,** тоді множина *Y* функціонально залежить від *X* позначається ($X \rightarrow Y$) тоді і тільки тоді, коли кожне значення множини *X* відношення *R* зв'язано у точності з одним значенням множини *Y* відношення *R*.

Детермінант	ліва частина (*X*) символьного запису функціональної залежності ($X \rightarrow Y$).

Залежна частина права частина (*Y*) символьного запису функціональної залежності (*X*→*Y*).

Існує п’ять нормальних форм. При цьому третя нормальна форма уточнюється додатковою нормальною формою Бойса-Кодда (НФБК).

***Перша нормальна форма* (*1НФ або 1NF*).** визначає, що відношення має атрибути тільки з скалярними значеннями, щоб усі таблиці даних не містили даних, які повторюються, у різних рядках, і комірки не містили більше одного значення. Кожна таблиця повинна мати основний ключ: мінімальний набір колонок, які ідентифікують запис.

Атомарність кожен атрибут повинен мати лише одне значення, а не множину значень (комірки не містили більше одного значення).

***Друга нормальна форма* (*2НФ або 2NF*)** визначає необхідність, крім виконання вимог першої нормальної форми, забезпечення можливості однозначного визначення значення кожного неключового поля за допомогою значень первинного ключа. Якщо для однозначного визначення використовується складений первинний ключ, то це правило застосовується до кожного значення з полів, що входять у складений ключ.

***Третя нормальна форма* (*3НФ або 3NF*)** визначає, що відношення знаходиться у 2НФ та неключові атрибути є взаємно незалежними:

- ✓ поля, що не є ключовими, повинні залежати від первинного ключа таблиці і не залежати від усіх інших полів;
- ✓ будь-яке поле, що залежить від основного ключа та від будь-якого іншого поля, має виноситись в окрему таблицю.

Третя нормальна форма не зовсім підходить для відношень з двома (або більше) потенційними складними ключами, які перекриваються (тобто мають, принаймні, один спільний атрибут).

У зв’язку з цим була введена додаткова третя нормальна форма Бойса-Кодда.

Нормальна форма Бойса-Кодда (НФБК або BCNF) визначає, що всі детермінанти у відношенні повинні бути потенційними ключами.

Якщо це правило не виконується, то, щоб привести вказане відношення до НФБК, його слід розділити на два відношення шляхом двох операцій проекції на кожну функціональну залежність, детермінант якої не є потенційним ключем:

1. Проекція без атрибутів залежної частини такої функціональної залежності.
2. Проекція на всі атрибути цієї функціональної залежності.

Зауваження:

Якщо відношення знаходиться в НФБК, то воно автоматично знаходиться і в 3НФ. Дійсно, це відразу випливає з визначення 3НФ.

Четверта нормальна форма (4НФабо 4NF) потребує, аби в схемі баз даних не було нетривіальних багатозначних залежностей множин атрибутів від будь чого, окрім надмножини ключа-кандидата. Відношення R знаходиться в 4НФ, якщо всякий раз, коли існує багатозначна залежність X→Y
(де Y – непорожня множина і вона не є підмножиною X, а XY складається не з всіх атрибутів R), також існує функціональна залежність X→A для будь-якого атрибута A у R.

ЗАУВАЖЕННЯ:

Вважається, що таблиця знаходиться у 4НФ тоді і лише тоді, коли вона знаходиться в НФБК та багатозначні залежності є функціональними залежностями. Четверта нормальна форма усуває небажані структури даних — багатозначні залежності.

ВАЖЛИВО:

Відношення, які знаходяться у 4НФ формуються за допомогою декомпозиції початкового відношення на дві проекції без втрат даних при їх з'єднанні. Але є відношення, які неможливо піддати

декомпозиції на дві проекції без втрат , але які можна піддати декомпозиції на три (або більшу кількість) проекції без втрат.

П'ята нормальна форма (5НФ або 5NF, PJ/NF) вимагає, аби не було нетривіальних залежностей об'єднання, котрі б не витікали із обмежень ключів. Вважається, що таблиця в п'ятій нормальній формі тоді і лише тоді, коли вона знаходиться в 4НФ та кожна залежність об'єднання зумовлена її ключами-кандидатами.

Відношення R знаходиться в п’ятій нормальній формі, тоді і тільки тоді, коли кожна залежність з’єднання (первинні ключі з’єднання проекцій відношення) в відношенні R визначається (виявляються) потенційними ключами відношення R.

Питання для самоперевірки

1. *Скільки існує варіантів зв’язків між таблицями? Дайте характеристику кожному типу відношень.*
2. *У чому полягає цілісність інформації?*
3. *Яким чином можуть бути задані межі допустимих значень даних?*
4. *На чому заснована цілісність усієї бази даних?*
5. *У чому полягає цілісність сутностей і посилань?*
6. *На чому заснована теорія нормалізації?*
7. *Скільки існує нормальних форм? Охарактеризуйте їх.*
8. *Дайте визначення першій нормальній формі.*
9. *Дайте визначення другій нормальній формі.*
10. *Дайте визначення третій нормальній формі.*
11. *Що визначає третя нормальна форма Бойса-Кодда?*
12. *Дайте визначення четвертій нормальній формі.*
13. *Дайте визначення п’ятій нормальній формі.*
14. *Які існують міжтабличні зв'язки в реляційній базі даних? Охарактеризуйте їх.*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 2

на тему: «Нормалізація відношень у базах даних**»**

МЕТА: *закріпити теоретичні знання та набути практичних навичок із застосування алгоритму нормалізації відношень у базах даних. Навчитися виявляти функціональні та мультизначні залежності між атрибутами, усувати надмірність та аномалії оновлення, вставки й видалення даних шляхом декомпозиції відношень до нормальних форм (від 1НФ до 5НФ).*

ОПИС ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

У даному прикладі розглядається облік книг у базі даних. Початкова таблиця зберігає інформацію про авторів та їхні книги, проте вона не відповідає вимогам нормалізації:

Початкова не нормалізована таблиця:

Автор	Найменування книги
Хейли А.	"Аеропорт", "Діагноз", "Готель"
Шекспір В.	"Король Лір", "Ромео і Джульєтта"
Шевченко Т.	"Кобзар"
Драйзер Т.	"Американська трагедія", "Геній", "Сестра Керрі"

Атрибут «Найменування книги» містить множину значень в одній комірці, що порушує першу нормальну форму (1НФ). Це ускладнює обробку даних, пошук окремих книг і внесення змін, оскільки інформація дублюється у вигляді списку всередині однієї клітинки. Для подальшої роботи з даними таблицю необхідно буде привести до нормалізованого вигляду:

Таблиця, приведена до1НФ:

Автор	Найменування книги
Хейли А.	"Аеропорт"

Автор	Найменування книги
Хейли А.	"Діагноз"
Хейли А.	"Готель"
Шекспір В.	"Король Лір"
Шекспір В.	"Ромео і Джульєтта"
Шевченко Т.	"Кобзар"
Драйзер Т.	"Американська трагедія"
Драйзер Т.	"Геній"
Драйзер Т.	"Сестра Керрі"

Після приведення таблиці до 1НФ усі багатозначні атрибути були розбиті на окремі записи. Кожен рядок таблиці тепер містить тільки одне значення атрибута «Найменування книги», що забезпечує унікальність та атомарність даних.

Далі її необхідно привести до другої нормальної форми. Для цього спочатку потрібно визначити **первинний ключ**. У даному випадку оптимальним вибором є введення **унікального ідентифікатора книги (ID книги)**. Це дозволяє зробити ключ простим і забезпечити однозначну ідентифікацію кожного запису. В результаті отримуємо відношення, у якому кожен запис унікально ідентифікується за допомогою ID книги, а всі інші атрибути залежать від цього ключа:

Таблиця, приведена до 2НФ

ID книги	Автор	Найменування книги
1001	Хейли А.	"Аеропорт"
1002	Хейли А.	"Діагноз"
1003	Хейли А.	"Готель"
1004	Шекспір В.	"Король Лір"
1005	Шекспір В.	"Ромео і Джульєтта"
1006	Шевченко Т.	"Кобзар"
1007	Драйзер Т.	"Американська трагедія"
1008	Драйзер Т.	"Геній"
1009	Драйзер Т.	"Сестра Керрі"

Для приведення таблиці до третьої нормальної форми (3НФ) необхідно

усунути транзитивні залежності, коли один неключовий атрибут залежить від іншого неключового атрибута, а не безпосередньо від первинного ключа.

У початковій таблиці «ID книги – Автор – Найменування книги» спостерігалася надмірність: ім’я автора повторювалося для кожної книги одного автора. Це створювало ризик аномалій при оновленні та дублювання даних. Для усунути даної аномалії необхідно ввести окреме відношення «Довідник авторів», де кожному автору присвоєно унікальний ID Автора.

У головній таблиці «Довідник книг» замість повного імені автора зберігається лише *ID Автора*, а інші атрибути *ID Книги*» та *Найменування книги* залежать безпосередньо від первинного ключа *ID Книги*.

Відношення «Довідник книг»

ID Книги	ID Автора	Найменування книги
1001	A01	"Аеропорт"
1002	A01	"Діагноз"
1003	A01	"Готель"
1004	A01	"Король Лір"
1005	A01	"Ромео і Джульєтта"
1006	A03	"Кобзар"
1007	A04	"Американська трагедія"
1008	A04	"Геній"
1009	A04	"Сестра Керрі"

Відношення «Довідник авторів»

ID Автора	Автор
A01	Хейли А.
A02	Шекспір В.
A03	Шевченко Т.
A04	Драйзер Т.

В результаті у відношенні «Довідник книг» усунене дублювання імен авторів. Відношення «Довідник авторів» містить унікальні записи про авторів і забезпечує можливість відновлення повної інформації через зв’язок по *ID Автора.*

Таким чином структура двох відношень відповідає 3НФ: кожен неключовий атрибут залежить тільки від первинного ключа і транзитивних

залежностей більше немає.

Розглянемо ще один приклад нормалізації віжнощень та приведення до таблиці до 3НФ. Приклад той же: облік продажу книг букіністичним магазином. Здається логічним, щоб у відношення з оперативною інформацією про книги в магазині входили у число інших такі атрибути: «Кількість книг», «Ціна за примірник» та «Загальна вартість книг». Вони не є ключовими. Для одержання значення атрибута «Загальна вартість книг» необхідно добуток атрибутів: «Кількість книг» та «Ціна за примірник», тобто значення атрибута «Загальна вартість книг» залежить від величин двох атрибутів, що не входять до складу первинного ключа. Це суперечить визначенню 3НФ. Щоб дане відношення відповідало третій нормальній формі з нього треба вилучити атрибут «Загальна вартість книг».

Наступним кроком, розглянемо приклад приведення таблиці до нормальної форми Бойса-Кодда (НФБК).

Наприклад, нехай потрібно зберігати дані про книги, в яких ціна за примірник залежить від видавництва. Дані про реалізацію книг можна зберігати в такому відношенні:

Відношення «Ціна примірника у розрізі видавництв»

ID книги	Найменування книги	ID видавництва	Ціна за примірник
1001	"Аеропорт"	V01	200,50
1004	"Король Лір"	V01	215,00
1006	"Кобзар"	V01	202,50
1001	"Аеропорт"	V02	178,00
1004	"Король Лір"	V02	196,50
1006	"Кобзар"	V02	300,00
1001	"Аеропорт"	V03	215,50
1004	"Король Лір"	V03	216,00
1006	"Кобзар"	V03	285,50

Дане відношення містить два потенційних ключа – *ID книги* і *ID видавництва*. Очевидно, що дані зберігаються в відношенні з надмірністю – у разі зміни найменування книги, це найменування потрібно змінити у всіх кортежах, де воно зустрічається. Для усунення такої аномалії за допомогою алгоритму нормалізації, необхідно виявити наявні функціональні залежності:

- *ID книги* ↠ *Найменування книги,* тобто атрибут *Найменування книги* залежить лише від *ID книги*,
- *ID книги*, *ID Видавництва* ↠ *Ціна за примірник*, тобто атрибут *Ціна за примірник* визначається комбінацією (*ID книги*, *ID_Видавництва*).

Тому аномалія даного відносини усувається шляхом декомпозиції його на два наступних відносини і усувається мультизначна залежність:

Відношення «Книги»

ID книги	Найменування книги
1001	"Аеропорт"
1004	"Король Лір"
1006	"Кобзар"

Відношення «Ціна примірника»

ID книги	ID видавництва	Ціна за примірник
1001	V01	200,50
1004	V01	215,00
1006	V01	202,50
1001	V02	178,00
1004	V02	196,50
1006	V02	300,00
1001	V03	215,50
1004	V03	216,00
1006	V03	285,50

Таким чином, ми уникаємо дублювання назв книг у кількох рядках, зберігаємо інформацію в мінімально необхідному вигляді і приводимо відношення до НФБК.

Розглянемо приклад приведення таблиці до четвертої нормальної форми (4НФ).

Наприклад, нехай потрібно зберігати дані про книги, їх перекладачів та мови видання. Для кожної книги може існувати кілька перекладачів та кілька мов, причому ці множини незалежні. У даному випадку наявні мультизначні незалежні залежності.

Відношення «Переклад книг»

ID Книги	Перекладач	Мова
1004	Пантелеймон Куліш	українська
1004	Ганс Р. Коч	німецька
1005	Пантелеймон Куліш	українська
1005	Леся Українка	українська
1006	Віра Річ	англійська
1006	Ганс Р. Коч	немецька

У представленому відношенні є аномалія оновлення, пов'язана з тим, що дублюються прізвища перекладачів і найменування мов перекладу.

У цьому відношенні спостерігається мультизначна залежність:

- книга може бути перекладена кількома перекладачами,
- книга може бути перекладена кількома мовами.

Виникає надмірність: наприклад, для книги 1005 повторюється інформація про українську мову. Це може спричиняти аномалії при оновленні, додаванні або видаленні даних (наприклад, видаливши один рядок, можна втратити інформацію про наявність перекладу певною мовою, навіть якщо інші перекладачі його виконували).

Модифіковане відношення «Переклад книг»

ID Книги	ID Перекладача	ID Мови
1004	T01	L01
1004	T02	L03
1005	T01	L01
1005	T03	L01
1006	T04	L02
1006	T02	L03

Перелік мов

ID Мови	Мова
L01	українська
L02	англійська
L03	немецька

Довідник перекладачів

ID Перекладача	Перекладач
T01	Пантелеймон Куліш
T02	Ганс Р. Коч
T03	Леся Українка
T04	Віра Річ

Отримали три таблиці відношень, які нормалізовані за 4НФ.

Розглянемо приклад приведення таблиці до п’ятої нормальної форми (5НФ).

Наприклад, нехай потрібно зберігати дані про постачання книг у різні філії магазину різними постачальниками. Магазин має таку модель:

- книги постачаються різними постачальниками;
- книги продаються в різних філіях магазину;
- і при цьому кожен постачальник може працювати лише з певними філіями

Якщо все зберігати в одній таблиці, то буде тринарна залежність:

Відношення тринарної залежності «Книги-Постачальники-Магазини»

ID_Книги	***ID_Постачальника***	***ID_Магазину***
1001	P01	M01
1001	P01	M02
1001	P02	M01

Зберігання всіх даних в одному відношенні призводить до надлишкового дублювання і можливих аномалій оновлення, вставки та видалення.

Щоб усунути ці проблеми та привести відношення до п’ятої нормальної форми (5НФ), відношення розкладається на три окремі бінарні таблиці, кожна з яких зберігає одну пару атрибутів:

- таблиця «Книга-Постачальник» – зберігає, які постачальники постачають які книги;
- таблиця «Постачальник-Магазин» – містить інформацію про постачальників, які працюють з якими філіями магазину;
- таблиця «Книга-Магазин» – вміщує дані про книги, які реалізуються у філіях магазину.

Ці три таблиці дозволяють повністю відновити всю інформацію про постачання книг у магазин.

Відношення «Книги_Магазини»

ID_Книги	***ID_Магазину***
1001	M01
1001	M02

Відношення «Книги-Постачальники»

ID_Книги	***ID_Постачальника***
1001	P01
1001	P02

Відношення «Постачальники_Магазини»

ID_Постачальника	***ID_Магазину***
P01	M01
P01	M02
P02	M01

Якщо потрібен прямий зв’язок, то таблиця «Книги-Магазин» будується через JOIN

Таким чином, отримано дкомпозицію відношення на проекції (які мають багатозначні функціональні залежності) та їх з’єднання без втрат даних.

ТЕМА 3. Архітектура Microsoft SQL Server та основи використання мови SQL у реляційних базах даних

ПЛАН:

3.1. Вступ до системи керування базами даних Microsoft SQL Server.
3.2. Архітектура та основні компоненти SQL Server.
3.3. Керування доступом у SQL Server.
3.4. Введення в SQL. Команди SQL.
3.5. Припустимі типи даних.

3.1. Вступ до системи керування базами даних Microsoft SQL Server

Microsoft SQL Server – одна з найпоширеніших у світі систем керування реляційними базами даних (Relational Database Management System, RDBMS), розроблена корпорацією Microsoft. Вона поєднує класичні принципи організації даних у вигляді таблиць із розширеними можливостями для збереження, обробки й аналітики великих обсягів інформації.

Основні характеристики та особливості SQL Server:

- використовується для створення, збереження та керування базами даних у найрізноманітніших сферах: від невеликих корпоративних систем до масштабних високонавантажених застосунків;
- забезпечення надійного та безпечного зберігання даних, надання швидкого доступу до них і зручно роботи з великими обсягами інформації;
- реалізована за допомогою клієнт-серверної архітектури:
  - серверна частина відповідає за обробку запитів, збереження даних, виконання транзакцій, управління безпекою та підтримку механізмів відмовостійкості;
  - клієнтська частина реалізується у вигляді інтерфейсів для користувачів і розробників (наприклад, SQL Server Management Studio), які дозволяють створювати запити мовою SQL, управляти об’єктами бази даних і здійснювати адміністрування;
- підтримує роботу з різними операційними системами (переважно Windows, а також Linux у сучасних версіях), здатен масштабуватися від локальної інсталяції на одному комп’ютері до

використання у великих хмарних інфраструктурах (Azure SQL Database).

Функціональні можливості *Microsoft SQL Server* .

- підтримка транзакцій із гарантіями *ACID* (атомарність, узгодженість, ізольованість, довговічність);
- розширені механізми безпеки: автентифікація користувачів, авторизація на основі ролей, аудит дій;
- потужні засоби для створення резервних копій і відновлення даних;
- інтеграція з мовами програмування (.NET, C#, Python, R тощо);
- підтримка аналітики та бізнес-інтелекту через модулі *SQL Server Analysis Services (SSAS), SQL Server Integration Services (SSIS) та SQL Server Reporting Services (SSRS).*

Використання SQL Server дозволяє централізовано зберігати дані, забезпечувати багатокористувацький доступ, виконувати складні запити та звіти, інтегруватися з іншими програмними продуктами Microsoft (Excel, Power BI, Visual Studio). Це робить його універсальним інструментом як для адміністраторів баз даних, так і для розробників та аналітиків.

3.2. Архітектура та основні компоненти Microsoft SQL Server

Архітектура *Microsoft SQL Server* побудована за модульним принципом і передбачає чітке розмежування функцій між основними компонентами системи, забезпечуючи масштабованість, продуктивність і гнучкість у використанні. Умовно можна виділити декілька ключових рівнів: ядро системи (Database Engine), служби SQL Server, модулі обробки даних, підсистема безпеки та інструменти управління.

✍ Ядро системи – Database Engine

Database Engine є центральним компонентом *Microsoft SQL Server*, який забезпечує основні функції зберігання, обробки та управління даними. Саме цей модуль відповідає за виконання всіх транзакційних і аналітичних операцій, реалізацію мовних конструкцій SQL, управління доступом користувачів і підтримання цілісності даних. Умовно *Database Engine* можна розглядати як «серце» системи, через яке проходять усі запити й операції.

Основою роботи *Database Engine* є реляційна модель даних і передбачає організацію інформації у вигляді таблиць, які складаються з рядків і стовпців. Кожна таблиця зберігає дані певного типу (наприклад, інформацію про клієнтів, товари чи транзакції), а між таблицями можуть встановлюватися

зв'язки (*PRIMARY KEY, FOREIGN KEY*). *Database Engine* також підтримує зберігання індексів, представлень (*VIEWS*), тригерів, процедур і функцій, що дозволяють будувати складні логічні конструкції для роботи з даними.

Коли користувач або застосунок надсилає запит до *SQL Server, Database Engine* виконує його аналіз і оптимізацію. Запит, поданий користувачем або додатком, спочатку надходить у компілятор запитів (*Query Parser*). На цьому етапі здійснюється синтаксичний аналіз (Syntactic Analysis) тобто відбувається перевірка структури SQL-виразу на відповідність правилам мови SQL. Якщо конструкції SQL сформовані некоректно, сервер повертає повідомлення про помилку. Далі виконується семантичний аналіз (*Semantic Analysis*) виконується перевірка правильності використання ідентифікаторів (назв таблиць, стовпців, функцій, операторів), відповідності типів даних та наявності об'єктів у базі даних. Лише після цього формується попереднє внутрішнє подання запиту, яке передається оптимізатору.

Цікавий факт:

Оптимізатор запитів (Query Optimizer) формує оптимальний план виконання, обираючи найбільш ефективні стратегії доступу до даних. Зокрема, він визначає, чи доцільно застосовувати індекси для пошуку рядків, у якій послідовності виконувати операції сортування та які алгоритми з'єднання таблиць (наприклад, Nested Loop Join, Merge Join або Hash Join) використовувати. Такий підхід дозволяє SQL Server мінімізувати витрати ресурсів процесора та пам'яті й забезпечувати стабільно високу продуктивність навіть за роботи з великими обсягами даних.

Однією з ключових особливостей Database Engine є підтримка транзакцій. Транзакція являє собою логічну одиниця роботи з даними, яка виконується повністю або не виконується зовсім. SQL Server реалізує транзакційну модель відповідно до властивостей ACID:

- *Атомарність* (*Atomicity*) усі операції в межах транзакції виконуються як одне ціле.
- *Узгодженість* (*Consistency*) після завершення транзакції дані залишаються у коректному стані.
- *Ізольованість* (*Isolation*) одночасні транзакції не впливають одна на одну.

🕮 *Стійкість / Збереження (Durability)* після підтвердження транзакції (*Commit*) всі зміни остаточно зберігаються в базі даних і не можуть бути втрачені навіть у разі відмови сервера або збою живлення..

Database Engine забезпечує доступ великої кількості користувачів до бази даних одночасно. Для цього використовуються механізми блокувань (*Locking*) і знімків даних (*Snapshot Isolation*). *SQL Server* може застосовувати різні рівні ізоляції транзакцій, починаючи від *Read Uncommitted* (мінімальний рівень контролю) до *Serializable* (максимальна ізоляція). Це дозволяє балансувати між продуктивністю та точністю результатів.

На фізичному рівні Database Engine оперує файлами даних (*.mdf, .ndf*) та журналами транзакцій (*.ldf*). Файли даних містять таблиці, індекси й інші об'єкти бази, тоді як журнали транзакцій забезпечують відновлення цілісності у випадку збоїв. Організація даних у сторінках і розподіл їх у «екстенти» (*Extents*) дозволяє SQL Server оптимально використовувати дисковий простір і прискорювати доступ.

Окрім зберігання даних, Database Engine підтримує виконання збережених процедур, функцій і тригерів. Це дозволяє переносити частину бізнес-логіки безпосередньо в базу даних. Такий підхід зменшує навантаження на клієнтські додатки й уніфікує правила обробки даних.

Ядро SQL Server здатне ефективно працювати як на малих системах, так і у великих корпоративних середовищах. Підтримка розподілених запитів, паралельної обробки даних, реплікації й кластеризації дозволяє масштабувати систему під зростаючі потреби організації.

🖉 Служби SQL Server

Служба — особливий вид програми, яка виконується в системі у фоновому режимі.

Окрім ядра, SQL Server включає набір спеціалізованих служб, кожна з яких виконує окремі завдання:

SQL Server Agent — служба автоматизації, яка дозволяє створювати й виконувати завдання за розкладом (наприклад, резервне копіювання, оновлення статистики, регламентні процедури).

Служба використовується для створення, планування та моніторингу виконання робіт (*Jobs*), які можуть включати резервне копіювання бази,

відновлення даних, запуск скриптів або обробку великих обсягів інформації у визначений час. Таким чином, *Agent* виступає ключовим елементом у побудові автоматизованої інфраструктури адміністрування баз даних.

SQL Server Reporting Services (SSRS)	інструмент для побудови та візуалізації звітів, забезпечує створення, публікацію та керування інтерактивними звітами.

Використовуючи SSRS, користувачі можуть отримувати структуровані аналітичні звіти у вигляді таблиць, графіків або інтерактивних панелей. Це робить *SQL Server* повноцінною платформою не лише для зберігання даних, а й для побудови звітності на всіх рівнях підприємства.

SQL Server Integration Services (SSIS)	служба інтеграції, що використовується для завантаження, трансформації та очищення даних (ETL).

SSIS дозволяє отримувати дані з різноманітних джерел, таких як текстових файлів, XML-документів, сторонніх баз даних, хмарних сервісів, і перетворювати їх відповідно до вимог корпоративного сховища даних чи аналітичних систем.

SQL Server Analysis Services (SSAS)	модуль багатовимірного та табличного аналізу даних (OLAP, Data Mining). Використовується для побудови аналітичних моделей, зокрема багатовимірних кубів та табличних моделей.

SSAS дозволяє реалізовувати складні сценарії бізнес-аналітики, проводити обчислення у реальному часі та підтримувати прийняття управлінських рішень на основі агрегованих показників.

SQL Server Browser	служба, що відповідає за маршрутизацію запитів користувачів до потрібного екземпляра SQL Server у мережі.

Служба *SQL Server Browser* дозволяє організовувати масштабовані та надійні розподілені додатки без необхідності залучення зовнішніх систем чергування. Завдяки Service Broker SQL Server може взаємодіяти з іншими сервісами, забезпечуючи гарантовану доставку повідомлень і збереження порядку їх обробки.

Кожна служба може встановлюватися та використовуватися незалежно, що дозволяє адаптувати SQL Server під конкретні потреби організації.

✍ Модулі обробки та розширені можливості Microsoft SQL Server

Однією з характерних рис *Microsoft SQL Server* є наявність розширюваної архітектури, що включає не лише базові механізми збереження та обробки даних у ядровому компоненті *Database Engine*, але й низку спеціалізованих модулів, які забезпечують розширені функціональні можливості. Ці модулі дають змогу інтегрувати *SQL Server* у складні корпоративні рішення, автоматизувати процеси управління даними, а також підвищувати продуктивність та масштабованість системи.

Модулі обробки та розширені можливості – це функціональні підсистеми ядра *SQL Server* (*Database Engine*), які не є окремими службами *Windows*, але розширюють базову функціональність СКБД. Вони вмикаються чи конфігуруються адміністратором, і працюють безпосередньо в межах ядра бази даних. Серед важливих розширень SQL Server варто виокремити:

Обробка транзакцій у пам'яті (In-Memory OLTP)	технологія, яка дозволяє зберігати таблиці повністю в оперативній пам'яті. Доступ до даних здійснюється надзвичайно швидко завдяки відмові від блокувань і використанню оптимізованих алгоритмів багатопоточної обробки.

Такий підхід зменшує затримки та суттєво підвищує продуктивність транзакційних систем із великим навантаженням.

Повнотекстовий пошук (Full-Text Search)	модуль забезпечує можливість пошуку за текстовими даними з урахуванням морфології, синонімів і складних мовних конструкцій. Він дозволяє виконувати пошукові запити за словоформами, фразами чи префіксами, що значно перевищує можливості стандартного *LIKE*.

Full-Text Search корисний для роботи з документами, описами товарів, статтями чи будь-якими великими текстовими масивами.

Системно-версійовані таблиці (Temporal Tables)	автоматично зберігають історію змін у даних. При кожному оновленні чи видаленні запису система зберігає попередній стан у спеціальній історичній таблиці.

Модуль дозволяє виконувати аналітичні запити «стан даних на певний момент часу», відновлювати попередні версії та реалізовувати аудит без додаткових механізмів.

Реплікація (Replication)	Модуль реплікації дозволяє копіювати й синхронізувати дані між кількома базами чи серверами.

Підтримуються різні типи реплікації:

- *Snapshot* – одноразове копіювання даних.
- *Transactional* – передача змін у режимі реального часу.
- *Merge* – двостороннє об'єднання даних із різних джерел.

Завдяки цьому забезпечується відмовостійкість, балансування навантаження та інтеграція розподілених систем.

Секціонування таблиць (Partitioning)	можливість розділити велику таблицю на кілька логічних частин (секцій), які фізично зберігаються окремо. Наприклад, таблиця з мільярдом рядків може бути розбита за роками чи діапазонами значень, а це в своючергу, підвищує продуктивність запитів, спрощує архівацію й дозволяє ефективно працювати з «великими даними».
Стиснення та шифрування даних (Compression & Encryption)	SQL Server підтримує row/page compression для економії місця та зменшення обсягу введення/виведення.

Також є механізми шифрування:

- *TDE (Transparent Data Encryption)* – шифрування всієї бази.
- *Always Encrypted* – шифрування окремих колонок із захистом на рівні клієнта.
- *Column-Level Encryption* – шифрування даних у конкретних полях.

Дані механізми забезпечують як економію ресурсів, так і захист від несанкціонованого доступу.

PolyBase	модуль, який дає змогу працювати з зовнішніми джерелами даних (*Hadoop, Azure Blob Storage, Oracle, Teradata* та інші БД) так, ніби це звичайні таблиці *SQL Server*. Це спрощує інтеграцію розподілених систем і дозволяє використовувати *T-SQL* для обробки «великих даних» без дублювання чи міграції.
Підсистеми машинного навчання (Machine Learning Services)	SQL Server інтегрує підтримку мов *R* та *Python*, що дозволяє виконувати складну статистичну обробку та побудову моделей машинного навчання прямо у базі. Завдяки цьому скорочується передавання даних між БД та зовнішніми інструментами, а аналітика відбувається безпосередньо на сервері.

Окремо варто відзначити розширені можливості *SQL Server* щодо роботи з ***нереляційними даними***. Система підтримує збереження та обробку XML-структур, JSON-документів, просторових і графових даних. Це значно розширює сферу застосування СКБД, дозволяючи використовувати її у проєктах, пов'язаних із геоінформаційними системами, соціальними мережами, документно-орієнтованими базами тощо.

Завдяки поєднанню ядрових механізмів обробки транзакцій та розширених модулів, *Microsoft SQL Server* перетворюється з класичної реляційної СКБД на комплексну платформу управління та аналітики даних. Кожен модуль має власну сферу застосування, а їх інтеграція дозволяє створювати єдине середовище для зберігання, обробки, аналізу й візуалізації інформації у масштабах великого підприємства.

✍ Підсистема безпеки

Підсистема безпеки *Microsoft SQL Server* являє собою комплекс механізмів, що забезпечують захист даних від несанкціонованого доступу, контроль прав користувачів та аудит дій у системі. Вона побудована на багаторівневій архітектурі, яка охоплює автентифікацію, авторизацію, шифрування, управління сертифікатами, моніторинг безпеки та протидію загрозам. Такий підхід гарантує відповідність вимогам корпоративної та нормативної політики інформаційної безпеки.

На першому рівні функціонує автентифікація (*Authentication*), яка підтверджує особу користувача. *SQL Server* підтримує два основні режими: автентифікацію *Windows* (*Windows Authentication*) та змішаний режим (*Mixed*

Mode), де додатково застосовується автентифікація на рівні *SQL Server*. Це дозволяє гнучко інтегрувати сервер у корпоративні домени та забезпечувати централізоване керування обліковими записами.

Другий рівень реалізує авторизацію (*Authorization*), яка визначає, які дії може виконувати автентифікований користувач. Для цього застосовується рольова модель безпеки (*Role-Based Security*), де права доступу розподіляються через серверні ролі та ролі на рівні бази даних. Такий підхід дає змогу чітко регламентувати доступ до таблиць, подань, збережених процедур та інших об'єктів бази даних.

Важливою складовою є механізми шифрування. *SQL Server* підтримує *Transparent Data Encryption* (*TDE*) для захисту даних «на диску», *Always Encrypted* – для шифрування конфіденційних полів у таблицях, а також протоколи *SSL*/*TLS* для захисту даних «у русі». Завдяки цим інструментам забезпечується конфіденційність інформації навіть у випадках компрометації середовища зберігання даних, або при перехопленні мережевого трафіку.

Додатково функціонує підсистема керування сертифікатами, ключами та обліковими даними (*Credentials*), що дозволяє реалізувати гнучку політику доступу до зовнішніх ресурсів та послуг. Адміністратор може використовувати симетричні та асиметричні ключі, сертифікати, а також інтегрувати *SQL Server* із системами керування ключами (*Key Management Services*).

Не менш значущим компонентом при забезпечення безпеки даних є аудит та моніторинг безпеки. *SQL Server Audit* надає засоби для реєстрації подій доступу до об'єктів, змін у схемах бази та спроб автентифікації. Зібрані дані прописуються в журналі та можуть бути використані для внутрішнього контролю у відповідності з міжнародним стандартам (наприклад, ISO/IEC 27001, GDPR чи HIPAA).

Окрему групу утворюють засоби протидії загрозам. Вбудований механізм *Dynamic Data Masking* приховує чутливі дані від некритичних користувачів, *Row-Level Security* забезпечує контроль доступу на рівні рядків таблиць, а *Data Classification* допомагає ідентифікувати та маркувати конфіденційну інформацію. У сукупності це формує сучасну модель багаторівневого захисту.

Інструменти управління

Інструменти управління *Microsoft SQL Server* становлять комплекс програмних засобів, що забезпечують адміністрування, налаштування, моніторинг і супровід роботи системи управління базами даних. Вони призначені як для початкового розгортання та конфігурації серверного

середовища, так і для його щоденної експлуатації. Наявність розвинених інструментів управління дозволяє знизити адміністративні витрати, підвищити ефективність роботи адміністраторів та забезпечити дотримання корпоративних стандартів обробки даних. До таких інструментів відносяться:

SQL Server Management Studio (SSMS)	інтегроване середовище адміністрування та розробки, надає графічний інтерфейс для створення та модифікації баз даних, управління безпекою, виконання запитів, налагодження збережених процедур і тригерів.

SSMS об'єднує інструменти для управління об'єктами бази, засоби виконання T-SQL-коду та інтерфейс для моніторингу продуктивності. Завдяки цьому адміністратори отримують універсальне рішення для централізованої роботи з усіма компонентами SQL Server.

SQL Server Data Tools (SSDT)	інструмент, що інтегрується з Visual Studio, орієнтований на розробників і дозволяє здійснювати проєктування баз даних, створення пакетів *SQL Server Integration Services (SSIS)*, кубів аналітики в *SQL Server Analysis Services (SSAS)* та звітів для *SQL Server Reporting Services (SSRS)*. *SSDT* підтримує підхід *Database Project*, де база даних розробляється у вигляді коду, що значно спрощує контроль версій та автоматизацію деплойменту.
SQL Server Profiler	інстумент контролю продуктивності, який у поєднанні з інфраструктурою *Extended Events* забезпечує збір і аналіз подій, що відбуваються на сервері, зокрема виконання запитів, блокування, підключення та виявлення помилок. Це дозволяє виявляти вузькі місця продуктивності, оптимізувати SQL-запити та підтримувати високу ефективність роботи системи.
Azure Data Studio	кросплатформенний інструмент для роботи з хмарними сервісами та аналітичними сценаріями, орієнтований на інтерактивний аналіз даних, має підтримку розширень і дозволяє працювати з різними джерелами даних, не обмежуючись *SQL Server*.

Таким чином, інструменти управління *SQL Server* охоплюють увесь життєвий цикл роботи з базами даних: від початкового проєктування та розгортання до щоденного адміністрування, їхньої оптимізації й інтеграції з хмарними платформами.

3.3. Керування доступом у SQL Server

Управління користувачами та правами доступу є ключовим елементом роботи з будь-якою системою керування базами даних, оскільки воно визначає, хто і в якому обсязі може взаємодіяти з інформацією. У *Microsoft SQL Server* реалізовано модель безпеки, яка поєднує автентифікацію користувачів, їх авторизацію та аудит дій. Такий підхід дозволяє гарантувати захист як на рівні операційної системи, так і безпосередньо на рівні бази даних.

Процес доступу до SQL Server на рівні сервера починається з автентифікації **(***Authentication***)**, тобто перевірки достовірності облікових даних користувача. Система підтримує два основні режими автентифікації: *Windows Authentication*, де використовується інтеграція з доменними обліковими записами *Active Directory*, та *SQL Server Authentication*, де користувач і пароль зберігаються у самій СКБД. Використання *Windows*-автентифікації вважається більш захищеним, оскільки воно дозволяє централізовано керувати доступом і політиками безпеки (рис. 3.1).

Після автентифікації відбувається етап авторизації (*Authorization*), коли *SQL Server* визначає, які саме дії дозволено виконувати користувачу. Для цього використовуються логіни (*Logins*), користувачі баз даних (*Database Users*) та ролі (*Roles*).

Логін	обліковий запис на рівні серверу.
Користувач бази даних	сутність, що пов’язує логін із конкретною базою даних.

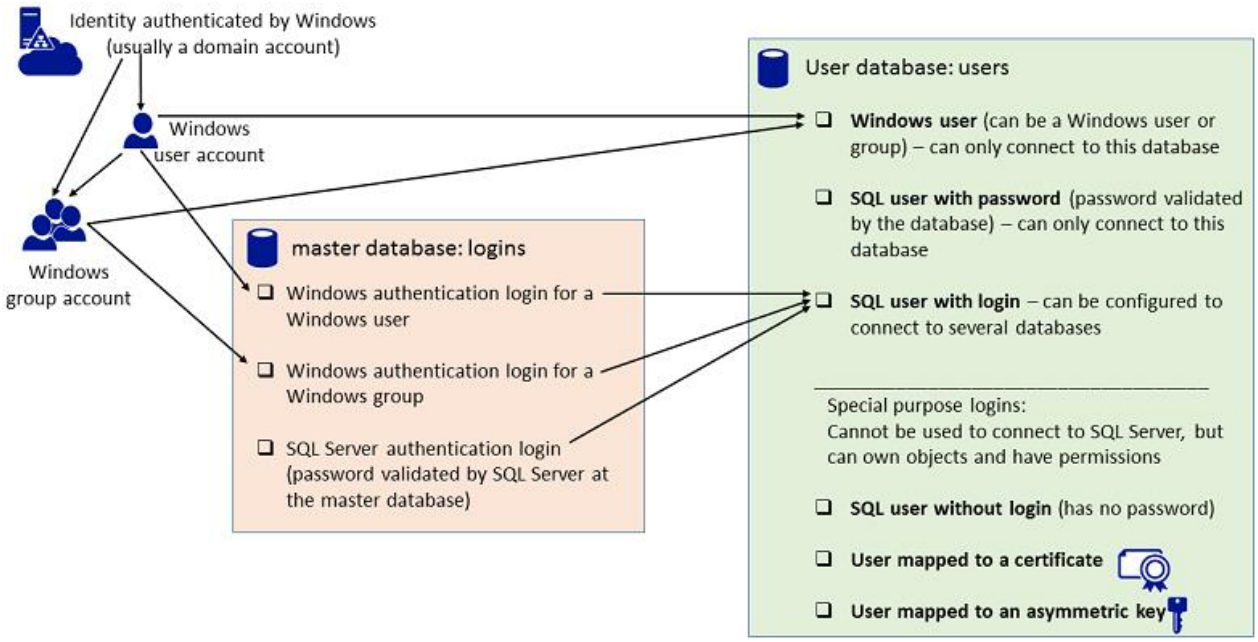


Рис.3.1. Керування доступом

У SQL Server на рівні бази даних існує кілька типів користувачів, кожен з яких має власне призначення та особливості використання. Типи користувачів SQL Server:

Користувач із логіном (SQL user with login)	створюється на основі облікового запису, який має право підключення до екземпляра SQL Server.
Користувач із паролем (User with password)	незалежний користувач, який автентифікується безпосередньо в межах конкретної бази даних.
Користувач без логіна (User without login)	використовується для механізму імперсоналізації, коли збережена процедура або функція виконується від імені іншого користувача.
Користувач, зіставлений із сертифікатом (User mapped to certificate)	забезпечує доступ на основі криптографічної автентифікації.
Користувач, зіставлений з асиметричним ключем	аналогічно використовує механізми шифрування для підтвердження особи.

(User mapped to asymmetric key)	
Користувач Windows (Windows user)	автентифікація здійснюється через облікові записи та групи операційної системи Windows.

Якщо особі чи групі потрібно отримати доступ до кількох баз даних на одному екземплярі *SQL Server*, спочатку створюється логін (*Login*), а вже потім він може бути пов'язаний із конкретними користувачами бази даних. Водночас існують сценарії, коли створюється користувач лише в межах окремої бази даних без логіна (наприклад, для ізольованого доступу до одного сховища).

Важливо розрізняти поняття логіна (*Login*) та користувача бази даних (*Database user*). Логін надає право підключення до екземпляра SQL Server, а користувач бази даних – це відображення цього логіна всередині конкретної бази. Імена логіна та користувача можуть збігатися, але це не є обов'язковим.

Для централізованого управління правами доступу в *SQL Server* застосовуються ролі (*Roles*). Вони виступають аналогом груп у *Windows* і дозволяють групувати користувачів для колективного призначення дозволів. Існують два основні рівні ролей:

- *Фіксовані ролі сервера* (*Fixed Server Roles*) – діють на рівні всього екземпляра *SQL Server* і регулюють адміністрування системи (наприклад, роль sysadmin має повні права на всі дії). В таблиці 3.1 надається список фіксованих ролей екземпляра (сервера) СКБД.
- *Ролі бази даних* (*Database Roles*) – визначають права в межах конкретної бази даних, можуть бути як вбудованими (наприклад, *db_datareader, db_datawriter*), так і користувацькими, створеними адміністратором. В таблиці 3.2 надається список фіксованих ролей на рівні бази даних.

Таблиця 3.1.

Фіксовані ролі екземпляра (серверу) СКБД

Назва фіксованої ролі	***Коротка характеристика***
sysadmin	Виконує всі команди в системі команд СКБД
serveradmin	Виконує конфігурування параметрів екземпляра
setupadmin	Виконує реплікацію та встановлення компонентів екземпляра

securityadmin	Здійснює керування обліковими записями користувачів та їх повноваженнями
processadmin	Здійснює системними процесами
dbcreator	Створює та змінює бази даних
diskadmin	Здійснює керування дисковими файлами

Таблиця 3.2 показує ролі фіксованих баз даних та їх можливості. Ці ролі існують у всіх базах даних. За винятком ролі загальнодоступної бази даних, дозволи, призначені ролей фіксованої бази даних, не можуть бути змінені.

Таблиця 3.2.

Ролі фіксованих баз даних

Ім'я ролі з фіксованою базою даних	***Опис***
1	***2***
db_owner	Учасники фіксованої ролі бази даних db_owner можуть виконувати всі дії з налаштування та обслуговування в базі даних, а також можуть скидати бази даних в SQL Server. (У базі даних SQL і сховищі даних SQL деякі дії з обслуговування вимагають дозволів на рівні сервера і не можуть бути виконані db_owners .)
db_securityadmin	Учасники фіксованої ролі бази даних db_securityadmin можуть змінювати членство в ролях і керувати правами доступу. Додавання принципалів до цієї ролі може дозволити ненавмисну ескалацію привілеїв.
db_accessadmin	Члени фіксованої ролі бази даних db_accessadmin можуть додавати або видаляти доступ до бази даних для входу в систему Windows, груп Windows і входів SQL Server.
db_backupoperator	Члени фіксованої ролі бази даних db_backupoperator можуть створити резервну копію бази даних.
db_ddladmin	Учасники фіксованої ролі бази даних db_ddladmin можуть запускати будь-яку команду Data Definition Language (DDL) у базі даних.

1	*2*
db_datawriter	Члени фіксованої ролі бази даних db_datawriter можуть додавати, видаляти або змінювати дані у всіх таблицях користувачів.
db_datareader	Учасники фіксованої ролі бази даних db_datareader можуть читати всі дані з усіх таблиць користувачів.
db_denydatawriter	Члени фіксованої ролі бази даних db_denydatawriter не можуть додавати, змінювати або видаляти будь-які дані в таблицях користувачів у базі даних.
db_denydatareader	Учасники фіксованої ролі бази даних db_denydatareader не можуть читати будь-які дані в таблицях користувачів у базі даних.

Адміністратор може використовувати вбудовані ролі або створювати власні користувацькі ролі для більш гнучкого контролю доступу.

Окремі права доступу задаються за допомогою операторів *GRANT* (надання доступу), *DENY* (заборона) та *REVOKE* (відкликання). Це дозволяє точково визначати, які саме дії користувач може виконувати щодо конкретних об’єктів бази даних: таблиць, подань, збережених процедур чи функцій. Наприклад, користувачу можна надати право лише на читання даних без можливості їх модифікації чи видалення.

Важливим елементом системи безпеки є аудит дій користувачів. *SQL Server* підтримує журналювання операцій входу, зміни даних і конфігураційних параметрів, що дає змогу відстежувати потенційні інциденти безпеки та забезпечувати відповідність нормативним вимогам, зокрема *GDPR* чи *SOX*.

Спеціальні ролі для бази даних SQL і сховища даних SQL існують тільки в базі даних віртуального майстра. Їх дозволи обмежуються діями, виконаними в *master*. До цих ролей можуть бути додані лише користувачі бази даних майстра. Логіни не можуть бути додані до цих ролей, але користувачі можуть створюватися на основі логінів, а потім ці користувачі можуть бути додані до ролей. До цих ролей можна також додати користувачів баз даних у *master*.

Таблиця 3.3

Спеціальні ролі для бази даних SQL і сховища даних SQL

Назва ролі	***Опис***
Dbmanager	Можна створювати і видаляти бази даних. Член ролі dbmanager, який створює базу даних, стає власником тієї бази даних, яка дозволяє користувачеві

Назва ролі	***Опис***
	підключатися до цієї бази даних як користувач dbo. Користувач dbo має всі права доступу до бази даних в базі даних. Члени dbmanager ролі не обов'язково мають дозвіл на доступ до баз даних, якими вони не володіють.
Loginmanager	Можна створювати та видаляти логіни у віртуальній базі даних.

База даних msdb містить спеціальні ролі, які показані в таблиці 3.4.

Таблиця 3.4.

Ролі msdb

Ім'я ролі msdb	***Опис***
db_ssisadmin, db_ssisoperator,db_ssisltduser	Члени цих ролей баз даних можуть адмініструвати та використовувати SSIS. Екземпляри SQL Server, які оновлюються з попередньої версії, можуть містити стару версію ролі, яку було названо за допомогою служб перетворення даних (DTS) замість SSIS. Для отримання додаткових відомостей див. Ролі служб інтеграції (служба SSIS) .
dc_admin, dc_operato, dc_proxy	Члени цих ролей баз даних можуть адмініструвати і використовувати збирач даних. Для отримання додаткової інформації див. Збір даних .
Роль адміністратора політики	Учасники ролі бази даних db_ PolicyAdministratorRole можуть виконувати всі операції налаштування та обслуговування з політик та умов керування на основі політик. Додаткову інформацію див. У розділі Адміністрування серверів за допомогою керування на основі політики .

Server Group Administrator Role, Server Group Reader Role – члени цих ролей баз даних можуть адмініструвати і використовувати зареєстровані групи серверів.

ВАЖЛИВО:

Члени ролі db_ssisadmin і роль dc_admin можуть підняти свої привілеї до sysadmin. Це підвищення права може відбуватися, оскільки ці ролі можуть змінювати пакети Integration Services і пакети Integration Services можуть виконуватися SQL Server за допомогою контексту безпеки sysadmin агента SQL Server. Щоб запобігти цьому підвищенню привілеїв під час запуску планів технічного обслуговування, наборів даних та інших пакетів служб інтеграції, налаштуйте завдання SQL Server Agent, які запускають пакунки, щоб використовувати проксі-обліковий запис з обмеженими привілеями або додайте лише елементи sysadmin до ролей db_ssisadmin і dc_admin.

MS SQL Server містить чотири системні бази даних: *master, model, msdb, tempdb.*

База даних *master* є головною системною базою даних *SQL Server*. Вона зберігає критично важливу інформацію про інсталяцію сервера, включаючи метадані всіх інших баз даних, інформацію про логіни, користувачів та серверні ролі, параметри конфігурації *SQL Server*, відомості про файли баз даних та шляхи до них, а також інформацію про точки відновлення та налаштування сервера. Без бази *master SQL Server* не може запуститися, оскільки саме вона містить дані про всі інші бази та облікові записи.

База даних *model* слугує шаблоном для створення нових баз даних. Коли адміністратор створює нову базу, *SQL Server* копіює структуру та параметри *model* у нову базу. В *model* можна додавати таблиці, типи даних, збережені процедури та інші об’єкти, щоб вони автоматично включалися у всі новостворені бази. Крім того, вона містить базові налаштування, такі як колація, опції обліку та початковий розмір файлів бази.

База даних *msdb* використовується для роботи внутрішніх механізмів SQL Server, зокрема для планування та автоматизації завдань. Вона містить інформацію про завдання (*Jobs*), операторів (*Operators*) та сповіщення (*Alerts*), керовані через *SQL Server Agent*. Також *msdb* зберігає дані про резервні копії, історію відновлення та виконання процесів *SQL Server Integration Services*

(SSIS). База застосовується для автоматизації адміністративних процедур, моніторингу та планування повторюваних дій.

База даних *tempdb* є тимчасовою робочою базою, яка створюється під час запуску *SQL Server* і очищується при його перезапуску. Вона використовується для зберігання тимчасових таблиць, змінних таблиць, курсорів, проміжних результатів запитів, сортувань та операцій з'єднання таблиць. *tempdb* активно застосовується під час виконання складних запитів і обчислень, які потребують проміжного зберігання даних. Оскільки база перезаписується при кожному старті сервера, її резервне копіювання не є необхідним.

3.4. Введення в SQL. Команди SQL.

SQL (Structured Query Language) – мова структурованих запитів.

SQL це спеціальна мова, що використовується для визначення даних, доступу до даних і їх обробки.

Мова *SQL* відноситься до непроцедурних (nonprocedural) мов, тобто лише описує потрібні компоненти (наприклад, таблиці) і бажані результати, не вказуючи, як саме ці результати повинні бути отримані.

Кожна реалізація *SQL* є надбудовою над процесором бази даних (database engine), який інтерпретує оператори *SQL* і визначає порядок звернення до структур БД для коректного і ефективного формування бажаного результату.

Стандарт *SQL* визначається *ANSI – American National Standarts Institute* (Американським Національним Інститутом Стандартів) і в даний час прийнятий *ISO – International Standarts Organization* (Міжнародною Організацією по Стандартизації).

SQL непроцедурна мова: серверу бази даних повідомляється, що потрібно зробити і яким чином. Для обробки запиту сервер бази даних транслює команди *SQL* у внутрішні процедури. Завдяки тому, що *SQL* приховує деталі обробки даних, його легко використовувати.

За допомогою SQL можна:

- створювати таблиці даних;
- зберігати дані;
- отримувати дані;

- змінювати дані;
- змінювати структуру таблиць;
- об'єднувати дані;
- виконувати обчислення;
- забезпечувати захист даних.

Команди *SQL* поділяються на такі групи:

> ***Команди мови визначення даних – DDL (Data Definition Language).***
>
> Призначені для створення та модифікації структури об’єктів бази даних (таблиць, індексів, схем, користувачів тощо).

До команд мови визначення даних (*Data Definition Language, DDL*) належать *CREATE* (створення), *ALTER* (модифікація) та *DROP* (видалення), які застосовуються до більшості типів об’єктів бази даних — таблиць, подань, процедур, тригерів, табличних областей, користувачів тощо. Тобто існує безліч DDL команд, наприклад:

- *CREATE SCHEMA* – створити схему БД;
- *DROP SCHEMA* – видалити схему БД;
- *CREATE TABLE* – створити таблицю;
- *ALTER TABLE* – змінити таблицю;
- *DROP TABLE* – видалити таблицю;
- *CREATE DOMAIN* – створити домен;
- *ALTER DOMAIN* – змінити домен;
- *DROP DOMAIN* – видалити домен;
- *CREATE COLLATION* – створити послідовність;
- *DROP COLLATION* – видалити послідовність;
- *CREATE VIEW* – створити подання (запит);
- *DROP VIEW* – видалити подання (запит);
- *CREATE PROCEDURE* – створити процедуру;
- *DROP PROCEDURE* – видалити процедуру;
- *CREATE USER* – створити користувача;
- *ALTER USER* – змінити користувача;

🕮 *DROP USER* – видалити користувача.

Деяким здається, що застосування *DDL* є прерогативою адміністраторів бази даних, а оператори *DML* повинні писати розробники, але ці дві мови не так-то просто розділити. Складно організувати ефективний доступ до даних і їх обробку, не розуміючи, які структури доступні і як вони пов'язані. Також складно проектувати відповідні структури, не знаючи, як вони будуть оброблятися.

✍ ***Команди мови управління даними – DCL (Data Control Language).***
Використовуються для керування доступом до бази даних. Вони дозволяють призначати або скасовувати права користувачів на виконання певних дій.

Основними командами *DCL* є *GRANT, REVOKE* та *DENY*.

Команда *GRANT* використовується для надання привілеїв користувачам або ролям у базі даних. Вона дозволяє виконувати певні операції, такі як *SELECT, INSERT, UPDATE, DELETE, EXECUTE* тощо.

Команда *REVOKE* відкликає раніше надані привілеї у користувача або ролі. Це означає, що користувач більше не зможе додавати нові записи у зазначену таблицю, але інші раніше надані дозволи залишаться чинними.

Команда *DENY* використовується для явної заборони доступу до певних операцій, навіть якщо користувач успадковує дозвіл через роль або групу. Якщо користувач отримує дозвіл *DELETE* через роль або групу, команда *DENY* матиме вищий пріоритет і заблокує виконання цієї операції.

Використання команд *GRANT, REVOKE* та *DENY* допомагає налаштовувати гнучку систему доступу до бази даних, захищаючи її від несанкціонованого доступу та помилкових змін.

✍ ***Команди мови управління транзакціями – TCL (Transaction Control Language).***
Використовуються для керування транзакціями в базі даних. Вони дозволяють контролювати виконання змін та визначати, коли ці зміни слід зберегти або скасувати.

Транзакція(або логічна одиниця роботи) неподільна з точки зору впливу на базу даних послідовність операторів маніпулювання даними (читання, видалення, вставки, модифікації) така, що або результати всіх операторів, що входять в транзакцію,

відображаються в БД, або вплив всіх цих операторів повністю відсутня.

Команди управління транзакціями включають в себе:

COMMIT закінчує (підтверджує) поточну транзакцію і робить постійними (зберігає в базі даних) зміни, здійснені цією транзакцією. Також стирає точки збереження цієї транзакції і звільняє її блокування. Можна також використовувати цю команду для того, щоб вручну підтвердити сумнівну розподілену транзакцію.

ROLLBACK виконує відкат транзакції, тобто скасовує всі зміни, зроблені в поточному транзакції. Можна також використовувати цю команду для того, щоб вручну скасувати роботу, виконану сумнівною розподіленої транзакцією.

Поняття транзакції має безпосередній зв'язок з поняттям цілісності бази даних. Дуже часто база даних може мати такі обмеженнями цілісності, які просто неможливо не порушити, виконуючи лише один оператор зміни БД. Наприклад, неможливо прийняти співробітника у відділ, назва і код якого відсутній в базі даних.

У системах з розвиненими засобами обмеження та контролю цілісності кожна транзакція починається при цілісному стані бази даних і повинна залишити цей стан цілісними після свого завершення. Недотримання цієї умови призводить до того, що замість фіксації результатів транзакції відбувається її відкат (тобто замість оператора *COMMIT* виконується оператор *ROLLBACK*), і база даних залишається в такому стані, в якому перебувала до моменту початку транзакції, тобтов цілісному стані.

У зв'язку з властивістю збереження цілісності БД транзакції є придатними одиницями ізольованості користувачів, тобто, якщо з кожним сеансом роботи з базою даних асоціюється транзакція, то кожен користувач починає роботу з узгодженим станом бази даних, тобто з таким станом, в якому база даних могла б знаходитися, навіть якби користувач працював з нею поодинці.

> ✍ ***Команди мови маніпулювання даними – DML (Data Manipulation Language)***.
> Використовуються для роботи з даними в базі даних. Вони дозволяють додавати, змінювати, видаляти та вибирати дані в таблицях, не змінюючи їхню структуру.

До команд мови маніпулювання даними належать:

- *INSERT* – здійснює вставку рядків в таблицю;
- *DELETE* – здійснює видалення рядків з таблиці;
- *UPDATE* – здійснює модифікацію даних в таблиці;
- *SELECT* – здійснює вибірку даних з таблиць за запитом.

Детальніше вказані команди будуть розглядатися у наступних темах.

3.5. Припустимі типи даних.

SQL підтримують три загальних типи даних: строковий, числовий і тип для подання дати й часу. Завдання типу даних визначає значення й довжина даних, а також формат їхнього подання при візуалізації.

Типи даних в SQL Server об'єднані в наступні категорії.

- точні числа;
- приблизні числа;
- дата та час;
- символьні рядки;
- символьні рядки в Юнікоді;
- двійкові дані;
- інші типи даних.

Таблиця 3.5.

Точні числа

Тип даних	***Опис***	***Обсяг пам'яті***
BIGINT	Використовується для зберігання значень, що виходять за діапазон, підтримуваний типом даних *int*. Діапазон від -9223372036854775808 до 9223372036 854775807	8 байт
INT	Основний тип цілочисельних даних в SQL Server. Діапазон від -2147483648 до 2147483647	4 байта
SMALLINT	Межа значень числа в два байти.Діапазон від -32768 до 32767	2 байта
TINYINT	Діапазон від 0 до 255	1 байт
MONEY	Представляє грошові (валютні) значення. Діапазон від -922337203685477,5808 до 922337203685477,5807	8 байт

Тип даних	***Опис***	***Обсяг пам'яті***	
SMALLMONEY	Діапазон від -214 748,3648 до 214 748,3647	4 байта	
BIT	Цілочисельний тип даних, який може приймати значення 1, 0 або *NULL*.	–	
DECIMAL	Числовий тип даних із фіксованою десятковою крапкою. Він дозволяє зберігати числа з точно визначеною точністю та масштабом. DECIMAL(p, s) p (precision) – загальна кількість значущих цифр числа, допустиме значення від 1 до 38, за замовчуванням p = 18. s (scale) – кількість цифр після десяткової крапки. Значення s не може перевищувати p.	**Точ-ність**	**Схови-ще**
		1 - 9	5
		10-19	9
		20-28	13
NUMERIC	Використовується для зберігання грошових значень. *NUMERIC (p, s).* Ідентичний з *DECIMAL*.	29-38	17

Типи даних *MONEY* і *SMALLMONEY* мають точність до однієї десятитисячної грошової одиниці, яку вони представляють.

Щоб відокремити дробові грошові одиниці, наприклад центи, від цілих грошових одиниць, використовуйте кому. Наприклад, 2,15 відповідає 2 доларам і 15 центам.

Типи приблизних числових даних, що використовуються для числових даних з плаваючою комою.

Таблиця 3.6.

Приблизні числа

Тип даних	***Опис***	***Обсяг пам'яті***
FLOAT	*FLOAT (n)* Де *n* - кількість бітів, використовуваних для зберігання мантиси числа в форматі *FLOAT* при експоненційному поданні. Визначає точність даних і розмір для	Залежить від значення *n*

Тип даних	***Опис***	***Обсяг пам'яті***	
	зберігання. Значення параметра *n* повинно лежати в межах від 1 до 53. За замовчуванням параметра n є53. Діапазон від $1{,}18 \cdot 10^{-38}$ до $3{,}34 \cdot 10^{38}$		
REAL	Визначає точність даних Кількість значущих цифр11 .. 12. Діапазон порядку від -39 до 39.	***Точ-ність***	***Обсяг пам'яті***
	значення *n:* 1-24	7 знакі	4 байта
	значення *n:*25-53	15 знакі	8 байт

Таблиця 3.7.

Тип даних дата та час

Тип даних	***Опис***	***Обсяг пам'яті***
DATE()	Дата в форматі РРРР-ММ-ДД. Діапазон від 0001-01-01 до 9999-12-31 От 1 січня 1 року нашої ери до 31 грудня 9999 року нашої ери	3 байта, фіксований
DATETIME()	Визначає дату, що включає час дня з частками секунди в 24-годинному форматі. Дата і час в форматі РРРР-ММ-ДД ГГ:ХХ:СС діапазон дати: 1 січня 1753 року - 31 грудня 9999 року діапазон часу: від 00:00:00 до 23: 59: 590,997	8 байт
TIME()	Визначає час дня. Час без урахування часового поясу в 24-годинному форматі Час в форматі ГГ:ХХ:СС Діапазон: від 00: 00: 00.0000000 до 23: 59: 59.9999999	5 байт

<table>
<tr><th>Тип даних</th><th>Опис</th><th colspan="2">Обсяг пам’яті</th></tr>
<tr><td>DATETIMEOFFSET</td><td>Визначає дату, об'єднану з часом дня, з урахуванням часового поясу в 24-годинному форматі.
Формати РРРР-ММ-ДД ГГ:ХХ:СС
Діапазон дати Від 0001-01-01 до 9999-12-31
Діапазон часу Від 00:00:00 до 23: 59: 59.9999999
Діапазон зміщення часового поясу Від -14: 00 до +14: 00</td><td colspan="2">10 байт</td></tr>
<tr><td>SMALLDATETIME</td><td>Визначає дату, поєднується з часом дня. Час представлено в 24-годинному форматі з секундами, завжди рівними нулю (: 00), без часток секунд.
Діапазон дати: від 01.01.1900 до 06.06.2079
Діапазон часу: від 00:00:00 до 23:59:59
2007-05-09 23:59:59 округляється до 2007-05-10 00:00:00</td><td colspan="2">4 байта</td></tr>
<tr><td>DATETIME2</td><td rowspan="4">Тип даних для зберігання дати та часу з точністю до часток секунди в 24-годинному форматі. Він є розширенням класичного типу DATETIME і відрізняється більш широким діапазоном дат та підвищеною точністю зберігання часу. DATETIME2 дозволяє визначати додаткову точність часу за допомогою параметру (Fractional Seconds Precision, FSP), який вказує, скільки десяткових знаків використовувати для зберігання часток секунди.
Допустиме значення FSP від 0 до 7, де:</td><td>Точ-ність</td><td>Обсяг пам'яті</td></tr>
<tr><td>3 цифри</td><td>6 байт</td></tr>
<tr><td>3 і 4 цифри</td><td>7 байт</td></tr>
<tr><td>інші значення</td><td>8 байт</td></tr>
</table>

Тип даних	*Опис*	*Обсяг пам'яті*	
	0 – секундна точність без дробової частини; 7 – максимальна точність до 100 наносекунд.		

У DATETIME2, визначається користувачем. Приклад визначення стовпця:

MyDate DATETIME2(*3*)

де 3 означає, що частки секунди будуть зберігатися з точністю до тисячних долей секунди (милісекунд). Таким чином, користувач самостійно визначає рівень точності збереженого часу відповідно до потреб додатку та ресурсів зберігання.

Таблиця 3.8.

Символьні рядки

Тип даних	*Опис*	*Обсяг пам'яті*
CHAR	Символьний тип даних фіксованою довжою *CHAR* [(*n*)]. Аргумент n визначає довжину рядка і повинен мати значення від 1 до 8000.	Розмір при зберіганні складає *n* байт.
VARCHAR	Символьний тип даних змінної довжини. *VARCHAR* [(*n* | *MAX*)], де n – максимальна довжина рядка в символах, допустиме значення від 1 до 8000. *MAX* – дозволяє зберігати рядки довжиною до 2^{31}-1 символів (2 ГБ). Тип *VARCHAR* ефективний для рядків, довжина яких може змінюватися, оскільки *SQL Server* виділяє пам'ять лише під фактичну довжину даних.	Фактична довжина введених даних плюс 2 байта
TEXT	Тип даних фіксованої довжини призначений для зберігання символьних і двійкових даних. Може зберігати не більше 65535 символів.	Обсяг не перевищує 2147483647 байт.

Зауваження:

При використанні типу даних CHAR або VARCHAR, рекомендується наступний підхід:

- *якщо розміри записів даних стовпців постійні, використовуйте CHAR;*
- *якщо розміри записів даних стовпців значно змінюються, використовуйте VARCHAR.*
- *якщо розмір може перевищити 8000байт, використовуйте VARCHAR (max).*

Дані в форматі Юнікод представляються символами кодування UNICODE UCS-2.

Таблиця 3.9.

Символьні рядки в Unicode

Тип даних	***Опис***	***Обсяг пам'яті***
NCHAR	Символьний тип даних, що має постійну довжину, містить дані в Unicode і використовує набір символів UCS-2. Аргументи *NCHAR* [(*n*)] Параметр n визначає довжину рядка і повинен мати значення від 1 до 4000.	Розмір складає подвоєне значення *n* в байтах.
NVARCHAR	Символьний тип даних, що має змінну довжину, містить дані в Unicode і використовує набір символів UCS-2. Аргументи *NVARCHAR* [(*n* | *MAX*)] Параметр n визначає довжину рядка і повинен мати значення від 1 до 4000. Значення *MAX* вказує, що максимальний розмір при зберіганні складає 2^{31}-1 байт (2 ГБ).	Розмір сховища в байтах вдвічі більше числа введених символів + 2 байта.
NTEXT	Тип даних змінної довжини призначений для зберігання символьних і двійкових даних в форматі Unicode і інших форматах. Може зберігати не більше 255 символів.	Розмір пам'яті в байтах вдвічі перевищує довжину введеного рядка.

Символьні рядки в Unicode використовуються тоді, коли необхідно зберігати тексти з різними мовами та наборами символів, що виходять за межі стандартного ASCII.

У *SQL Server* для цього застосовуються типи даних *NCHAR*(n) і *NVARCHAR*(n | *MAX*):

- дозволяють коректно зберігати символи кирилиці, китайські ієрогліфи, арабські, японські та інші алфавіти;
- використовуються у багатомовних застосунках, міжнародних вебсервісах та системах, де важлива сумісність із Unicode;
- забезпечують точне відображення та обробку символів без ризику втрати або некоректного кодування.

Таким чином, символьні типи даних у Unicode рекомендовано застосовувати для глобалізованих рішень і баз даних з мультимовними записами.

Таблиця 3.10.

Двійкові дані

Тип даних	***Опис***	***Обсяг пам'яті***
BINARY	Тип двійкових даних фіксованої довжини. Аргументи *BINARY* [(*n*)], де *n* визначає кількість байтів, яку виділяє SQL Server для зберігання значення (допустиме значення від 1 до 8000). Використовується для зберігання невеликих двійкових значень, які завжди мають однаковий розмір, наприклад контрольні суми, ключі шифрування або маленькі бінарні ідентифікатори.	Розмір при зберіганні складає n байт.
VARBINARY	Тип двійкових даних змінної довжини. Аргументи *VARBINARY* [(*n* | *MAX*)], де *n* – максимальна довжина в байтах (1–8000). *MAX* дозволяє зберігати об'єкти до 2^{31}-1 байт (2 ГБ). Ефективний для даних, розмір яких може змінюватися, наприклад файлів, серіалізованих об'єктів або зашифрованих	Розмір – фактична довжина введених даних плюс 2 байта.

	блоків, для файлів, зображень та інших великих бінарних даних.	

В SQL існує ряд визначених системних змінних, які можна використовувати у виразах замість імен колонок і констант. До таких змінних відносять такі:

- *NULL* – для подання невизначених значень;
- *USER* –ім'я користувача, активного в цей момент;
- *SYSDATETIME* – системний поточний час і дата;
- *SYSDATE* – системна поточна дата;
- *SYSTIME* – системний поточний час;
- *SYSTIMEZONE* – часовий пояс, установлений у системі.

Питання для самоперевірки

1. *Перерахуйте основні компоненти SQL Server*
2. *Що зазвичай забезпечують служби?*
3. *В чому полягає призначення модулів обробки та розширених можливостей?*
4. *Охарактеризуйте можливості фіксованих ролей SQL серверу.*
5. *Назвіть спеціальні ролі бази даних SQL.*
6. *Дайте визначення мові SQL.*
7. *Що описує мова SQL?*
8. *Чим визначається стандарт мови SQL?*
9. *На які групи поділяються команди SQL?*
10. *Охарактеризуйте команди мови визначення даних.*
11. *Дайте визначеннякоманди мови управління даних.*
12. *У чому полягає відмінність команди мови управління транзакціями.*
13. *Охарактеризуйте команди мови маніпулювання даними.*
14. *Перерахуйте припустимі типи даних.*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 3

на тему: *«Побудова інформаційної моделі предметної області»*

МЕТА: *познайомитися з основними етапами проектування інформаційної моделі предметної області. Набути практичних навичок щодо побудови концептуальної, логічної та фізичної моделей бази даних предметної області.*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. На основі індивідуального завдання визначити предметну область, для якої будується інформаційна модель.
2. Проаналізувати задачі, які необхідно вирішувати, з використанням побудованої БД.
3. На основі поставлених задач сформувати концептуальну модель предметної області, яку подати у вигляді схеми:
   - ✓ виділити основні об’єкти (сутності);
   - ✓ визначити зв’язки та типи зв’язків між об’єктами.
4. На основі реляційної моделі даних побудувати логічну модель БД, яку також подати у вигляді схеми:
   - ✓ визначити атрибути, первісні та зовнішні ключі об’єктів;
   - ✓ виявити типи зв’язків між атрибутами об’єктів;
   - ✓ перевірити нормалізацію реляційної моделі (1НФ, 2НФ, 3НФ) і, якщо вона ненормалізована, привести її до третьої нормальної форми (3НФ).
5. Виконати описання фізичної моделі для бази даних:
   - ✓ скласти проекти таблиць, які будуть реалізовані в *MS SQL Server Management Studio*;
   - ✓ визначити ідентифікатори полів;
   - ✓ визначити типи полів та їх властивості.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Побудова інформаційної моделі включає такі етапи:

✍ Проектування концептуальної моделі предметної області – збір даних,

визначення основних інформаційних об'єктів і зв'язків між ними.

- Визначення атрибутів і ключів об'єктів.
- Побудова логічної моделі – співставлення кожному об'єктові концептуальної моделі таблиці, що містить відповідні атрибути, і встановлення зв'язків між таблицями логічної моделі за допомогою первинних ключів.
- Нормалізація моделі – приведення моделі до необхідного рівня нормальної форми.
- Фізичний опис моделі – визначення розміщення даних, методів доступу і техніки індексування з урахуванням забезпечення цілісності бази даних.

Розглянемо діяльність книжкового магазину, що займається реалізацією книг середнім або дрібним гуртом. Припустімо, що ціни закупівлі та реалізації книг фіксовані. Реалізація книг відбувається за готівковим або безготівковим розрахунком. Одержимо такі об'єкти інформаційної моделі предметної області:

- «Прайс книг»;
- «Довідник клієнтів»;
- «Замовлення книг».

Між цими об'єктами існують такі логічні зв'язки:

- ✓ одна книга може реалізовуватися неодноразово, тобто проходити по декількох документах;
- ✓ один клієнт (у даній предметній області під клієнтом розглядається юридична особа, тобто організація, установа, заклад освіти, які регулярно поповнюють власний бібліотечний фонд) може неодноразово закуповувати книги.

З огляду на логічні зв'язки між об'єктами обраної предметної області, одержимо таку концептуальну модель цієї предметної області (рис. 1).

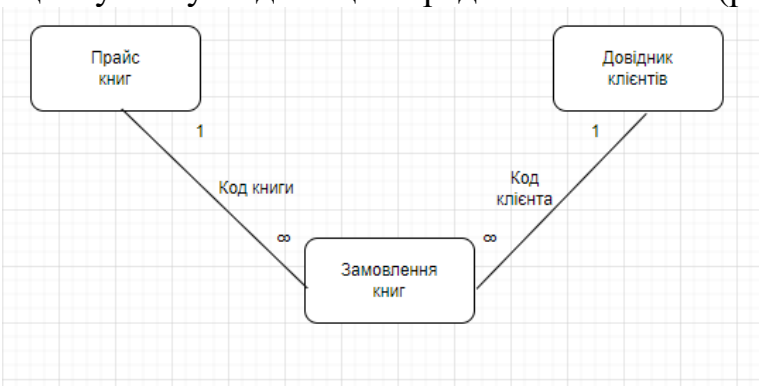


Рис. 1. Концептуальна модель бази даних

Логічна модель бази даних – це абстрактне представлення структури даних, що описує сутності, їхні атрибути та взаємозв’язки без урахування

особливостей конкретної системи керування базами даних (СКБД). Вона є проміжним етапом між концептуальною моделлю, яка фокусується на предметній області, та фізичною моделлю, що враховує технічні параметри реалізації. Основна мета логічної моделі – забезпечити повноту, узгодженість та логічну цілісність даних, зберігаючи при цьому незалежність від конкретної технологічної платформи.

Головними компонентами логічної моделі є сутності, атрибути, ключі та зв’язки між об’єктами предметної області. Сутності відображають основні об'єкти, що підлягають зберіганню у базі даних, наприклад «Прайс книг», «Довідник клієнтів», «Замовлення книг». Кожна сутність характеризується набором атрибутів, які описують її властивості, наприклад, Для таблиці «Довідник клієнтів» будуть такі сутності: *Id_Клієнта, Найменування клієнта, Адреса клієнта, Телефон* тощо. На цьому рівні визначаються як первинні ключі, що забезпечують унікальність записів, так і зовнішні, що встановлюють зв’язки між сутностями.

Особливу увагу у логічній моделі приділяють опису зв’язків між сутностями. Вони можуть бути типу *один-до-одного*, *один-до-багатьох* або *багато-до-багатьох*. Зв'язок типу *багато-до-багатьох* зазвичай реалізуються через додаткові зв’язкові таблиці, які містять зовнішні ключі та допоміжні атрибути. Також на цьому рівні формуються правила цілісності даних, включно з обов’язковістю заповнення певних полів, перевіркою допустимих значень та забезпеченням унікальності. Наприклад, зв’язок *Клієнт – Замовлення* реалізується через додавання в таблицю «Замовлення книг» атрибута *Id Клієнта*, який є зовнішнім ключем і вказує на таблицю «Довідник клієнтів». У випадку відношень «багато-до-багатьох» будується проміжна таблиця «Вміст замовлення», що містить два зовнішні ключі, кожен із яких посилається на відповідні таблиці-сутності.

Важливою складовою логічного проєктування є процес нормалізації таблиць. Він полягає у приведенні структури даних до такої форми, яка усуває дублювання інформації, зменшує ризики аномалій при додаванні, оновленні чи видаленні даних та забезпечує узгодженість бази. Зазвичай нормалізація виконується до третьої нормальної форми (3НФ), але за потреби застосовуються також нормальна форма Бойса-Кодда (НФБК), четверта (4НФ) та п’ята нормальні форми (5НФ).

У нашому випадку нормалізація розпочинається із вихідної сутності «Прайс книг», яка відображає узагальнену інформацію про літературні видання, а саме: назву книги, автора, видавництво, жанр, країну та ціну. На етапі попереднього проєктування така структура може здаватися зручною, проте вона містить значні недоліки, зокрема дублювання даних та порушення принципів цілісності. Наприклад, інформація про автора чи видавництво буде

повторюватися для кожної книги, що ускладнює оновлення та підтримку даних.

Для усунення зазначених проблем застосовується процес нормалізації, що передбачає поступове розділення початкової сутності на окремі логічні таблиці відповідно до принципів нормальних форм. У результаті з єдиної таблиці «Прайс книг» було виокремлено п’ять взаємопов’язаних таблиць: «Прайс книг», «Довідник жанрів», «Довідник видавництв», «Довідник авторів» та «Довідник країн», які будуть пов’язані між собою за допомогою первинних (PK) та зовнішніх ключів (FK). Зокрема, таблиця «Прайс книг» міститиме зовнішні ключі, які посилаються на довідники, де зберігається детальна інформація про автора, країну, видавництво та жанр. Такий підхід дозволяє кожному атрибуту зберігатися в єдиному місці, виключає дублювання та забезпечує можливість легкого оновлення інформації.

Нижче наведено схему логічної моделі предметної області (рис. 2). Ключові поля виділені жирним шрифтом та позначені їх індексація (тобто первинний ключ чи зовнішній ключ).

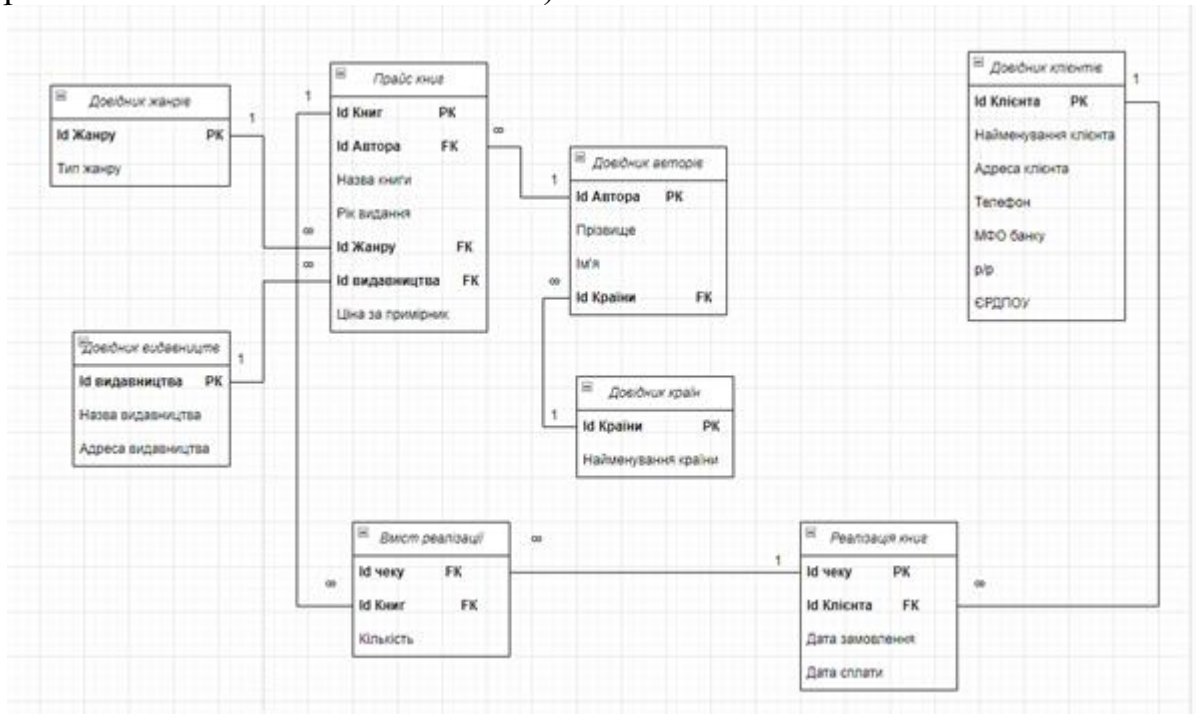


Рис. 2. Логічна модель бази даних

Логічна модель використовується як основа для побудови фізичної моделі бази даних.

Наступним кроком формування інформаційної моделі бази даних – це створення фізичної моделі. Вона є завершальним етапом проєктування, у якому логічна модель перетворюється у фізичні об’єкти SQL Server (таблиці,

індекси, ключі, представлення, секції зберігання). Нагадаємо, що фізична модель бази даних визначає конкретну реалізацію логічної структури даних, оптимізована для зберігання, доступ та обробку даних з урахуванням продуктивності, надійності й масштабованості системи в межах конкретної СКБД.

Мета – створення такої структури бази даних у середовищі MS SQL Server, яка забезпечить цілісність даних, продуктивність обробки запитів, масштабованість та відповідність апаратним і бізнес-вимогам.

У випадку MS SQL Server фізична модель описує, як сутності та їхні атрибути перетворюються на фізичні об’єкти: таблиці, стовпці, ключі, індекси та обмеження. Особливості фізичної моделі для *SQL Server* включають:

1. Типи даних. *MS SQL Server* передбачає використання конкретних типів даних, що відповідають природі інформації та забезпечують ефективне зберігання. Для числових значень застосовуються типи *INT, BIGINT* або *SMALLINT*, а для точних обчислень і фінансових даних *DECIMAL*(*p, s*) або *NUMERIC*(*p, s*). Текстові дані зберігаються у полях *NVARCHAR*(*n*) або *NCHAR*(*n*), що підтримують Юнікод, що є важливим для багатомовних даних. Дати і часові позначки реалізуються через *DATE* та *DATETIME2*, а логічні значення використовуються через *BIT*. Для зберігання двійкових даних (наприклад, сканів або зображень обкладинок книг) застосовуються типи *VARBINARY*(*MAX*).

2. Ключі та обмеження цілісності даних. У фізичній моделі чітко визначаються первинні ключі (*PRIMARY KEY*), які забезпечують унікальність кожного запису у таблиці та зовнішні ключі (*FOREIGN KEY*), які встановлюють зв’язки між таблицями та гарантують цілісність даних у межах бази. Крім того, застосовуються обмеження *NOT NULL*, які не допускають відсутності значень, *CHECK*, що контролюють допустимі значення полів, та *DEFAULT*, які визначають значення за замовчуванням для конкретних колонок.

3. Організація фізичного зберігання даних. MS SQL Server дозволяє оптимально організувати розміщення таблиць і індексів у файлових групах (*filegroups*), що забезпечує балансування навантаження між фізичними носіями та підвищує продуктивність. Для великих обсягів даних може застосовуватися *partitioning* таблиць, що розділяє інформацію за заданими критеріями (наприклад, за датою замовлень або роком видання книг). Така організація зберігання дозволяє спрощувати адміністрування бази та покращує масштабованість системи.

Отже, вказавши, дані якого типу повинні зберігатися в кожному з полів кожної таблиці, отримаємо фізичну модель. Для опису фізичної моделі обраної предметної області розробимо таку структуру для кожної таблиці (розробленої

в логічній моделі) окремо. Структура має такий вид:

Найменування таблиці

Назва поля	Ключ	Індексоване поле	Тип даних	Null-значення

Отже, для кожної таблиці необхідно заповнити даними вищенаведену структуру фізичної моделі у вигляді таблиці (тобто, якщо, наприклад, логічна модель містить 8 таблиць, то й структура фізичної моделі містить теж 8 таблиць).

Для прикладу заповнимо одну таблицю для «Довідник клієнтів». Тобто замість «Найменування таблиці» пишемо «Довідник клієнтів». Вказана таблиця містить сім полів, назву яких зазначмо у полі «Назва поля». Для поля «Id_Клієнта» зазначимо, що воно являється ключовим та визначемо його індексацію – Primary Key (PK – первинний ключ). Далі для нього зазначимо тип даних та відсутність Null-значення.

ВАЖЛИВО:

Всі ключові поля не можуть містити Null-значення, решта – може.

Заповнена структуру фізичної моделі для таблиці «Довідник клієнтів» має такий вид:

Довідник клієнтів

Назва поля	Ключ	Індексоване поле	Тип даних	Null-значення
Id_Клієнта	+	Primary Key	INT	NOT NULL
Найменування клієнта			NVARCHAR(100)	NOT NULL
ЄРДПОУ			CHAR(12)	NOT NULL
Адреса клієнта			NVARCHAR(100)	NULL
Телефон			CHAR(15)	NULL
МФО			INT	NULL
Р/р			CHAR(15)	NULL

Наступним кроком буде заповнення подібним чином решта 5 таблиць

ТЕМА 4. ФАЙЛИ ТА ФАЙЛОВІ ГРУПИ В MICROSOFT SQL SERVER

ПЛАН:

4.1. Основи фізичної організації бази даних у SQL Server.
4.2. Типи файлів та файлові групи.
4.3. Логічні і фізичні параметри файлів та управління дисковим простором.
4.4. Сторінки файлів даних.
4.5. Створення нової бази даних з використанням файлових груп.
4.6. Рекомендації з оптимізації.

4.1. Основи фізичної організації бази даних у SQL Server

На фізичному рівні база даних у *Microsoft SQL Server* являє собою сукупність файлів операційної системи, що включає файли даних та окремі файли журналу транзакцій. Тобто, фізична організація бази даних базується на використанні двох основних типів файлів: файлів даних та файлів журналу транзакцій. Файли даних містять усі об'єкти бази, зокрема таблиці, індекси, представлення та службову інформацію, що забезпечує функціонування системи. Журнал транзакцій призначений для реєстрації всіх змін, виконаних у базі даних, з метою забезпечення її відновлення до узгодженого стану у разі збоїв чи відкату операцій. Розділення даних і журналу є обов’язковим: ці компоненти мають різні профілі доступу (випадкове читання/запис проти послідовного запису), що впливає на продуктивність і відмовостійкість, тому їх зазвичай розміщують на різних носіях. Файли мають визначені типи та структуру і ідентифікуються логічними й фізичними іменами, що дозволяє керувати їх розмірами, зростанням і розподілом дискового простору.

Усі файли бази даних ідентифікуються за логічними та фізичними іменами. Логічне ім’я використовується на рівні системи керування базами даних для посилання на конкретний файл, тоді як фізичне ім’я відповідає розташуванню файлу в операційній системі. Така подвійна ідентифікація забезпечує зручність адміністрування та спрощує перенесення бази даних між різними середовищами. У процесі функціонування бази даних важливе значення має управління розміром файлів та їх зростанням. SQL Server дозволяє встановлювати початковий розмір файлу, граничний обсяг та параметри автоматичного збільшення. Це забезпечує гнучке використання

дискового простору, запобігає перевищенню ресурсів та дає змогу планувати розширення без зупинки роботи системи.

Ще одним елементом фізичної організації є використання файлових груп. Файлова група об'єднує один або кілька файлів даних і дозволяє розподіляти об'єкти бази між різними носіями для досягнення балансу навантаження. Основна файлова група завжди містить первинний файл, тоді як додаткові групи застосовуються для масштабування та підвищення продуктивності. Розподіл даних по файлових групах дає змогу оптимізувати виконання запитів та резервне копіювання. Наприклад, великі таблиці та індекси можна розташовувати в окремих файлових групах для підвищення ефективності операцій читання та запису. Це також дозволяє виконувати резервування та відновлення вибіркових груп, що зменшує час простою системи.

4.2. Типи файлів та файлові групи

У склад бази даних SQL Server входять два принципово різні класи файлів: файли даних і файли журналу транзакцій. Розділення цих класів є обов'язковим: дані та службові структури зберігаються у файлах даних, а послідовність змін фіксується у файлах журналу транзакцій. Таке розділення дозволяє забезпечити відновлення до узгодженого стану та коректну роботу механізмів транзакційності.

Первинний файл даних слугує початковою точкою доступу до вмісту бази даних. У ньому містяться метадані про склад бази та відомості, необхідні для визначення інших файлів, зокрема їхніх логічних і фізичних імен.

Для первинного файла прийнято використовувати розширення *.mdf*. Наявність саме одного первинного файла є вимогою до структури будь-якої бази даних SQL Server.

Вторинні файли даних є необов'язковими, база може не містити жодного такого файла або містити один чи кілька. Вони застосовуються для розширення обсягу збереження, розміщення великих об'єктів на окремих носіях або для організаційного поділу даних між файловими групами.

Для вторинних файлів прийнято розширення *.ndf*. Зміст таблиць, індексів

і інших об’єктів може бути розподілений між кількома вторинними файлами відповідно до політики розміщення.

Файл журналу транзакцій обов’язковий компонентом кожної бази (дозволяється визначати один або більше таких файлів). Файл жрналу не входить до складу жодної файлової групи і має власну модель організації, орієнтовану на послідовний запис записів журналу.

Саме журнал транзакцій забезпечує можливість відкату транзакцій і відновлення після збоїв згідно з обраною моделлю відновлення. Для них прийнято розширення *.ldf*.

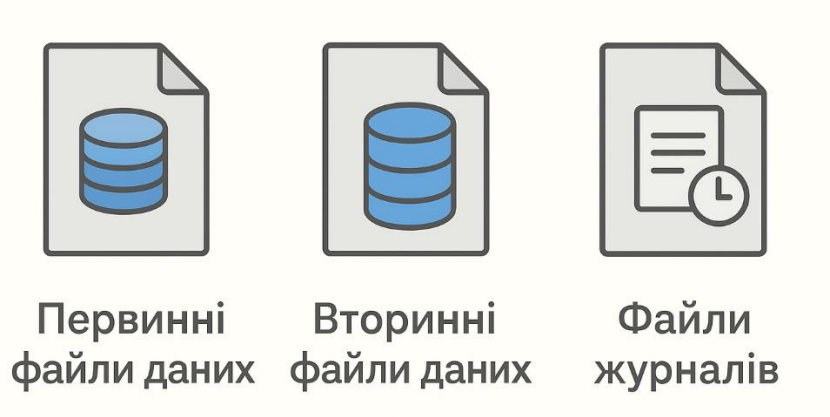


Рис. 4.1. Типи файлів SQL Server

Файлові групи становлять іменовані колекції файлів даних і використовуються для логічного об’єднання та керованого розміщення об’єктів.

Кожен файл даних належить рівно одній файловій групі; файли журналу транзакцій до файлових груп не включаються. Розміщуючи об’єкти «на» конкретній файловій групі, адміністратор може керувати використанням дискового простору та сценаріями резервного копіювання і відновлення.

Первинна файлова група (PRIMARY) існує в кожній базі за замовчуванням і містить первинний файл даних. Системні об’єкти бази даних зберігаються саме в цій групі.

За відсутності іншого налаштування саме *PRIMARY* виконує роль файлової групи за замовчуванням, до якої потрапляють нові об’єкти, якщо під час їх створення не вказано іншу групу. За потреби роль групи за

замовчуванням може бути призначена користувацькій файловій групі.

Користувацькі файлові групи інстумент контролю продуктивності, який у створюються для організаційного поділу даних та керування життєвим циклом великих об'єктів. Об'єкти можна спрямовувати до таких груп під час створення, що дозволяє, наприклад, відокремити історичні дані, виділити простір під специфічні схеми індексації або підготувати окремі групи до резервного копіювання чи відновлення на рівні групи.

У кожен момент часу лише одна файлова група може бути позначена як група за замовчуванням.

Розподіл даних між кількома файлами в одній файловій групі відбувається із застосуванням механізмів пропорційного заповнення. За рівних умов обсягу та параметрів зростання це забезпечує рівномірне використання файлів. Для коректної роботи таких механізмів доцільно підтримувати співмірні початкові розміри та політики автозростання для файлів, що належать одній групі.

ВАЖЛИВО:

Важливою є сукупність обмежень:

- *файл даних не може входити більше ніж до однієї файлової групи;*
- *системні об'єкти бази прив'язані до групи PRIMARY;*
- *файли журнальнів транзакцій завжди керуються окремо та не можуть бути включені до файлових груп;*
- *у базі має існувати рівно один первинний файл даних і щонайменше один файл журналу.*

Дотримання цих вимог забезпечує коректність операцій резервного копіювання, відновлення і керування простором.

Файлова група з ознакою CONTAINS FILESTREAM спеціалізована файлова група в SQL Server, призначена для зберігання великих двійкових об'єктів (*VARBINARY*(*MAX*) із атрибутом *FILESTREAM*) у файловій системі операційної системи замість традиційних 8-КБ сторінок. При

цьому метадані про *FILESTREAM*-об’єкти (посилання, атрибути, транзакційні зв’язки) зберігаються в стандартних сторінках бази даних, а самі бінарні потоки зберігаються як файли/каталоги в заданому *NTFS* каталозі, керованому *SQL Server*.

Файлова група з *CONTAINS FILESTREAM* створюється з вказанням фізичного каталогу (шляху) для *FILESTREAM*-контейнерів у момент визначення файлу/групи, і перед використанням цієї можливості екземпляр *SQL Server* має бути відповідним чином увімкнений для *FILESTREAM*.

Ключові властивості такої файлової групи:

- *FILESTREAM*-дані беруть участь у механізмах транзакційності та входять до складу резервних копій і відновлення бази;
- доступ до цих даних відбувається через файлову систему (що дає переваги при роботі з великими BLOB-об’єктами);
- *FILESTREAM*-об’єкти не зберігаються у звичних сторінках даних і керуються окремими файловими контейнерами, зв’язаними з відповідними записами у базі.

Файлова група з ознакою CONTAINS MEMORY_OPTIMIZED_DATA спеціальний різновид фалової групи, призначений для збереження стійких (*Durable*) об’єктів In-Memory OLTP. У такій групі розміщуються файлові контейнери, що містять контрольні пари файлів (*Checkpoint Files*) даних і дельт, необхідні для відновлення пам’яті-оптимізованих таблиць.

У межах однієї бази дозволяється лише одна така файлова група, і вона використовується виключно для об’єктів відповідного типу.

Зауваження:

У практичному аспекті типи файлів і файлові групи використовуються для узгодженого планування розміщення об’єктів, визначення політик зростання та реалізації сценаріїв обслуговування. Правильне визначення складу файлів, ролей файлових груп і їхніх параметрів закладає основу керованості бази даних, передбачуваності

використання дискового простору та коректності процедур відновлення.

4.3. Логічні та фізичні параметри файлів та управління дисковим простором

Логічні та фізичні параметри файлів визначають, яку роль конкретний файл виконує в базі даних, як система і адміністратор посилаються на цей файл, а також які обмеження і політики застосовуються до його розміру та розміщення. Під поняттям дисковий простір у цьому контексті мається на увазі весь вільний і зайнятий простір файлової системи або томів, на яких розміщені файли бази даних (файли даних, файли журналів транзакцій, *FILESTREAM*-контейнери та інші контейнерні ресурси, що використовуються *SQL Server*). Управління цим простором включає як налаштування окремих файлів, так і моніторинг та планування їхнього зростання й розподілу по фізичних носіях.

Цікавий факт:

Екстент (extent) *— це базова одиниця виділення простору на диску в багатьох системах керування базами даних (наприклад, у Microsoft SQL Server).*

- *Один екстент складається з* ***8 сторінок (pages)*** *по* ***8 КБ*** *кожна.*
- *Відповідно, загальний розмір одного екстенту дорівнює* ***64 КБ****.*
- *Коли таблиці або індекси зростають, система виділяє їм не окремі сторінки, а цілі екстенти для ефективнішого керування пам'яттю.*

Приклад: *якщо таблиця починає займати більше ніж 8 сторінок, їй виділяється новий екстент.*

Логічні параметри файлу набір атрибутів, які використовуються СКБД і адміністратором при посилання на файл у *T-SQL* та при плануванні розміщення об'єктів.

До ключових логічних параметрів належать:

- логічне ім'я файлу (*logical_file_name*),
- приналежність до файлової групи (для файлів даних)
- ідентифікатор файлу – внутрішній числовий код, який SQL Server призначає файлу для унікальної ідентифікації в системних таблицях. Він використовується на рівні системних процесів.

Логічне ім’я використовують у командах на кшталт *ALTER DATABASE ... MODIFY FILE* або при відновленні/приєднанні бази, тому воно має бути унікальним у межах бази та відповідати правилам ідентифікаторів.

Фізичні параметри файлу визначають, як саме файли бази даних зберігаються та використовуються на рівні операційної системи, тлбто визначають конкретну реалізацію файлу в операційній системі: повний шлях до файлу та його ім’я (*os_file_name*), початковий розмір, максимальний допустимий розмір (*MAXSIZE*) і параметри автоматичного зростання (*FILEGROWTH*).

Ключові фізичні параметри файлів у *SQL Server*:

- фізичне ім’я файлу (*os_file_name*), має бути унікальним у межах файлової системи та відповідати правилам для імен файлів *Windows*, з вказанням повного шляху до файлу в операційної системи. Цей параметр визначає, де саме зберігаються дані або журнал транзакцій на диску, і використовується *SQL Server* під час операцій введення-виведення, резервного копіювання та відновлення бази;
- тип файлу (*DATA, LOG, FILESTREAM*) – визначає функціональне призначення файлу. Файли *DATA* зберігають основні дані та об’єкти бази даних, *LOG*-файли містять журнал транзакцій, який використовується для відновлення бази до певного стану, а *FILESTREAM*-файли забезпечують зберігання великих двійкових об’єктів (наприклад, зображень чи документів) безпосередньо у файловій системі із зв’язком із записами бази даних. Тип файлу впливає на політику резервного копіювання та доступ до даних;
- початковий розмір файлу (*SIZE*) – обсяг дискового простору, який резервується під файл при його створенні. Значення *SIZE* можна задавати в кілобайтах (KB), мегабайтах (MB). Початковий розмір визначає, скільки даних може бути записано без необхідності автоматичного зростання, що зменшує ризик фрагментації і забезпечує стабільну продуктивність;
- максимальний розмір файлу (*MAXSIZE*) – обмеження, до якого файл може збільшуватися. Цей параметр задається у тих самих одиницях (KB, MB) або може мати значення *UNLIMITED*, що дозволяє файлу зростати до повного вільного простору на носії. *MAXSIZE* використовується для контролю використання дискового простору та запобігання неконтрольованому заповненню томів;

- крок автоматичного зростання (*FILEGROWTH*) – величина, на яку файл збільшується після досягнення заповненості. Значення *FILEGROWTH* можна вказати у KB, MB або у відсотках (%) від поточного розміру файлу;
- ідентифікатор файлу (*file_id*) – унікальний числовий код, який SQL Server призначає кожному файлу для внутрішньої ідентифікації. Ідентифікатор використовується для зв'язку сторінок і екстентів із конкретним файлом у системних таблицях, а також у представленнях *sys.database_files* та *sys.master_files* для адміністрування і моніторингу. Для користувачів він не є обов'язковим, але важливий для внутрішніх процесів і оптимізації керування файлами.

ВАЖЛИВО:

Політики автозростання потребують обережного підходу. Часті малі прирости призводять до фрагментації файлів і до частих операцій розширення дискового простору, що може викликати затримки у роботі системи; надто великі прирости можуть створити великі невикористані області і ускладнити керування простором. Тому в продуктивному середовищі рекомендовано попередньо виділяти достатній обсяг дискового простору та встановлювати FILEGROWTH в розмірі, адекватному очікуваному росту (фіксований крок у MB), а також, за потреби, вказувати MAXSIZE, щоб уникнути неконтрольованого заповнення тома.

Якщо у файловій групі присутні кілька файлів, SQL Server розподіляє нові екстенти між файлами за принципом пропорційного заповнення, тобто розподіл залежить від відсотку вільного місця у кожному файлі. Через це адміністраторам доцільно підтримувати співмірні початкові розміри та узгоджені політики автозростання для файлів однієї групи, щоб уникнути нерівномірного використання носіїв і непередбачуваного впливу на продуктивність. Автозростання відбувається на рівні окремого файлу; коли один файл у групі заповнений і потребує розширення, саме йому застосовується його налаштування *FILEGROWTH*.

Окрему увагу слід приділяти файлам журналу транзакцій. Налаштування розміру журналу і параметрів його зростання впливають на внутрішню структуру журналу (кількість внутрішніх віртуальних лог-файлів *VLF*), що, у свою чергу, впливає на швидкість відновлення і операції резервного копіювання логів. Часте автозростання журналу невеликими кроками породжує багато дрібних *VLF* і може погіршити продуктивність, тому для журналів також рекомендують задавати помірні фіксовані кроки росту або заздалегідь виділяти достатній розмір.

Для спостереження та управління параметрами файлів використовують як системні збережені процедури, так і представлення каталогу (таблиця 4.1.)

Окремим обмеженням і важливим чинником є операції стиснення/зменшення файлу (*DBCC SHRINKFILE / DBCC SHRINKDATABASE*). Хоча інструменти для зменшення розміру бувають необхідні у разі одноразового звільнення простору, вони спричиняють фрагментацію даних і, як правило, не рекомендовані як регулярна практика. Краще запланувати корекцію розмірів через попереднє провиділення та архівацію/очищення даних, ніж постійно зменшувати і знову розширювати файли.

Таблиця 4.1.

Інструменти для спостереження та управління файлами бази даних у SQL Server

Інструмент	***Тип***	***Призначення***	***Особливості***
sp_helpdb [база_даних]	Збережена процедура	Відображає інформацію про базу даних, включаючи налаштування та розташування файлів	Якщо база не вказана, показує інформацію по всіх базах
sp_helpfile ['ім'я_файлу']	Збережена процедура	Показує інформацію про фізичні файли поточної бази даних	Якщо ім'я не вказано, відображає всі файли бази
sp_helpfilegroup ['ім'я_групи']	Збережена процедура	Показує інформацію про файлові групи бази даних та файли в них	Можна переглянути всі групи або конкретну групу
sp_spaceused ['об'єкт']	Збережена процедура	Показує обсяг дискового простору, який використовується базою даних або	Корисно для моніторингу зайнятого місця

		конкретним об’єктом	
sys.database_files	Системне представлення	Містить інформацію про всі файли поточної бази даних	Використовується для адміністрування та скриптів
sys.master_files	Системне представлення	Містить інформацію про файли всіх баз на сервері	Дає глобальний огляд стану всіх файлів
sys.filegroups	Системне представлення	Містить інформацію про файлові групи бази даних	Використовується для управління та моніторингу груп
sys.dm_io_virtual_file_stats	Динамічне представлення продуктивності	Показує статистику I/O по кожному файлу	Дозволяє оцінити навантаження та продуктивність

Підсумовуючи, керування логічними та фізичними параметрами файлів і дисковим простором — це комплекс заходів: точне визначення логічних і фізичних атрибутів файлів при створенні бази, обґрунтоване налаштування початкових розмірів і політик автозростання, вибір розміщення файлів на фізичних носіях з урахуванням профілю навантаження (розділення даних і журналу, виділення окремих груп під «гарячі» таблиці або індекси), регулярний моніторинг використання простору та обережне застосування операцій зменшення розміру. Дотримання цих принципів забезпечує передбачуваність використання дискового простору, зменшує ризики пов’язані з фрагментацією та сприяє підтримці стабільної продуктивності системи.

4.4. Сторінки файлів даних

У системі керування базами даних *Microsoft SQL Server* мінімальною одиницею зберігання інформації є сторінка даних. Кожна сторінка має фіксований розмір у 8 КБ, незалежно від типу даних, що зберігаються, і містить структуровану інформацію, що може включати рядки таблиць, елементи індексів, великі об’єкти (текстові або графічні дані) чи службові структури системних каталогів (метадані). Саме через сторінки *SQL Server* організовує роботу з фізичними файлами бази даних, забезпечуючи ефективний доступ до інформації.

Перші сторінки файлу мають спеціальне призначення:

- 🕮 *Сторінка заголовка файлу* (*File Header Page*) містить метадані про файл, такі як тип, розмір, поточний стан та інші атрибути.
- 🕮 *Системні сторінки* (*Allocation Map Pages: GAM, SGAM, PFS* тощо) зберігають інформацію про розподіл простору, карти виділення

екстентів та сторінок, що дозволяє *SQL Server* ефективно керувати вільним та зайнятим місцем (детальний опис та призначення системих сторінок подано у таблиці 4.2).

- 🕮 *Завантажувальна сторінка бази даних* (*Database Boot Page*) розташована у первинному файлі даних (*.mdf*). Вона містить ключові атрибути бази та необхідна для її ініціалізації при відкритті, відновленні або приєднанні. Завантажувальна сторінка завжди має номер **9.**

Логічна структура сторінки у SQL Server є чітко організованою та складається з кількох обов'язкових компонентів. У верхній частині розташований *заголовок сторінки (Page Header),* який містить службову інформацію: номер сторінки, її тип, стан, ідентифікатор файлу, посилання на попередню та наступну сторінку, а також відомості про вільний простір. Основна частина сторінки відводиться під *область даних* (*Data Rows*), де безпосередньо зберігаються рядки таблиць, індексні записи чи інші об'єкти залежно від призначення сторінки. Між заповненими рядками і кінцем сторінки залишається *область вільного простору (Free Space)*, що дозволяє розміщувати нові записи або модифікувати наявні без негайного виділення нової сторінки. У нижній частині розташований *масив зміщень рядків* (*Row Offset Array*), який містить покажчики на фізичні позиції кожного рядка всередині сторінки, забезпечуючи швидкий доступ до даних незалежно від їх розташування. Така організація гарантує ефективне використання простору та оптимізує операції пошуку, вставки й модифікації даних.

Таблиця 4.1.

Спеціальні сторінки даних у SQL Server

Назва сторінки (укр.)	***Призначення***	***Розташування***
Сторінка заголовка файлу *File Header Page*	Містить службові метадані про сам файл: тип, розмір, ідентифікатор, стан та інші атрибути	На початку кожного файлу даних
Сторінка завантаження бази *Database Boot Page*	Зберігає ключові відомості про базу: ім'я, ID, версію, параметри сумісності, час створення; використовується для відкриття/відновлення	Лише у первинному файлі даних (*.mdf*), зазвичай сторінка 9
Сторінка карти виділеного простору *Page Free Space (PFS) Page*	Відстежує рівень заповненості сторінок (вільне місце, ступінь використання)	Кожні 8 088 сторінок (~64 МБ)
Глобальна карта виділення *Global Allocation Map (GAM) Page*	Фіксує інформацію про вільні екстенти, доступні для виділення	Кожні 64 000 сторінок (~512 МБ)

Загальнодоступна карта виділення *Shared Global Allocation Map (SGAM) Page*	Відстежує екстенти, які одночасно є частково виділеними для кількох об’єктів	Кожні 64 000 сторінок (~512 МБ)
Диференційна карта *Differential Changed Map (DCM) Page*	Відзначає екстенти, що змінилися з моменту останнього повного резервного копіювання	Кожні 64 000 сторінок (~512 МБ)
Карта змінених під час журналювання *Bulk Changed Map (BCM) Page*	Відстежує екстенти, змінені під час операцій bulk insert, для полегшення резервування	Кожні 64 000 сторінок (~512 МБ)

Сторінки файлів нумеруються послідовно в межах конкретного файлу, причому перша сторінка файлу має номер 0. Для унікальної ідентифікації сторінки у всій базі даних використовується комбінація ідентифікатора за схемою «ідентифікатор файлу : ідентифікатор сторінки» (*File ID : Page ID*). Наприклад, сторінка *01:0000* належить первинному файлу даних з *File ID 01*, а сторінка *02:0001* – вторинному файлу з *File ID 02* (рис. 4.2). Такий принцип уніфікованої адресації дозволяє швидко визначати фізичне розташування даних у базі та оптимізує роботу механізмів пошуку. Це забезпечує точне визначення розташування будь-якого об’єкта чи фрагмента даних у фізичному файлі.

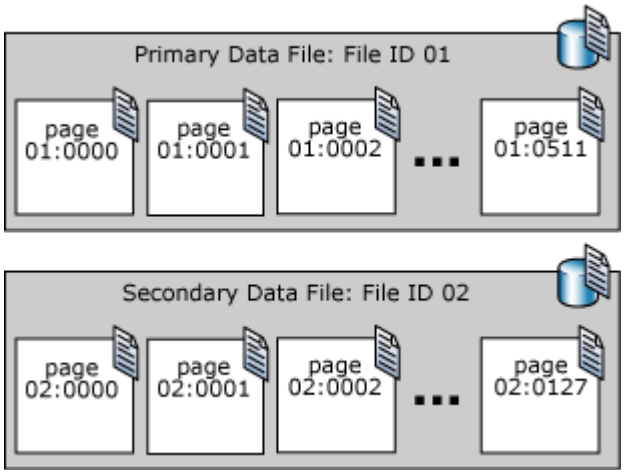


Рис. 4.2. Приклад сторінкової організації первинного та вторинного файлів даних у SQL Server

SQL Server використовує різні типи сторінок залежно від призначення:

- *Data Page* зберігає рядки таблиць;
- *Index Page* містить елементи індексів;
- *Text/Image Page* зберігаються великі об’єкти (*LOB*-даних);

- *IAM Page* (*Index Allocation Map*) відстежує екстенти, виділені певному об'єкту (таблиці чи індексу).

ВАЖЛИВО:

IAM Page (Index Allocation Map) відстежує групи сторінок пам'яті, які виділені конкретному об'єкту (таблиці чи індексу). Такі групи складаються з 8 послідовних сторінок (по 8 КБ кожна) і утворюють єдиний блок обсягом 64 КБ, що називається ***екстентом (Extent)***.

Таким чином, сторінкова організація забезпечує баланс між ефективним зберіганням, швидким доступом до даних і можливістю масштабування бази. Вона є основою для механізмів резервування, відновлення та адміністрування, а також визначає принципи взаємодії між файлами та файловими групами у *SQL Server*.

4.5. Створення нової бази даних з використанням файлових груп

Створення нової бази даних за допомогою файлових груп передбачає копіювання базової конфігурації із системної бази *model*, а також налаштування фізичних і логічних параметрів файлів даних та журналів транзакцій. Користувачі з ролями *sysadmin* та *dbcreator* мають право виконувати такі операції, що гарантує контроль за створенням та структурою баз даних у межах організації.

Синтаксис створення нової бази даних з розподілом по файлам та файловим групам:

```
CREATE DATABASE ім'я_бази_даних
[ON [PRIMARY] (NAME = 'логічне_ ім'я _файлу',
FILENAME = 'фізичне_ ім'я_файлу'
[, SIZE = розмір]
[, MAXSIXE = {максимальний_розмір | UNLIMITED} ]
[, FILEGROWTH = крок_збільшення_розміру [Mb | Kb | %] )
[ {FILEGROUP ім'я_файлової_групи} ]
```

[, ...n]
[LOG ON (NAME = 'логічне_ім'я_файлу',
FILENAME = 'фізичне_ім'я_файлу'
[, SIZE = розмір]
[, MAXSIXE = { максимальний_розмір | UNLIMITED}]
[, FILEGROWTH = крок_збільшення_розміру [Mb | Kb | %])
[, ...n]

Основні ключові слова і параметри:

PRIMARY – визначає файл як первинний або як член первинної файлової групи, якщо опущено, то основним файлом стає перший файл в операторі і для зберігання використовується первинна файлова група;

NAME – визначає логічне ім'я файлу. За замовчуванням збігається з фізичним ім'ям файлу, визначеному в параметрі *FILENAME*;

FILENAME – вказує повний шлях і ім'я фізичної файлу;

SIZE – вказує розмір файлу: в мегабайтах, кілобайтах. Мінімально можливе значення 512 Кб. Розмір основного файлу за замовчуванням дорівнює розміру БД *model*. За замовчуванням розмір додаткових файлів даних і журналу дорівнює 1 Мб;

MAXSIZE – вказує максимальний розмір, до якого може збільшуватися файл. Якщо цей параметр не вказано, то встановлюється значення *UNLIMITED*, що дозволяє збільшувати файлів розмір без обмежень;

FILEGROWTH – задає крок збільшення файлу, причому нуль означає заборону збільшення розміру. Значення вказується в мегабайтах, кілобайтах або відсотках. За замовчуванням приріст – 10%, якщо не вказані одиниці, то цифра сприймається в мегабайтах;

FILEGROUP – визначає ім'я групи файлів, в яку поміщається файл.

Для перегляду інформації про базу даних, файли і групи файлів використовуються наступні збережені процедури (подані у таблиці 4.1.)

Крім перерахованих вище фізичних параметрів база даних має ще й логічні параметри. Тільки власник і системний адміністратор може змінити ці параметри. Управління параметрами здійснюється за допомогою системної збереженої процедури sp_dboption:

sp_dboption [[@dbname=] 'ім'я_бази'] [, [@option=] ''] [, [@value=] ON | OFF]

Використання файлових груп при створенні бази даних у *Microsoft SQL*

Server дозволяє ефективно організовувати зберігання даних, розподіляти навантаження між фізичними носіями та підвищувати продуктивність системи. Ретельне налаштування логічних і фізичних параметрів файлів бази, а також застосування системних збережених процедур для моніторингу та управління ресурсами забезпечує надійність, масштабованість та контроль за структурою бази даних. Такий підхід є важливим етапом у професійному адмініструванні баз даних та створює основу для оптимізації їхньої роботи у будь-яких інформаційних системах.

Рекомендації при роботі з файлами і файловими групами:

- 🕮При використанні багатьох файлів даних створіть другу файлову групу з додатковим файлом і зробіть її файлової групою за замовчуванням. Тоді в первинному файлі будуть зберігатися тільки системні таблиці та об'єкти.
- 🕮Щоб збільшити продуктивність, рознесіть файли та файлові групи на декілька доступних дисків. Об'єкти, які активно конкурують за вільний простір, розмістити в різні файлові групи.
- 🕮Використовуйте файлові групи для цілеспрямованого розміщення об'єктів на конкретних фізичних дисках.
- 🕮Розміщуйте різні таблиці, що використовуються в одних і тих же запитах, в різних файлових групах.
- 🕮Використання різних груп файлів збільшить продуктивність, так як можна буде використовувати паралельне введення і виведення, якщо файли знаходяться на різних жорстких дисках.
- 🕮Ніколи не кладіть файли журналу транзакцій на той же фізичний диск, де знаходяться інші файли та файлові групи.

Питання для самоперевірки

1. *Які типи файлів та файлових груп ви знаєте?*
2. *Дайте визначення первинному файлу даних.*
3. *Для чого використовують вторинні файли даних?*
4. *Який файл містить розширення .ldf?*
5. *Який файл містить розширення .ndf?*

6. *Скільки файлів база даних створюєть за замовченням?*
7. *З чого складається сторінки файлів даних?*
8. *Для чого використовується файлова група?*
9. *У чому полягає особливість файлової групи PRIMARY*
10. *У яких випадках з Використовується файлова група з ознакою CONTAINS MEMORY_OPTIMIZED_DATA?*
11. *Назвіть спеціальне призначення для перших сторінок файлу.*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 4

на тему: *«Файли і файлові групи бази даних»*

МЕТА: *набути практичних навичок щодо створення файлових груп різними способами у середовищі MS SQL Server Management Studio. Навчитися створювати різні типи файлів бази даних. Освоїти способи переміщення файлів бази даних з одного диску на інший, опанувати команди Detach та Attach для виконання даних дій.*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

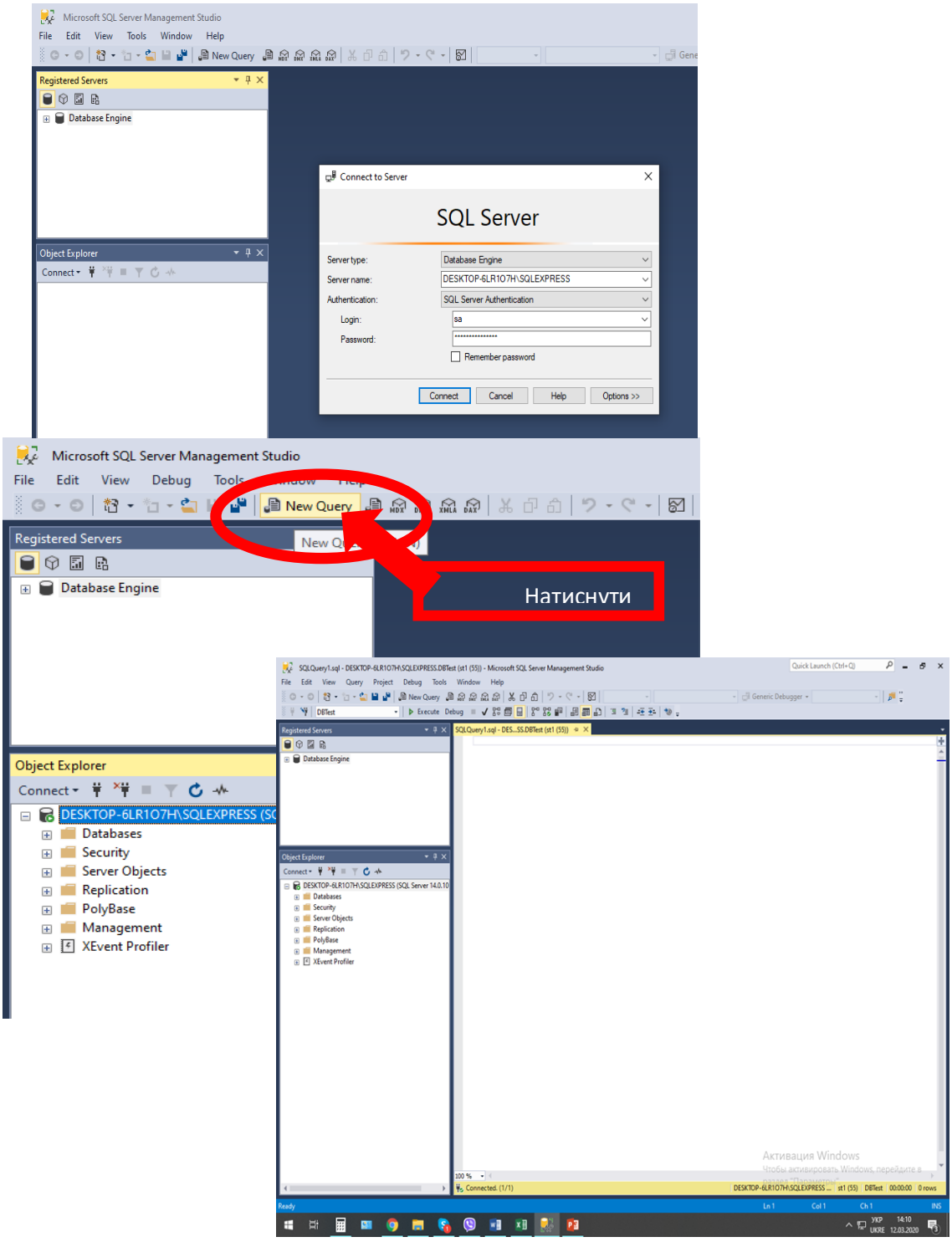
1. У середовищі *MS SQL Server Management Studio* створити нову базу даних.
2. За допомогою графічного інтерфейсу середовища *MS SQL Server Management Studio* створити нову файлову групу.
3. До наявних файлів (первиного з роширенням *.mdf* та файл журналу транзакцій з розширенням *.ldf*) додати мінімум два нових файли БД з розширенням *.ndf*. Надати новоствореним файлам:
   - логічне та фізичне ім’я;
   - вказати фізичне розташуванням на диску *C:*;
   - приріст зростання та максимальний розмір.
4. Повторити пп. 2 та 3 використовуючи SQL-команди у середовищі *MS SQL Server Management Studio*.
5. Виконати переміщення файлів бази даних виконуючи команди від’єднання від серверу *Detach* та приєднання файлів БД *Attach*.
6. Створити звіт про хід виконання лабораторної роботи.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

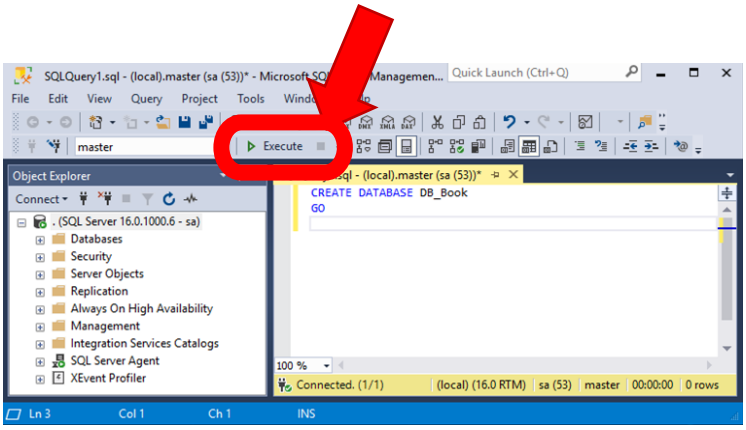
Для виконання даної лабораторної роботи необхідно встановити відповідне програмне середовище *MS SQL Server* та *MS SQL Server Management Studio*.

Після встановлення даних програмних продуктів, необхідно виконати вхід у середовище *MS SQL Server Management Studio* під логіном системного

адміністратора *SA* та ввести пароль, який було введено під час інсталяції *MS SQL Server*.

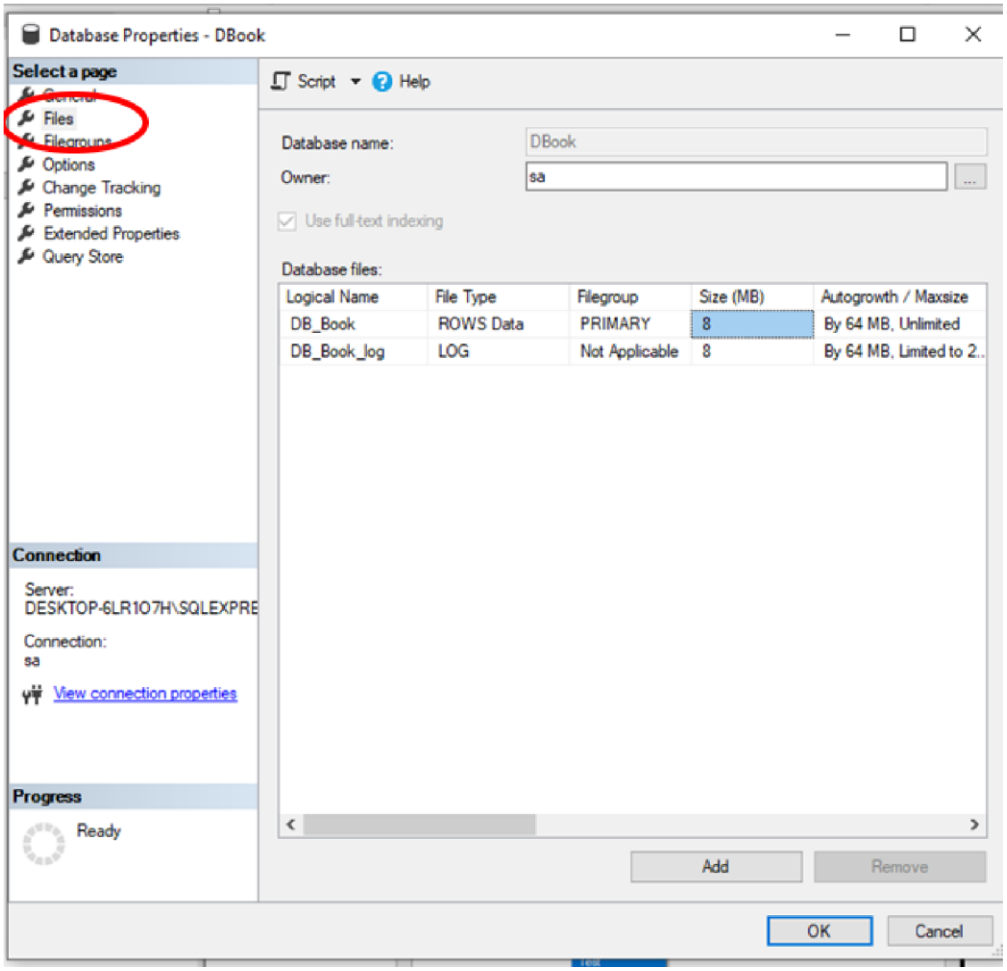


1. У вікні ***SQLQuery*** набрати код створення БД.
2. Натиснути кнопку ***Execute*** (виконати)



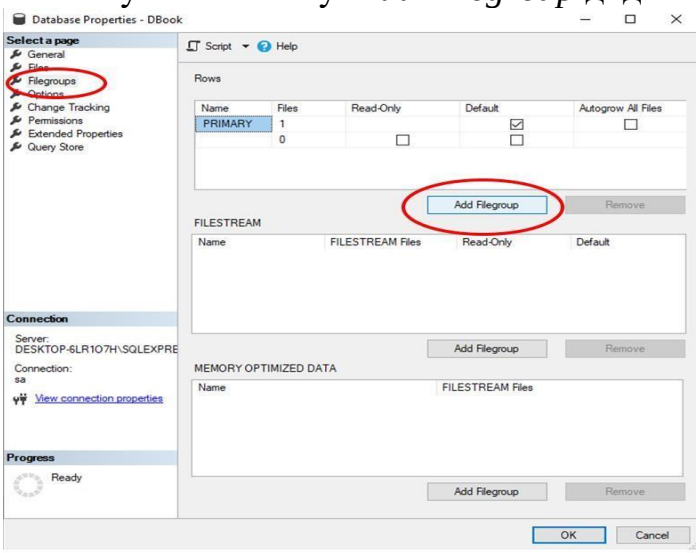
Виконання наступних операцій будемо проводити у БД *master*.

Якщо БД вже створення, то вона містить одну файлову групу PRIMARY та два файли: первинний файл даних з розширенням *.mdf* та файл журналу транзакцій з розширенням *.ldf*. Щоб переглянути, натиснути правою кнопкою миші по БД, обрати команду *Properties*, у діалоговому вікні обрати опцію *Files*.

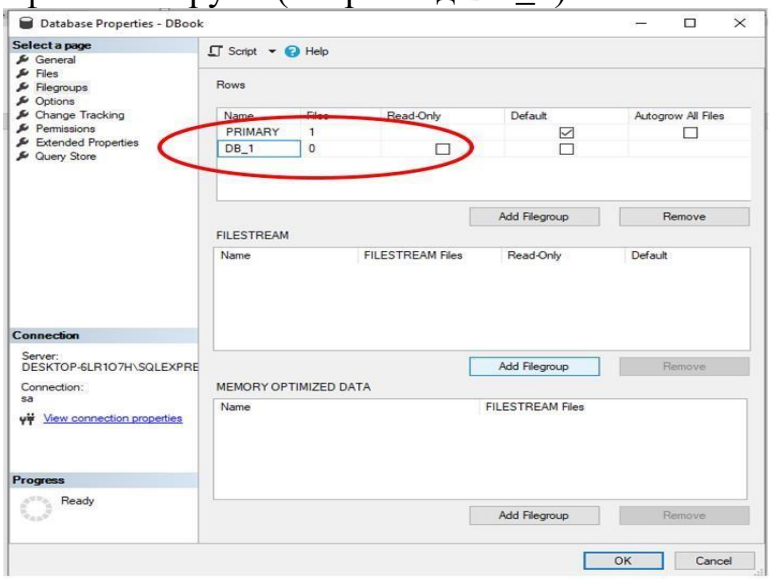


Для додавання файлової групи з іншою назвою у вікні властивості обрати опцію *Filegroups*. У вікні зі списком Rows вже створено за замовчуванням *Defult* файлову групу *PRIMARY*, в якій вже знаходиться первинний файл даних з розширенням *.mdf* (файл журналу транзакцій з розширенням *.ldf* не може входити до жодних груп файлів).

Натиснувши кнопку *Add Filegroup* додаємо нову групу.

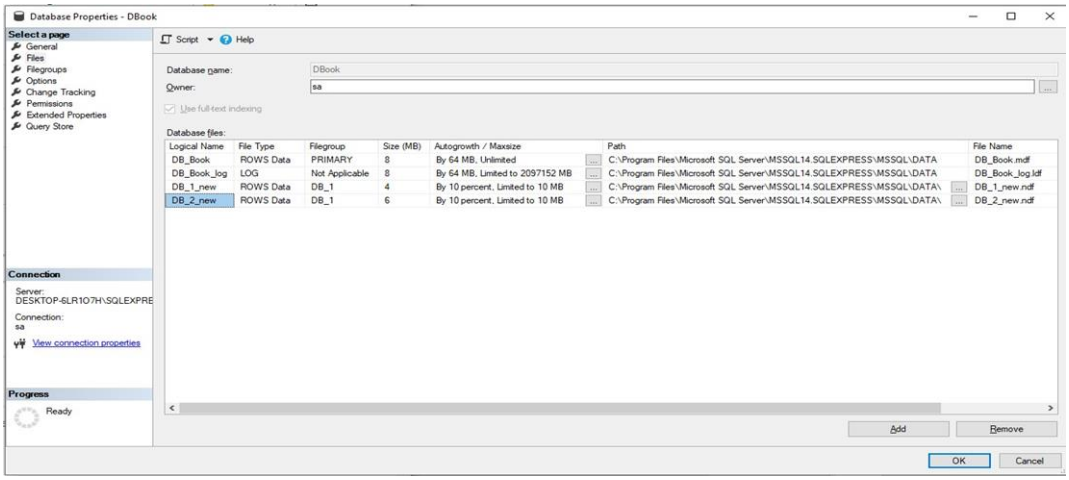


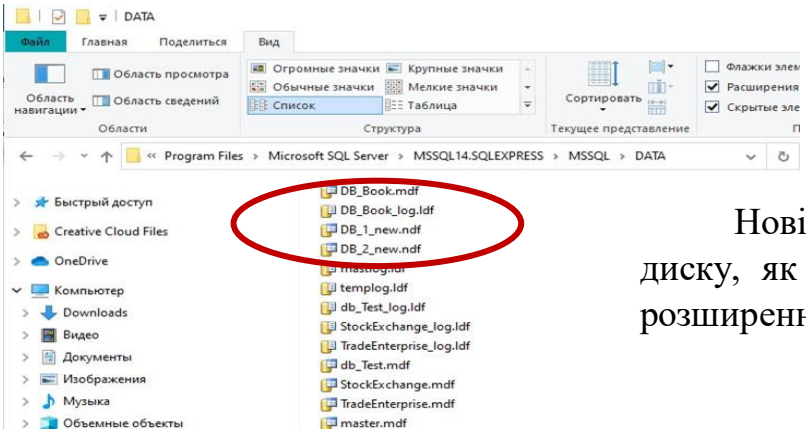
У вікні зі списком *Rows* додається новий рядок, в якому задається ім’я нової файлової групи (наприклад *DB_1*) та натискаємо кнопку ОК.



Після створення нової файлової групи *DB_1* необхідно аби вона містила хоча б один файл. Тому треба створити новий файл та призначити йому створену файлову групу.

Для цього обрати команду *Properties* для БД, у діалоговому вікні обрати опцію Files. По-черзі додаємо нових два файли з логічним ім’ям (*DB_1_new* та *DB_2_new*), обираємо групу *DB_1*, вказуємо розміри файлів 4 та 6 МВ відповідно, максимальний розмір – 10 МВ, крок збільшення файлу – у відсотках (10%), та задаємо фізичні імена файлам з розширенням *.ndf* (*DB_1_new.ndf* та *DB_2_new.ndf*):





Нові файли створені на диску, як решта файлів, але з розширенням .ndf.

Тепер, коли будуть створюватись таблиці, їх можна свідомо прив'язувати до файлової групи *DB_1*. Наприклад:

```
USE DB_Book
GO

CREATE TABLE TableMy
( Name CHAR (10)      )     ON DB_1
```

Створення нової бази даних за допомогою SQL-команд

Для створення нової бази даних на основі файлових груп необхідно написати програмний код за допомогою SQL-команд (приклад подпно у Листингу 1).

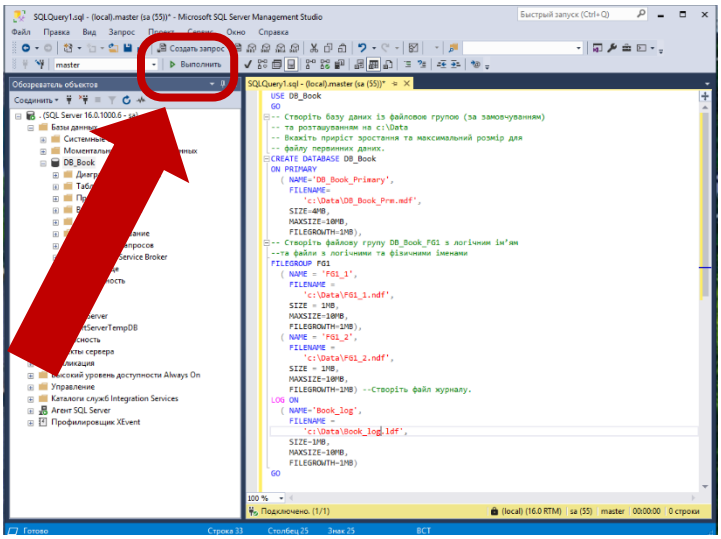
Лістинг 1. Створення бази даних

```
USE DB_Book
GO
-- Створіть базу даних із файловою групою (за замовчуванням) та розташуванням на
-- c:\Data
-- Вкажіть приріст зростання та максимальний розмір для файлу первинних даних.
```

```
CREATE DATABASE DB_Book
ON PRIMARY
 ( NAME='DB_Book_Primary',
   FILENAME=
     'c:\Data\DB_Book_Prm.mdf',
   SIZE=4MB,
   MAXSIZE=10MB,

FILEGROWTH=1MB),
-- Створіть файлову групу DB_Book_FG1 з логічним ім'ям
--та файли з логічними та фізичними іменами
FILEGROUP FG1
 ( NAME = FG1_1',
   FILENAME =
     'c:\Data\ FG1_1.ndf',
   SIZE = 1MB,
   MAXSIZE=10MB,

FILEGROWTH=1MB),
 ( NAME = FG1_2',
   FILENAME =
     'c:\Data\ FG1_2.ndf',
   SIZE = 1MB,
   MAXSIZE=10MB,
   FILEGROWTH=1MB)
FILEGROUP FG2
 ( NAME = FG2_1',
   FILENAME =
     'c:\Data\ FG2_1.ndf',
   SIZE = 1MB,
   MAXSIZE=10MB,

FILEGROWTH=1MB),
 ( NAME = FG2_2',
   FILENAME =
     'c:\Data\ FG2_2.ndf',
   SIZE = 1MB,
   MAXSIZE=10MB,
   FILEGROWTH=1MB) --
--Створіть файл журналу.
LOG ON
 ( NAME=Book_log',
   FILENAME =
     'c:\Data\ Book_log.ldf',
   SIZE=1MB,
   MAXSIZE=10MB,

FILEGROWTH=1MB);
GO
```

В результаті отримуємо дві файлові групи та шість файлів, один з них містить мета дані, один – журнал транзакцій.

У середовищі *MS SQL Server Management Studio* даний лістинг матиме вид:

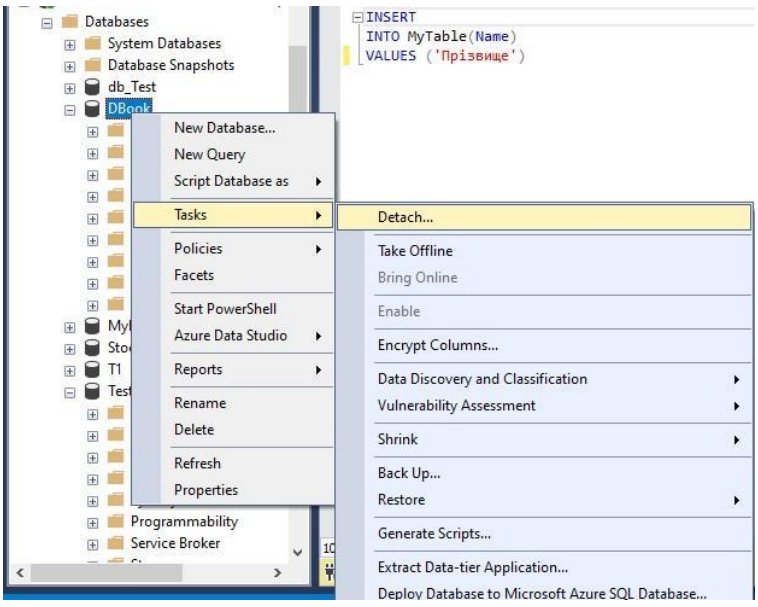


Наступним кроком буде натиснути кнопку *Execute* (виконати) або клавішу F5.

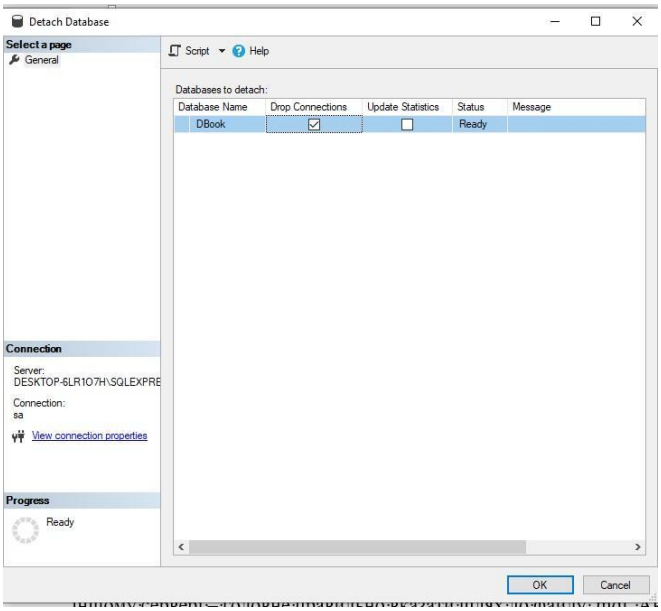
Переміщення файлів (БД) з одного диску на інший

При переміщенні БД *mdf*-файл не можна витісняти, лише *ndf* -файли у межах файлової групи.

Спочатку створимо на іншому диску теку *Test* (*e:\Test*). Від'єднати БД від серверу використовуючи команду *Detach* (користувачі в цей момент не будуть бачити БД).

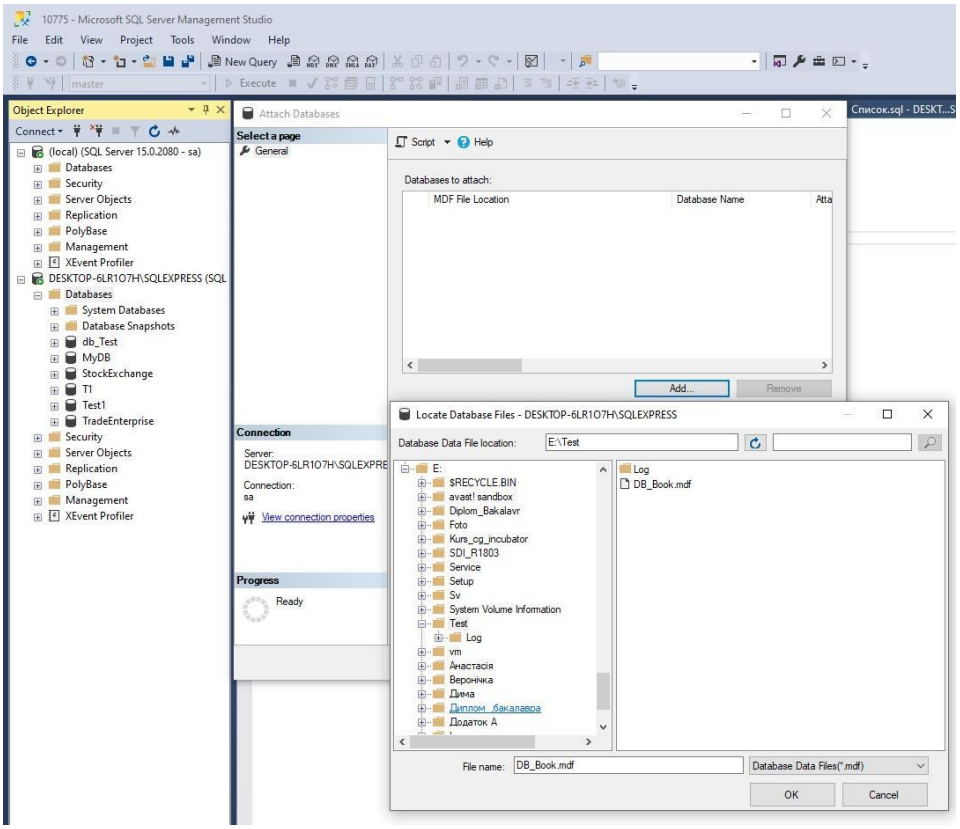


Для того, щоб коректно від'єднати БД, необхідно від'єднати всіх користувачів. Для цього в меню обирається прапорець *Drop connections*, яка обриває всі активні з'єднання.

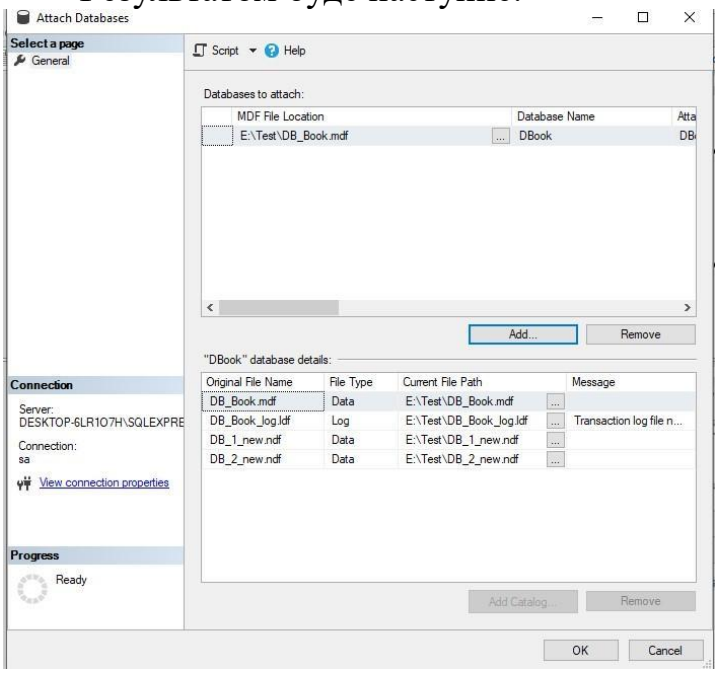


Після від'єднання SQLServer «забуває» про файли БД, викреслює її із системних таблиць, в списках її вже немає, але БД находиться на тому місці диску, де й знаходилась, а файли всі розблоковані. Далі перенести файли у створений каталог.

Наступний крок – приєднання файлів БД за допомогою команди *Attach* (права кнопка миші по *Databases*). Вказану команду можна виконати на іншому сервері – головне правильно вказати шлях до файлу .mdf. Активується кнопка додати та обирається переміщений файл .mdf (решта файлів автоматично будуть приєднані, т.к. в цьому файлі знаходяться мета дані, де в тому числі, записано список файлів вказаної БД, їх назва та місцезнаходження). Можна в одному вікні приєднати декілька БД одночасно.



Результатом буде наступне:



Після приєднання БД буде активною.

ТЕМА 5. СТВОРЕННЯ ТАБЛИЦЬ БАЗИ ДАНИХ ТА ОБРОБКА ДАНИХ У ТАБЛИЦЯХ

ПЛАН:

5.1. Технологія створення таблиць бази даних.
5.2. Застосування обмежень.
5.3. Створення схеми даних.
5.4. Обробка даних у таблицях.
5.5. Модифікація структури таблиць.
5.6. Засоби перевірки та контролю даних.

5.1. Технологія створення таблиць бази даних

Створення таблиць є ключовим етапом фізичної реалізації бази даних, оскільки саме в таблицях зберігаються всі дані предметної області. У реляційних СКБД, зокрема *Microsoft SQL Server*, таблиця визначається як набір рядків (записів), організованих за стовпцями (полями), кожне з яких має певний тип даних та набір обмежень цілісності.

Структура таблиць формується на основі логічної моделі, але при фізичній реалізації враховуються особливості СКБД, обмеження щодо типів даних, продуктивності та масштабованості. Важливим аспектом є вибір схеми бази даних (*Schema*)**,** у межах якої створюються об’єкти, що дозволяє логічно групувати таблиці за функціональним призначенням і спрощує адміністрування (створення схеми даних буде розглянуто у наступному пункті даної теми).

Процес створення таблиць включає такі основні етапи:

- Визначення структури таблиці – перелік полів, їх типів даних та логічних обмежень.
- Формування первинного ключа (*PRIMARY KEY*) для однозначної ідентифікації кожного запису.
- Застосування зовнішніх ключів (*FOREIGN KEY*) для забезпечення зв’язку між таблицями та підтримання референційної цілісності.
- Додавання додаткових обмежень (*NOT NULL, UNIQUE, CHECK, DEFAULT*) – для підвищення коректності збережених даних.

SQL дозволяє створювати таблиці окремо.

SQL-команда CREATE TABLE – створення таблиці.

Команда *CREATE TABLE* використовується для створення нової таблиці у базі даних. Вона задає логічну структуру майбутнього об'єкта зберігання даних, визначаючи його назву, атрибути (стовпці), їхні типи даних та обмеження.

Синтаксис SQL-команди CREATE TABLE

```
CREATE TABLE [схема.]ім'я_таблиці
( ім'я_стовпця тип_даних [NULL | NOT NULL]
   [CONSTRAINT ім'я_обмеження тип_обмеження],
   ...
   [CONSTRAINT ім'я_ключа PRIMARY KEY (стовпці)],
   [CONSTRAINT ім'я_ключа FOREIGN KEY (стовпці)
      REFERENCES таблиця_посилання (стовпці)]);
```

Основні ключові слова і параметри команди CREATE TABLE:

ім'я_стовпця – назва поля таблиці;

тип_даних – визначає формат збереження даних (наприклад, *INT, DECIMAL, NVARCHAR, DATE*);

NULL / NOT NULL – вказує, чи допускається відсутність значення;

CONSTRAINT – задає ім'я та тип обмеження цілісності;

PRIMARY KEY – визначає унікальний ідентифікатор запису;

FOREIGN KEY – встановлює зв'язок з іншою таблицею;

CHECK, DEFAULT, UNIQUE – додаткові механізми контролю даних, обмеження.

ПРИКЛАД 5.1. Створення нової таблиці з інформацією про авторів за допомогою SQL-команди *CREATE TABLE*, яка включає в себе поля код автора, прізвище та ім'я автора.

Лістинг 5.1. Створення таблиці, яка містить інформацію про авторів

```
USE DB_Book
GO
CREATE TABLE Avtor
(IdAvtor INT PRIMARY KEY , Surname VARCHAR (30),
Name   VARCHAR (30))
GO
```

*SELECT * FROM Avtor*

Зауваження:

При використанні типу даних CHAR або VARCHAR, рекомендується наступний підхід.

- ✓ *Використовуйте CHAR, коли розміри значень постійні.*
- ✓ *Використовуйте VARCHAR, коли розміри мінливі.*
- ✓ *Використовуйте VARCHAR(MAX), якщо очікується, що розміри значень можуть перевищувати 8000 байт.*

Пояснення:

- Ефективність та витрати простору *CHAR*: тип даних *CHAR* завжди займає фіксований об'єм, навіть слово "Hi" у *CHAR*(5) буде збережено як "Hi" (із заповненням пробілами, щоб у сумі було 5 символів), що може призвести до зайвого використання диску. Проте при постійному розмірі значень це забезпечує швидшу обробку та прогнозовану роботу з індексами.
- Гнучкість і додаткові витрати *VARCHAR*: тип даних *VARCHAR* використовує лише потрібний обсяг плюс кілька байтів на збереження довжини даних. Це ефективно для значень, які мають довільну довжину і варіативність. Але може мати трохи меншу продуктивність через додаткову обробку.
- Великі текстові поля з обмеженнями й особливостями *VARCHAR(MAX):* тип даних *VARCHAR*(*MAX*) автоматично переходить на LOB-формат, якщо дані перевищують ~8000 байт (у межах 8000 байт зберігається в рядку (*in-row*), **але** при перевищенні – у спеціальному LOB-розділі). Індексування обмежено, тобто не можна створити звичайний B-tree індекс на VARCHAR(MAX) (тільки включений стовпець).

Потенційні ризики застосування VARCHAR(MAX):

- 24 байти додаткового резервування пам'яті на стовпець можуть впливати на максимальний розмір рядка (8,060 байт).
- Деякі функції можуть обробляти лише перші ~8,000 символів, а все решта – можуть ігноруватись.

Баланс між практичністю та перформансом:

- *VARCHAR*(*MAX*) корисно використовувати лише тоді, коли це справді необхідно (наприклад, для зберігання довгих текстів, *JSON*, *XML* тощо);
- надмірне використання *VARCHAR*(*MAX*) без потреби може знизити продуктивність, масово перевищити обсяг пам'яті під час запитів, ускладнити індексування, тощо.

5.2. Застосування обмежень

Обмеження (Constraints) це спеціальні правила, які забезпечують дотримання умов збереження та підтримку цілісності даних у таблицях бази даних. Їх використання гарантує відповідність даних визначеним вимогам, запобігає дублюванню та мінімізує ризик логічних помилок.

У *SQL Server* та інших системах керування базами даних підтримуються такі основні типи обмежень: *PRIMARY KEY, FOREIGN KEY, CHECK, DEFAULT, UNIQUE, NOT NULL***.**

Обмеження *PRIMARY KEY* визначає унікальний ідентифікатор кожного рядка в таблиці. Таблиця може містити лише один первинний ключ. Поле або набір полів, позначених цим ключем, не може містити значення *NULL* і має бути унікальним. Первинний ключ автоматично створює кластеризований індекс для оптимізації пошуку.

Приклад використання *Унікальна ідентифікація коду автора для таблиці «Довідник авторів»:*

```
CREATE TABLE AUTHORS
  (IdAvtor INT PRIMARY KEY , Surname
    VARCHAR (30), Name VARCHAR (30))
```

Обмеження *FOREIGN KEY* встановлює логічний зв’язок між двома таблицями та гарантує дотримання референційної цілісності даних. Воно визначає, що значення у стовпці або наборі стовпців підлеглої таблиці повинні відповідати значенням первинного ключа або унікального ключа пов’язаної таблиці. Це забезпечує узгодженість даних, запобігаючи вставці некоректних значень, які не існують у базовій таблиці, а також блокує видалення або зміну записів, на які є активні посилання з інших таблиць. Додатково обмеження може застосовуватися для реалізації каскадних операцій відповідно до визначених правил (*CASCADE***,** *SET NULL***,** *SET DEFAULT***,** *NO ACTION*).

Приклад використання *Забезпечити, щоб для кожного автора була вказана країна походження, яка зберігається в таблиці «Довідник країн» (Countries), шляхом встановлення зовнішнього ключа:*

```
CREATE TABLE Authors (IdАвтора INT
  PRIMARY KEY,  Прізвище NVARCHAR(30), Ім'я
  NVARCHAR(30), IdКраїни INT FOREIGN KEY
  REFERENCES  Bookstore.Countries(IdКраїни))
      GO
```

CASCADE – це режим зовнішнього ключа, який автоматично поширює зміни з батьківської таблиці (таблиця, що містить первинний ключ) на підлеглу (таблиця, яка містить зовнішній ключ, що посилається на первинний ключ батьківської таблиці). Якщо видаляється запис у батьківській таблиці, всі пов'язані записи у підлеглій таблиці автоматично видаляються. Якщо змінюється значення ключа в батьківській таблиці, значення у підлеглій таблиці теж оновлюється автоматично.

Приклад використання

Таблиця «Довідник авторів» (Avtor) є батьківською по відношенню до таблиці «Прайс-лист» (PriceList), тобто містить первинний ключ IdАвтора, а у підлеглій таблиці «Прайс-лист» ключ IdАвтора є вже зовнішнім. Якщо видаляємо автора, всі його книги в прайс-листі повинні видалятися, щоб не залишилися «сиротами».

```
CREATE TABLE PriceList (IdBook INT PRIMARY
KEY, NameBook TEXT, IdAvtor INT FOREIGN
KEY REFERENCES Avtor (IdAvtor), Cena INT
CONSTRAINT FK_PriceList_Avtor
    FOREIGN KEY (IdAvtor)
    REFERENCES Avtor(IdAvtor)
    ON DELETE CASCADE);
```

У даному прикладі додається обмеження до таблиці *PriceList*, що має назву *FK_PriceList_Authors* для зручності ідентифікації. Після цього йде тип обмеження *FOREIGN KEY* (*IdАвтора*) і вказівка, на яку таблицю воно посилається *REFERENCES Authors* (*IdАвтора*), та режим дії *ON DELETE* CASCADE. У результаті при видаленні автора автоматично видаляються всі його книги, при оновленні *IdАвтора* в таблиці *Authors* автоматично оновлюється *IdАвтора* в таблиці *PriceList*.

SET NULL – це режим дії зовнішнього ключа, за якого під час видалення або оновлення запису в батьківській таблиці значення зовнішнього ключа, у відповідних записах підлеглої таблиці, автоматично встановлюється *NULL*. У

результаті підлеглі записи залишаються в таблиці, але втрачають посилання на батьківський запис.

Приклад використання

Наявні дві таблиці «Довідник видавництв» (Publishers) та «Прайс-лист» (PriceList). При чому таблиця Publishers є батьківською по відношенню до PriceList, ключовим полем у цьому зв'язку виступає IdВидавництва.
Якщо видаляємо інформацію з даними про видавництво, книги залишаються в прайс-листі, але без прив'язки до цього видавництва (IdВидавництва = NULL):

```
ALTER TABLE Bookstore.PriceList
ADD CONSTRAINT FK_PriceList_Publishers
 FOREIGN KEY (IdВидавництва) REFERENCES
Bookstore.Publishers(IdВидавництва)
ON DELETE SET NULL;
```

Що відбувається на практиці: при видаленні даних про видавництво з таблиці *Publishers,* в яких поле *IdВидавництва* = 2, у таблиці *PriceList* всі книги цього видавництва приймають значення *IdВидавництва = NULL*. Дані про книги зберігаються, але більше не мають посилання на видалене видавництво.

SET DEFAULT – це режим зовнішнього ключа, який дозволяє підлеглому запису залишатися в таблиці, але отримувати значення за замовчуванням, коли батьківський запис видаляється. Значення за замовчуванням потрібно визначити при створенні колонки.

Приклад використання

Забезпечити, щоб при видаленні даних про жанр з таблиці «Довідник жанрів» (Genres), у таблиці «Прайс-лист» (PriceList) залишаються книги, але їхній IdGenre автоматично встановлюється на значення за замовчуванням
(наприклад, 1 – «Невизначено»):

```
ALTER TABLE PriceList
 ADD CONSTRAINT FK_PriceList_Genres
  FOREIGN KEY (IdGenre)
   REFERENCES Genres(IdGenre)
```

ON DELETE SET DEFAULT
GO

NO ACTION – це режим зовнішнього ключа, який забороняє видаляти або змінювати запис у батьківській таблиці, якщо є пов'язані записи в підлеглій таблиці, тим самим забезпечується максимальний контроль над цілісністю даних.

Приклад використання

Забезпечити унеможливлення видалення даних про клієнта з батьківської таблиці «Довідник клієнта» (Clients), якщо даний клієнт придбав книги у магазині і, відповідно, у підлеглій таблиці «Реалізація книг» (Sales) є про це запис:

```
ALTER TABLE Sales
 ADD CONSTRAINT FK_Sales_Clients
  FOREIGN KEY (IdClient)
   REFERENCES Clients(IdClient)
    ON DELETE NO ACTION;
```

Що відбувається на практиці: при спробі видалити клієнта ввевши SQL-запит:

DELETE FROM Clients WHERE IdClient = 10;

Операція не виконується, а СКБД видає помилку. При цьому інформація про продажі зберігаються, цілісність даних не порушується.

Обмеження *CHECK* застосовується для встановлення умов, яким мають відповідати значення у певному стовпці або комбінації стовпців, запобігаючи внесенню некоректних або нелогічних даних. Це дозволяє забезпечити допустимі діапазони, формати та логічні залежності даних на рівні таблиці.

Приклади використання

Перевірка діапазону цін:

```
CHECK (Price > 0 AND Price <= 10000)
```

Контроль значення року видання:

```
CHECK (YearPublished BETWEEN 1500 AND
       YEAR(GETDATE()))
```

Значення за замовчуванням *DEFAULT* дозволяє автоматично

заповнювати поле стандартним значенням, якщо під час додавання запису воно не було явно вказане, з метою зменшення кількості *NULL*-значень і спрощення введення даних.

Приклади використання

Автоматичне встановлення поточної дати замовлення:

DEFAULT (GETDATE())

Встановлення стандартного статусу для нового клієнта:

DEFAULT ('Active')

Обмеження унікальності UNIQUE гарантує, що значення у певному стовпці або комбінації стовпців не повторюватимуться. Дане обмеження забезпечує унікальність атрибутів, які не є первинним ключем, але мають бути неповторними.

Приклади використання

Заборона повторення електронної пошти клієнта:

UNIQUE (Email)

Контроль унікальності для коду:

UNIQUE (IdBook)

5.3. Створення схеми даних

Схема даних логічний контейнер, за допомогою якого об'єднуються об'єкти в межах однієї бази даних.

Тобто, ***схема даних*** у реляційних СКБД є логічною структурою, що визначає організацію об’єктів бази даних, таких як таблиці, подання, збережені процедури, функції та індекси. Вона виконує роль простору імен, що дозволяє групувати взаємопов’язані об’єкти та забезпечує зручність у керуванні доступом до них. Завдяки використанню схем забезпечується більш чітка структуризація бази даних, розмежування обов’язків між користувачами та підвищується рівень безпеки інформаційної системи.

Правильне використання схем дозволяє:

- 🕮 уникнути конфліктів з ім'ям об'єктів;
- 🕮 ефективно управляти правами користувачів на доступ до даних;
- 🕮 спростити процес супроводу бази даних.

Всі об'єкти бази даних належать тій чи іншій схемі. За замовчуванням, в базі даних створюється одна схема з ім'ям *dbo*. Всі об'єкти, незалежно від того, який користувач їх створив, розміщуються в цій схемі.Так як ця схема є «домашньою», або *схемою за замовчуванням*, для всіх користувачів бази даних, то при створенні об'єктів без явної вказівки схеми (*Schema-Qualified Name*), *SQL Server* автоматично розміщує їх у схемі *dbo*.

У *Microsoft SQL Server* створення схеми здійснюється за допомогою *SQL*-команди *CREATE SCHEMA*, яка дозволяє визначити власника схеми та одразу створити в її межах необхідні об’єкти.

✍ ***SQL-команда CREATE SCHEMA* – створення схеми даних.**
Команда створює нову схему даних.

Синтаксис SQL-команди CREATE SCHEMA

```
CREATE SCHEMA ім’я_схеми
  [AUTHORIZATION власник]
  [об’єкти_схеми];
```

Основні ключові слова і параметри команди CREATE SCHEMA:

ім’я_стовпця – назва поля таблиці;

ім’я_схеми – унікальне ім’я, яке ідентифікує схему в межах бази даних;

AUTHORIZATION власник – визначає користувача або роль, якій належатиме схема;

об’єкти_схеми – необов’язковий блок, що дозволяє створити таблиці, представлення або інші об’єкти безпосередньо під час створення схеми.

Створення схем має низку переваг. По-перше, це дає змогу уникнути конфліктів імен об’єктів, оскільки кожен об’єкт однозначно ідентифікується за допомогою формату *схема.об’єкт* (наприклад, *Sales.Orders*). По-друге, схеми спрощують адміністрування доступу: права можна надавати не окремим таблицям, а всій групі об’єктів, що міститься у схемі.

ПРИКЛАД 5.2. Створення нової схеми даних за допомогою SQL-команди *CREATE TABLE*, яка включає в себе інформацію про реалізацію книг букіністичним магазином. Таблиці нормалізовані і відповідають фізичній моделі БД (приклад побудови фізичної моделі БД букіністичного магазину наведено у практичній роботі №3). Кожна таблиця має приналежність до файлової групи *DB_Book_FG1*, яка була створена у практичній роботі №4 та має додаткові механізми контролю даних *CHECK, DEFAULT, UNIQUE.*

Лістинг 5.2. Створення схеми даних та таблиць для букіністичного магазину

```
USE DB_Book
GO

CREATE SCHEMA Bookstore AUTHORIZATION dbo
GO

-- Створення таблиці «Довідник жанрів»
CREATE TABLE Bookstore.Genres (IdЖанру INT PRIMARY KEY,
ТипЖанру VARCHAR(100) DEFAULT N'Невизначено')
ON FG1;

-- Створення таблиці «Довідник видавництв»
CREATE TABLE Bookstore.Publishers (IdВидавництва INT
PRIMARY KEY, НазваВидавництва VARCHAR(200) NOT NULL
UNIQUE,  АдресаВидавництва VARCHAR(250))
ON FG1;

-- Створення таблиці «Довідник країн»
CREATE TABLE Bookstore.Countries (  IdКраїни INT PRIMARY
KEY,  НайменуванняКраїни VARCHAR(30) NOT NULL
UNIQUE) ON FG1;

-- Створення таблиці «Довідник авторів»
CREATE TABLE Bookstore.Authors (IdАвтора INT PRIMARY KEY,
Прізвище VARCHAR(35) UNIQUE, Імя VARCHAR(35)
UNIQUE, IdКраїни INT FOREIGN KEY REFERENCES
Bookstore.Countries(IdКраїни)) ON FG1;
```

```
-- Створення таблиці «Прайс-лист»
CREATE TABLE Bookstore.PriceList (IdКниги INT PRIMARY KEY,
IdАвтора INT FOREIGN KEY REFERENCES
Bookstore.Authors(IdАвтора), НазваКниги VARCHAR(250) NOT
NULL UNIQUE,  РікВидання INT CHECK (РікВидання >= 1900),
IdЖанру INT FOREIGN KEY REFERENCES
Bookstore.Genres(IdЖанру), IdВидавництва INT FOREIGN KEY
REFERENCES Bookstore.Publishers(IdВидавництва),
Ціна DECIMAL(8,2) CHECK (Ціна >= 0) DEFAULT 0)
ON FG1;

-- Створення таблиці «Довідник клієнтів»
CREATE TABLE Bookstore.Clients (IdКлієнта INT PRIMARY KEY,
НайменуванняКлієнта VARCHAR(200) UNIQUE, АдресаКлієнта
VARCHAR(250),  Телефон NVARCHAR(20) UNIQUE  CHECK
(Телефон LIKE '+38([0-9][0-9][0-9])%' AND Телефон LIKE '%-%'),
МФОБанку NVARCHAR(20),  РР NVARCHAR(30),
ЄДРПОУ NVARCHAR(20) UNIQUE ) ON FG1;

-- Створення таблиці «Реалізація книг»
CREATE TABLE Bookstore.Sales (IdЧеку INT PRIMARY KEY,
IdКлієнта INT FOREIGN KEY REFERENCES
Bookstore.Clients(IdКлієнта), ДатаЗамовлення DATE NOT NULL
DEFAULT GETDATE(), ДатаСплати DATE) ON FG2;

-- Створення таблиці «Вміст реалізації»
CREATE TABLE Bookstore.SalesItems (IdЧеку INT FOREIGN KEY
REFERENCES Bookstore.Sales(IdЧеку), IdКниги INT FOREIGN
KEY REFERENCES Bookstore.PriceList(IdКниги), Кількість INT
CHECK (Кількість > 0) DEFAULT 1) ON FG2;
```

У реляційних базах даних дуже важливим є **правильний порядок створення таблиць**, оскільки між ними існують зв’язки через *первинні* (*PRIMARY KEY*) та *зовнішні ключі* (*FOREIGN KEY*). Якщо створити таблицю з зовнішнім ключем раніше, ніж таблицю, на яку він посилається тобто з наявним первинним ключем, *SQL Server* видасть помилку.

Логіка формування порядку:

1. Спочатку створюються незалежні довідники – таблиці, які не мають зовнішніх ключів і на які інші таблиці посилаються, тобто містять лише

одне ключове поле *PRIMARY KEY*. Це так звані *батьківські таблиці*. Наприклад: таблиці *Genres, Publishers, Countries, Clients*.

2. Далі створюються залежні довідники, які окрім первинних ключів *PRIMARY KEY*, містять також зовнішні ключі *FOREIGN KEY* із директивою *REFERENCES*, що забезпечує посилальну цілісність даних та встановлює зв'язки з раніше створеними таблицями. Наприклад: таблиця *Authors* (має зв'язок із таблицею *Countries*).
3. Потім створюються фактологічні таблиці (операційні), які посилаються одночасно на кілька довідників. Наприклад: таблиця *PriceList* (посилання на таблиці *Authors, Genres, Publishers*).
4. Останніми створюються транзакційні таблиці, які описують події чи операції та обов'язково мають зовнішні ключі, які посилаються на довідники й фактологічні таблиці. Наприклад: *Sales* (посилання на *Clients*), а вже після нього – *SalesItems* (посилання на *Sales* та *PriceList*).

✍ *Рекомендації щодо найменування об'єктів бази даних та роботи з датами*

Правильне проєктування назв об'єктів бази даних та коректне зберігання дат є важливою складовою підтримки цілісності, зрозумілості та ефективності роботи БД. Основні рекомендації:

- 🕮 Використовуйте, що чітко відображає зміст даних (наприклад, *Books, Authors, Sales*).
- 🕮 Якщо назва складається з двох або більше слів, застосовуйте нижнє підкреслення (_), наприклад: Price_List, а не Price List чи Price.List.
- 🕮 Уникайте використання точок у назвах, оскільки вони можуть сприйматися як роздільники між схемою та базою даних.
- 🕮 Пишіть назви об'єктів з великих літер (наприклад, *PriceList*), щоб уникнути проблем у середовищах, де імена чутливі до регістру.
- 🕮 Використовуйте формат *DATETIME* або *DATE* для зберігання дат. Це дозволяє оптимізувати запити та забезпечує правильне сортування, фільтрацію та обчислення.
- 🕮 Не зберігайте дати у вигляді рядків *VARCHAR*, оскільки це ускладнює обробку, збільшує ризик помилок та знижує продуктивність.
- 🕮 Не розділяйте дату на кілька колонок (*YEAR, MONTH, DAY*), оскільки це ускладнює виконання запитів та операцій агрегування.

Учасники ролі *sysadmin і* власники бази даних є користувачами бази з ім'ям *dbo* і схемою за замовчуванням *dbo*.Створені ними об'єкти розміщуються в цій схемі за замовчуванням, якщо не вказано інше.

Лайфхаки для ефективної роботи SQL

Додаючи стовпець з обмеженням NOT NULL для всіх полів таблиці, розробник або адміністратор БД повинні врахувати ряд обставин:

- ✓ *спочатку потрібно створити стовпець без обмеження, а потім ввести значення в усі його рядки*
- ✓ *після того як всі значення стовпця стануть не NULL-значення, до нього можна застосувати обмеження NOT NULL.*

Якщо з обмеженням NOT NULL користувач намагається додати новий рядок для внесення даних, то повертається повідомлення про помилку.
Помилка вказує на те, що або таблиця повинна бути порожньою, або в стовпці повинні міститися значення для кожного існуючого рядка.
Нагадаємо, що після накладення на стовпець обмеження NOT NULL в ньому не можуть бути присутніми NULL-значення ні в одній з існуючих рядків.

5.4. Обробка даних у таблицях

Обробка даних у таблицях баз даних – це комплекс операцій, які забезпечують введення, зміну, видалення, сортування, фільтрацію та перегляд інформації, що зберігається у структурованому вигляді. У реляційних базах даних таблиці є основною одиницею організації даних, і обробка інформації відбувається безпосередньо на рівні таблиць та їхніх стовпців.

✍ ***SQL-команда INSERT* – додавання рядку записи до таблиці.**

Операція додавання використовується для вставки нових рядків у таблиці. Вона може виконуватися у двох основних формах:

- із явним зазначенням усіх стовпців та значень;
- із зазначенням лише необхідних стовпців, при цьому для решти застосовуються значення за замовчуванням (*DEFAULT*) або допускається *NULL*.

Таким чином, оператор *INSERT* дозволяє поступово наповнювати таблиці фактичними даними з урахуванням обмежень, накладених первинними та зовнішніми ключами, а також перевірками цілісності (*CHECK, UNIQUE*).

Синтаксис SQL-команди INSERT

```
INSERT INTO table_name (column_name1, ... column_nameN)
VALUES (expressions, ...)
GO
```

Основні ключові слова і параметри команди INSERT:

table_name– ім'я таблиці, в яку повинні бути вставлені рядки; якщо зазначене подання, то рядки вставляються в основну таблицю;

column_name1– стовпець таблиці або подання, в який для кожного вставленого рядку вводиться значення з фрази *VALUES* або підзапиту; якщо один з стовпців таблиці опускається з цього списку, значенням стовпця для вставленого рядка є значення за замовчуванням стовпчика, певне при створенні таблиці.

Якщо повністю опускається список стовпчика, пропозиція *VALUES* або запит повинен визначити значення для всіх стовпців в таблиці.

VALUES– визначає рядок значень, які будуть вставлені в таблицю або подання; значення має бути визначено в реченні *VALUES* для кожного стовпця в списку стовпців.

Так само, значення можуть бути записані і без вказівки стовпців:

```
INSERT INTO table_name
VALUES (expressions, ...)
GO
```

ПРИКЛАД 5.3. У таблицю *«Довідник клієнта»* необхідно ввести дані про нового клієнта.

Лістинг 5.3. Додавання записи про нового клієнта у таблицю «Довідник клієнта»

```
USE DB_Book
GO
INSERT INTO Bookstore.Clients (IdКлієнта,
НайменуванняКлієнта, АдресаКлієнта, Телефон, МФОБанку, РР,
ЄДРПОУ)
VALUES (777, 'Знайка', 'вул.Хмельницька, 3б', '0442796438',
'121121', '123456789123', '5025025')
GO
```

У *SQL Server* різні типи даних вимагають різного синтаксису при вставці значень у таблицю. Числові типи, такі як *INT*, *DECIMAL* або *FLOAT*, представляють собою числа, над якими СКБД може виконувати математичні операції. Такі значення не беруться в апострофи, оскільки база даних розуміє їх як числа без додаткових позначень. У прикладі *IdКлієнта* = 777 означає число цілого типу *INT*, тому лапки не потрібні.

Натомість текстові дані, що зберігаються у стовпцях типу *NVARCHAR, VARCHAR, CHAR, NCHAR* або *TEXT* обов'язково беруться в одинарні апострофи '...'. Це дозволяє *SQL* чітко відокремлювати текст від команд та чисел. Наприклад, *'Знайка'* або *'вул.Хмельницька, 3б'* – це рядки символів, які потрібно трактувати як текстові значення.

Іноді числові значення зберігають у текстових полях, наприклад, телефон, МФО банку або ЄДРПОУ, щоб зберегти форматування. У цьому випадку навіть цифри беруться в лапки, бо колонка визначена як *NVARCHAR VARCHAR, CHAR* або *NCHAR*. Наприклад, '0442796438' – це цифри, але зберігаються як текст, а не як числа або ідентифікатори і SQL потребує апострофів.

Таким чином, правило просте: числа у числових стовпцях записуються без одинарних лапок, а всі текстові значення у текстових стовпцях завжди беруться в апострофи, навіть якщо вони складаються лише з цифр. Це забезпечує правильну обробку даних і уникнення помилок під час вставлення даних.

ПРИКЛАД 5.4. Якщо потрібно вставити *NULL*-значення, необхідно вказати його як звичайне значення наступним чином:

Лістинг 5.4. Додавання записи з NULL-значенням

```
USE DB_Book
```

```
GO
INSERT INTO Bookstore.Authors
VALUES (88, 'Хенкс', NULL, 5);
GO
```

✍ *SQL-команда UPDATE* – змінює (оновлює) записі в таблиці SQL

Операція оновлення застосовується для модифікації наявних записів у таблицях. Вона дозволяє змінювати значення одного або кількох стовпців, використовуючи умовні вирази (*WHERE*) для вибіркового впливу лише на ті рядки, які відповідають визначеним критеріям.

Правильне застосування *UPDATE* є важливим для уникнення випадкової модифікації всієї таблиці. Додатково можуть застосовуватися обмеження тригерів (*TRIGGERS*), які контролюють допустимість змін або забезпечують каскадне оновлення пов'язаних таблиць.

Синтаксис SQL-команди UPDATE

```
UPDATE table_name
SET column_name expression
WHERE condition
GO
```

Основні ключові слова і параметри команди UPDATE:

table_name – ім'я таблиці *SQL*, в якій змінюються дані;

column_name – стовпець таблиці *SQL* або подання *SQL*, значення якого змінюється; якщо стовпець таблиці з пропозиції *SET* опускається, значення стовпця залишається незмінним;

expression – нове значення, яка призначається відповідним стовпцем; це вираз може містити головні змінні і необов'язкові індикаторні змінні;

WHERE – визначає діапазон змінюваних значень тим рядкам, для яких умова *condition*риймає твердження *TRUE*; якщо опускається ця фраза, тоді модифікуються всі рядки в таблиці або поданні.

Заміна значення стовпця в усіх рядках таблиці, як правило, використовується рідко. Тому в команді *UPDATE*, як і в команді *DELETE*, можна використовувати предикат *WHERE*, який дозволяє задати умову відбору рядків для модифікації чи видалення. Використання предиката забезпечує вибірковість обробки даних і запобігає масовим змінам у всій таблиці, що могло б призвести до втрати актуальності або узгодженості інформації.

Цікавий факт:

Предикат (від лат. Praedicare – проголошувати, заявляти, присуджувати) у сучасній логіці зазвичай означає булеву функцію.

Якщо предикат *WHERE* не вказати, команда буде застосована до всіх рядків таблиці без винятку. Це може бути корисним лише у виняткових випадках, коли необхідно одночасно оновити або видалити всі записи. Проте в більшості ситуацій коректніше формулювати чіткі умови, що визначають конкретні рядки для обробки.

Умови предиката можуть включати як прості порівняння (наприклад, *WHERE Id* = 10), так і складні логічні вирази з використанням операторів *AND*, *OR*, *NOT*, а також підзапитів (*Subqueries*) чи функцій агрегування. Це робить команди *UPDATE* і *DELETE* гнучкими інструментами для підтримки цілісності та актуальності даних у реляційних таблицях (більш детальну інформацію про предикат *WHERE* буде розглянуто у наступній темі).

ПРИКЛАД 5.5. У таблиці «Прайс-лист» змінити значення поля ціна на 380 для коду книги, який дорівнює 1006

Лістинг 5.5. Зміна значення ціни для обраної книги

```
USE DB_Book
GO
UPDATE Bookstore.PriceList
SET   Ціна = 380
WHERE IdКниги = 1006;
```

ПРИКЛАД 5.6. Зміна рейтингу для всіх покупців на значення, рівне 200.

Лістинг 5.6. Зміна значення рейтингу

```
UPDATE Customers
SET rating = 200
GO
```

ПРИКЛАД 5.7. У реченні *SET* можна вказати будь-яку кількість значень для стовпців, розділених комами:

Лістинг 5.7. Зміна декілька значень для різних стовпців

```
UPDATE emp
SET job = 'MANAGER', sal = sal +1000, deptno = 20
WHERE ename = 'JONES'
GO
```

ПРИКЛАД 5.8. У реченні SET можна вказати значення NULL без використання будь-якого спеціального синтаксису (наприклад, такого як IS NULL).

Лістинг 5.8. Встановлення значень рейтингу покупців, які проживають у місті Лондон рівними NULL.

```
UPDATE Customers
SET rating = NULL
WHERE city = 'London'
GO
```

✍ *SQL-команда DELETE* – видалення записі з таблиці

Операція видалення дозволяє вилучати записи з таблиці. Як і у випадку з оновленням, застосування умовного оператора *WHERE* є обов'язковим для безпечної роботи, інакше команда *DELETE* очистить усю таблицю. У разі наявності зовнішніх ключів система може забороняти видалення записів, якщо на них існують посилання в інших таблицях, або ж реалізовувати каскадне видалення *CASCADE DELETE*. Це забезпечує узгодженість даних у реляційній моделі.

Синтаксис SQL-команди DELETE

```
DELETE FROM table_name
WHERE < condition>
GO
```

Основні ключові слова і параметри команди DELETE:

WHERE–видаляє тільки рядки, які задовольняють умові;
condition–умова може посилатися на таблицю і містити підзапит.

ПРИКЛАД 5.9. Видалення даних з таблиці *Klient*, код клієнта дорівнює 777.

Лістинг 5.9. Видалення значень, в яких код клієнта = 777

```
USE DB_Book
   GO
DELETE FROM Bookstore.Clients
WHERE IdKlient = 777
   GO
```

ПРИКЛАД 5.10. Видаленнявсіх рядків з таблиці *«Прайс-лист»*

Лістинг 5.10. Видалення всіх значень

```
USE DB_Book
   GO
DELETE FROM [dbo].[ Book]
GO
```

ПРИКЛАД 5.11. Видалити з таблиці всіх продавців, у яких комісійні менше 100 у.о. на місяць.

Лістинг 5.11. Видалення інформації про продавців, комісійні яких менше 100 у.о.

```
DELETE FROM emp
WHERE Job = 'Salesman' AND Comm <100;
GO
```

В даному прикладі команда *DELETE* видаляє всі рядки, які потрапляють під умову *Job = 'Salesman' AND Comm <100*.

Попередній приклад можна записати по-іншому:

```
DELETE FROM (select * from emp)
WHERE Job = 'Salesman' AND Comm <100
GO
```

5.5. Модифікація структури таблиць

У процесі експлуатації бази даних виникає потреба змінювати структуру вже існуючих таблиць. Це може бути пов’язано з появою нових вимог до інформаційної системи, необхідністю оптимізації зберігання даних,

уточненням атрибутів або забезпеченням цілісності. Для внесення таких змін у *SQL Server* використовується команда *ALTER TABLE*.

✍ *SQL-команда ALTER TABLE* – модифікація структури таблиці

Команда *ALTER TABLE* змінює визначення таблиці одним із таких способів:

- додає стовпець;
- видаляє стовпець;
- перевизначає стовпець (тип даних, розмір, замовчувана значення);
- модифікує характеристики пам'яті або інші параметри;
- додає,включає, вимикає або видаляє обмеження цілісності або тригер.

УМОВА! Таблиця повинна бути в схемі користувача, або користувач повинен мати системну привілей *ALTER ANY TABLE*.

Для включення системного привілею *ALTER ANY TABLE* необхідно викликати контекстно-залежне меню (п. к. м.) *DB_Book* та обрати команду *Properties*

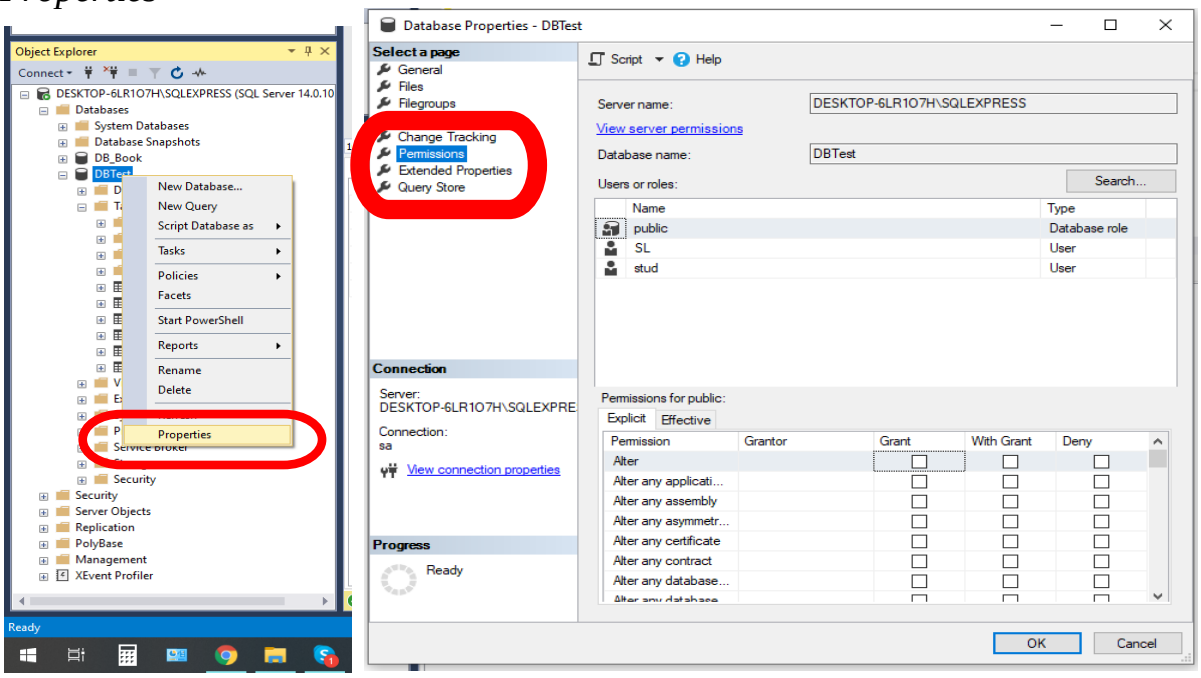


Рис. 5.1. Етапи включення системного привілею

У розділі *Select a page* обрати команду *Permissions*. Далі включити такі дозволи *Alter any assembly*т а *Alter any data space*.

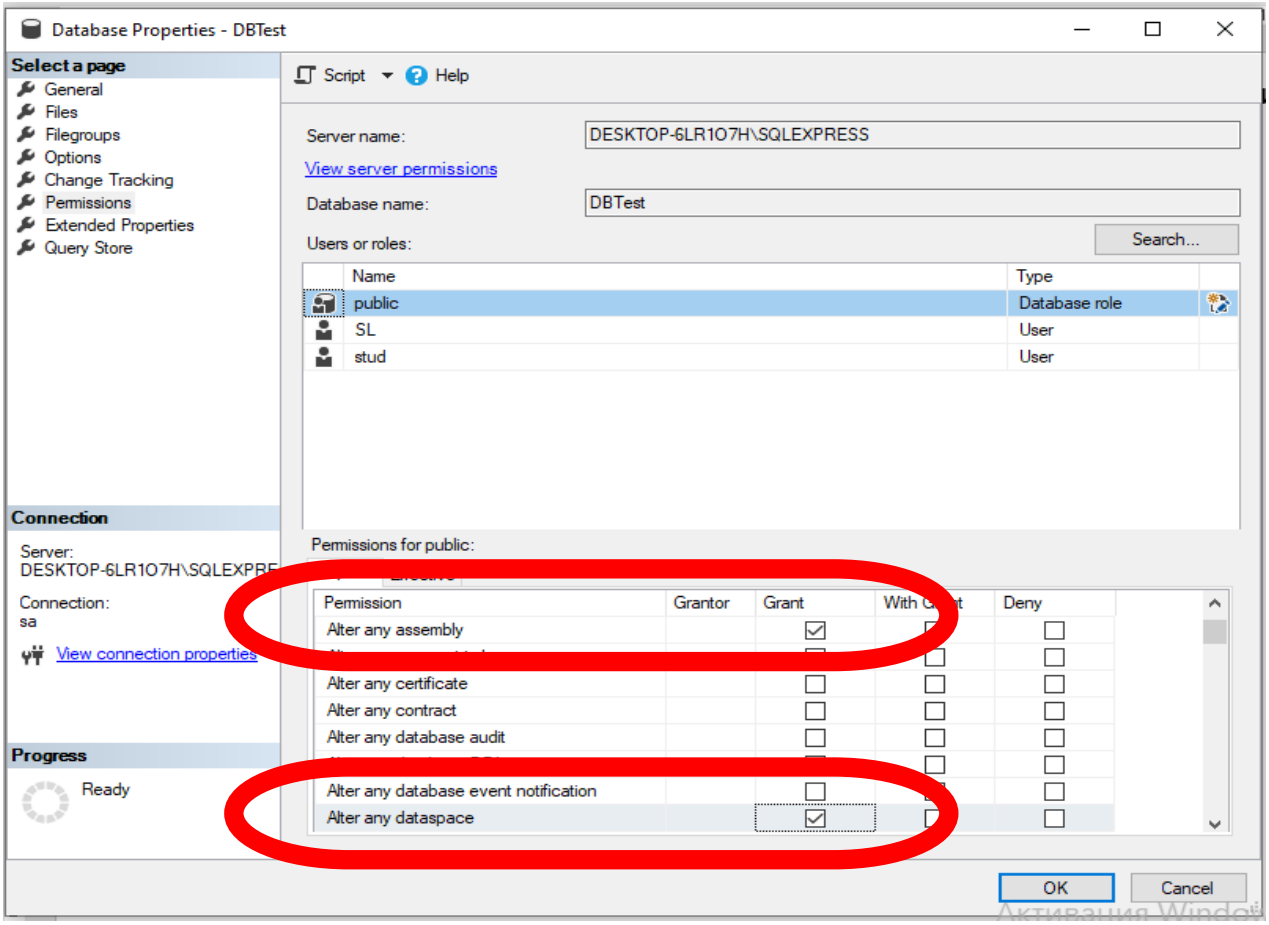


Рис. 5.2. Включення дозволів на зміни

Змінюючи типи даних існуючих стовпців або додаючи стовпчики в таблицю бази даних, потрібно дотримуватися ряду умов.

Під час зміни структури вже існуючих таблиць у базі даних (наприклад, шляхом зміни типів даних або додавання нових стовпців) слід враховувати низку обмежень, зумовлених як логікою збереження даних, так і технічними можливостями СКБД.

Допустимі розширення (safe changes) – це такі модифікації структури бази даних, які не спричиняють втрату наявних даних і, як правило, не потребують виконання додаткових умов або обмежень. До цієї категорії належать:

- збільшення розміру стовпців символьних типів (CHAR, NCHAR, VARCHAR, NVARCHAR), наприклад, розширення VARCHAR(50) до VARCHAR(100);

- додавання нових стовпців до таблиці**,** за умови що вони допускають значення NULL або мають визначене значення за замовчуванням (DEFAULT).

Допустимі, але обмежені зміни (restricted changes) – це такі модифікації структури бази даних, які дозволяються лише за дотримання певних умов. До них належать:

- зменшення розміру стовпців символьних типів (CHAR, NCHAR, VARCHAR, NVARCHAR), що можливе тільки у випадках, коли стовпець не містить значень (усі записи є NULL), або усі наявні дані гарантовано вміщуються у новий розмір;
- зміна типу даних стовпця, яка допускається за умови, що стовпець є порожнім (усі значення NULL), або існує явна та безпечна конверсія між типами (наприклад, з INT у BIGINT, з DECIMAL(5,2) у DECIMAL(7,2) тощо).

Обмеження:

Зміни у структурі бази даних, які не допускаються або вимагають особливої обережності, оскільки можуть призвести до втрати чи пошкодження даних. До таких змін належать:

- ✓ *зміна типу даних стовпця, що може спричинити втрату інформації (наприклад, перехід із VARCHAR(100) на VARCHAR(10) за наявності довших значень);*
- ✓ *заміна числового типу на символьний або навпаки без явного перетворення даних;*
- ✓ *видалення стовпців, якщо перед цим не виконано перенесення або архівування важливої інформації.*

Синтаксис SQLкоманди ALTER TABLE додати рядок в таблиці:

```
ALTER TABLE table_name
ADD column_name datatype
GO
```

Синтаксис SQLкоманди ALTER TABLE змінити стовпця в таблиці:

```
ALTER TABLE table_name
ALTER COLUMN column_a datatype
 GO
```

Синтаксис SQLкоманди ALTER TABLE видалення стовпця таблиці:

```
ALTER TABLE table_name
 DROP COLUMN table_name
 GO
```

Якщо при створенні таблиці бази даних не був визначений первинний ключ *PRIMARY KEY*, то це може бути зроблено за допомогою команди *ALTER TABLE*.

СинтаксисSQLкоманди ALTER TABLE в разі додавання простого первинного ключа:

```
ALTER TABLE table_name
ADD PRIMARY KEY (id_name)
GO
```

✍ ***SQL-команда DROP*–видалення.**
Команда видаляє таблиці ***TABLE***, індекси ***INDEX*** і бази даних ***DATABASE***.

Зазвичай з таблицею в базі даних пов'язано кілька об'єктів, наприклад індекс, створюваний первинним ключем, або обмеження *UNIQUE* , що накладається на стовпці таблиці.

При видаленні таблиці автоматично видаляє і будь-який пов'язаний з нею індекс. Для видалення таблиці з БД необхідно виконати команду *DROP TABLE*.

СинтаксисSQL-команди DROP TABLE:

```
DROP TABLE table_name
```

GO

Однак видалити таблицю не завжди настільки просто. У будь-який момент ми можемо створити таблицю з обмеженнями цілісності. Обмеження цілісності *Integrityconstraint*–це правило, яке встановлюється для таблиці і обмежує тип даних, які можна вводити в цю таблицю.

Якщо спробувати видалити таблицю з обмеженнями цілісності, повертається повідомлення про помилку наступного виду: «Unique / Primary keys in table referenced by foreign keys» (первинні або унікальні ключі таблиці використовуються у зовнішніх ключах інших таблиць).

Коли існують обмеження для інших таблиць, на які посилається видаляється таблиця, можна користуватися каскадної конструкцією *CASCADECONSTRAINTS* :

Синтаксис SQL-команди DROP TABLE видалення таблиці з обмеженнями цілісності:

DROP TABLE table_name CASCADE CONSTRAINTS

GO

DROP DATABASE видаляє одну або кілька користувацьких баз даних або моментальних знімків бази даних з екземпляра *SQL Server.*

СинтаксисSQL-команди DROP DATABASE видалення бази даних:

DROP DATABASE database

GO

DROP INDEX використовується для видалення індексів в таблиці.Коли індекс в базі даних більше не потрібен, розробник може видалити його командою ***DROP INDEX***.

Синтаксис оператора ***DROP INDEX*** однаковий для видалення індексу будь-якого типу (унікальності, бітової карти або B-дерева). Щоб якимось чином поліпшити індекс, потрібно спочатку видалити його, а потім створити новий.

Синтаксис SQL-команди DROP INDEX видалення індексу:

```
DROP INDEX index_name
ON {database_name.schema_name.table_name | schema_name.table_name | table_name }
GO
```

Основні ключові слова і параметри команди DROP INDEX

index_name – ім'я індексу, який необхідно видалити;

database_name – ім'я бази даних;

schema_name– ім'я схеми даних, якій належить таблиця або представлення;

table_or_view_name – ім'я таблиці або представлення, пов'язаних з індексом. Просторові індекси підтримуються тільки для таблиць.

Щоб відобразити звіт за індексами об'єкта, слід скористатися поданням каталогу *sys.indexes*.

Цікавий факт:

Після видалення індексу ефективність пошуку з використанням стовпця або стовпців, обмежених індексом, більше не підвищується і згадка про індекс зникає зі словника даних. Індекс, що застосовується для первинного ключа, видалити не можна.

5.6. Засоби перевірки та контролю даних

Після створення структури бази даних і заповнення її початковими даними виникає потреба у постійній перевірці якості, достовірності та узгодженості інформації, що зберігається в таблицях. З часом унаслідок помилок користувачів, збоїв у процесах завантаження або інтеграції з іншими системами у базі можуть з'являтися некоректні, суперечливі або застарілі дані. Щоб запобігти цьому, у *Microsoft SQL Server* передбачено комплекс засобів контролю та перевірки даних, які застосовуються вже під час експлуатації бази.

Передусім *SQL Server* підтримує механізми перевірки цілісності даних на фізичному та логічному рівнях. До фізичних належать регулярні перевірки стану сторінок бази, цілісності індексів та зв'язності сторінок за допомогою службових команд, таких як *DBCC CHECKDB, DBCC CHECKTABLE і DBCC CHECKALLOC*. Вони дозволяють виявляти пошкоджені сторінки, порушення внутрішніх зв'язків і логічні невідповідності у структурі таблиць та індексів.

Виконання цих перевірок є обов'язковим елементом профілактичного обслуговування БД.

Для логічного контролю вмісту даних застосовуються запити перевірки якості (*Data Validation Queries*), що дозволяють виявити аномалії, дублікати, пропущені значення, невірні коди або порушення довідкових зв'язків. Як правило, ці запити створюються адміністраторами або аналітиками та регулярно виконуються як частина плану обслуговування. Їх доцільно автоматизувати за допомогою *SQL Server Agent* або інтегрувати у пакети *SQL Server Integration Services (SSIS),* які реалізують завдання з очищення та трансформації даних під час завантаження ззовні.

Важливим засобом контролю є моніторинг порушень обмежень і помилок при транзакціях**,** що дозволяє оперативно виявляти спроби збереження некоректних даних. Для цього використовують вбудоване журналювання *SQL Server*, а також створюють тригери аудиту, які фіксують усі невдалі спроби змінити дані та передають відомості в окремі таблиці журналу. Надалі ці журнали аналізуються для виявлення причин порушення правил введення.

Ще одним компонентом системи перевірки **є** служби забезпечення якості даних, зокрема *Data Quality Services (DQS)* і *Master Data Services (MDS)*. *DQS* застосовується для автоматизованого виявлення та виправлення помилок у даних (дедуплікації, виправлення форматів, усунення пропусків), а *MDS* дозволяє централізовано керувати довідниковими даними та підтримувати їх у актуальному та узгодженому стані між різними базами і прикладними системами. Їх використання особливо доцільне у великих організаціях із багатьма джерелами даних.

У комплексі із цими засобами застосовуються також регулярні аудити та ревізії вмісту БД**.** Для цього розробляються контрольні звіти, які порівнюють фактичні дані з нормативними значеннями, перевіряють відповідність бізнес-правилам, коректність зв'язків і повноту заповнення полів. Результати таких перевірок документуються, а виявлені помилки виправляються засобами транзакційного оновлення, щоб не порушити цілісність бази.

Таким чином, засоби перевірки та контролю даних у *SQL Server* формують багаторівневу систему забезпечення якості**,** яка поєднує автоматичні механізми СКБД (перевірки цілісності, тригери, служби якості даних) із організаційними процедурами (аудит, планове тестування та моніторинг). Використання цих засобів дозволяє підтримувати дані у надійному, узгодженому та аналітично придатному стані протягом усього життєвого циклу бази.

Питання для самоперевірки

1. *Дайтевизначення схемі даних.*
2. *Скільки схем даних за замовчуванням створюються в базі?*
3. *Яка команда SQL створює схему даних?*
4. *У чому полягає відмінність між ключовими полями PRIMARY KEY і FOREIGN KEY?*
5. *Яку опцію підтримує команда CREATE TABLE?*
6. *Яка команда додає рядки в таблицю? Назвіть синтаксис команди.*
7. *Опишіть синтаксис команди, яка змінює існуючі значення в таблиці або в основній таблиці уявлення.*
8. *Яка команда видаляє рядки з таблиці або представлення основної таблиці бази даних?*
9. *Синтаксис команди, що змінює колонки таблиці.*
10. *Перерахуйте способи, якими команда ALTER TABLE змінює визначення таблиці.*
11. *Опишіть синтаксис команди, яка додає простий первинний ключ у таблицю.*
12. *Які об'єкти видаляє SQL-команда* **DROP.**

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 5

на тему: «Створення та модифікація таблиць у схемі бази даних усередовищі MS SQL Server Management Studio»

Мета: *набуття практичних навичок у виконанні операцій з обробки даних у таблицях реляційних баз даних, включаючи створення таблиць, додавання, оновлення та видалення записів, а також у застосуванні команд для зміни структури таблиць та видалення об'єктів бази даних.*

План лабораторної роботи

1. У середовищі *MS SQL Server Management Studio* створити схему даних та таблиць у ній.
2. Заповнення таблиць даними за допомогою команди *INSERT INTO*.
3. Оновлення даних у таблицях із використанням команди *UPDATE*.
4. Видалення записів із таблиць за допомогою команди *DELETE*.
5. Модифікація структури таблиць *ALTER TABLE*.
6. Створити звіт про хід виконання лабораторної роботи.

Хід виконання лабораторної роботи

У середовищі *MS SQL Server Management Studio* створення схеми здійснюється за допомогою *SQL*-команди *CREATE SCHEMA*. Прикладом може бути створення схеми *Bookstore*, що використовується для об'єктів, пов'язаних з інформаційною системою книжкового магазину:

```
CREATE SCHEMA Bookstore AUTHORIZATION dbo;
```

Після визначення схеми відбувається поетапне створення таблиць у ній. Послідовність формування структури ґрунтується на принципах реляційного проєктування:

- спочатку створюються довідникові таблиці, які містять лише первинні ключі та не мають зовнішніх залежностей (наприклад, *Genres, Countries, Publishers, Clients*);

- наступним кроком формуються залежні довідники, що включають як власні первинні ключі, так і зовнішні ключі для зв'язку з базовими довідниками (наприклад, *Authors*, який посилається на *Countries*);
- далі визначаються операційні таблиці, що поєднують кілька зовнішніх ключів і описують предметну область більш детально (наприклад, *PriceList*, який містить посилання на авторів, жанри та видавництва);
- завершальним етапом є створення транзакційних таблиць, які відображають фактичні операції, зокрема реалізацію книг (*Sales*) та деталізацію продажів (*SalesItems*).

У процесі створення таблиць застосовуються ключові механізми забезпечення цілісності даних: *PRIMARY KEY* для унікальної ідентифікації рядків, *FOREIGN KEY* для встановлення логічних зв'язків між таблицями, *CHECK* для перевірки коректності введених значень, *DEFAULT* для автоматичного призначення типових параметрів, *NOT NULL* для заборони порожніх значень, а також *UNIQUE* для гарантування неповторюваності окремих атрибутів.

Першою створемо таблицю «Довідник жанрів» у межах схеми *Bookstore*. Таблиця призначена для зберігання інформації про літературні жанри, що використовуються в інформаційній системі.

```
-- Створення таблиці «Довідник жанрів»
CREATE TABLE Bookstore.Genres (IdЖанру INT PRIMARY KEY,
НазваЖанру NVARCHAR(100) NOT NULL UNIQUE, ТипЖанру
NVARCHAR(100) DEFAULT N'Невизначено') ON FG1;
```

CREATE TABLE Bookstore.Genres – команда створення нової таблиці з ім'ям *Genres*, яка належить до схеми *Bookstore*. Використання схеми дозволяє впорядкувати структуру бази даних і відокремити логічно пов'язані об'єкти.

IdЖанру INT PRIMARY KEY – оголошується поле *IdЖанру* як ціле число (*INT*), що виконує роль первинного ключа. Первинний ключ гарантує унікальність кожного запису та забороняє значення *NULL* та слугує ідентифікатором жанру й забезпечує можливість встановлення зв'язків із іншими таблицями за допомогою зовнішніх ключів.

НазваЖанру *NVARCHAR(100) NOT NULL UNIQUE* – поле для зберігання назви жанру, використовується тип *NVARCHAR*(*100*), що дозволяє зберігати текстові значення до 100 символів у кодуванні *Unicode*, що важливо для підтримки багатомовності. Обмеження *NOT NULL* забороняє створення запису без вказівки назви жанру. Обмеження *UNIQUE* забезпечує унікальність

значень у цьому стовпці, тобто одна й та сама назва жанру не може повторюватися.

ТипЖанру *NVARCHAR(100) DEFAULT N'Невизначено'* – поле для уточнення типу жанру, використовується символьний тип *NVARCHAR*(*100*). Додано значення за замовчуванням *DEFAULT N'Невизначено'*. Це означає, що якщо під час вставки нового рядка значення цього поля не буде явно вказане, система автоматично запише текст «Невизначено».

ON FG1 – визначає файлову групу, у якій фізично розміщуватиметься таблиця. У даному випадку таблиця зберігається на файловій групі FG1. Це налаштування використовується для розподілу даних між різними фізичними носіями та може підвищувати ефективність роботи бази даних.

За таким принципом створюємо наступні дві таблиці «Довідник видавництв» та «Довідник країн».

```
-- Створення таблиці «Довідник видавництв»
    CREATE TABLE Bookstore.Publishers (IdВидавництва INT PRIMARY KEY,
     НазваВидавництва NVARCHAR(200) NOT NULL UNIQUE,
     АдресаВидавництва NVARCHAR(250)) ON FG1;

-- Створення таблиці «Довідник країн»
    CREATE TABLE Bookstore.Countries ( IdКраїни INT PRIMARY KEY,
     НайменуванняКраїни NVARCHAR(150) NOT NULL UNIQUE) ON FG1;
```

Далі створюємо таблицю **«**Довідник авторів**»** у межах схеми *Bookstore*, яка призначена для зберігання інформації про авторів книжкової продукції та містить такі поля. В ній, окрім первиного ключа *PRIMARY KEY* оголошується зовнішній ключ, який встановлює зв’язок з таблицею *Bookstore.Countries*.
IdКраїни INT FOREIGN KEY REFERENCES
Bookstore.Countries(IdКраїни)

Це означає, що для кожного автора обов’язково повинна бути вказана країна походження, яка вже наявна у довіднику країн. Таким чином забезпечується референційна цілісність даних.

```
-- Створення таблиці «Довідник авторів»
    CREATE TABLE Bookstore.Authors (IdАвтора INT PRIMARY KEY,
     Прізвище NVARCHAR(35) UNIQUE, Імя NVARCHAR(35) UNIQUE,
     IdКраїни INT FOREIGN KEY REFERENCES Bookstore.Countries(IdКраїни))
      ON FG1;
```

Далі створюємо таблицю **«Прайс-лист»** у межах схеми *Bookstore*, яка призначена для зберігання інформації про книжковий асортимент магазину,

включно з авторами, видавництвами, жанрами та цінами на книги. Таблиця містить такі поля:

- первинний ключ таблиці, який забезпечує унікальну ідентифікацію кожної книги у прайс-листі *IdКниги INT PRIMARY KEY;*
- зовнішній ключ, який встановлює зв'язок із таблицею *Bookstore.Authors*. *IdАвтора INT FOREIGN KEY REFERENCES Bookstore.Authors(IdАвтора)*

  Це означає, що для кожної книги повинен бути зазначений автор, який вже існує у довіднику авторів, що забезпечує референційну цілісність даних;
- обов'язкове поле назва книги, значення якого не може повторюватися, оскільки кожна книга у каталозі повинна мати унікальну назву *НазваКниги VARCHAR(250) NOT NULL UNIQUE*;
- поле рік видання книги має обмеження CHECK: *РікВидання INT CHECK (РікВидання >= 1900)* – гарантує, що значення не буде меншим за 1900 рік і відповідатиме часовим межам реального книжкового асортименту;
- зовнішній ключ *IdЖанру INT FOREIGN KEY REFERENCES Bookstore.Genres(IdЖанру)*, який пов'язує книгу з відповідним жанром у таблиці *Bookstore.Genres*. Це дозволяє класифікувати книги за типом жанру.
- зовнішній ключ *IdВидавництва INT FOREIGN KEY REFERENCES Bookstore.Publishers(IdВидавництва),* що встановлює зв'язок із таблицею *Bookstore.Publishers*. Завдяки цьому для кожної книги вказується видавництво, яке її опублікувало.
- поле для збереження вартості книги з точністю до копійок *Ціна DECIMAL(8,2) CHECK (Ціна >= 0) DEFAULT 0 в якому* передбачено перевірку на невід'ємність ціни та значення за замовчуванням – 0.

```
-- Створення таблиці «Прайс-лист»
CREATE TABLE Bookstore.PriceList (IdКниги INT PRIMARY KEY,
IdАвтора INT FOREIGN KEY REFERENCES
Bookstore.Authors(IdАвтора), НазваКниги VARCHAR(250) NOT
NULL UNIQUE,  РікВидання INT CHECK (РікВидання >= 1900),
IdЖанру INT FOREIGN KEY REFERENCES
Bookstore.Genres(IdЖанру), IdВидавництва INT FOREIGN KEY
REFERENCES Bookstore.Publishers(IdВидавництва),
Ціна DECIMAL(8,2) CHECK (Ціна >= 0) DEFAULT 0)
ON FG1;
```

Наступною створимо таблицю «Довідник клієнтів» у межах схеми *Bookstore*, яка призначена для зберігання інформації про клієнтів букіністичного магазину, включно з їхніми контактними та банківськими реквізитами.

У таблиці *IdКлієнта INT PRIMARY KEY* є первинним ключем, що забезпечує унікальну ідентифікацію кожного клієнта. Поле *НайменуванняКлієнта VARCHAR(200) UNIQUE* використовується для збереження назви клієнта, при цьому обмеження *UNIQUE* запобігає дублюванню записів.

Поле *Телефон NVARCHAR(20)* має одразу два обмеження: *UNIQUE* – для унеможливлення повторень телефонних номерів, і *CHECK (Телефон LIKE '+38([0-9][0-9][0-9])%' AND Телефон LIKE '%-%'),* яке забезпечує дотримання формату українського номера телефону. Також поле *ЄДРПОУ NVARCHAR(20) UNIQUE* гарантує унікальність коду в межах таблиці, що є обов'язковою вимогою для юридичних осіб.

```
-- Створення таблиці «Довідник клієнтів»
CREATE TABLE Bookstore.Clients (IdКлієнта INT PRIMARY KEY,
НайменуванняКлієнта VARCHAR(200) UNIQUE, АдресаКлієнта
VARCHAR(250),  Телефон NVARCHAR(20) UNIQUE  CHECK
(Телефон LIKE '+38([0-9][0-9][0-9])%' AND Телефон LIKE '%-%'),
МФОБанку NVARCHAR(20),  РР NVARCHAR(30),
ЄДРПОУ NVARCHAR(20) UNIQUE ) ON FG1;
```

Таблиця «Реалізація книг» призначена для зберігання даних про оформлені замовлення клієнтів та факти продажу книжкової продукції. На відміну від попередніх таблиць, ця таблиця фізично розміщується у файловій групі *FG2*, що дозволяє розподілити навантаження між різними групами файлів бази даних і підвищити ефективність доступу до даних.

Поле *IdЧеку INT PRIMARY KEY* є первинним ключем і унікально ідентифікує кожен продаж (чек). Поле *IdКлієнта INT FOREIGN KEY REFERENCES Bookstore.Clients(IdКлієнта)* встановлює зв'язок із таблицею клієнтів, забезпечуючи, що кожна операція продажу пов'язана з конкретним клієнтом. Це обмеження підтримує референційну цілісність між замовленнями та довідником клієнтів.

Поле *ДатаЗамовлення DATE NOT NULL DEFAULT GETDATE()* фіксує дату створення замовлення, автоматично підставляючи поточну дату системи при додаванні нового запису. Поле *ДатаСплати DATE* відображає дату фактичної оплати, яка заповнюється користувачем після проведення платежу.

```
-- Створення таблиці «Реалізація книг»
```

```
CREATE TABLE Bookstore.Sales (IdЧеку INT PRIMARY KEY,
IdКлієнта INT FOREIGN KEY REFERENCES
Bookstore.Clients(IdКлієнта),
ДатаЗамовлення DATE NOT NULL DEFAULT GETDATE(),
ДатаСплати DATE) ON FG2;
```

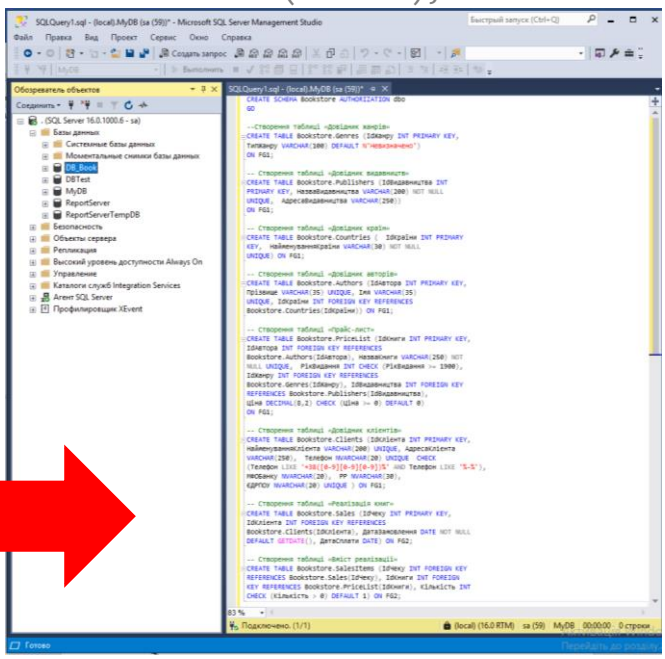
Останньою створюємо таблицю «Вміст реалізації», яка призначена для деталізації складу кожного продажу, тобто для зберігання інформації про те, які саме книги та в якій кількості були реалізовані за певним чеком. Таблиця розміщується у файловій групі *FG2*, що узгоджується з розподілом даних між різними групами файлів і забезпечує оптимізацію роботи із транзакційними таблицями.

Поле *IdЧеку INT FOREIGN KEY REFERENCES Bookstore.Sales(IdЧеку)* встановлює зв'язок із таблицею Bookstore.Sales, що дає змогу об'єднувати позиції продажу з відповідним замовленням. Поле *IdКниги INT FOREIGN KEY REFERENCES Bookstore.PriceList(IdКниги)* пов'язує кожен запис із конкретною книгою з прайс-листа, забезпечуючи коректність відображення асортименту.

Поле *Кількість INT CHECK (Кількість > 0) DEFAULT 1* визначає кількість проданих примірників книги. Обмеження *CHECK* гарантує, що кількість не може бути від'ємною або нульовою, а значення за замовчуванням встановлюється рівним одному, що є типовим для більшості продажів.

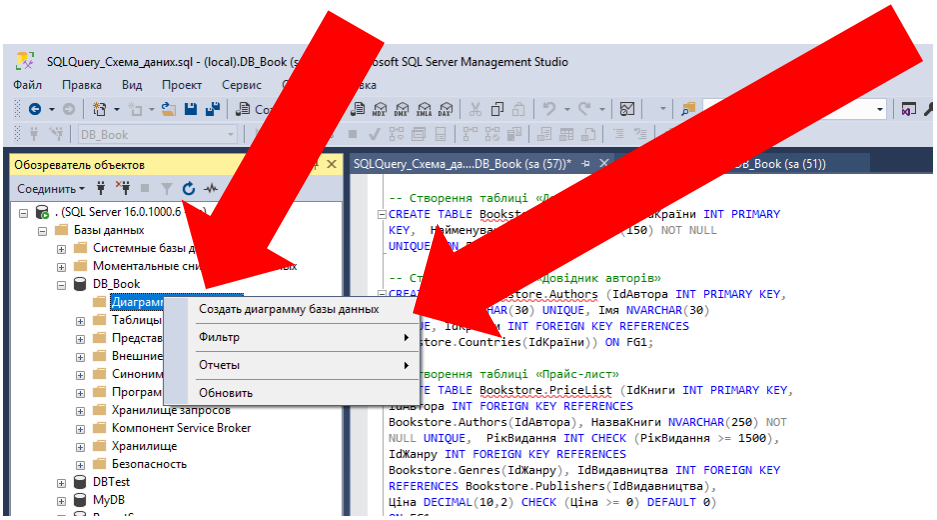
```
-- Створення таблиці «Вміст реалізації»
CREATE TABLE Bookstore.SalesItems
(IdЧеку INT FOREIGN KEY REFERENCES Bookstore.Sales(IdЧеку),
IdКниги INT FOREIGN KEY REFERENCES Bookstore.PriceList(IdКниги),
Кількість INT
CHECK (Кількість > 0) DEFAULT 1)
ON FG2;
```

Вигляд створення схеми даних та таблиць у середовищі *MS SQL Server Management Studio*

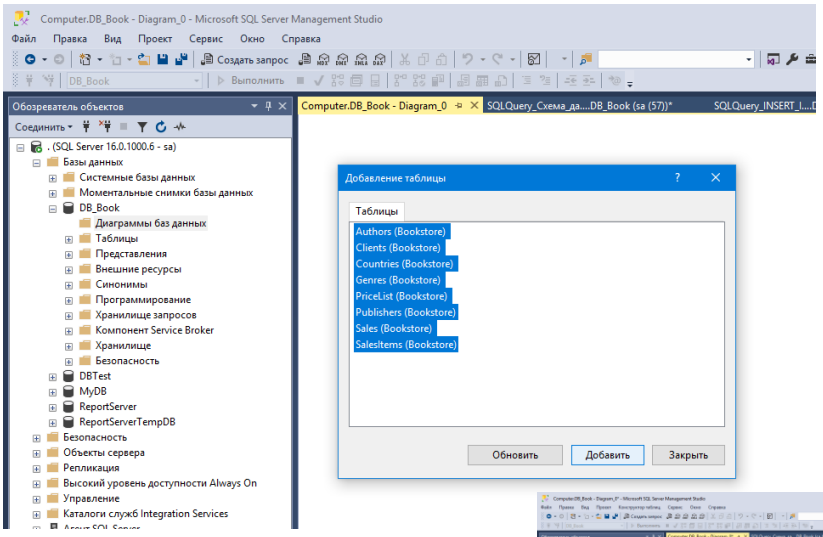


Для побудови фізичних зв'язків між таблицями (створення логічної моделі БД), необхідно викликати контекстно-залежне

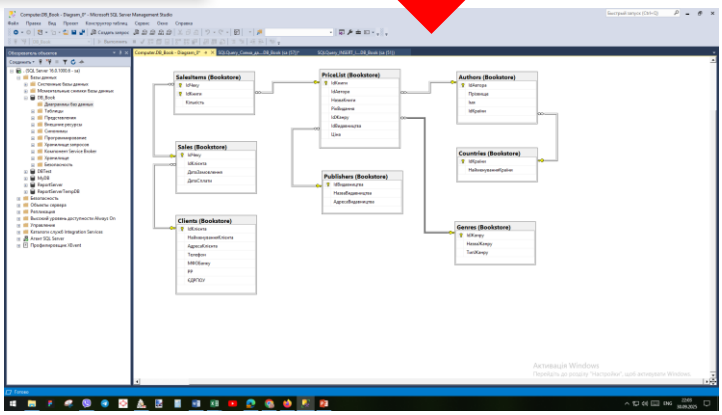
меню (натиск правою кнопкою мишки) *Database Diagram* та обрати команду *New Database Diagram*



У діалогову вікні обирається необхідні таблиці та натискається кнопка ***Add***



РЕЗУЛЬТАТ



Заповнення таблиць даними за допомогою команди *INSERT INTO*

Наступним кроком буде заповнення таблиць інформаційними даними. Порядок внесення даних у таблиці бази даних *Bookstore* визначається логічною послідовністю створених між ними зв'язків через зовнішні ключі. Оскільки деякі таблиці є довідниковими, а інші залежними від них, заповнення має виконуватись від базових довідників до таблиць, що містять зовнішні посилання.

Спочатку заповнюються довідникові таблиці, які не мають зовнішніх ключів. У першу чергу вноситься інформація до таблиці *Bookstore.Genres*, що містить типи жанрів книг, далі до таблиці *Bookstore.Publishers*, де зберігаються дані про видавництва, та таблиці *Bookstore.Countries*, яка містить перелік країн. Вказані таблиці формують основу для подальшого заповнення пов'язаних довідників.

Після цього заповнюється таблиця *Bookstore.Authors*, яка має зовнішній ключ *IdКраїни*, що посилається на таблицю Countries. Тому спочатку необхідно, щоб усі країни були вже внесені. Аналогічно, на наступному етапі вносяться дані до таблиці *Bookstore.PriceList*, що містить зовнішні ключі на авторів, жанри та видавництва, тобто всі довідники, які повинні бути заповнені заздалегідь.

Потім додаються записи до таблиці *Bookstore.Clients*, оскільки вона не залежить від інших таблиць і використовується для зберігання інформації про покупців. Після цього переходять до заповнення таблиці *Bookstore.Sales*, у якій фіксуються факти продажів. Вона містить зовнішній ключ *IdКлієнта,* тому дані про клієнтів повинні бути внесені раніше.

Завершальним етапом є внесення інформації до таблиці *Bookstore.SalesItems*, яка є деталізацією продажів і містить два зовнішні ключі *IdЧеку* (посилання на таблицю *Sales*) та *IdКниги* (посилання на таблицю PriceList). Відповідно, ці дві таблиці повинні бути заповнені до додавання записів у *SalesItems*.

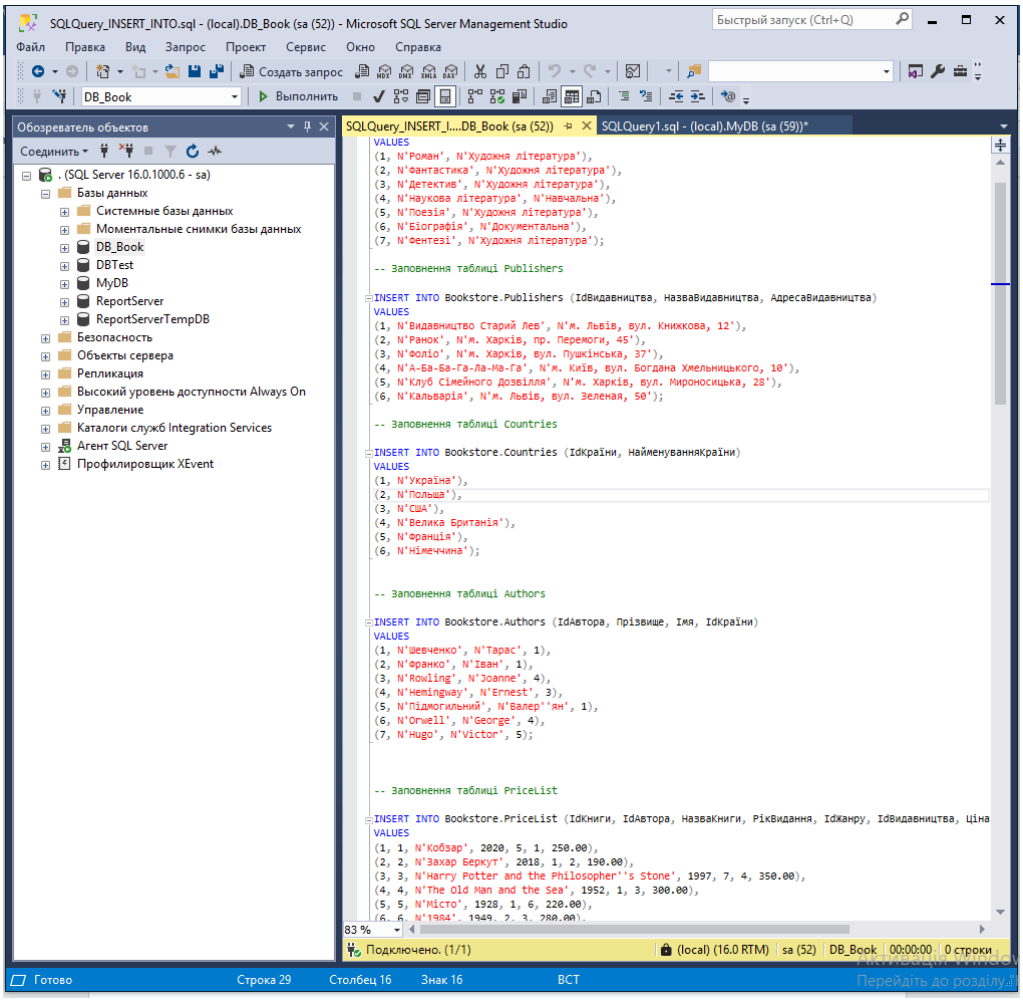
Таким чином, правильна послідовність заповнення таблиць має вигляд: *Genres* → *Publishers* → *Countries* → *Authors* → *PriceList* → *Clients* → *Sales* → *SalesItems*. Дотримання цього порядку забезпечує цілісність даних і відсутність порушень зовнішніх зв'язків під час вставлення записів у базу даних.

Приклад заповнення таблиці «Довідник видавництв»

```
-- Заповнення таблиці Publishers
INSERT INTO Bookstore.Publishers (IdВидавництва, НазваВидавництва, АдресаВидавництва)
VALUES
(1, N'Видавництво Старий Лев', N'м. Львів, вул. Книжкова, 12'),
(2, N'Ранок', N'м. Харків, пр. Перемоги, 45'),
(3, N'Фоліо', N'м. Харків, вул. Пушкінська, 37'),
```

```
(4, N'А-Ба-Ба-Га-Ла-Ма-Га', N'м. Київ, вул. Богдана Хмельницького, 10'),
(5, N'Клуб Сімейного Дозвілля', N'м. Харків, вул. Мироносицька, 28'),
(6, N'Кальварія', N'м. Львів, вул. Зеленая, 50');
```

Аналогічним чином заповнюються всі інші таблиці бази даних *Bookstore*. Для кожної з них використовуються оператори *INSERT INTO*, що дають змогу внести дані відповідно до визначених полів та типів даних.Приклад запоанення таблиці даними у середовищі *MS SQL Server Management Studio:*



Оновлення даних у таблицях із використанням команди *UPDATE*

Після заповнення таблиць необхідними даними може виникнути потреба в оновленні певної інформації, наприклад, зміни ціни книги, виправлення помилки у назві чи зміні адреси видавництва. Для цього в середовищі *MS SQL Server Management Studio* застосовується команда *UPDATE*, яка дозволяє змінювати значення одного або кількох полів у записах таблиці.

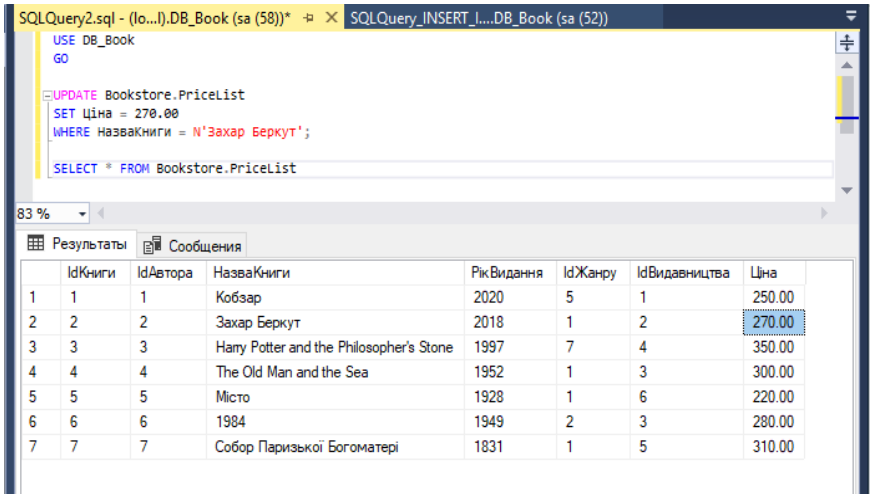
Загальний синтаксис команди:
UPDATE <ім'я_таблиці>
SET <назва_поля> = <нове_значення>
WHERE <умова_відбору>;

Ключове слово *WHERE* задає умову відбору рядків, які потрібно змінити. кщо умову не вказано, зміни буде застосовано до всіх записів таблиці, тому в практичній роботі *WHERE* обов'язково слід використовувати.

Приклад 1. Зміна ціни книги у таблиці *Bookstore.PriceList:*

```
USE DB_Book
 GO

UPDATE Bookstore.PriceList
SET Ціна = 270.00
WHERE НазваКниги = N'Захар Беркут';
```

Команда оновлює ціну книги «Захар Беркут» з 190,00 грн на 270,00 грн. Це типова операція коригування дантх прайс-листа при зміні ринкової вартості товару.



	IdКниги	IdАвтора	НазваКниги	РікВидання	IdЖанру	IdВидавництва	Ціна
1	1	1	Кобзар	2020	5	1	250.00
2	2	2	Захар Беркут	2018	1	2	270.00
3	3	3	Harry Potter and the Philosopher's Stone	1997	7	4	350.00
4	4	4	The Old Man and the Sea	1952	1	3	300.00
5	5	5	Місто	1928	1	6	220.00
6	6	6	1984	1949	2	3	280.00
7	7	7	Собор Паризької Богоматері	1831	1	5	310.00

Результат виконання команди *UPDATE* у середовищі *MS SQL Server Management Studio:*

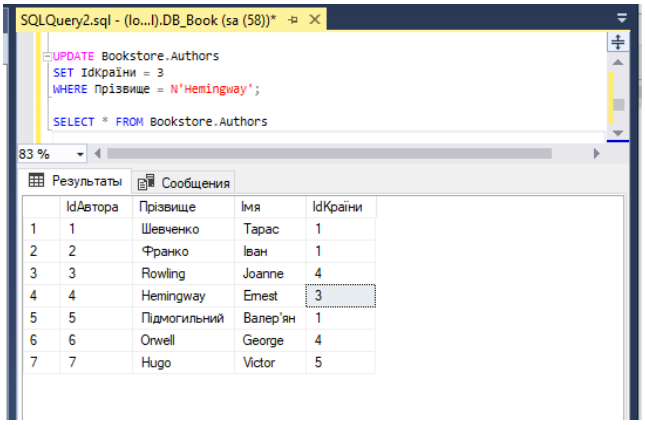
Приклад 2. Оновлення адреси видавництва у таблиці *Bookstore.Publishers*:

```
SELECT * FROM Bookstore.PriceList

UPDATE Bookstore.Publishers
SET АдресаВидавництва = N'м. Львів, вул. Літературна, 21'
WHERE НазваВидавництва = N'Видавництво Старий Лев';
```

У результаті в записі для видавництва «Видавництво Старий Лев» оновлюється адреса офісу. Подібні зміни відображають актуалізацію контактних даних.

Приклад 3. Оновлення країни для автора у таблиці *Bookstore.Authors:*

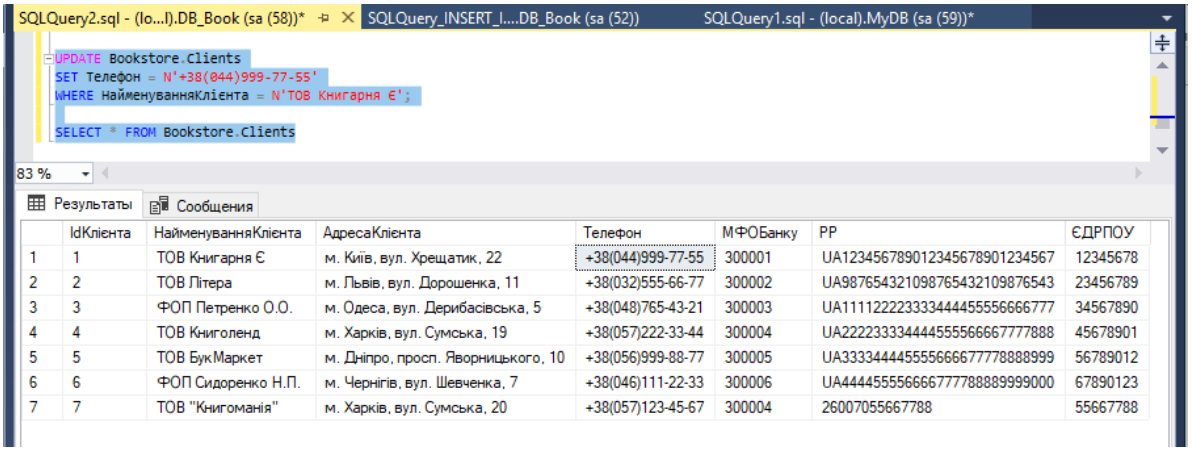
```
UPDATE Bookstore.Authors
SET IdКраїни = 3
WHERE Прізвище = N'Hemingway';
```



Цей запит змінює країну походження автора Ернеста Гемінґвея, вказавши посилання на країну з ідентифікатором 3 – США. Таким чином демонструється використання зовнішнього ключа для підтримки логічних зв’язків між таблицями.

Приклад 4. Зміна телефонного номера клієнта у таблиці *Bookstore.Clients*:

```
UPDATE Bookstore.Clients
SET Телефон = N'+38(044)999-77-55'
WHERE НайменуванняКлієнта = N'ТОВ Книгарня Є';
```



Оновлюється телефон клієнта «ТОВ Книгарня Є». Такі зміни використовуються для актуалізації контактної інформації клієнтів.

Приклад 5. Виправлення дати оплати замовлення у таблиці *Bookstore.Sales*:

```
UPDATE Bookstore.Sales
SET ДатаСплати = '2024-01-13'
WHERE IdЧеку = 1;
```

У цьому прикладі дата фактичної оплати для першого замовлення змінюється з 12.01.2024 на 13.01.2024.

Приклад 6. Коригування кількості проданих примірників книги у таблиці *Bookstore.SalesItems*:

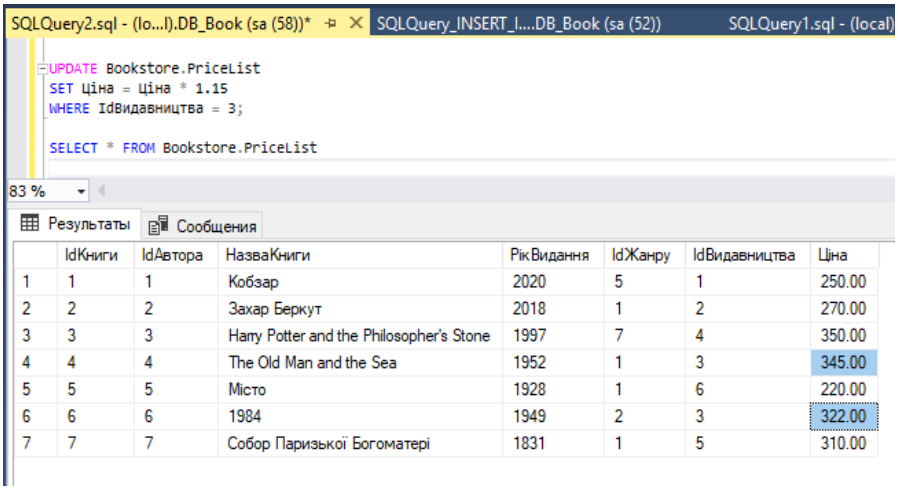
```
UPDATE Bookstore.SalesItems
SET Кількість = 25
WHERE IdЧеку = 2 AND IdКниги = 6;
```

Оновлено кількість проданих книг «1984» для другого чеку з 15 на 25 примірників. Це демонструє оновлення транзакційних даних у таблиці, що містить зовнішні ключі.

Приклад 7. Масове підвищення цін на всі книги певного видавництва у таблиці *Bookstore.PriceList*:

```
UPDATE Bookstore.PriceList
SET Ціна = Ціна * 1.15
WHERE IdВидавництва = 3;
```

Дана команда збільшує ціни всіх книг видавництва «Фоліо» на 15%. Це приклад групового оновлення, яке часто використовується для регулювання прайсів у межах одного постачальника.



	IdКниги	IdАвтора	НазваКниги	РікВидання	IdЖанру	IdВидавництва	Ціна
1	1	1	Кобзар	2020	5	1	250.00
2	2	2	Захар Беркут	2018	1	2	270.00
3	3	3	Harry Potter and the Philosopher's Stone	1997	7	4	350.00
4	4	4	The Old Man and the Sea	1952	1	3	345.00
5	5	5	Місто	1928	1	6	220.00
6	6	6	1984	1949	2	3	322.00
7	7	7	Собор Паризької Богоматері	1831	1	5	310.00

Видалення записів із таблиць за допомогою команди *DELETE*

Команда *DELETE* використовується для видалення окремих записів із таблиці без зміни її структури. Вона дозволяє вилучати застарілі, дубльовані або помилкові дані. При цьому важливо враховувати, що якщо таблиця має зв’язки через зовнішні ключі (*FOREIGN KEY*), система не дозволить видалити запис, на який існують посилання в інших таблицях, доки не буде знято залежність або не видалено відповідні записи у дочірніх таблицях.

Загальний синтаксис команди:

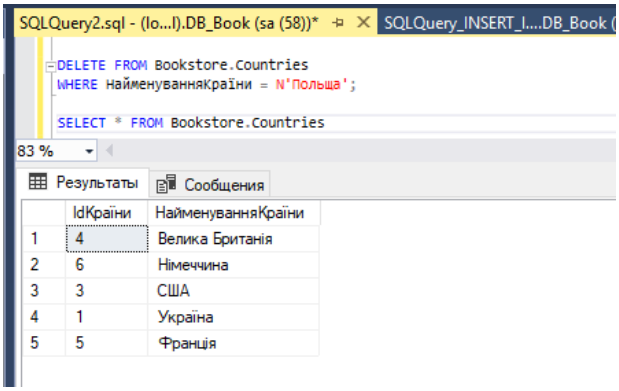
DELETE FROM <ім'я_таблиці>
WHERE <умова_відбору>;

Якщо умову *WHERE* не вказати, команда видалить усі записи таблиці, тому обов'язковим є уточнення, який саме запис потрібно вилучити.

Приклад 8. Видалення країни з таблиці *Bookstore.Countries*:

```
DELETE FROM Bookstore.Countries
WHERE НайменуванняКраїни = N'Польща';
```

Якщо у таблиці *Authors* відсутні автори, які посилаються на цю країну (тобто *IdКраїни = 2* ніде не використовується), команда успішно виконається. У протилежному випадку видалення буде заборонене механізмом цілісності даних.



	IdКраїни	НайменуванняКраїни
1	4	Велика Британія
2	6	Німеччина
3	3	США
4	1	Україна
5	5	Франція

Приклад 9. Видалення всіх продажів за певний період (масова операція):

```
DELETE FROM Bookstore.Sales
WHERE ДатаЗамовлення < '2024-03-01';
```

Команда видалить усі записи про замовлення, оформлені до 1 березня 2024 року. Подібні операції застосовуються для архівації старих транзакцій у великих базах даних.

Модифікація структури таблиць *ALTER TABLE*

У процесі розроблення або супроводу бази даних може виникнути необхідність змінити структуру таблиці: додати нове поле, змінити тип даних, видалити атрибут або встановити нове обмеження. Для цього використовується команда *ALTER TABLE*.

Загальний синтаксис:

ALTER TABLE <ім'я_таблиці>
<операція_зміни_структури>;

Приклад 10. Додавання нового поля:

```
ALTER TABLE Bookstore.Clients
ADD Email NVARCHAR(100) UNIQUE;
```

У цьому прикладі до таблиці клієнтів додається нове поле *Email* із типом *NVARCHAR(100),* для якого встановлено обмеження унікальності.

Приклад 11. Зміна типу даних:

```
ALTER TABLE Bookstore.PriceList
ALTER COLUMN Ціна DECIMAL(10,2);
```

Команда змінює розмірність числового типу поля *Ціна*, зберігаючи його дві десяткові позиції.

Приклад 12. Видалення стовпця:

```
ALTER TABLE Bookstore.Publishers
DROP COLUMN АдресаВидавництва;
```

У цьому випадку з таблиці *Publishers* буде вилучено поле *АдресаВидавництва.*

ТЕМА 6. МОВА ЗАПИТІВ SQL: ОПЕРАТОР SELECT ТА ЗАСОБИ ОБРОБКИ ДАНИХ

ПЛАН:

6.1. Оператор SELECT як основний засіб доступу до даних

Оператор *SELECT* є центральною конструкцією мови *SQL* і використовується для отримання даних із бази. Саме за допомогою *SELECT* реалізуються практично всі запити до таблиць: від простого вибору кількох стовпців до складних аналітичних запитів. У результаті виконання цього оператора завжди формується нова таблиця (так звана результатна множина), яка може бути використана як самостійно, так і у вигляді проміжного результату для подальших обчислень чи вкладених запитів.

Цікавий факт:

Історично саме можливість вкладати один оператор SELECT у інший стала однією з головних особливостей мови SQL та причиною використання терміна «структурована» у її назві.

Ця властивість дозволяє поступово ускладнювати запити, поєднувати різні етапи обробки та робити вибірку максимально гнучкою.

Оператор *SELECT* застосовується не лише для відбору даних, але й для їхньої трансформації. Завдяки використанню виразів і функцій можна створювати обчислювані стовпці, формувати нові значення на основі існуючих, а також підсумовувати або агрегувати інформацію. Це робить

SELECT не просто засобом пошуку, а універсальним інструментом для формування необхідного уявлення про дані.

Пропозиція *SELECT* може використовуватися як:

- самостійна команда на отримання і виведення рядків таблиці, сформованої з стовпців і рядків однієї або декількох таблиць (подань);
- елемент *WHERE*- або *HAVING*-умови (скорочений варіант пропозиції, так званий «вкладений запит»);
- фраза вибору в командах *CREAT VIEW, DECLARE CURSOR* або *INSERT*;
- засіб присвоєння глобальним змінним значень з рядків сформованої таблиці (INTO-фраза).

6.2. Структура запиту та логічний порядок його виконання.

Оператор *SELECT* у мові *SQL* має суворо визначену структуру, яка забезпечує єдиний підхід до формування запитів незалежно від складності задачі. Стандартний синтаксис передбачає використання низки ключових слів, що утворюють послідовні розділи запиту. Кожен з них виконує окрему функцію у процесі обробки даних:

Синтаксис пропозиції SELECT

```
SELECT [ALL|DISTINCT] визначення_полів
[INTO нова_таблиця ]
FROM назва_таблиці
[WHERE умови_пошуку]
[GROUP BY умова_групування
[HAVING умова_пошуку]]
[ORDER BY умова_сортування]
```

Основні ключові слова і параметри пропозиції SELECT:

Необов'язкова частина виділяється квадратними дужками.

SELECT – визначає набір полів, які потрібно отримати в результаті запиту (*ALL*– повторювані рядки присутні, *DISTINCT*– відсутні);

FROM – встановлює набір таблиць, звідки витягується результат;

WHERE – визначає умова отримання даних;

GROUP BY –задає умову угруповання даних;

HAVING – визначає умову відбору всередині угруповання;

ORDERBY – задає поля і порядок сортування.

Логічний порядок виконання запиту відрізняється від послідовності запису у синтаксисі. Система керування базами даних (СКБД) обробляє запит поетапно, формуючи проміжні результати:

1. *FROM* – формується джерело даних (у випадку багатотабличних запитів – декартовий добуток або результат з'єднання таблиць).
2. *WHERE* – відсіюються рядки, які не задовольняють умову відбору.
3. *GROUP BY* – відбувається угруповання залишених рядків.
4. *HAVING* – відсіюються цілі групи, що не відповідають умовам.
5. *SELECT* – обчислюється перелік виразів і формується набір результатних стовпців.
6. *ORDER BY* – впорядковується результатна множина.

Таким чином, користувач у запиті визначає логічну модель обробки даних, а СКБД трансформує її у конкретний план виконання. Це розмежування є принциповим, тобто програміст оперує декларативним описом бажаного результату, а не алгоритмом обчислень.

Розуміння логічного порядку виконання запиту має не лише теоретичне, а й практичне значення. Воно дозволяє:

- правильно інтерпретувати результати при використанні умов *WHERE* та *HAVING*;
- уникати поширених помилок при посиланні на псевдоніми стовпців, які фактично формуються лише на етапі *SELECT*;
- оптимізувати запити шляхом усвідомленого використання умов відбору, угрупування та сортування.

Символи, що використовуються в синтаксичних конструкціях:

зірочка (*) для позначення «все»–вживається в звичайному для програмування сенсі, тобто «Всі випадки, що задовольняють визначенню»;

квадратнідужки ([])–означають, що конструкції, укладені в ці дужки, є необов'язковими (тобто можуть бути опущені);

фігурнідужки ({})– означають, що конструкції, укладені в ці дужки, повинні розглядатися як цілі синтаксичні одиниці, тобто вони дозволяють уточнити порядок розбору синтаксичних конструкцій, замінюючи звичайні дужки, які використовуються в синтаксисі *SQL*;

трикрапки (...)–вказують на те, що синтаксична одиниця якабезпосередньо передує їм, факультативно може повторюватися один або більше разів;

прямариска (|)– означає наявність вибору з двох або більше можливостей. Наприклад, позначення *ASC | DESC* вказує, що можна вибрати один з термінів *ASC* або *DESC*; коли ж один з елементів вибору укладений у квадратні дужки, то це означає, що він вибирається за замовчуванням (так, [*ASC*] | *DESC* означає, що відсутність всієї цієї конструкції буде сприйматися як вибір *ASC*);

крапказ комою (;)– завершальний елемент пропозицій *SQL*;

кома (,) –використовується для поділу елементів списків;

пробіл () –може вводитися для підвищення наочності між будь-якими синтаксичними конструкціями пропозицій *SQL*;

прописніжирні латинські букви і символи– використовуються для написання конструкцій мови *SQL* і повинні, якщо це спеціально не обумовлено, записуватися в точності так, як показано;

малі літери– використовуються для написання конструкцій, які повинні замінюватися конкретними значеннями, обраними користувачем, причому для визначеності окремі слова цих конструкцій зв'язуються між собою символом підкреслення (_);

термінтаблиця– використовується для узагальнення таких видів таблиць, як *базова_табліця*, уявлення або псевдонім; тут псевдонім служить для тимчасового (на момент виконання запиту) перейменування і (або) створення робочої копії *базовій_таблиці* (подання).

ім'я_поля– позначає поле з ім'ям «ім'я_поля»;

імя_таблиці.імя_поля– позначає поле з ім'ям «ім'я_поля» таблиці «ім'я_таблиці»;

SQL-вираз– задає вираз, куди входять поля таблиць і константи.

Вирази, умови та оператори.

Відповідно до стандарту *ANSI* поняття *SQL*-вираз визначено так: вирази з арифметичними операціями «плюс», «мінус», «помножити», «розділити», де в якості аргументів використовуються поля таблиць бази даних або константи;

SQL-функція, як аргумент застосовує поле таблиці, константу або їх комбінацію за допомогою арифметичних операцій.

ВАЖЛИВО:

Будь-яка сучасна СКБД містить велику кількість стандартних функцій, які можуть застосовуватися в операторі SELECT при отриманні даних з БД. Використання таких функцій в деяких випадках дуже ефективно, так як економить час при розробці ПЗ і значно прискорює процес отримання кінцевого результату внаслідок того, що

SQL-оператор SELECT виконується на сервері, що володіє значно більшою потужністю, ніж робоча станціякористувача.

Припустимо, що необхідний результат формується шляхом обчислення синуса над значеннями, що зберігаються. 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.

6.3. Команда FROM: вибір таблиці та формування джерела даних

Команда *FROM* є обов'язковим елементом оператора *SELECT*, оскільки саме вона визначає джерело даних, з яким працює запит. У найпростішому випадку це може бути одна таблиця, представлення (*View*) або навіть інший підзапит. Коли джерело даних визначено, система керування базами даних формує проміжну результатну множину, яка надалі передається на обробку наступним частинам запиту (*WHERE, GROUP BY*, тощо).

У теоретичному розумінні, результат виконання *FROM* можна розглядати як розширений декартовий добуток таблиць, указаних у запиті. Якщо у виразі *FROM* вказана лише одна таблиця, то проміжна множина просто відтворює всі її рядки. Якщо таблиць кілька, то на цьому етапі формується всі можливі комбінації рядків, після чого інші частини запиту (зокрема *WHERE*) уточнюють відбір і звужують множину результатів.

У практиці проектування баз даних це має важливе значення:

- при роботі з однією таблицею *FROM* фактично визначає простір пошуку даних.
- при роботі з кількома таблицями (які будуть розглядатися в наступній лекції) саме розділ *FROM* задає логіку поєднання даних через операції з'єднання (*JOIN*).

Особливість конструкції *FROM* полягає в тому, що до результату цього етапу можуть додаватися обчислювані стовпці, створювані на основі виразів чи функцій. Наприклад, у запиті можна використати арифметичні операції над існуючими атрибутами, створивши похідні поля, які відразу стають частиною результатної множини.

Отже, *FROM* є логічною відправною точкою будь-якого запиту *SELECT*. Саме він визначає об'єкти бази, над якими буде здійснюватися подальша обробка, і без чіткого розуміння його ролі неможливо адекватно опановувати більш складні *SQL*-конструкції. Для початкових прикладів найчастіше використовують просту форму:

Синтаксис пропозиції FROM:

```
SELECT стовпець_1, стовпець_2, ...
FROM назва_таблиці;
```

ПРИКЛАД 6.1. Вибірка полів *NameKl i Tel* з таблиці *Klient*

Лістинг 6.1. Простої вибірки SELECT

```
SELECT НайменуванняКлієнта, Телефон
FROM Bookstore.Clients
GO
```

ПРИКЛАД 6.2. Вибірка всіх полів з таблиці Klient без перерахування назв полів.

Лістинг 6.2. Вибір всіх полів

```
USE DB_Book
GO
SELECT * FROM Bookstore.Clients
GO
```

Результуюча таблиця може включати не тільки поля вхідних таблиць, а й результат обчислення над полями таблиці. У виразах можуть використовуватися арифметичні оператори і функції.

Конструкція ***SELECT*** може включати арифметичні вирази, а також прості імена полів. Крім того, можна додати константи в результат вибірки.

ПРИКЛАД 6.3. Вибірка полів *NameBook, IdAvtor, Cena* та обчислювальне поле ціна з ПДВ.

Лістинг 6.3. Використання у вибірки обчислювального полю

```
USE DB_Book
GO
```

```
SELECT НазваКниги, IdАвтора, Ціна, Ціна *1.20 ЦінаPDV
FROM Bookstore.PriceList
GO
```

ВАЖЛИВО:

Арифметичні вирази лівої і правої частин предиката порівняння будуються за загальними правилами побудови арифметичних виразів і можуть включати, в загальному випадку, імена стовпців таблиць з розділу FROM та константи. Типи даних арифметичних виразів повинні бути порівнянними (наприклад, якщо тип стовпця a таблиці A є типом символьних рядків, то предикат "a = 5" неприпустимий).

Якщо правий операнд операції порівняння задається підзапитом, то додатковим обмеженням є те, що потужність результату підзапиту повинна бути не більше одиниці. Якщо хоча б один з операндів операції порівняння має невизначене значення, або якщо правий операнд є підзапитом з порожнім результатом, то значення предиката порівняння одно unknown.

6.4. Команда WHERE: предикати та умови пошуку.

Якщо в табличному вираженні присутній розділ *WHERE*, то наступним обчислюється він. Обчислення розділу *WHERE* проводиться за такими правилами: Нехай R – результат обчислення розділу *FROM*. Тоді умова пошуку застосовується до всіх рядків *R*, і результатом розділу *WHERE* є таблиця *SQL* , що складається з тих рядків *R*, для якого результатом обчислення умови пошуку є *true*. Якщо умова вибірки включає підзапити, то кожен підзапит обчислюється для кожного кортежу таблиці *R* (в стандарті використовується термін «effectively» в тому сенсі, що результат повинен бути таким, як якби кожен підзапит дійсно обчислювався заново для кожного кортежу *R*).

Команда *WHERE* використовується в операторі *SELECT* для встановлення критеріїв відбору рядків із таблиці. Її головне призначення – обмежити обсяг даних, що потрапляє до результатної множини, залишивши лише ті кортежі, які задовольняють заданим умовам. Таким чином, *WHERE* виконує роль фільтра, що застосовується до проміжної таблиці, сформованої командою *FROM*.

Умови пошуку у *WHERE* задаються за допомогою предикатів, тобто виразів, які повертають логічне значення: *TRUE*, *FALSE* або *UNKNOWN* (у випадку наявності *NULL*). Якщо результат обчислення умови дорівнює *TRUE*, рядок потрапляє до результатної таблиці; в інших випадках – відкидається.

Синтаксис запиту з використанням комегди WHERE:

```
SELECT стовпець_1, стовпець_2, ...
FROM назва_таблиці
WHERE умови_пошуку...
GO
```

Умова, наступне за ключовим словом *WHERE***,** може включати:

- **Логічні предикати** – дозволяють комбінувати прості умови, формуються з одного або декількох умов, з'єднаних логічними операторами:
  - ✓ *AND* – коли повинні задовольнятися обидві умови;
  - ✓ *AND* – вираз істинний тоді й тільки тоді, коли істинні обидва операнди. (Логічно: A *AND* B ≡ істинно лише якщо A = TRUE і B = TRUE.)
  - ✓ *OR* – вираз істинний, якщо істинний хоча б один з операндів (інклюзивне OR). (Логічно: A *OR* B ≡ істинно якщо A = TRUE або B = TRUE або обидва.)
  - ✓ *AND NOT* – скорочення для A *AND NOT* B; істинно, коли перший операнд істинний, а другий — не істинний (тобто A = TRUE і B = FALSE або B IS NULL, залежно від семантики NULL у конкретній СКБД).
  - ✓ *OR NOT* – скорочення для A *OR NOT* B; істинно, коли перший операнд істинний або другий – не істинний (тобто A = TRUE або B = FALSE / B IS NULL).

Предика́т (відлат. *praedicare*– проголошувати, заявляти, присуджувати) у сучасній логіці зазвичай означає булево-значну функцію.

ВАЖЛИВО:

Члени існує пріоритет AND над OR (спочатку виконуються всі операції AND і тільки після цього операції OR). Для отримання бажаного результату WHERE умови повинні бути введені в правильному порядку, який можна організувати введенням дужок.

📖 **Предикат порівняння** з виразами або результатами підзапиту. Умова визначається з двох виразів, розділених одним із знаків операції відносини:

= (дорівнює),
<> (не дорівнює),
< (менше),
<= (менше або дорівнює),
> (більше),
> = (більше або дорівнює),

які можуть передувати оператором *NOT*, створюючи, наприклад, відносини «не менше» і «не більше».

ВАЖЛИВО:

При обробці умови числа порівнюються алгебраїчно - негативні числа вважаються меншими, ніж позитивні, незалежно від їх абсолютної величини. Рядки символів порівнюються відповідно до їх поданням до коді, що використовується в конкретній СКБД, наприклад, в коді ASCII. Якщо порівнюються два рядки символів, що мають різні довжини, більш короткий рядок доповнюється справа пробілами для того, щоб вони мали однакову довжину перед здійсненням порівняння.

📖 **Діапазонний предикат *BETWEEN*** (між) – перевіряє, чи входить значення у визначений інтервал.

Синтаксис предиката BETWEEN в більш загальному випадку:

<Ім'я_поля> [NOT] BETWEEN <початкове_значення> AND <останнє_значення>

ПРИКЛАД 6.4. Визначити з таблиці «Книги» інформацію про ціну, що належить діапазону [200, 300], виключити з діапазону число 210.

Лістинг 6.4. Використання предиката BETWEEN

```
SELECT *   FROM Bookstore.PriceList
WHERE (Ціна BETWEEN 200 AND 300)
AND Not Ціна IN (210)
GO
```

📖 **Множинний предикат *IN*** (належить) – перевіряє, чи входить значення до певного набору.

ПРИКЛАД 6.5. Визначити з таблиці «Клієнти» інформацію про клієнтів, які не мають назву «Дивосвіт».

Лістинг 6.5. Використання предиката IN

```
SELECT * FROM Bookstore.Clients
WHERE НайменуванняКлієнта NOT IN ('Дивосвіт')
GO
```

📖 **Шаблонні предикати *LIKE*** (схоже на) – застосовуються до символьних даних для пошуку за зразком.

Синтаксис предиката LIKE в більш загальному випадку:

<Ім'я_стовбця> [NOT] LIKE <зразок> (<рядок> [NOT] LIKE <підрядок>)

При перевірці умови вибірки числа порівнюються алгебраїчно: негативні числа вважаються менше, ніж позитивні, незалежно від їх абсолютної величини. Рядки порівнюються відповідно до їх поданням до коді *ANSI* . При порівнянні двох рядків, що мають різні довжини, попередньо більш короткий рядок доповнюється справа пробілами для того, щоб обидві рядки мали однакову довжину.

Зауваження: *Оператор LIKE застосовується тільки до строкових полів типу CHAR (NCHAR) або*

VARCHAR (NVARCHAR), оскільки він використовується для пошуку підрядків в предикаті реченні WHERE. Отже, він здійснює перегляд рядка для перевірки умови – чи входить заданий підрядок в зазначений рядок. Для гнучкою реалізації цієї мети використовують шаблони (маски) – спеціальні символи для конструювання зразка рядки.

Існують такі типи шаблонів, які використовуються з *LIKE*:

- ✓ *Символ «підкреслення»* (_) – замінює будь-який одиночний символ.

НАПРИКЛАД:

LIKE 'б_т' – зразком 'б_т' відповідає 'біт' або 'бут';

LIKE '___ст' – будь-яке слово з п'яти символів, що має в кінці слова дві букви 'ст', наприклад 'крест';

LIKE '__ ав___' – будь-яке слово з семи символів, в середині якого є пара символів 'ав', наприклад 'правило'.

- ✓ *Символ «відсоток»* (%) – замінює послідовність символів довільної довжини, в тому числі і нульовий.

НАПРИКЛАД:

LIKE 'А%' – будь-який рядок, що починається з букви *Р,* наприклад 'Аєропорт';

LIKE ' % н %'– будь-який рядок, що містить букву н, наприклад *'Геній'*;

LIKE' % о % а'–відповідають *'Полтава'*;

LIKE' % о_е %' – істинно для будь-якого слова, що містить шаблон *' о_е'*, наприклад *' Готель'*.

ПРИКЛАД 6.6. Необхідно витягти з таблиці «Автори» інформацію про авторів, в яких прізвища починаються з літери 'Г' або 'Ш' сформуємо наступний запит:

Лістинг 6.6. Використання предиката LIKE

```
SELECT *
FROM Bookstore.Authors
WHERE Прізвище LIKE N'Г%' OR Прізвище LIKE N'Ш%';
```

📖 **Перевірка на невизначеність *IS NULL*** (не визначено), який може передувати оператором (не). Використовується для виявлення відсутніх значень.

📖 **Предикат існування *EXISTS*** (існує), який також може передувати оператором NOT (не). Перевіряє наявність рядків у підзапиті. Він часто застосовується у складніших конструкціях, але навіть у базових прикладах демонструє потужність *WHERE*.

ПРИКЛАД 6.7. Знайти клієнтів, які ніколи не робили замовлень:

Лістинг 6.7. Приклад застосування предикату EXISTS та оператора NOT.

```
SELECT c.НайменуванняКлієнта, c.Телефон
FROM Bookstore.Clients c
WHERE NOT EXISTS (
   SELECT 1
   FROM Bookstore.Sales s
   WHERE s.IdКлієнта = c.IdКлієнта
```

Для забезпечення переносимості прикладних програм потрібно уважно оцінювати специфіку роботи з невизначеними значеннями в конкретній СКБД.

6.5. Вирази та функції у запитах

У мові *SQL* важливу роль відіграють вирази та функції, які забезпечують можливість формування похідних даних, виконання обчислень, перетворення значень та здійснення логічного контролю умов. Застосування виразів і функцій дозволяє підвищити інформативність запитів, скоротити обсяг допоміжних обчислень поза СКБД та реалізувати складні бізнес-правила безпосередньо на рівні бази даних.

Вираз (Expression) комбінація констант, ідентифікаторів стовпців, операторів та функцій, що має певне значення.

Результатом виразу є скалярне значення, яке може бути використане:

- у списку вибірки (*SELECT*),
- у предикатах (*WHERE, HAVING*),
- у виразах сортування (*ORDER BY*),
- у визначеннях обчислюваних стовпців, представлень чи обмежень.

Вирази можуть бути:

- арифметичними (salary * 1.15, price - discount),

- ✍ логічними (age > 18 AND status = 'active'),
- ✍ рядковими (CONCAT('Hello ', surname)),
- ✍ умовними (CASE WHEN grade >= 60 THEN 'Passed' ELSE 'Failed' END).

Функція вбудована або користувацька процедура, яка приймає аргументи та повертає значення.

ВАЖЛИВО:

Будь-яка сучасна СКБД містить велику кількість стандартних функцій, які можуть застосовуватися в операторі SELECT при отриманні даних з БД. Використання таких функцій в деяких випадках дуже ефективно, так як економить час при розробці ПЗ і значно прискорює процес отримання кінцевого результату внаслідок того, що SQL-оператор SELECT виконується на сервері, що володіє значно більшою потужністю, ніж робоча станціякористувача.

Припустимо, що необхідний результат формується шляхом обчислення синуса над значеннями, що зберігаються. 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.

Всі *SQL-функції* класифікують наступним чином:

- 🕮 *скалярні*– для обробки одного даного;
- 🕮 *агрегатні*– для обробки безлічі даних.

Скалярні функції поділяють на:

- 🕮 робота з числами;
- 🕮 робота з символами;
- 🕮 робота з датою і часом;

- явне перетворення типів даних;
- шифрування / дешифрування даних;
- решта.

У табл. 6.1 наведені основні математичні функції *MS SQL Server*.

Таблиця 6.1.

Математичні функції

Функція	***Призначення***
ABS (n)	Повертає абсолютне значення *n*
ACOS (n)	Повертає арккосинус *n*
ATAN (n)	Арктангенс *n*
ATN2	Арктангенс кута, описаного двома кутами
CEILING (n)	Найближче ціле, що перевищує *n*
COS (n)	Косинус *n*
COT (n)	Котангенс кута *n*
DEGREES	Перетворює значення кута із радіан у градуси
FLOOR (n)	Найближче ціле, менше *n*
LOG	Логарифм по підставі 2
LOG10	Логарифм по підставі 10
POWER(m,n)	Повертає m в ступені *n*
RADIANS	Перетворення градусів в радіани
RAND	Генератор випадкових чисел
ROUND	Округлення чисел
SIGN	Повертає знак вираження
SIN (n)	Синус *n*
SQRT (n)	Корінь квадратний від *n*
SQUARE	Зведення в квадрат
TAN (n)	Тангенс *n*

Приклади SQL-виразів:

x + y - z; SIN(x + y); SIN(x) + COT (y);

ABS(x); TAN(x); SIN(y) + COS(x) + SQRT(y).

В даному прикладі *x, y* і *z*– поля деякої таблиці бази даних, *SIN* і *COS* - функції отримання синуса і косинуса.

У *SQL-функцій*є невеликий набір функцій для маніпулювання колонками з типом *date*. Список основних функцій обробки дати й часу наведений у таблиці6.2.

Таблиця 6.2.

Функції для обробки дати

Функція	***Призначення***
DatePart ()	Повертає частину дати: рік, квартал, місяць, тиждень, день, годину, хвилини, секунди
Year (), Month ()	Повертає рік і місяць відповідно
Hour (), Minute (), Second ()	Повертає годину, хвилини і секунди зазначеної дати
WeekdayName ()	Повертає назву дня тижня
SYSDATE	Повертає поточну дату й час
ROUND(D[,F])	Округлює значення дати *D* відповідно до заданого шаблону
TRANC(D[,F])	Зменшує значення дати *D* відповідно до заданого шаблону
NEXT_DAY(D,S)	Повертає дату дня, що є першим днем, більш пізнім, ніж поточна дата із назвою *S*

Функція *DatePart ()* має додатковий параметр, який дозволяє відобразити необхідну частину дати (табл. 6.3.).

Таблиця 6.3.

Параметри функції*DatePart ()*

Параметр	***Відображення***
‘m’	номер місяця
‘yyyy’	номер року
‘q’	квартал
‘d’	день
‘w’	тиждень
‘h’	годину
‘n’	хвилини
‘s’	секунди

ВАЖЛИВО:

Ключове слово SYSDATE завжди повертає поточну дату. У цьому прикладі також показано, як використовується арифметичний оператор додавання зі змінними типу "дата". До змінного типу "дата" можна додавати й віднімати від нього ціле число днів, місяців, років, годин, хвилин, секунд, мікросекунд. Для цього

використовуються відповідні ключові слова (DAY, MONTH тощо), що випливають за цілою константою (дробова частина ігнорується, якщо ви вказуєте число з десятковою крапкою). Є обмеження на використання дужок у таких виразах (так, висновок у дужках виразу 1 DAYS + 1 YEARS призведе до помилки).

ПРИЛАД 6.8. Якщо потрібний список нових службовців, які з'явились в організації за останній квартал, то лістинг запиту має таку запис:

Лістинг 6.8. Нові службовці, які з'явились за останній квартал

```
SELECT ENAME, HIREDATE,
   HIREDATE + 92  DAYS
FROM   EMPLOYEE
WHERE HIREDATE + 92 DAYS > SYSDATE
  AND DEPNO=30;
```

В табл.6.4 наведені основні строкові функції.

Таблиця 6.4.

Строкові функції

Функція	***Призначення***
LEFT	Відбирає символи з лівого кінця рядка, функція LEFT ('qwertyefg', 5) поверне рядок qwert
LEN	Повертає довжину символьного рядка
LOWER	Перетворює рядок в нижній регістр
LTRIM	Видаляє початкові пробіли з рядка
REPLACE	Замінює задані фрагменти рядка іншим рядком, функція *REPLACE* (*'qwt', 'w', 'e'*) поверне рядок *qet*
RIHGT	Відбирає символи з правого кінця рядка
RTRIM	Видаляє кінцеві пробіли з рядка
UPPER	Перетворює рядок у верхній регістр
+	Операція конкатенації рядків

Агрегатні функції SQL (табл.6.5.) повертають одинарне значення, обчислене зі значень у стовпці в командах *GROUP BY, HAVING, ORDER BY* SQL-оператора *SELECT*. Всі функції, за винятком *COUNT*, не враховують у своїй роботі поля зі значенням*NULL* і мають наступну загальну форму подання: *ім'я_функції* ([*DISTINCT* | *ALL*] вираз) .

За замовчуванням перед виразом розміщується *ALL*, яка стверджує, що в обробку потрапляють всі значення множини.

Таблиця 6.5.

Функції агрегування (агрегатні функції)

Функція	***Призначення***
AVG ([ALL | DISTINCT] <expression>)	Функція *AVG* повертає середнє арифметичне значень, представлених в параметрі *<expression>*. Параметр *expression* повинен містити числові значення. *NULL*-значення ігноруються
*COUNT ([ALL | DISTINCT] <expression> | *)*	Функція *COUNT* повертає дані типу *Int*про кількість елементів, представлених впараметрі *<expression>*. Параметр не може ставитися до типу даних *Text*, *Ntext*або *Image*. При використанні значення параметра * відбувається повернення даних про кількість рядків в таблиці, при це дублюючі значення або *NULL* - значення не виключаються
*COUNT_BIG ([ALL |DISTINCT] < expression> | *)*	Повертає дані про кількість елементів у групі. Аналогічна функції *COUNT*, але яке значення має тип даних *Biqint*
GROUPING(<column_name>)	Функція *GROUPING*додає додатковий стовпець до висновку оператора *SELECT*
MAX ([ALL | DISTINCT] <expression>)	Функція *MAX* повертає максимальне з значень, представлених в параметрі *<expression>*. При обчисленні функції *MAX* всі *NULL*-значення ігноруються
MIN ([ALL | DISTINCT] <expression>)	Функція *MIN* повертає мінімальне з значень, представлених в параметрі *<expression>*. При обчисленні функції *MIN* всі *NULL*-значення ігноруються

Функція	***Призначення***
STDEV (<expression>)	Функція *STDEV* повертає результат обчислення середньоквадратичного відхилення за всіма значеннями, представленим в параметрі <expression>. При обчисленні функції *STDEV* всі *NULL*- значення ігноруються
SUM ([ALL | DISTINCT] <expression>)	Функція *SUM* повертає суму всіх значень, представлених в параметрі <expression>. При обчисленні функції *SUM* всі *NULL*-значення ігноруються
VAR (<expression>)	Функція *VAR* повертає результат обчислення дисперсії за всіма значеннями, представленим в параметрі <expression>. При обчисленні функції *VAR* всі *NULL*- значення ігноруються

Вирази та функції у запитах SQL є ключовим інструментом для гнучкої роботи з даними: від простих арифметичних обчислень до складного аналітичного аналізу. Їх правильне використання забезпечує підвищення ефективності та інформативності запитів.

6.6. Команда GROUP BY: угруповання даних

GROUP BY ініціює перекомпонування формованої таблиці по групах, кожна з яких має однакове значення в стовпцях, включених до переліку *GROUP BY*. Далі до цих груп застосовуються агрегатні функції, зазначені у фразі SELECT, що призводить до заміни всіх значень групи на єдине значення (сума, кількість тощо).

Якщо в табличному вираженні присутній розділ *GROUP BY SQL* , то наступним виконується *GROUP BY*.

Якщо позначити через *R* таблицю, що є результатом попереднього розділу (*FROM* або *WHERE*), то результатом розділу *GROUP BY* є розбиття *R* на безліч груп рядків, що складається з мінімального числа груп таких, що для кожного стовпця зі списку стовпців розділу *GROUP BY* у всіх рядках кожної групи, що включає більше одного рядка, значення цього стовпця рівні. Для позначення результату розділу *GROUP BY* в стандарті використовується термін «згрупована таблиця».

Логіка роботи *GROUP BY: п*ри виконанні запиту з оператором GROUP BY система керування базами даних виконує такі кроки:

1. Вибирає дані із заданого джерела (таблиці чи результату підзапиту).

2. Формує групи рядків за значеннями одного чи кількох стовпців.
3. До кожної групи застосовуються агрегатні функції (*SUM, AVG, COUNT, MAX, MIN* тощо).
4. Повертається по одному рядку для кожної групи.

Таким чином, *GROUP BY* перетворює множину рядків таблиці на агреговане представлення даних.

Синтаксис запиту з використанням GROUP BY:

```
SELECT стовпець_1, ..., агрегатна_функція (стовпець)
FROM таблиця
GROUP BY стовпець_1, стовпець_2, ....
GO
```

ПРИКЛАД 6.9. Обчислити загальний обсяг продажу книг для кожного автора.

Лістинг 6.9. Загальна кількість проданих книг у розрізі авторів

```
SELECT IdАвтора,  SUM (Ціна) CenaSum
FROM Bookstore.PriceList
GROUP BY IdАвтора
GO
```

Команду GROUP BY можна використовувати разом з умовою відбору.

ПРИКЛАД 6.10. Згрупувати дані за кожною книгою та вивести код автора і ціну, виключивши записи про книгу з кодом *1002*.

Лістинг 6.10. Використання команди GROUP BY з умовою відбору

```
USE [DB_Book]
GO
SELECT IdАвтора,  SUM (Ціна) CenaSum
FROM Bookstore.PriceList
WHERE IdКниги <> 1002
GROUP BY IdАвтора
GO
```

Зауваження:

Рядки, що не задовольняють умові WHERE , виключаються перед групуванням даних.

Рядки таблиці можна групувати по будь-якій комбінації її полів. Якщо поле, за значеннями якого здійснюється групування, містить будь-які невизначені значення, то кожне з них породжує окрему групу.

6.7. Команда HAVING: умови для груп

Оператор *HAVING* у мові *SQL* використовується для встановлення умов відбору, що застосовуються до груп даних, сформованих за допомогою оператора *GROUP BY*. На відміну від *WHERE*, який фільтрує окремі рядки до моменту групування, *HAVING* дозволяє накладати обмеження вже на результати агрегування. Це робить його важливим інструментом при створенні зведених запитів і статистичних вибірок.

Логіка виконання: під час обробки запиту з *GROUP BY* і *HAVING* система діє в такій послідовності:

1. Виконується вибірка даних із джерела (*FROM*).
2. Фільтруються рядки за умовами *WHERE* (якщо вони вказані).
3. Формуються групи рядків відповідно до *GROUP BY*.
4. Для кожної групи обчислюються значення агрегатних функцій.
5. На результати групування накладаються обмеження, визначені в *HAVING*.
6. Виводяться лише ті групи, що задовольняють умови.

Таким чином, *HAVING* застосовується на більш пізньому етапі обробки запиту, ніж *WHERE*.

Синтаксис оператора HAVING

```
SELECT стовпець_1, агрегатна_функція (стовпець_2)
FROM таблиця
GROUP BY стовпець_1
HAVING [NOT] condition [[AND | OR] [NOT] condition] ...
GO
```

Зауваження:

Умова пошуку розділу HAVING будується за тими ж синтаксичними правилами, що і умова пошуку розділу WHERE, і може включати ті ж

самі предикати. Однак є спеціальні синтаксичні обмеження по частині використання в умови пошуку специфікацій стовпців таблиць з розділу FROM даного табличного вираження. Ці обмеження випливають з того, що умова пошуку розділу HAVING задає умову на цілу групу, а не на індивідуальні рядки.

Тому в арифметичних виразах предикатів, що входять в умову вибірки розділу HAVING, прямо можна використовувати тільки специфікації стовпців, зазначених в якості стовпців групування в розділі GROUP BY . Інші стовпці можна специфікувати тільки всередині специфікацій агрегатних функцій COUNT, SUM, AVG, MIN і MAX , що обчислюють в даному випадку деякий агрегатний значення для всієї групи рядків. Аналогічно іде справа у підзапитах, що входять в предикати умови вибірки розділу HAVING : якщо в підзапиті використовується характеристика поточної групи, то вона може здаватися тільки шляхом посилання на стовпці групування.

Результатом виконання пропозиції *HAVING* є згрупована таблиця, яка містить тільки ті групи рядків, для яких результат обчислення умови пошуку є *TRUE***.** Зокрема, якщо розділ *HAVING* присутній в табличному вираженні, що не містить *GROUP BY*, то результатом його виконання буде або порожня таблиця, або результат виконання попередніх розділів табличного вираження, що розглядається як одна група без стовпців групування.

ПРИКЛАД 6.11. Вибрати коди товарів, що купуються більш ніж одним покупцем.

Лістинг 6.11. Використання пропозиції HAVING COUNT

```
SELECT stock FROM ordsale
GROUP BY stock
HAVING COUNT (*)> 1;
```

ПРИКЛАД 6.12. Отримати значення мінімального і максимального окладу для клерків кожного відділу, де найнижче платню складає менше 1000.

Лістинг 6.11. Використання пропозиції HAVING MIN та MAX

```
SELECT deptno, MIN (sal), MAX (sal)
FROM emp WHERE job = 'CLERK'
GROUP BYdeptno
HAVING MIN (sal) <1000;
```

ПРИКЛАД 6.12. Сформувати запит для відбору мінімальної ціни продажу кожного товару за умови, що продаж здійснено 01.06.2025.

Пропозиція *HAVING* може мати тільки ті аргументи, що мають єдине значення для групи вихідних даних, тому наступна команда буде заборонена:

Лістинг 6.12. Приклад не правилного використання оператора HAVING

```
SELECT Код_товару,  MIN (Ціна)
FROM Продажі   GROUP BY Код_товару
HAVING DATE= 01.06.2025
```

ПРИКЛАД 6.13. Наведемо правильний спосіб виконати розглянутий запит.

Лістинг 6.13. Приклад правилного запиту для відбору мінімальної ціни продажу, яка була здійснена 01.06.2025

```
SELECT Код_товару, MIN (Ціна)
FROM Продажі
WHERE DATE = 01.06.2025
GROUP BY Код_товару
```

Команда *HAVING* є важливим доповненням до механізму групування в SQL, дозволяє відбирати лише ті групи даних, які відповідають визначеним умовам та робить її незамінною при створенні підсумкових звітів та аналітичних запитів.

6.8. Команда ORDER BY: сортування результатів

У *SQL* результати запиту за замовчуванням повертаються у довільному порядку, який залежить від способу зберігання рядків у таблицях та стратегії

обробки, обраної системою керування базами даних. Щоб отримати впорядкований набір даних, використовується оператор *ORDER BY*, який визначає критерії сортування результатів.

Оператор *ORDER BY* дозволяє:

- впорядковувати результати за одним або кількома стовпцями;
- визначати напрямок сортування (за зростанням чи спаданням);
- застосовувати сортування за виразами, обчислюваними полями та функціями;
- комбінувати кілька рівнів сортування (наприклад, спочатку за містом, а всередині міста — за прізвищем).

Синтаксис пропозиції ORDER BY:

```
SELECT стовпець_1, стовпець_2, ...
FROM назва_таблиці
ORDER BY умова_сортування1 [ASC | DESC] [,умова_сортування2 [ASC | DESC], ...]
GO
```

Основні ключові слова і параметри пропозиції ORDER BY:

умова_сортування – задає поле оператора *SELECT*, за яким виконується сортування поля, де відбувається упорядкування даних (повинно бути вказано в операторі *SELECT*);

ASC (за зростанням) і *DESC* (за убуванням) визначають порядок (напрямок) сортування.Зазамовчуванням сортування виконується по зростанню *ASC*, одночасно *ASC* і *DESC* не можуть бути задані.

ВАЖЛИВО:

У реченні ORDER BY можна задавати декілька виразів. Спочатку упорядковані рядки, групуються на значеннях для першого виразу, потім упорядковуються по другому вислову і так далі. NULL-значення групується після всіх інших при впорядковані в порядку зростання і перед усіма іншими, при сортуванні в порядку спадання .

Замість імені стовпця можна вказати його позицію для скорочення запису довгого виразу.

Крім того, при складанні складних запитів, що містять множинні оператори UNION, INTERSECT, MINUS, або UNION ALL ,

в реченні ORDER BY краще використовувати позиції, ніж безпосередньо самі вирази. Пропозиція ORDER BY може з'являтися тільки в останнійфразі запиту і сортує рядки, отримані всім складовим запитом в цілому.

Пропозиція *ORDER BY* підпорядковане наступним обмеженням:

- Якщо в утвердженні *SELECT* використовуються і оператор *ORDER BY* і оператор *DISTINCT*, то пропозиція *ORDER BY* не може посилатися на стовпці, які не згадуються в списку вибору обраних стовпців.
- Пропозиція *ORDERBY* не може з'являтися в підзапитах всередині інших тверджень.

ПРИКЛАД 6.11. Вивести список книг, відсортований за зростанням найменувань книг:

Лістинг 6.11. Використання команди ORDER BY

```
SELECT *
FROM Bookstore.PriceList
ORDER BY НазваКниги ASC;
GO
```

ПРИКЛАД 6.12. Використання пропозиції *ORDER BY* в зростаючому і спадному порядку. Вибрати з *EMP* записи по службовцям, впорядковані спочатку по зростанню номера відділу а потім по спадаючій розміру окладу.

Лістинг 6.12. Використаннія пропозиції ORDER BY в зростаючому і спадному порядку

```
SELECT Ename, Deptno, Sal
FROM Emp
ORDER BY Deptno ASC, Sal DESC ;
```

Питання для самоперевірки

1. *Дайте визначення пропозиції SELECT.*
2. *Назвіть основні ключові слова і параметри пропозиції SELECT.*
3. *Назвіть символи, що використовуються в синтаксичних конструкціях пропозиції SELECT.*
4. *Охарактеризуйте класифікацію SQL-функцій.*
5. *Дайте визначення пропозиції FROM.*
6. *Назвіть символи, що використовуються в синтаксичних конструкціях пропозиції WHERE.*
7. *Перерахуйте предикати умови пошуку (булевскі оператори).*
8. *Опишіть синтаксис предиката BETWEEN та наведіть приклад.*
9. *В яких випадках використовується предикат BETWEEN? Наведіть синтаксис та приклад.*
10. *Наведіть приклади використання предикатів ISNULL, NOTNULL, EXISTS, NOTEXISTS.*
11. *Дайте визначення пропозиції GROUP BY.*
12. *Перелічить функції, які відносяться до агрегатних функцій.*
13. *Синтаксис пропозицій HAVING та ORDER BY.*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 6

на тему: «Реалізація однотабличних запитів до бази даних засобами мови SQL»

МЕТА: *ознайомитися з принципами формування однотабличних запитів до бази даних у середовищі MS SQL Server, навчитися отримувати, фільтрувати, групувати та впорядковувати дані за допомогою базових конструкцій мови SQL.*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. Створити запити на вибірку даних із використанням оператора *SELECT*.
2. Сформувати умов відбору даних за допомогою оператора *WHERE* та предикатів порівняння, *BETWEEN, IN, LIKE, IS NULL*.
3. Застосовувати вирази та вбудовані скалярні функції *SQL* для обчислення й відображення похідних значень у запитах.
4. Створити запити з використанням агрегатних функцій та операторів *GROUP BY, HAVING* для групування та узагальнення даних.
5. Виконати сортування результатів запитів за допомогою оператора *ORDER BY*.
6. Проаналізувати результати виконання однотабличних запитів і зробити висновки щодо ефективності застосованих операторів.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Для виконання лабораторної роботи необхідно завантажити базу даних *DB_Book* у середовище *MS SQL Server Management Studio*, яке використовується як основний інструмент для роботи з системою керування базами даних *Microsoft SQL Server*. Після завантаження бази потрібно ознайомитися зі структурою таблиць, що належать до схеми *Bookstore*. Зокрема, до її складу входять таблиці *Genres, Publishers, Authors, PriceList, Clients, Sales* та *SalesItems,* які забезпечують зберігання довідникової, транзакційної та аналітичної інформації про діяльність книжкового магазину.

Після ознайомлення зі структурою бази даних необхідно переконатися у наявності заповнених даних у всіх таблицях. Це дозволить виконувати запити

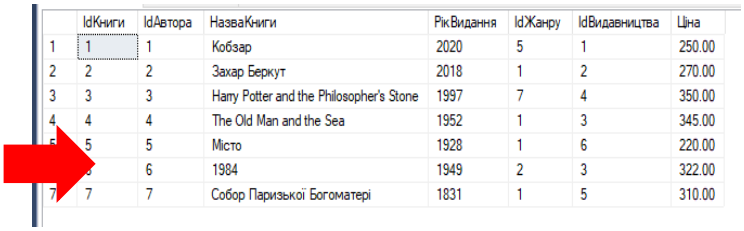
вибірки, фільтрації, групування та сортування даних на основі реальних записів, що забезпечує повноцінну імітацію роботи з прикладною базою даних.

Створити запити на вибірку даних із використанням оператора *SELECT*

Приклад 1. Визначити повний перелік записів таблиці *PriceList* для перегляду всієї інформації про книги, наявні в базі даних.

```
USE DB_Book
GO

SELECT *
FROM Bookstore.PriceList;
```

Оператор *SELECT* * повертає всі поля таблиці, вказаній у пропозиції *FROM* без фільтрації. Такий запит доцільно використовувати на етапі первинного перегляду вмісту таблиці. У результаті виконання даного запиту буде така результуюча таблиця:

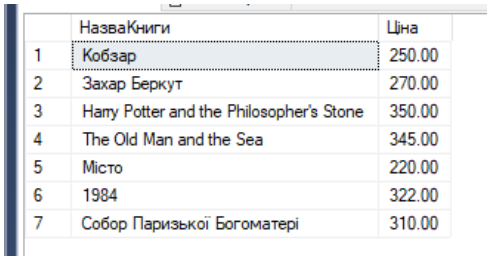


	IdКниги	IdАвтора	НазваКниги	РікВидання	IdЖанру	IdВидавництва	Ціна
1	1	1	Кобзар	2020	5	1	250.00
2	2	2	Захар Беркут	2018	1	2	270.00
3	3	3	Harry Potter and the Philosopher's Stone	1997	7	4	350.00
4	4	4	The Old Man and the Sea	1952	1	3	345.00
5	5	5	Місто	1928	1	6	220.00
6	6	6	1984	1949	2	3	322.00
7	7	7	Собор Паризької Богоматері	1831	1	5	310.00

Приклад 2. Отримати перелік назв книг та їх вартості з таблиці *PriceList*, обмеживши вибірку лише двома полями.

```
USE DB_Book
GO

SELECT НазваКниги, Ціна
FROM Bookstore.PriceList;
```

У цьому прикладі оператор *SELECT* використовується для відбору окремих атрибутів таблиці. Виводяться лише назви та ціни книг.



	НазваКниги	Ціна
1	Кобзар	250.00
2	Захар Беркут	270.00
3	Harry Potter and the Philosopher's Stone	350.00
4	The Old Man and the Sea	345.00
5	Місто	220.00
6	1984	322.00
7	Собор Паризької Богоматері	310.00

Приклад 3. Для авторів виконати об’єднання (конкатенацію) прізвища та ініціалу імені в один стовпець для відображення скороченого запису імені автора.

```
USE DB_Book
GO

SELECT Прізвище + ' ' + LEFT(Імя, 1) + '.' AS Автор
FROM Bookstore.Authors;
```

	Автор
1	Шевченко Т.
2	Франко І.
3	Rowling J.
4	Hemingway E.
5	Підмогильний В.
6	Orwell G.
7	Hugo V.

Використано функцію *LEFT()* для отримання першої літери імені та оператор конкатенації + для об'єднання рядкових значень. Після поля прізвища додано пробіл, з метою розділення значень двох полів між собою У результаті вийшло нове поле, якому присвоїлась назва *Автор*.

Формування умов відбору даних за допомогою оператора *WHERE*

Приклад 4. Визначити книги, вартість яких перевищує 250 грн.

```
USE DB_Book
GO

SELECT НазваКниги, Ціна
FROM Bookstore.PriceList
WHERE Ціна > 250;
```

	НазваКниги	Ціна
1	Захар Беркут	270.00
2	Harry Potter and the Philosopher's Stone	350.00
3	The Old Man and the Sea	345.00
4	1984	322.00
5	Собор Паризької Богоматері	310.00

Оператор *WHERE* використовується для відбору рядків, які відповідають умові порівняння. У результаті виконання запиту виведені лише ті книги, ціна яких більша за 250 грн.

Приклад 5. Отримати перелік клієнтів, що зареєстровані у місті Київ.

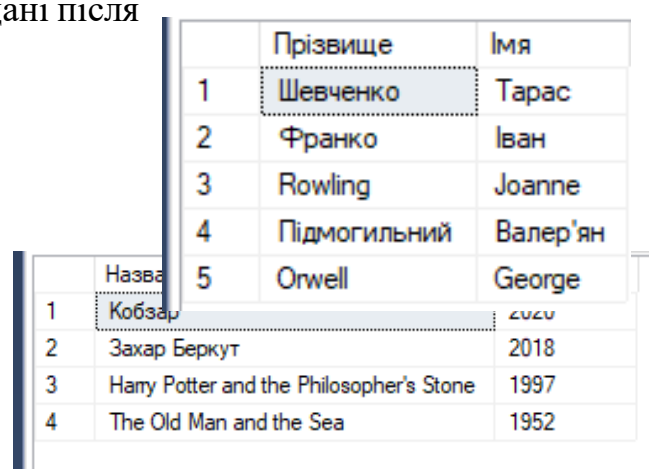
```
USE DB_Book
GO

SELECT НайменуванняКлієнта, АдресаКлієнта
FROM Bookstore.Clients
WHERE АдресаКлієнта LIKE N'%Київ%';
```

	НайменуванняКлієнта	АдресаКлієнта
1	ТОВ Книгарня Є	м. Київ, вул. Хрещатик, 22

Предикат *LIKE* дозволяє виконувати пошук підрядка в текстовому полі. Символи % позначають довільну кількість будь-яких символів.

Приклад 6. Визначити книги, видані після 1950 року.

```
USE DB_Book
GO

SELECT НазваКниги, РікВидання
FROM Bookstore.PriceList
WHERE РікВидання > 1950;
```



	Прізвище	Ім'я
1	Шевченко	Тарас
2	Франко	Іван
3	Rowling	Joanne
4	Підмогильний	Валер'ян
5	Orwell	George

	Назва	
1	Кобзар	2020
2	Захар Беркут	2018
3	Harry Potter and the Philosopher's Stone	1997
4	The Old Man and the Sea	1952

Умовою *РікВидання > 1950* здійснюється фільтрація записів за числовим критерієм. У результаті виводяться книги, що видані після зазначеного року.

Приклад 7. Визначити книги, ціна яких знаходиться в межах від 200 до 300 грн.

```
USE DB_Book
GO

SELECT НазваКниги, Ціна
FROM Bookstore.PriceList
WHERE Ціна BETWEEN 200 AND 300;
```

	НазваКниги	Ціна
1	Кобзар	250.00
2	Захар Беркут	270.00
3	Місто	220.00

Предикат *BETWEEN* використовується для відбору записів, значення яких належить до заданого діапазону включно з межами.

Приклад 8. Отримати дані про авторів, які є громадянами України або Великої Британії.

```
USE DB_Book
GO

SELECT Прізвище, Імя
FROM Bookstore.Authors
WHERE IdКраїни IN (1, 4);
```

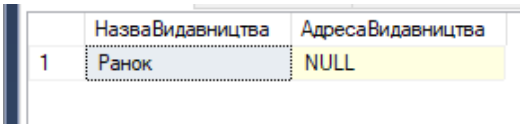
У цьому запиті використовується предикат IN, який дозволяє перевіряти, чи належить значення конкретного стовпця до заданого набору констант. Конструкція *IdКраїни IN (1, 4)* означає, що будуть відібрані лише ті записи з таблиці *Bookstore.Authors*, у яких значення поля *IdКраїни* дорівнює *1* або *4*. Ці числові значення є зовнішніми ключами, що посилаються на таблицю

Countries ідентифікаторів країн, в якій значення 1 відповідає Україні, а 4 – Великій Британії. Таким чином, запит повертає прізвища та імена авторів, які походять із зазначених країн, використовуючи числові ідентифікатори замість текстових назв.

Приклад 9. Визначити видавництва, у яких не вказано адресу.

```
USE DB_Book
GO

SELECT НазваВидавництва, АдресаВидавництва
FROM Bookstore.Publishers
WHERE АдресаВидавництва IS NULL;
```

Предикат *IS NULL* дозволяє знаходити записи, у яких значення поля відсутнє. Це корисно для контролю повноти даних у таблицях.



	НазваВидавництва	АдресаВидавництва
1	Ранок	NULL

Застосовування виразів та вбудованих скалярних функцій *SQL*

Приклад 10. Вивести назву книги та її ціну, округлену до цілих гривень.

```
USE DB_Book
GO

SELECT НазваКниги, ROUND(Ціна, 0) AS Ціна_округлена
FROM Bookstore.PriceList;
```

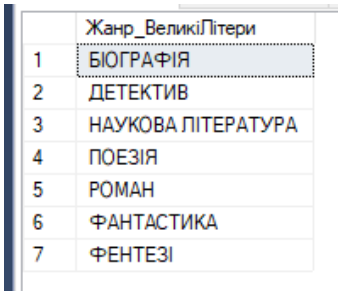
У даному запиті використовується вбудована скалярна функція *ROUND*, яка дозволяє округлити числове значення до зазначеної кількості десяткових розрядів. У цьому випадку аргумент 0 означає округлення до цілих чисел.

Застосування цієї функції дозволяє представити цінові показники у більш зручному для аналізу та звітності вигляді без дробових частин.

Приклад 11. Вивести всі жанри великими літерами.

```
USE DB_Book
GO

SELECT UPPER(НазваЖанру) AS Жанр_ВеликіЛітери
FROM Bookstore.Genres;
```

У цьому запиті застосовано вбудовану скалярну функцію *UPPER*, яка перетворює всі символи рядка у верхній регістр. Це дозволяє уніфікувати відображення назв жанрів, що забезпечує єдину стилістику в інтерфейсі користувача або у звітах, а також полегшує пошук даних за текстовими ознаками.

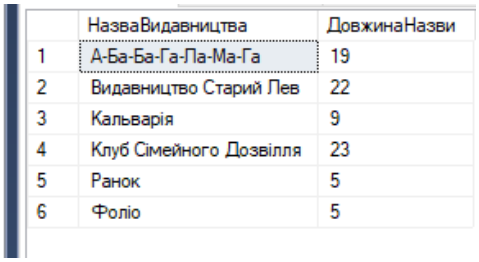


	Жанр_ВеликіЛітери
1	БІОГРАФІЯ
2	ДЕТЕКТИВ
3	НАУКОВА ЛІТЕРАТУРА
4	ПОЕЗІЯ
5	РОМАН
6	ФАНТАСТИКА
7	ФЕНТЕЗІ

Приклад 12. Вивести назву видавництва та кількість символів у його назві.

```
USE DB_Book
GO

SELECT НазваВидавництва, LEN(НазваВидавництва) AS ДовжинаНазви
FROM Bookstore.Publishers;
```

Запит використовує функцію *LEN*, яка обчислює кількість символів у рядковому значенні. Таке застосування дозволяє аналізувати довжину назв видавництв, що може бути корисним для форматування інтерфейсу, підготовки звітів або контролю за відповідністю стандартам оформлення даних.



	НазваВидавництва	ДовжинаНазви
1	А-Ба-Ба-Га-Ла-Ма-Га	19
2	Видавництво Старий Лев	22
3	Кальварія	9
4	Клуб Сімейного Дозвілля	23
5	Ранок	5
6	Фоліо	5

Приклад 13. Для кожного замовлення визначити кількість днів між датою оформлення та датою оплати.

```
USE DB_Book
GO

SELECT IdЧеку, DATEDIFF(DAY, ДатаЗамовлення, ДатаСплати) AS
Кількість_Днів
FROM Bookstore.Sales;
```

У цьому запиті використовується вбудована функція *DATEDIFF*, яка обчислює різницю між двома датами у зазначених одиницях виміру. Аргумент *DAY* вказує, що обчислення здійснюється у днях.

Таке застосування дозволяє оцінити тривалість обробки кожного замовлення, що є корисним для аналізу оперативності виконання замовлень та контролю бізнес-процесів.

Приклад 14. Для кожного замовлення, оформленого у 2025 році, вивести рік його оформлення.

```
USE DB_Book
GO

SELECT IdЧеку, YEAR(ДатаЗамовлення) AS Рік_Замовлення
FROM Bookstore.Sales
WHERE YEAR(ДатаЗамовлення) = 2025;
```

	IdЧеку	Рік_Замовлення
1	13	2025
2	14	2025
3	16	2025
4	17	2025

Функція *YEAR* дозволяє виділити рік із значення дати. Умовою *WHERE* відбираються лише ті замовлення, які були оформлені у *2025* році. Це забезпечує аналіз активності клієнтів протягом конкретного календарного року та спрощує підготовку статистичних звітів і планування продажів.

Приклад 15. Вивести всі замовлення, оформлені у другому кварталі року, та показати номер кварталу.

```
USE DB_Book
GO

SELECT IdЧеку, ДатаЗамовлення, DATEPART(QUARTER, ДатаЗамовлення) AS
Квартал
FROM Bookstore.Sales
WHERE DATEPART(QUARTER, ДатаЗамовлення) = 2;
```

	IdЧеку	ДатаЗамовлення	Квартал
1	4	2024-04-20	2
2	5	2024-05-11	2
3	6	2024-06-25	2

Функція *DATEPART* дозволяє виділити певну частину дати, у цьому випадку номер кварталу. Значення *QUARTER* означає, що з дати буде отримано номер кварталу (1–4).

Умовою *WHERE* відбираються лише ті записи, які належать до другого кварталу року. Це дозволяє аналізувати активність замовлень за кварталами та підготовляти квартальні звіти.

Використання агрегованих функцій

Приклад 16. Підрахувати загальну кількість клієнтів у таблиці *Clients*

```
USE DB_Book
```

```
GO

SELECT COUNT(*) AS [Кількість клієнтів]
FROM Bookstore.Clients;
```

	Кількість клієнтів
1	7

Функція *COUNT(*)* обчислює загальну кількість рядків у таблиці, незалежно від наявності значень у конкретних полях і дозволяє визначити повну чисельність клієнтської бази, що є необхідним показником для управління взаємодією з клієнтами та аналітики продажів.

Приклад 17. Визначити максимальну та мінімальну вартість книг у таблиці *PriceList*.

```
USE DB_Book
GO

SELECT MAX(Ціна) AS [Найвища ціна], MIN(Ціна) AS [Найнижча ціна]
FROM Bookstore.PriceList;
```

	Найвища ціна	Найнижча ціна
1	350.00	220.00

Функції *MAX()* та *MIN()* дозволяють швидко визначити граничні значення числового поля. Використання цих функцій дає змогу оцінити діапазон цін на книги, що важливо для формування цінової стратегії та аналізу ринку.

Приклад 18. Визначити кількість книг у кожному жанрі.

```
SELECT IdЖанру, COUNT(*) AS [Кількість книг]
FROM Bookstore.PriceList
GROUP BY IdЖанру;
```

	IdЖанру	Кількість книг
1	1	4
2	2	1
3	5	1
4	7	1

Оператор *GROUP BY* дозволяє об’єднати записи за спільною ознакою — у даному випадку за ідентифікатором жанру. Функція *COUNT(*)* підраховує кількість книг у кожній групі. Це дозволяє оцінити розподіл книг за жанрами та планувати асортимент відповідно до популярності категорій.

Приклад 19. Вивести лише ті жанри, у яких кількість книг перевищує 1.

```
SELECT IdЖанру, COUNT(*) AS [Кількість книг]
FROM Bookstore.PriceList
```

```
GROUP BY IdЖанру
HAVING COUNT(*) > 1;
```

	IdЖанру	Кількість книг
1	1	4

Оператор *HAVING* застосовується для накладання умов на агреговані результати, отримані після групування.

У даному випадку умова *COUNT(*) > 1* дозволяє відфільтрувати жанри, які містять більше ніж одну книгу. Це важливо для аналізу насиченості асортименту за категоріями.

Сортування результатів запитів

Приклад 20. Впорядкувати книги за зростанням ціни.

```
USE DB_Book
GO

SELECT НазваКниги, Ціна
FROM Bookstore.PriceList
ORDER BY Ціна ASC;
```

	НазваКниги	Ціна
1	Місто	220.00
2	Кобзар	250.00
3	Захар Беркут	270.00
4	Собор Паризької Богоматері	310.00
5	1984	322.00
6	The Old Man and the Sea	345.00
7	Harry Potter and the Philosopher's Stone	350.00

Оператор *ORDER BY* забезпечує сортування результатів запиту за зазначеним полем. Ключове слово *ASC* вказує на зростаючий порядок. Використання сортування дозволяє легше аналізувати асортимент за ціною та виявляти найдешевші або найдорожчі позиції.

Приклад 16. Вивести перелік авторів, впорядкований за алфавітом у спадному порядку прізвищ.

```
USE DB_Book
GO

SELECT Прізвище, Імя
FROM Bookstore.Authors
ORDER BY Прізвище DESC;
```

	Прізвище	Імя
1	Шевченко	Тарас
2	Франко	Іван
3	Підмогильний	Валер'ян
4	Rowling	Joanne
5	Orwell	George
6	Hugo	Victor
7	Hemingway	Ernest

Ключове слово *DESC* забезпечує сортування у зворотному алфавітному порядку за прізвищем автора. Такий підхід дозволяє формувати впорядковані списки авторів для звітів, каталогів або інших аналітичних задач, де потрібен зворотний порядок відображення.

ТЕМА 7. ВКЛАДЕНІ ТА БАГАТОТАБЛИЧНІ ЗАПИТИ

ПЛАН:

7.1. Поняття вкладеного підзапиту: визначення та загальні принципи.
7.2. Прості вкладені підзапити.
7.3. Корельовані вкладені підзапити.
7.4. Багаторівневі вкладені підзапити.
7.5. Операції об'єднання результатів запитів.
7.6. Природнє об'єднання таблиць.
7.7. Практичні рекомендації з написання та оптимізації запитів.

7.1. Поняття вкладеного підзапиту: визначення та загальні принципи

У мові SQL важливе місце займає механізм вкладених підзапитів (nested subqueries), що забезпечує можливість багаторівневої обробки даних.

Вкладений підзапит — оператор *SELECT*, розташований усередині іншого запиту та використаний як його складова частина. Інакше кажучи, підзапит є внутрішнім запитом, результат виконання якого виступає проміжним джерелом даних або умовою для обробки у зовнішньому запиті.

Вкладені підзапити можуть з'являтися в різних частинах зовнішнього запиту: у виразах після ключових слів *WHERE, HAVING, FROM, SELECT*. Найчастіше вони використовуються у розділі *WHERE* для формування умов відбору, що ґрунтуються на даних іншої таблиці або навіть на результаті складного обчислення. Завдяки цьому підзапити розширюють можливості мови *SQL* у напрямі створення гнучких та багатокрокових критеріїв пошуку.

Загальний принцип функціонування вкладеного підзапиту полягає в тому, що система керування базами даних спочатку виконує внутрішній запит, формуючи його результатну множину, а потім використовує цю множину як умову або джерело для обробки у зовнішньому запиті. Така багаторівнева схема дозволяє розділити складну задачу на окремі логічні частини, зберігаючи декларативний характер *SQL*-запитів.

Залежно від характеру зв'язку із зовнішнім запитом виділяють два основні типи вкладених підзапитів: прості (некорельовані) та корельовані. У

першому випадку внутрішній запит виконується один раз і його результат використовується для обчислення умов зовнішнього запиту. У другому випадку підзапит безпосередньо залежить від значень рядків зовнішнього запиту, а отже, виконується багаторазово, окремо для кожного рядка результатної множини.

Таким чином, вкладені підзапити є універсальним інструментом вираження складних умов і зв'язків між таблицями без необхідності явного використання з'єднань (*JOIN*). Вони підвищують гнучкість запитів, забезпечують більш природне відображення складних умов пошуку і дозволяють скоротити обсяг коду завдяки використанню вкладеної логіки.

Зауваження:

Слід зазначити, що SQL володіє великою надмірністю в тому сенсі, що він часто надає кілька різних способів формулювання одного і того ж запиту.І незважаючи на те, що частина з них успішно реалізується за допомогою з'єднань, в даній темі все ж таки будуть приведені їх варіанти з використанням вкладених підзапитів. Це пов'язано з необхідністю детального знайомства зі створенням і принципом виконання вкладених підзапитів, так як існує чимало завдань (особливо на видалення і зміна даних), які не можуть бути реалізовані в інший спосіб.

7.2. Прості вкладені підзапити

Простий вкладений підзапит — *SQL*-вираз, розташований у дужках усередині іншого запиту, який виконується раніше та передає свій результат до зовнішнього запиту. Він розглядається як «запит у запиті» й використовується для формування проміжної результатної множини, що може виступати в ролі операнда в умовах пошуку (*WHERE*), у виразах вибірки (*SELECT*) або в інших частинах *SQL*-інструкції.

Тобто, прості вкладені підзапити обробляються системою «знизу до верху». Першим обробляється вкладений підзапит самого нижнього рівня, а

значення, отриманні в результаті його виконання, використовується при реалізації підзапиту більш високого рівня тощо.

Головною ознакою простого підзапиту є його незалежність від зовнішнього запиту. Це означає, що підзапит може бути виконаний окремо, а його результат не залежить від значень рядків, які обробляє зовнішній запит. Унаслідок цього прості підзапити обчислюються лише один раз під час виконання всієї інструкції, а не повторно для кожного рядка результату.

Семантично простий підзапит може повертати:

- одне скалярне значення (наприклад, мінімальну або максимальну ціну проданих книги);
- один стовпець зі списком значень (наприклад, усі ідентифікатори клієнтів, які зробили замовлення);
- повну таблицю з кількома стовпцями (хоча такий варіант частіше застосовується у вигляді віртуальної таблиці в секції *FROM*).

Найчастіше прості вкладені підзапити застосовують у секції *WHERE* разом з операторами:

- *IN* – перевірка належності значення до множини, сформованої підзапитом;
- порівняння (=, <, >, <=, >=, <>) – коли підзапит повертає скалярне значення;
- ANY / SOME, ALL – коли потрібно зіставити значення з кожним або хоча б одним елементом множини, що повертає підзапит.

ПРИКЛАД 7.1. Вивести список авторів книг, у яких ціна примірника більше 300:

Лістинг 7.1. Використання предикату IN у простому вкладенному підзапиті

```
USE DB_Book
GO
-- Автори книг, ціна яких перевищує 300
SELECT Прізвище, Ім'я
FROM Bookstore.Authors
WHERE IdАвтора IN (
SELECT IdАвтора
FROM Bookstore.PriceList
WHERE Ціна > 300
);
```

Отже, прості вкладені підзапити є потужним засобом для побудови багатокрокових умов відбору даних. Вони дозволяють уникати складних об’єднань таблиць у простих випадках, роблять код більш зрозумілим і логічним, а також сприяють модульності запитів: кожен підзапит можна окремо перевіряти та налагоджувати.

7.3. Корельовані вкладені підзапити

Корельовані вкладені підзапити є важливим інструментом мови SQL, який дозволяє виконувати більш складні логічні перевірки та формувати динамічні критерії відбору даних.

Корельований підзапит є особливим видом вкладеного запиту, який виконується не самостійно, а залежить від значень, що беруться із зовнішнього запиту.

На відміну від звичайних (некорельованих) підзапитів, які обчислюються один раз і передають свій результат у зовнішній запит, корельований підзапит виконується повторно для кожного рядка, що розглядається зовнішнім запитом. Саме ця властивість обумовлює як його гнучкість, так і відносно високу обчислювальну вартість.

Цікавий факт:

Механізм виконання корельованого підзапиту полягає у тому, що система керування базами даних для кожного кортежу зовнішньої таблиці передає у підзапит значення відповідних атрибутів, після чого обчислює результат вкладеного запиту. Отримане значення або булевий результат порівнюється з умовами, визначеними у зовнішньому запиті, що й визначає, чи буде конкретний рядок включено до результатної множини. Цей процес повторюється для кожного рядка, що робить виконання корельованих підзапитів менш продуктивним, але більш гнучким порівняно з простими підзапитами.

Класичним прикладом корельованого підзапиту є використання предиката *EXISTS* або *NOT EXISTS*. У цьому випадку зовнішній запит формує початковий набір рядків, а підзапит перевіряє наявність відповідних даних у пов’язаній таблиці для кожного конкретного рядка. Наприклад, можна знайти

всіх клієнтів, які зробили хоча б одне замовлення. Тут підзапит перевіряє існування записів у таблиці замовлень для кожного клієнта, що забезпечує більш точну і контекстну вибірку даних.

У теоретичному плані корельований підзапит можна розглядати як форму зв'язку між двома рівнями запиту, коли внутрішній рівень має доступ до атрибутів зовнішнього рівня. Таким чином, результат підзапиту залежить від поточного кортежу, який обробляється зовнішнім запитом. Це відрізняє його від простих підзапитів, де внутрішня вибірка цілком незалежна і може бути виконана автономно.

Розглянемо типовий приклад. Припустімо, що у базі даних зберігаються таблиці Book (книги) та Author (автори). Завдання полягає у виборі тих авторів, які мають принаймні одну книгу з ціною понад 500 гривень:

ПРИКЛАД 7.2. Вивести список тих авторів, які мають принаймні одну книгу з ціною понад 500 гривень:

Лістинг 7.2. Використання корельованого підзапиту

```
SELECT a.Прізвище, a.Ім'я
FROM Bookstore.Authors a
WHERE EXISTS (
SELECT 1
FROM Bookstore.PriceList p
WHERE p.IdАвтора = a.IdАвтора
AND p.Ціна > 500
);
```

У цьому прикладі внутрішній підзапит виконується для кожного рядка зовнішньої таблиці *Author*. Зовнішній атрибут *a.AuthorID* використовується у внутрішньому запиті для перевірки умови відповідності книг конкретному автору. Якщо для автора знайдеться хоча б один рядок у таблиці *Book*, що задовольняє умову *Price* > 500, підзапит поверне істину, і відповідний автор буде включений до результату.

З погляду оптимізації важливо розуміти, що корельовані підзапити можуть виконуватися багаторазово, що збільшує навантаження на систему при великих обсягах даних. Тому на практиці їх часто замінюють на еквівалентні запити з використанням з'єднань (*JOIN*). Проте в низці випадків саме корельований підзапит дає більш інтуїтивно зрозумілий і декларативний опис задачі, особливо коли необхідна перевірка умови існування або відсутності певних записів (*EXISTS, NOT EXISTS*).

Отже, корельовані підзапити становлять потужний інструмент SQL, який дозволяє формулювати складні умови відбору на основі взаємозалежних даних, і водночас потребують усвідомленого підходу до оптимізації.

7.4. Багаторівневі вкладені підзапити

Багаторівневі вкладені підзапити становлять один із найпотужніших механізмів мови SQL, оскільки дозволяють реалізовувати складні схеми відбору та обробки даних.

Багаторівневий вкладений підзапит	конструкція у мові *SQL*, коли один підзапит містить у собі інший підзапит, утворюючи багаторівневу ієрархію.

Іншими словами, результати виконання одного підзапиту стають вхідними даними для наступного, утворюючи своєрідний каскадний ланцюг залежностей. Такі конструкції суттєво підвищують виразні можливості мови *SQL*, надаючи розробнику інструменти для поетапного формування складних умов відбору.

Основною передумовою використання багаторівневих вкладених підзапитів є необхідність поступової деталізації пошуку або зіставлення даних. Наприклад, один підзапит може відібрати множину ідентифікаторів об’єктів за певним критерієм, інший – уточнити цю множину за додатковими умовами, а зовнішній запит остаточно сформує результатну таблицю для користувача. Такий підхід дозволяє чітко розділяти логіку пошуку на окремі етапи, роблячи запит більш зрозумілим і структурованим.

З технічної точки зору, багаторівневі підзапити можуть бути реалізовані у будь-якій частині конструкції *WHERE, HAVING* чи навіть *FROM*. У найтиповішому випадку вони зустрічаються в умовах відбору, коли значення атрибутів порівнюються не з константою, а з результатом іншого запиту. При цьому глибина вкладення може бути довільною, хоча на практиці вона рідко перевищує два-три рівні через складність читання та відлагодження коду.

Слід зазначити, що багаторівневі вкладені підзапити мають низку особливостей виконання. Система керування базами даних обчислює найглибший підзапит і передає його результат на вищий рівень. Цей процес повторюється доти, доки не буде сформовано остаточну множину даних для зовнішнього запиту. Таким чином, порядок обчислень є ієрархічним і відповідає принципу «зсередини назовні». Однак фактичний план виконання може відрізнятися залежно від оптимізатора СКБД, який намагається мінімізувати витрати часу та пам’яті.

Перевагою багаторівневих підзапитів є їхня концептуальна наочність: складна задача розбивається на низку менших, кожна з яких розв'язується власним підзапитом. Це робить підхід особливо корисним для проектування систем, де прозорість логіки має критичне значення. Водночас надмірне використання багаторівневих конструкцій може призводити до зниження продуктивності, тому в практичних сценаріях часто рекомендується перетворювати їх на запити з'єднання (*JOIN*) або застосовувати тимчасові таблиці.

ПРИКЛАД 7.3. Вивести список авторів книг, у яких ціна примірника більше 300:

Лістинг 7.3. Використання багатотабличного запиту

```
SELECT a.Прізвище, a.Ім'я, p.НазваКниги, p.Ціна
FROM Bookstore.Authors a
JOIN Bookstore.PriceList p
   ON a.IdАвтора = p.IdАвтора
WHERE p.Ціна > 300;
```

Отже, багаторівневі вкладені підзапити становлять важливий інструмент у арсеналі розробника SQL. Вони забезпечують високу гнучкість у побудові запитів, дозволяють відокремлювати різні аспекти логіки пошуку та працювати з даними на кількох рівнях абстракції. Правильне використання таких конструкцій сприяє підвищенню зрозумілості програмного коду, проте вимагає обережності з огляду на можливі витрати обчислювальних ресурсів.

7.5. Операції об'єднання результатів запитів

Під час роботи з базами даних часто виникає необхідність отримати узагальнений набір результатів, який формується не з однієї таблиці чи запиту, а шляхом об'єднання кількох вибірок. Для цього у мові *SQL* передбачені спеціальні операції об'єднання результатів. Вони дозволяють поєднувати рядки з різних запитів у єдину таблицю результату, зберігаючи логіку реляційної моделі даних. Застосування таких операцій забезпечує більш високу гнучкість при формуванні інформаційних вибірок і розширює аналітичні можливості користувача.

Основними операторами об'єднання є *UNION, UNION ALL, INTERSECT* та *EXCEPT*. Всі вони виконують над множинами рядків дії, подібні до операцій з теорії множин: об'єднання, перетин або різницю. Для коректного

застосування цих операторів обов'язковою умовою є сумісність запитів, тобто однакова кількість стовпців у їхніх результатах і відповідність типів даних у кожному стовпці.

Оператор *UNION* використовується для об'єднання результатів двох або більше запитів у єдиний набір рядків із автоматичним вилученням дублюючих записів. Це означає, що кінцева вибірка є об'єднанням множин, а кожен рядок зберігається лише один раз, навіть якщо він з'являвся у кількох запитах. Якщо ж необхідно зберегти всі повтори, застосовується оператор *UNION ALL*. Він поєднує результати без вилучення дублюючих рядків, що може бути корисним при підрахунку статистичних показників, але водночас потребує більшої обережності, оскільки обсяг результату може значно збільшитися.

ПРИКЛАД 7.4. Видати код книг, в яких ціна за примірник дорівнює 300, а кількість проданих примірників більше 20:

Лістинг 7.4. Використання операції об'єднання UNION

```
-- Книги з ціною рівною 300
SELECT IdКниги
FROM Bookstore.PriceList
WHERE Ціна = 300
UNION
-- Книги, у яких загальна кількість проданих примірників >20
SELECT SI.IdКниги
FROM Bookstore.SalesItems SI
GROUP BY SI.IdКниги
HAVING SUM(SI.Кількість) > 20;
```

Для пошуку спільних рядків між двома вибірками використовується оператор *INTERSECT*. Він повертає лише ті записи, які одночасно присутні у результатах обох запитів. Така операція є аналогом математичного перетину множин і використовується тоді, коли важливо виділити дані, що мають однакові характеристики у кількох джерелах. Прикладом може бути пошук клієнтів, які зробили покупки у двох різних магазинах.

ПРИКЛАД 7.5. Знайти клієнтів, які купували книги і чиї дані вже зберігаються у таблиці клієнтів:

Лістинг 7.5. Використання операції об'єднання INTERSECT

```
SELECT НайменуванняКлієнта
```

```
FROM Bookstore.Clients
INTERSECT
SELECT C.НайменуванняКлієнта
FROM Bookstore.Clients C
JOIN Bookstore.Sales S ON C.IdКлієнта = S.IdКлієнта;
```

Якщо клієнт занесений у довідник, але ще нічого не купував, він до результату не потрапить.

Ще однією корисною операцією є *EXCEPT*, яка виконує різницю множин. Вона повертає ті рядки, які входять у результат першого запиту, але відсутні у другому. Такий підхід застосовується для виявлення унікальних даних, що не мають відповідників в іншій таблиці чи вибірці. Наприклад, це може бути пошук товарів, які наявні в одній базі, але відсутні у базі партнерського складу.

ПРИКЛАД 7.5. Знайти книги, що є у прайс-листі, але жодного разу не були продані**:**

Лістинг 7.5. Використання операції об'єднання EXCEPT

```
SELECT НазваКниги
FROM Bookstore.PriceList
EXCEPT
SELECT P.НазваКниги
FROM Bookstore.PriceList P
JOIN Bookstore.SalesItems SI ON P.IdКниги = SI.IdКниги;
```

Таким чином, операції об'єднання результатів запитів у SQL виступають важливим інструментом для формування складних вибірок та реалізації логіки роботи з даними на основі множинних відношень. Вони дозволяють зменшити кількість проміжних обчислень, підвищити наочність запитів та розширити можливості користувача у вирішенні практичних завдань аналітики. Однак при їх використанні слід враховувати як сумісність результатів запитів, так і вплив на продуктивність системи, особливо при роботі з великими обсягами даних.

7.6. Природнє об'єднання таблиць

Природне об'єднання таблиць у мові *SQL* є спеціальною формою об'єднання, яка дозволяє автоматично поєднувати рядки двох таблиць на основі спільних стовпців з однаковими іменами. При цьому користувачу не потрібно явно вказувати умову поєднання (*ON* або *USING*), *SQL* визначає

відповідність стовпців самостійно. Такий підхід значно спрощує запити та робить їх більш наочними, особливо коли таблиці мають багато однакових полів, що використовуються для зв'язку.

Природне об'єднання автоматично створює результат у вигляді таблиці, що містить усі стовпці обох таблиць, причому дублікати спільних стовпців виключаються. Це дозволяє уникнути повторного включення ідентичних колонок у результат, зберігаючи при цьому логічну цілісність даних. Таким чином, *NATURAL JOIN* виконує одночасно операцію поєднання та усунення надлишкових стовпців.

Логіка виконання *NATURAL JOIN* схожа на *INNER JOIN* із автоматично сформованою умовою поєднання. Рядки з обох таблиць з'єднуються лише у випадку, коли значення у всіх спільних стовпцях однакові. Якщо хоча б одне значення не збігається, відповідний рядок не потрапляє до результату. Це гарантує, що в кінцевій вибірці будуть лише повністю узгоджені записи.

У практичній роботі з базами даних *NATURAL JOIN* часто застосовується для отримання агрегованої інформації з пов'язаних таблиць. Наприклад, у книжковому магазині можна об'єднати таблиці *PriceList* і *Authors* за стовпцем IdАвтора, щоб отримати список всіх книг з іменами авторів і цінами. Такий підхід дозволяє швидко формувати звіти, уникати ручного прописування умов поєднання і підвищує наочність *SQL*-запитів.

ПРИКЛАД 7.6. Отримати список всіх проданих книг із назвою книги, прізвищем автора та ціною:

Лістинг 7.6. Використання природнього об'єднання

```
SELECT PriceList.НазваКниги, Authors.Прізвище, PriceList.Ціна
FROM Bookstore.PriceList
NATURAL JOIN Bookstore.Authors;
```

або

```
SELECT p.НазваКниги, a.Прізвище, p.Ціна
FROM Bookstore.PriceList p
INNER JOIN Bookstore.Authors a ON p.IdАвтора = a.IdАвтора;
```

Друга конструкція еквівалентна *NATURAL JOIN* і дає той самий результат, тільки рядки з однаковими значеннями у спільному стовпці об'єднуються в одну вибірку.

7.7. Практичні рекомендації з написання та оптимізації запитів

У процесі роботи з базами даних ефективність виконання SQL-запитів має важливе значення, оскільки навіть незначні помилки у формулюванні запиту можуть призвести до суттєвих витрат ресурсів і зниження продуктивності системи. Тому при написанні простих запитів доцільно дотримуватися таких рекомендацій:

- Чіткість у виборі стовпців, тобто використання команди *SELECT ** є небажаним, оскільки вона повертає всі стовпці таблиці, включно з тими, які не потрібні для конкретного завдання. Рекомендується вказувати лише необхідні поля для зменшення обсягу переданих даних і пришвидшує виконання запиту.
- Усі фільтри, що можуть бути застосовані до окремих рядків, слід розміщувати в предикаті WHERE. Це дозволяє обмежити кількість оброблюваних даних ще до етапу групування або сортування.
- Використання предикату HAVING тільки для груп. Умови, які стосуються окремих записів, не повинні розташовуватись у HAVING. Дана конструкція використовується виключно для накладання обмежень на агреговані групи даних.
- Вкладені підзапити часто ускладнюють обчислення. За можливості слід віддавати перевагу використанню об’єднань (JOIN) замість складних підзапитів, що в свою чергу, сприяє більш ефективному виконанню складних запитів.
- Якщо таблиця має індекси за полями, які використовуються у фільтрації (*WHERE*) або сортуванні (*ORDER BY*), це суттєво прискорює доступ до даних. При проєктуванні запитів варто враховувати наявність таких індексів.
- Уникнення надмірних обчислень у запиті. Використання складних функцій або виразів безпосередньо у фільтрах може сповільнити виконання. Наприклад, обчислення функцій над стовпцем у пропозиції *WHERE* унеможливлює використання індексів. Доцільно переносити подібні обчислення у прикладний код або підготовчі етапи обробки.
- Сортування лише за потреби, тобто виконання *ORDER BY* є ресурсномісткою операцією і її варто застосовувати лише тоді, коли впорядковане подання даних справді необхідне.
- Використання зрозумілих імен та коментарів. Для полегшення читання та супроводу запитів бажано застосовувати осмислені псевдоніми для таблиць і полів, а у складніших випадках — короткі коментарі.

Питання для самоперевірки

1. *Дайте визначення вкладеному підзапиту.*
2. *Опишіть прості вкладені підзапити.*
3. *Коли застосовується операція UNION?*
4. *Яка основна відмінність у виконанні простого підзапиту та корельованого підзапиту?*
5. *Які приклади практичних завдань доцільніше реалізувати за допомогою вкладених підзапитів, а не об'єднань (JOIN)?*
6. *У чому полягає принцип обчислення вкладених підзапитів «зсередини назовні»?*
7. *Які можливі результати може повертати простий підзапит (скалярне значення, стовпець, таблиця)?*
8. *Як саме працює предикат EXISTS у корельованому підзапиті?*
9. *Чому надмірне використання багаторівневих підзапитів може знижувати продуктивність системи?*
10. *У яких випадках доцільно застосовувати оператор INTERSECT?*
11. *Яке призначення має оператор EXCEPT і який приклад його використання можна навести?*
12. *Які переваги надає NATURAL JOIN у порівнянні з явним INNER JOIN?*
13. *Чому використання команди SELECT * вважається небажаним при написанні SQL-запитів?*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 6

на тему: «Вкладені та багатотабличні запити в SQL»

МЕТА: *ознайомитися з основами побудови вкладених (простих, корельованих, багаторівневих) підзапитів у мові SQL, навчитися застосовувати операції об'єднання результатів запитів (UNION, INTERSECT, EXCEPT) та природне об'єднання таблиць (NATURAL JOIN). Закріпити навички створення складних вибірок даних із використанням підзапитів і багатоетапних умов відбору.*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. Створення простих вкладених підзапитів із використанням предикатів *IN, ANY, ALL.*
2. Виконання корельованих вкладених підзапитів із застосуванням предикатів *EXISTS і NOT EXISTS.*
3. Розроблення багаторівневих підзапитів для багатокрокового відбору даних.
4. Виконання операцій об'єднання результатів запитів за допомогою *UNION, UNION ALL, INTERSECT, EXCEPT.*
5. Використання об'єднання таблиць для отримання узагальнених результатів.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Для виконання лабораторної роботи було використано базу даних *DB_Book*, яка містить такі основні таблиці:

- довідник авторів книг *Bookstore.Authors (IdАвтора, Прізвище, Ім'я, IdКраїни)*;
- перелік країн *Bookstore Countries (IdКраїни, НайменуванняКраїни)*;
- класифікатор жанрів літератури *Bookstore.Genres (IdЖанру, НазваЖанру, ТипЖанру)*;
- відомості про видавництва *Bookstore.Publishers* (*IdВидавництва, НазваВидавництва, АдресаВидавництва)*;

- інформація про книги та їхню вартість *Bookstore.PriceList (IdКниги, IdАвтора, НазваКниги, РікВидання, IdЖанру, IdВидавництва, Ціна)*;
- відомості про клієнтів *Bookstore.Clients (IdКлієнта, НайменуванняКлієнта, АдресаКлієнта, Телефон, МФОБанку, РР, ЄДРПОУ)*;
- дані про замовлення *Bookstore.Sales (IdЧеку, IdКлієнта, ДатаЗамовлення, ДатаСплати)*;
- деталізацію проданих книг у кожному замовленні *Bookstore.SalesItems* (*IdЧеку*, *IdКниги*, *Кількість).*

У процесі роботи з реляційними базами даних виникає потреба виконувати складні аналітичні запити, які охоплюють кілька таблиць, потребують порівняння даних між ними або поетапного відбору результатів. Одним із найважливіших механізмів, що забезпечує такі можливості, є вкладені та багатотабличні запити в мові *SQL*.

Вкладені підзапити (subqueries) дозволяють вбудовувати один запит у інший, формуючи багаторівневу логіку обробки даних. Вони використовуються для уточнення умов вибірки, перевірки наявності записів, обчислення проміжних результатів або створення тимчасових вибірок. Розрізняють прості (некорельовані) підзапити, які виконуються один раз незалежно від зовнішнього запиту, та корельовані, що залежать від поточного рядка зовнішнього запиту і виконуються багаторазово. Окрему категорію становлять багаторівневі підзапити, які поєднують кілька рівнів вкладення та застосовуються для реалізації складних умов відбору.

Крім вкладених запитів, важливе місце в *SQL* посідають операції об’єднання результатів (*UNION, INTERSECT, EXCEPT*), що дозволяють формувати узагальнені множини даних з кількох вибірок, а також природне об’єднання таблиць (*NATURAL JOIN*), яке забезпечує автоматичне поєднання записів за спільними атрибутами. Дані інструменти значно розширюють можливості користувача у побудові аналітичних і звітних запитів.

Створення простих вкладених підзапитів

Приклад 1. Визначити авторів, які мають книги дорожчі за середню ціну всіх книг.

```
SELECT Прізвище, Імя
FROM Bookstore.Authors
WHERE IdАвтора IN (
    SELECT IdАвтора
```

```
    FROM Bookstore.PriceList
    WHERE Ціна > (SELECT AVG(Ціна)
     FROM Bookstore.PriceList)
       );
```

	Прізвище	Ім'я
1	Rowling	Joanne
2	Hemingway	Ernest
3	Orwell	George
4	Hugo	Victor

У цьому запиті реалізовано багаторівневий вкладений підзапит, який дозволяє виконати послідовну аналітичну обробку даних. Внутрішній підзапит другого рівня обчислює середнє значення ціни книг у всій таблиці *PriceList* за допомогою агрегатної функції *AVG()*. Результат цього обчислення використовується підзапитом першого рівня, який формує перелік авторів, чиї книги мають ціну вищу за середню. Зовнішній запит вибирає прізвища та імена відповідних авторів із таблиці *Authors*.

Приклад 2. Визначити книги, які видані у видавництвах, розташованих у місті Львів.

```
SELECT НазваКниги
FROM Bookstore.PriceList
WHERE IdВидавництва IN (
    SELECT IdВидавництва
    FROM Bookstore.Publishers
    WHERE АдресаВидавництва LIKE N'%Львів%'
  );
```

	НазваКниги
1	Кобзар
2	Місто

Даний приклад ілюструє використання вкладеного підзапиту для логічного зв'язку між таблицями, без явного застосування оператора *JOIN*. Внутрішній підзапит вибирає ідентифікатори видавництв (*IdВидавництва*), у полі адреси яких міститься підрядок «Львів». Оператор *LIKE* використано для реалізації пошуку за частковим збігом рядка, що дозволяє врахувати всі адреси, які містять задану назву міста.

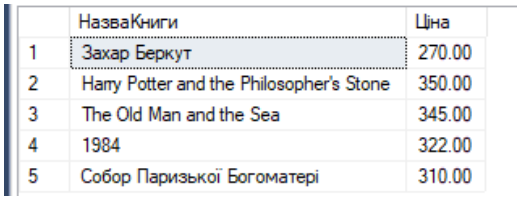
Зовнішній запит здійснює вибірку назв книг із таблиці *PriceList*, у яких поле *IdВидавництва* збігається з будь-яким із результатів внутрішнього запиту. Таким чином, формується перелік усіх книг, виданих у львівських видавництвах.

Приклад 3. Визначити книги, ціна яких перевищує будь-яку з цін книжок жанру «Поезія».

```
SELECT НазваКниги, Ціна
FROM Bookstore.PriceList
WHERE Ціна > ANY (
    SELECT Ціна
```

```
    FROM Bookstore.PriceList
    WHERE IdЖанру = (
        SELECT IdЖанру FROM Bookstore.Genres
         WHERE НазваЖанру = N'Поезія' )
);
```

Запит реалізує багаторівневу структуру вкладених підзапитів. Найвнутрішній підзапит отримує ідентифікатор жанру, для якого назва дорівнює «Поезія». На другому рівні формується множина цін усіх поетичних книг, а зовнішній запит виконує порівняння ціни кожної книги з цінами цієї множини за допомогою предиката *ANY*.



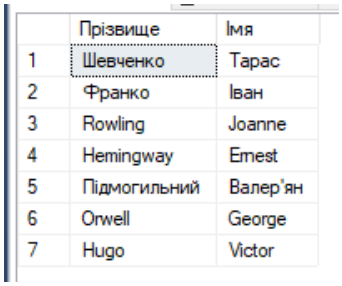
	НазваКниги	Ціна
1	Захар Беркут	270.00
2	Harry Potter and the Philosopher's Stone	350.00
3	The Old Man and the Sea	345.00
4	1984	322.00
5	Собор Паризької Богоматері	310.00

Предикат *ANY* (або синонім *SOME*) повертає істину, якщо умова виконується хоча б для одного елемента підмножини. Отже, у результат вибірки потрапляють книги, вартість яких перевищує принаймні одну з цін у жанрі «Поезія».

Виконання корельованих вкладених підзапитів

Приклад 4. Визначити авторів, чиї книги були продані хоча б один раз.

```
SELECT a.Прізвище, a.Імя
FROM Bookstore.Authors a
WHERE EXISTS (
    SELECT 1
    FROM Bookstore.PriceList p
    JOIN Bookstore.SalesItems si
    ON p.IdКниги = si.IdКниги
    WHERE p.IdАвтора = a.IdАвтора
);
```



	Прізвище	Імя
1	Шевченко	Тарас
2	Франко	Іван
3	Rowling	Joanne
4	Hemingway	Ernest
5	Підмогильний	Валер'ян
6	Orwell	George
7	Hugo	Victor

У цьому прикладі використано корельований вкладений підзапит, який пов’язаний із зовнішнім запитом через поле *IdАвтора*. Підзапит виконує об’єднання таблиць *PriceList* та *SalesItems* для перевірки наявності хоча б одного факту продажу книги, що належить певному авторові.

Предикат *EXISTS* повертає істинне значення, якщо підзапит знаходить хоча б один запис, який задовольняє умову з’єднання. Таким чином, до результатної множини включаються лише ті автори, книги яких були реалізовані принаймні один раз.

Такий підхід є типовим для аналізу комерційної активності авторів, коли необхідно визначити, які з них мають фактичні продажі. Використання *EXISTS* є оптимальним, оскільки перевірка припиняється одразу після знаходження першого відповідного запису, що підвищує ефективність виконання запиту.

Приклад 5. Визначити клієнтів, які здійснили хоча б одне замовлення у 2025 році.

```
SELECT НайменуванняКлієнта
FROM Bookstore.Clients c
WHERE EXISTS (
    SELECT 1
    FROM Bookstore.Sales s
    WHERE s.IdКлієнта = c.IdКлієнта
      AND YEAR(s.ДатаЗамовлення) = 2025
);
```

	НайменуванняКлієнта
1	ТОВ Книгарня Є
2	ТОВ Літера
3	ТОВ Книголенд
4	ФОП Сидоренко Н.П.

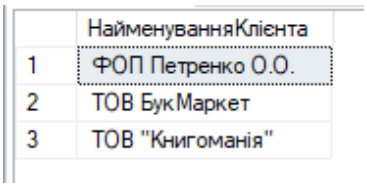
У цьому запиті використовується корельований підзапит, тобто такий, що пов'язаний із зовнішнім запитом через загальне поле *IdКлієнта*. Внутрішній підзапит перевіряє наявність принаймні одного запису в таблиці *Sales*, де клієнт із зовнішнього запиту (*c.IdКлієнта*) здійснив замовлення у 2025 році.

Предикат *EXISTS* повертає істинне значення *TRUE*, якщо хоча б один відповідний запис знайдено. Отже, основний запит відбирає тільки тих клієнтів, які фактично зробили покупки у зазначеному році.

Застосування *EXISTS* у таких випадках є більш ефективним, ніж пошук через *IN*, оскільки система припиняє перевірку одразу після знаходження першого збігу. Такий підхід є типовим для аналітичних звітів, орієнтованих на активність клієнтів у певний період часу.

Приклад 6. Визначити клієнтів, які не здійснили жодного замовлення у 2025 році.

```
SELECT НайменуванняКлієнта
FROM Bookstore.Clients c
WHERE NOT EXISTS (
    SELECT 1
    FROM Bookstore.Sales s
    WHERE s.IdКлієнта = c.IdКлієнта
      AND YEAR(s.ДатаЗамовлення) = 2025
);
```



	НайменуванняКлієнта
1	ФОП Петренко О.О.
2	ТОВ БукМаркет
3	ТОВ "Книгоманія"

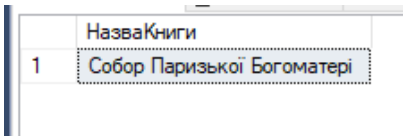
Цей приклад є логічною протилежністю попереднього. Використано предикат *NOT EXISTS*, який повертає істинне значення лише тоді, коли підзапит не знаходить жодного відповідного запису. Внутрішній підзапит здійснює пошук замовлень клієнта за 2025 рік, а якщо такі замовлення відсутні, клієнт включається до результатної множини.

Такий запит дозволяє виявити неактивних клієнтів, тобто тих, хто не здійснював покупок у визначеному часовому інтервалі. Подібний аналітичний підхід застосовується в CRM-системах та аналітиці продажів для подальшої розробки маркетингових стратегій, спрямованих на повторну активацію клієнтської бази.

Багаторівневі підзапити для багатокрокового відбору даних

Приклад 7. Визначити назви книг, автори яких походять з Франції.

```
SELECT НазваКниги
FROM Bookstore.PriceList
WHERE IdАвтора IN (
    SELECT IdАвтора
    FROM Bookstore.Authors
    WHERE IdКраїни = (
        SELECT IdКраїни
        FROM Bookstore.Countries
        WHERE НайменуванняКраїни = N'Франція' )
);
```



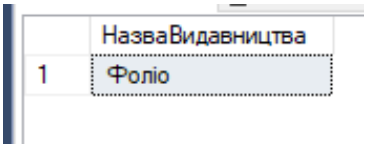
	НазваКниги
1	Собор Паризької Богоматері

Цей багаторівневий підзапит послідовно звертається до трьох таблиць. Спочатку внутрішній підзапит визначає ідентифікатор країни «Франція». Потім середній рівень запиту обирає авторів, що належать до цієї країни, а зовнішній запит формує перелік назв книг, написаних зазначеними авторами. Такий підхід дозволяє поетапно уточнювати умови відбору, зберігаючи логічну структурованість запиту.

Приклад 8. Визначити видавництва, у яких представлені книги щонайменше двох різних авторів.

```
SELECT НазваВидавництва
FROM Bookstore.Publishers p
```

```
WHERE p.IdВидавництва IN (
    SELECT IdВидавництва
    FROM Bookstore.PriceList
    GROUP BY IdВидавництва
    HAVING COUNT(DISTINCT IdАвтора) >= 2
);
```

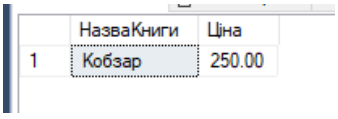


	НазваВидавництва
1	Фоліо

У цьому прикладі реалізовано два вкладених підзапити, які обмежують множину записів за двома різними умовами: походження автора та місце розташування видавництва. Перший підзапит визначає авторів із України, а другий – видавництва, розташовані у місті Києві. Основний запит повертає тільки ті книги, що одночасно задовольняють обидва критерії, тобто належать українським авторам і видані у Києві.

Приклад 9. Визначити найдорожчу книгу серед тих, що видані львівськими видавництвами.

```
SELECT НазваКниги, Ціна
FROM Bookstore.PriceList
WHERE Ціна = (
    SELECT MAX(Ціна)
    FROM Bookstore.PriceList
    WHERE IdВидавництва IN (
        SELECT IdВидавництва
        FROM Bookstore.Publishers
        WHERE АдресаВидавництва LIKE N'%Львів%' )
);
```



	НазваКниги	Ціна
1	Кобзар	250.00

Вкладений підзапит формує підмножину книг, виданих львівськими видавництвами, після чого визначає серед них максимальне значення ціни. Зовнішній запит відображає назву книги, якій відповідає ця максимальна вартість. Такий підхід є типовим для задач порівняльного аналізу даних, коли необхідно знайти екстремальні значення в обмеженій множині.

Виконання операцій об’єднання результатів запитів

Приклад 10. Об’єднати списки українських і британських авторів.

```
SELECT Прізвище, Імя FROM Bookstore.Authors
WHERE IdКраїни =
  (SELECT IdКраїни FROM Bookstore.Countries
```

```
  WHERE НайменуванняКраїни = N'Україна')
UNION
SELECT Прізвище, Ім'я FROM Bookstore.Authors
WHERE IdКраїни =
  (SELECT IdКраїни FROM Bookstore.Countries
   WHERE НайменуванняКраїни = N'Велика Британія');
```

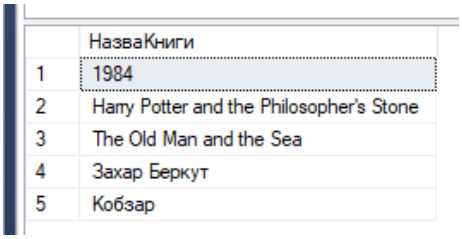
Операція *UNION* виконує об'єднання результатів двох запитів, видаляючи дублікати. У наведеному прикладі формується єдиний список авторів із двох різних країн України та Великої Британії. Така конструкція дозволяє ефективно узагальнювати інформацію з різних груп даних, забезпечуючи зручність подальшого аналізу.

	Прізвище	Ім'я
1	Orwell	George
2	Rowling	Joanne
3	Підмогильний	Валер'ян
4	Франко	Іван
5	Шевченко	Тарас

Приклад 11. Визначити книги, які продавались у 2024 році, але не продавались у 2025 році.

```
SELECT НазваКниги FROM Bookstore.PriceList
WHERE IdКниги IN (
    SELECT IdКниги FROM Bookstore.SalesItems si
    JOIN Bookstore.Sales s ON si.IdЧеку = s.IdЧеку
    WHERE YEAR(ДатаЗамовлення) = 2024
)
EXCEPT
SELECT НазваКниги FROM Bookstore.PriceList
WHERE IdКниги IN (
    SELECT IdКниги FROM Bookstore.SalesItems si
    JOIN Bookstore.Sales s ON si.IdЧеку = s.IdЧеку
    WHERE YEAR(ДатаЗамовлення) = 2025
);
```

Оператор *EXCEPT* повертає записи, що містяться у першому запиті, але відсутні у другому. Таким чином, у результаті будуть відображені назви книг, які реалізовувалися лише у 2024 році та не були зафіксовані у продажах наступного року, що корисно для аналізу змін у попиті на книжкову продукцію.

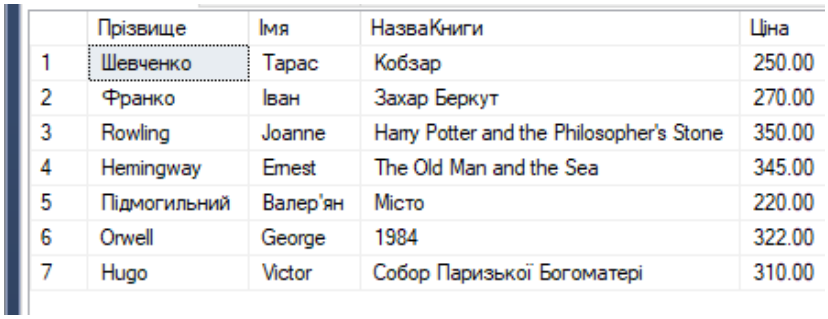


	НазваКниги
1	1984
2	Harry Potter and the Philosopher's Stone
3	The Old Man and the Sea
4	Захар Беркут
5	Кобзар

Використання об'єднання таблиць

Приклад 12. Отримати інформацію про книги разом із прізвищами авторів.

```
SELECT a.Прізвище, a.Імя, p.НазваКниги, p.Ціна
FROM Bookstore.Authors a
JOIN Bookstore.PriceList p
ON a.IdАвтора = p.IdАвтора;
```



	Прізвище	Імя	НазваКниги	Ціна
1	Шевченко	Тарас	Кобзар	250.00
2	Франко	Іван	Захар Беркут	270.00
3	Rowling	Joanne	Harry Potter and the Philosopher's Stone	350.00
4	Hemingway	Ernest	The Old Man and the Sea	345.00
5	Підмогильний	Валер'ян	Місто	220.00
6	Orwell	George	1984	322.00
7	Hugo	Victor	Собор Паризької Богоматері	310.00

У цьому прикладі здійснено явне внутрішнє об'єднання *JOIN* таблиць *Authors* та *PriceList* за спільним атрибутом *IdАвтора*. Запит формує узагальнену таблицю, що містить інформацію про авторів разом із назвами та цінами їхніх книг. Використання *JOIN ... ON* забезпечує чітке визначення умови з'єднання, зменшуючи ймовірність помилок та підвищуючи прозорість запиту.

Приклад 13. Визначити прізвища та ініціали авторів, назви їхніх книг, видавництва та жанри цих книг, які були продані протягом останнього кварталу 2024 року (з 1 жовтня по 31 грудня 2024 року). У результаті має бути представлено узагальнену інформацію про літераторів, їхні твори, відповідні видавництва та жанрову класифікацію проданих книг у визначений часовий інтервал.

```
SELECT     a.Прізвище + N' ' + LEFT(a.Імя, 1) + N'.' AS Автор,
    pL.НазваКниги, pub.НазваВидавництва, g.НазваЖанру
FROM Bookstore.Authors a
JOIN Bookstore.PriceList pL
ON a.IdАвтора = pL.IdАвтора
JOIN Bookstore.Genres g
ON pL.IdЖанру = g.IdЖанру
JOIN Bookstore.Publishers pub
ON pL.IdВидавництва = pub.IdВидавництва
JOIN Bookstore.SalesItems sI
ON pL.IdКниги = sI.IdКниги
JOIN Bookstore.Sales s
ON sI.IdЧеку = s.IdЧеку
WHERE s.ДатаЗамовлення BETWEEN '2024-10-01' AND '2024-12-31'
ORDER BY a.Прізвище, pL.НазваКниги;
```

	Автор	НазваКниги	НазваВидавництва	НазваЖанру
1	Hemingway E.	The Old Man and the Sea	Фоліо	Роман
2	Hugo V.	Собор Паризької Богоматері	Клуб Сімейного Дозвілля	Роман
3	Orwell G.	1984	Фоліо	Фантастика
4	Rowling J.	Harry Potter and the Philosopher's Stone	А-Ба-Ба-Га-Ла-Ма-Га	Фентезі
5	Підмогильний В.	Місто	Кальварія	Роман
6	Франко І.	Захар Беркут	Ранок	Роман

У наведеному запиті реалізовано багатотабличне об'єднання для отримання комплексної інформації з кількох сутностей бази даних. Таблиця *Authors* з'єднується з *PriceList*, що дозволяє встановити відповідність між авторами та їхніми книгами. Таблиці *Genres* і *Publishers* додають відомості про жанрову приналежність і видавництво кожного твору. Додаткове приєднання таблиць *SalesItems* і *Sales* забезпечує можливість відбору лише тих книг, які були реалізовані протягом останнього кварталу 2024 року. Для цього використовується умова

WHERE s.ДатаСплати BETWEEN '2024-10-01' AND '2024-12-31',

що обмежує вибірку конкретним часовим інтервалом. Отже, запит повертає повну інформацію про авторів, їхні книги, відповідні видавництва та жанри, які фігурували у продажах у кінці 2024 року.

Приклад 14. Визначити клієнтів, які придбали книги українських авторів упродовж 2024 року. Для кожного клієнта вивести його найменування, місто (із адреси), прізвище та ініціали автора, назву книги, видавництво, жанр, кількість проданих примірників і сумарну вартість купівлі. У результат включати лише ті продажі, де загальна сума придбання перевищує 3000 грн.

```
SELECT c.НайменуванняКлієнта, SUBSTRING(c.АдресаКлієнта,
    CHARINDEX(N'м.', c.АдресаКлієнта), LEN(c.АдресаКлієнта)) AS Місто,
    a.Прізвище + N' ' + LEFT(a.Імя, 1) + N'.' AS Автор,
    pL.НазваКниги, pub.НазваВидавництва, g.НазваЖанру, sI.Кількість,
    (sI.Кількість * pL.Ціна) AS СумаПродажу
FROM Bookstore.Sales s
JOIN Bookstore.SalesItems sI
ON s.IdЧеку = sI.IdЧеку
JOIN Bookstore.PriceList pL
ON sI.IdКниги = pL.IdКниги
JOIN Bookstore.Authors a
ON pL.IdАвтора = a.IdАвтора
JOIN Bookstore.Countries ctr
ON a.IdКраїни = ctr.IdКраїни
JOIN Bookstore.Publishers pub
ON pL.IdВидавництва = pub.IdВидавництва
JOIN Bookstore.Clients c
```

```
ON s.IdКлієнта = c.IdКлієнта
JOIN Bookstore.Genres g
ON pL.IdЖанру = g.IdЖанру
WHERE ctr.НайменуванняКраїни = N'Україна'
  AND YEAR(s.ДатаСплати) = 2024
  AND (sI.Кількість * pL.Ціна) > 3000
ORDER BY c.НайменуванняКлієнта, СумаПродажу DESC;
```

	НайменуванняКлієнта	Місто	Автор	НазваКниги	НазваВидавництва	НазваЖанру	Кількість	СумаПродажу
1	ТОВ БукМаркет	м. Дніпро, просп. Яворницького, 10	Франко І.	Захар Беркут	Ранок	Роман	15	4050.00
2	ТОВ Книгарня Є	м. Київ, вул. Хрещатик, 22	Франко І.	Захар Беркут	Ранок	Роман	25	6750.00
3	ТОВ Книгарня Є	м. Київ, вул. Хрещатик, 22	Шевченко Т.	Кобзар	Видавництво Старий Лев	Поезія	20	5000.00
4	ТОВ Книголенд	м. Харків, вул. Сумська, 19	Франко І.	Захар Беркут	Ранок	Роман	15	4050.00
5	ТОВ Книголенд	м. Харків, вул. Сумська, 19	Підмогильний В.	Місто	Кальварія	Роман	15	3300.00
6	ФОП Петренко О.О.	м. Одеса, вул. Дерибасівська, 5	Підмогильний В.	Місто	Кальварія	Роман	25	5500.00
7	ФОП Сидоренко Н.П.	м. Чернігів, вул. Шевченка, 7	Підмогильний В.	Місто	Кальварія	Роман	15	3300.00

У даному запиті здійснюється комплексне об’єднання кількох логічно пов’язаних таблиць для аналітичного дослідження продажів за 2024 рік. Таблиця *Sales* надає інформацію про операції продажу та клієнтів, які здійснили покупки. Таблиця *SalesItems* уточнює, які саме книги входили до кожного чеку, а також їх кількість. Через таблицю *PriceList* встановлюється зв’язок із авторами (*Authors*), жанрами (*Genres*) та видавництвами (*Publishers*). Додатково приєднано таблицю *Countries*, щоб відфільтрувати лише українських авторів.

Використана умова *YEAR(s.ДатаСплати) = 2024* обмежує вибірку календарним роком, а вираз *(sI.Кількість * pL.Ціна) > 3000* забезпечує відбір лише тих покупок, де сумарна вартість перевищує 3000 грн. Для наочності у результаті виводиться місто клієнта, що визначається з текстового поля адреси за допомогою функції *SUBSTRING*(). Отже, запит дозволяє проаналізувати найбільш вагомі операції купівлі українських книг у 2024 році, відобразивши ключові характеристики клієнта, автора, видавництво, жанр і фінансовий показник продажу.

ТЕМА 8. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ

ПЛАН:

8.1. Призначення та переваги збережених процедур.
8.2. Створення та виконання збережених процедур.
8.3. Використання параметрів у збережених процедурах.
8.4. Повернення результатів: вихідні параметри та оператор RETURN.
8.5. Управління збереженими процедурами.
8.6. Оптимізація та продуктивність збережених процедур.
8.7. Безпека і керування доступом до процедур.

8.1. Призначення та переваги збережених процедур

Збережена процедура (Stored Procedures) іменований програмний модуль, що містить блок інструкцій Transact-SQL, збережений у базі даних і виконуваний на сервері, яка може викликатися багаторазово з різних прикладних програм або інших об’єктів СКБД.

Основна ідея полягає у винесенні бізнес-логіки та операцій з даними ближче до місця їх зберігання, що забезпечує ефективність та узгодженість роботи всієї системи. Одним із головних призначень збережених процедур є інкапсуляція бізнес-логіки. Усі складні послідовності команд, що реалізують певну операцію (наприклад, оформлення замовлення, розрахунок вартості, формування звіту), можна помістити в одну процедуру. Це дає змогу відокремити логіку обробки від прикладного коду, зробивши її більш незалежною та гнучкою для супроводу.

Інша суттєва перевага полягає у підвищенні продуктивності. Під час першого виконання збереженої процедури *SQL Server* створює та кешує план виконання, який повторно використовується при наступних викликах. Це скорочує витрати на компіляцію запитів і значно пришвидшує обробку великих масивів даних, особливо у випадках багаторазового виконання однотипних операцій.

Цікавий факт:

Збережені процедури також підвищують безпеку доступу до даних. Користувачам можна надати права на виконання конкретної процедури без надання прямого доступу до таблиць чи подань. Таким чином, адміністратор може суворо контролювати, які саме дії дозволено виконувати користувачам, і запобігати несанкціонованим операціям.

Важливим є й аспект зменшення трафіку між клієнтським застосунком та сервером. Замість надсилання великої кількості окремих інструкцій клієнт може викликати одну процедуру, яка виконує всі необхідні дії на сервері. Це особливо актуально в розподілених системах та при роботі з віддаленими базами даних.

Ще однією перевагою є спрощення адміністрування та супроводу. Коли бізнес-логіка зосереджена в збережених процедурах, будь-які зміни до правил чи алгоритмів виконуються централізовано шляхом модифікації однієї процедури на сервері, без необхідності оновлювати клієнтські програми. Це значно скорочує витрати на підтримку системи.

З практичної точки зору, збережені процедури сприяють також повторному використанню коду. Один і той самий модуль можна викликати у різних сценаріях, що запобігає дублюванню логіки та зменшує кількість помилок, що в свою чергу, робить розробку більш стандартизованою й підвищує якість програмного забезпечення.

Таким чином, збережені процедури поєднують у собі одразу кілька ключових переваг:

- інкапсулюють бізнес-логіку;
- підвищують продуктивність;
- забезпечують безпеку;
- зменшують мережевий трафік;
- спрощують процеси адміністрування;
- сприяють повторному використанню коду.

Саме завдяки цим властивостям вони стали стандартним інструментом у сучасних реляційних СКБД.

8.2. Створення та виконання збережених процедур

Нерідко операція з даними представляє набір інструкцій, які необхідно виконати в певній послідовності. Наприклад, при додаванні інформацію про реалізацію книг необхідно внести дані в таблицю замовлень. Однак перед цим треба перевірити, а чи є в наявності книга. Можливо, при цьому знадобиться перевірити ще ряд додаткових умов. Тобто фактично процес продажу книг охоплює кілька дій, які повинні виконуватися в певній послідовності. І в цьому випадку оптимально буде інкапсулювати всі ці дії в один об'єкт – збережену процедуру.

Ідентифікатори спеціальні символи, які використовуються зі змінними для ідентифікування їх типу або для угруповання слів в змінну.

Типи ідентифікаторів:

@	ідентифікатор локальної змінної (користувальницької).
@@	ідентифікатор глобальної змінної (вбудованої).
#	ідентифікатор локальної таблиці або процедури.
##	ідентифікатор глобальної таблиці або процедури.
[]	ідентифікатор угруповання слів в змінну (працюють як стандартні "").

Для створення нової процедури використовується оператор **CREATE PROCEDURE**, після якого зазначається унікальне ім’я процедури, список параметрів (за потреби) та тіло процедури. Для її створення використовується команда *CREATE PROCEDURE*, для модифікації – *ALTER PROCEDURE*, для видалення – *DROP PROCEDURE*, а для виконання – *EXEC* або *EXECUTE*.

При створенні процедури визначаються:

- ім’я процедури;
- параметри (вхідні, вихідні або змішані);
- тіло процедури, яке містить *SQL*-інструкції.

Синтаксис процедури:

```
CREATE PROCEDURE [schema_name.]procedure_name
   [ @parameter_name data_type [= default] [OUTPUT] [READONLY] ]
   [ ,...n ]
```

```
AS
BEGIN
   -- T-SQL інструкції, що становлять тіло процедури
END;
```

Основні ключові слова і параметри процедури:

[schema_name.]procedure_name – повне ім'я процедури. Ім'я схеми є необов'язковим, однак у корпоративних середовищах його рекомендують завжди зазначати для уникнення неоднозначностей.

@parameter_name – параметр процедури, що починається зі знака @. Ім'я параметра має бути унікальним у межах процедури.

data_type – тип даних параметра, визначений за правилами *SQL Server* (наприклад, *INT, DECIMAL(10,2), NVARCHAR(50)*).

= *default* – значення за замовчуванням, яке буде використане, якщо користувач не передасть конкретне значення під час виклику.

OUTPUT – атрибут, який визначає параметр як вихідний. Такі параметри дозволяють повернути значення з процедури у змінну виклику. Якщо *OUTPUT* не зазначено, параметр вважається вхідним.

READONLY – атрибут, що використовується для параметрів типу таблиця (*Table-Valued Parameter*). Він обов'язковий і вказує, що дані у такому параметрі не можуть бути змінені всередині процедури.

AS – ключове слово, що відділяє сигнатуру процедури від її тіла.

BEGIN ... END – межі блоку, у якому розміщуються інструкції *T-SQL*. Всередині блоку можуть бути оператори вибірки (*SELECT*), модифікації даних (*INSERT, UPDATE, DELETE*), керуючі конструкції (*IF...ELSE, WHILE,* обробка винятків *TRY...CATCH*), виклики інших процедур чи функцій.

Для виконання збереженої процедури використовується оператор *EXEC* або його повна форма *EXECUTE*, після якого зазначається ім'я процедури та, за потреби, список значень для параметрів. Якщо процедура містить вихідні параметри, при виклику необхідно явно вказати ключове слово *OUTPUT*.

ПРИКЛАД 8.1. Створити збережену процедуру для додавання інформації про жанри в таблицю *«Довідник жанрів»*:

ЛИСТИНГ 8.1. Створення та виконання процедури

```
USE DB_Book;
GO

CREATE PROCEDURE Bookstore.AddGenre
   @IdЖанру INT,
```

```
    @НазваЖанру NVARCHAR(100),
    @ТипЖанру NVARCHAR(100) = N'Невизначено'
 AS
 BEGIN
    SET NOCOUNT ON;
    BEGIN TRY
       INSERT INTO Bookstore.Genres(IdЖанру, НазваЖанру,
ТипЖанру)
       VALUES (@IdЖанру, @НазваЖанру, @ТипЖанру);
    END TRY
    BEGIN CATCH
       -- повернемо інформацію про помилку
       DECLARE @ErrMsg NVARCHAR(4000) =
ERROR_MESSAGE();
       RAISERROR('Помилка при додаванні жанру: %s', 16, 1,
@ErrMsg);
       RETURN -1;
    END CATCH;
    RETURN 0;
 END;

  --Виконання процедури:
 EXEC Bookstore.AddGenre @IdЖанру = 101, @НазваЖанру =
N'Фантастика';
```

У цьому прикладі демонструється створення збереженої процедури, яка реалізує додавання нового жанру до таблиці *Bookstore.Genres.*

Алгоритм виконання процедури:

1. *Ініціалізація середовища*

Процедура створюється в базі даних *DB_Book*, у схемі *Bookstore*, оголошуються три параметри:

@IdЖанру – числовий ідентифікатор жанру;

@НазваЖанру – назва жанру (рядкове значення);

@ТипЖанру – тип жанру (рядкове значення). Для параметра даного типу встановлено значення за замовчуванням «Невизначено», що дозволяє викликати процедуру без явного зазначення цього аргументу

2. *Встановлення параметрів середовища виконання*

Виконується команда *SET NOCOUNT ON*, яка забороняє відображення службових повідомлень про кількість змінених рядків, з метою оптимізації роботи процедури.

3. *Основна спроба виконання (TRY)*

У тілі процедури використовується оператор *INSERT* для додавання нового запису у таблицю *Bookstore.*Genres зі значеннями, переданими через параметри процедури. Якщо вставка виконалась успішно, програма переходить до завершення процедури.

4. *Обробка виняткових ситуацій (CATCH)*

У разі виникнення помилки перехоплюється текст повідомлення за допомогою функції *ERROR_MESSAGE*(). За допомогою інструкції *RAISERROR* виводиться зрозуміле повідомлення користувачеві: «Помилка при додаванні жанру: …». Повертається код помилки -1 для сигналізації про невдале виконання.

5. *Завершення процедури*

Якщо помилок не виникло, процедура завершується оператором *RETURN 0*, що означає успішне виконання

6. *Виклик процедури*

Користувач викликає процедуру через *EXEC*, передаючи значення параметрів. У прикладі додається жанр із ідентифікатором 101 та назвою «Фантастика», при цьому параметр *@ТипЖанру* не передається, тому автоматично використовується значення за замовчуванням «Невизначено».

ПРИКЛАД 8.2. Створити збережену процедуру для отримання даних з таблиці «Довідник жанрів»:

ЛИСТИНГ 8.2. Створення та виконання процедури для отримання даних з таблиці «Довідник жанрів»

```
CREATE PROCEDURE GenreSummary
AS
BEGIN
   SET NOCOUNT ON;
   SELECT
      IdЖанру AS GenreID,
      НазваЖанру AS GenreName,
      ТипЖанру AS GenreType
   FROM Bookstore.Genres;
END;

--Виконання процедури
EXEC GenreSummary;
```

Алгоритм виконання процедури:

Спочатку виконується команда *CREATE PROCEDURE GenreSummary...,* яка визначає процедуру в базі даних *DB_Book*. На цьому етапі код процедури зберігається в системних об'єктах *SQL Server* і готовий до подальшого виконання, але сам по собі ще не запускається. Процедура існує лише як об'єкт бази даних, який можна викликати командою *EXEC.*

Коли користувач або програма виконує команду *EXEC GenreSummary;*, SQL Server розпочинає виконання процедури. Сервер входить у блок *BEGIN ... END* і виконує команду *SET NOCOUNT ON;*, яка вимикає повідомлення про кількість оброблених рядків. Це робить виконання більш оптимізованим і чистим з точки зору клієнта, оскільки SQL Server не надсилає зайві повідомлення про кількість рядків, що оброблені запитом.

Далі SQL Server обробляє SQL-запит *SELECT… FROM...* На цьому етапі сервер сканує таблицю *Bookstore.Genres*, отримує всі рядки та формує результуючу таблицю з трьома стовпцями. Для кожного рядка вибираються три стовпці: *IdЖанру, НазваЖанру, ТипЖанру*, для яких використовуються псевдоніми *AS* для більш зрозумілих назв у результаті *GenreID, GenreName, GenreType.*

Після формування таблиці результатів SQL Server повертає її користувачу (або додатку, який викликав процедуру, наприклад, Management Studio), тобто отримує набір усіх жанрів з відповідними колонками, що дозволяє переглядати дані у зручному вигляді.

Після завершення виконання *SELECT* процедура закінчує роботу, всі ресурси *SQL Server*, що використовувалися для її виконання, звільняються, і сервер готовий приймати наступні запити. Таким чином, процедура *GenreSummary* забезпечує ефективне отримання та представлення даних таблиці жанрів.

Таким чином, створення та виконання збережених процедур у *SQL Server* охоплює такі елементи: декларацію імені та параметрів, формування блоку логіки у межах *BEGIN...END*, компіляцію та збереження об'єкта у базі даних, а також подальший виклик за допомогою *EXEC/EXECUTE*. Збережені процедури забезпечують модульність, повторне використання коду та централізоване управління бізнес-логікою на рівні бази даних.

8.3. Використання параметрів у збережених процедурах

Один із ключових аспектів ефективного використання збережених процедур є робота з параметрами. Параметри дозволяють робити процедури

універсальними, гнучкими та адаптивними до різних умов виконання. Замість створення окремої процедури для кожного можливого запиту можна визначити одну процедуру з параметрами, значення яких задаються під час її виклику.

Параметри у збережених процедурах бувають трьох типів.

Вхідні параметри (INPUT)	параметри, значення яких передаються у процедуру під час її виклику та використовуються всередині тіла процедури для виконання операцій, таких як фільтрація даних, вставка записів або виконання обчислень.
Вихідні параметри (OUTPUT)	дозволяють процедурі повертати значення безпосередньо через параметр, що дає змогу передавати результати виконання або обчислені значення (наприклад, кількість оброблених рядків) без використання окремого *SELECT*-запиту.
Параметри зі значенням за замовчуванням	дозволяють викликати процедуру без явного зазначення певного параметра. У такому випадку використовується задане стандартне значення.

При створенні процедури параметри оголошуються в її сигнатурі після імені процедури. Кожен параметр має вказаний тип даних, який визначає допустимі значення. Також можна задавати значення за замовчуванням, що дозволяє викликати процедуру без явного передавання певного аргумента. Наприклад, якщо параметр *@ТипЖанру* у процедурі додавання жанру має значення за замовчуванням *Невизначено*, виклик процедури без цього параметра не викличе помилку і автоматично підставить стандартне значення.

При виконанні процедури користувач або додаток передає значення параметрів у команді *EXEC*. *SQL Server* підставляє ці значення у відповідні частини *SQL*-запиту всередині процедури. Це дозволяє отримувати різні результати при одному і тому ж наборі *SQL*-операторів. Наприклад, одна процедура може повертати дані про всі жанри, а при передачі параметра *@ІdЖанру* – тільки конкретний рядок із таблиці.

Для внутрішніх обчислень або тимчасового зберігання значень у тілі процедури можна використовувати локальні змінні, які оголошуються за допомогою команди *DECLARE*. Локальні змінні існують лише протягом виконання процедури і дозволяють, наприклад, зберігати проміжні результати, виконувати обчислення або формувати значення, які потім будуть повернені через *OUTPUT* параметри чи використані у *SELECT*. При цьому локальні

змінні не впливають на зовнішній стан бази даних і видимі лише всередині процедури.

Процедура може повертати результати кількома способами:

Перший спосіб являє собою табличний набір результатів (*Result Set*) через команду *SELECT*, який передається клієнту у вигляді стандартної таблиці рядків.

Другий спосіб через *OUTPUT* параметри, коли значення передаються безпосередньо у змінні викликаючого коду.

Третій – через код завершення процедури за допомогою команди *RETURN <int>;* традиційно значення 0 позначає успішне виконання, а будь-яке інше число використовується для передачі коду помилки або статусу виконання.

Використання параметрів також підвищує безпеку та продуктивність (всі ключові параметри подані у табомці 8.1.). Параметризовані процедури зменшують ризик *SQL*-ін'єкцій, оскільки значення параметрів передаються як дані, а не як частина динамічного *SQL*. Крім того, *SQL Server* може компілювати план виконання параметризованої процедури один раз і повторно використовувати його при різних викликах, що підвищує швидкодію.

Синтаксис параметризованої процедури:

```
CREATE PROCEDURE [schema_name.]procedure_name
   @Параметр1 <тип> [= значення_за_замовчуванням] [OUTPUT],
   @Параметр2 <тип> [= значення_за_замовчуванням] [OUTPUT]
 AS
 BEGIN
   -- SQL-оператори
 END;

 -- Виклик процедури з передачею параметрів
 DECLARE @Результат INT;
 EXEC <ім'я_процедури>
   @Параметр1 = <значення>,
   @Параметр2 = <значення> OUTPUT;
```

ПРИКЛАД 8.3. Створити процедуру, яка дозволяє отримувати дані про жанри книг. Процедура повинна підтримувати параметри:

@IdЖанру – для вибірки конкретного жанру (вхідний параметр),

@ТипЖанру – з можливістю вказати значення за замовчуванням (необов'язковий параметр),
@Кількість – вихідний параметр, який повертає кількість знайдених записів:

ЛИСТИНГ 8.3. Створення та виконання процедури для отримання даних про жанри книг

```
-- Створення процедури з вхідним, вихідним та параметром за замовчуванням
CREATE PROCEDURE Bookstore.GetGenres
    @IdЖанру INT,
     -- параметр за замовчуванням
    @ТипЖанру NVARCHAR(100) = N'Невизначено',
-- вихідний параметр
    @Кількість INT OUTPUT
AS
BEGIN
    SELECT *
    FROM Bookstore.Genres
    WHERE (@IdЖанру IS NULL OR IdЖанру = @IdЖанру)
     AND (@ТипЖанру IS NULL OR ТипЖанру = @ТипЖанру);

    -- Підрахунок кількості рядків, що задовольняють умову
    SELECT @Кількість = COUNT(*)
    FROM Bookstore.Genres
    WHERE (@IdЖанру IS NULL OR IdЖанру = @IdЖанру)
      AND (@ТипЖанру IS NULL OR ТипЖанру = @ТипЖанру);
END;

-- Виклик процедури
DECLARE @Рядків INT;
EXEC Bookstore.GetGenres
    @IdЖанру = 1,
    @Кількість = @Рядків OUTPUT;

-- Перегляд значення вихідного параметра
SELECT @Рядків AS КількістьЗнайденихЖанрів;
```

Пояснення:

- параметри *@IdЖанру* та *@ТипЖанру* дозволяють фільтрувати дані;

- вихідний параметр *@Кількість* повертає додаткову інформацію про обсяг вибірки;
- значення за замовчуванням дозволяє викликати процедуру без обов'язкового передавання всіх аргументів.

Параметризовані процедури підвищують продуктивність і безпеку, зменшуючи ризик SQL-ін'єкцій.

Таким чином, параметри у збережених процедурах є невід'ємним інструментом для створення універсальних, безпечних і ефективних процедур, здатних обробляти широкий спектр задач без дублювання коду. Вміння грамотно оголошувати та використовувати параметри є ключовою навичкою при розробці баз даних та програм, що працюють із ними.

8.4. Повернення результатів: вихідні параметри та оператор RETURN

У збережених процедурах *SQL Server* повернення результатів може здійснюватися кількома способами, і серед найбільш важливих є використання вихідних параметрів *OUTPUT* та оператора *RETURN*. Ці механізми дозволяють не лише отримати набір даних за допомогою *SELECT*, але й гнучко контролювати логіку завершення процедури та передавати конкретні значення назад до викликаного середовища.

Вихідні параметри *OUTPUT* застосовуються для повернення значень безпосередньо через параметри, оголошені у сигнатурі процедури. На відміну від звичайних вхідних параметрів, вони можуть змінюватися в тілі процедури й передавати результат на зовнішній рівень. Це особливо корисно у випадках, коли потрібно повернути одне або кілька обчислених значень, наприклад, кількість оброблених рядків, ідентифікатор новоствореного запису або статус виконання окремої частини алгоритму. Для роботи з такими параметрами їх потрібно оголошувати з ключовим словом *OUTPUT* і відповідним типом даних, а у виклику процедури явно вказувати, що параметр є вихідним.

Синтаксис параметра OUTPUT:

```
CREATE PROCEDURE [schema_name.]procedure_name
   @Параметр_Вхідний <тип>,
   @Параметр_Вихідний <тип> OUTPUT
AS
BEGIN
   -- тіло процедури
   SET @Параметр_Вихідний = <значення>
END;
```

```
-- Виклик:
DECLARE @Змінна <тип>;
EXEC <ім'я_процедури> @Вхідний = <значення>, @Параметр_Вихідний = @Змінна OUTPUT;
```

ПРИКЛАД 8.4. Створити збережену процедуру, яка повертає назву жанру за його *IdЖанру* через вихідний параметр:

ЛИСТИНГ 8.4. Використання OUTPUT для таблиці Genres

```
USE DB_Book
GO

  -- Оголошення процедури:
  --  @IdЖанру — вхідний параметр через який передається ідентифікатор жанру.
  -- @НазваЖанру OUTPUT — вихідний параметр, через який процедура поверне назву жанру.

CREATE PROCEDURE Bookstore.GetGenreName
   @IdЖанру INT,
   @НазваЖанру NVARCHAR(100) OUTPUT


AS
BEGIN
   SET NOCOUNT ON;

   -- Основний код: у змінну @НазваЖанру записується значення
-- з таблиці Genres.
   --  Використано TOP(1), щоб уникнути випадку, якщо є дублікати.

   SELECT TOP(1) @НазваЖанру = НазваЖанру
   FROM Bookstore.Genres
   WHERE IdЖанру = @IdЖанру;

   -- Обробка ситуації "жанр не знайдено"
  -- Якщо за вказаним IdЖанру запису відсутній, процедура поверне рядок "Невідомий жанр".
```

```
    IF @НазваЖанру IS NULL
        SET @НазваЖанру = N'Невідомий жанр';
END;

    -- Вмклик процедури: створюється змінна @GenreName.
    -- Виконується процедура, і значення в неї повертається через OUTPUT.
    -- Команда PRINT виводить результат.

DECLARE @GenreName NVARCHAR(100);
EXEC Bookstore.GetGenreName @IdЖанру = 101, @НазваЖанру = @GenreName OUTPUT;
PRINT N'Назва жанру: ' + @GenreName;
```

У результат виконання, якщо в таблиці є жанр з *IdЖанру = 101*, то повернеться його назва, у іншому випадку (якщо жанру не знайдено) результат буде:

```
Назва жанру: Невідомий жанр
```

Команда *SET NOCOUNT ON* використовується у збережених процедурах для керування тим, чи буде *SQL Server* повертати клієнту службові повідомлення про кількість оброблених рядків. За замовчуванням після виконання будь-якого запиту типу *INSERT*, *UPDATE*, *DELETE* або *SELECT* сервер надсилає повідомлення на кшталт (1 row(s) affected). Такі повідомлення не несуть корисної інформації для більшості випадків, але вони все одно передаються клієнту разом із результатами.

У процедурі це може створювати певні незручності. По-перше, такі повідомлення збільшують обсяг мережевого трафіку, що при великих обсягах запитів негативно впливає на продуктивність. По-друге, додаткові службові рядки можуть "засмічувати" результат, особливо тоді, коли процедура повинна повертати лише набір даних або значення через параметри *OUTPUT* чи оператор *RETURN*. У деяких клієнтських застосунках, особливо застарілих, це навіть може призвести до помилок інтерпретації результатів.

Використання *SET NOCOUNT ON* дозволяє уникнути цих проблем: процедура повертає лише потрібні дані, без службових повідомлень. Якщо ж є потреба знову вмикати стандартну поведінку, використовується команда *SET NOCOUNT OFF*. Таким чином, ця конструкція підвищує ефективність виконання процедур і робить їх результати чистішими та зручнішими для подальшої обробки.

Оператор *RETURN* використовується для повернення цілого числа, яке виконує роль коду завершення процедури. Традиційно значення 0 інтерпретується як успішне завершення, а будь-яке інше число, як код помилки або спеціальний статус. *RETURN* зручний у тих випадках, коли важливо швидко повідомити про результат виконання або виникнення помилки без передачі додаткових даних. Його можна використовувати разом із блоками *TRY...CATCH*, щоб реалізувати власну систему обробки помилок і повернення кодів.

Синтаксис оператора RETURN:

```
CREATE PROCEDURE [schema_name.]procedure_name
   @Параметр <тип>
AS
BEGIN
   -- тіло процедури
   IF <умова>
      RETURN 0; -- успіх
   ELSE
      RETURN 1; -- помилка/інший статус
END;

-- Виклик:
DECLARE @ResultCode INT;
EXEC @ResultCode = <ім'я_процедури> @Параметр = <значення>;
```

ПРИКЛАД 8.5. Створити збережену процедуру, яка перевіряє, чи є книга дорожчою за 300, і повертає код завершення:
0 – книга дешевша або дорівнює 300,
1 – книга дорожча,
2 – книга відсутня:

ЛИСТИНГ 8.5. Використання RETURN для таблиці Books

```
USE DB_Book
GO

  -- Оголошення процедури, де приймається лише один
   -- параметр  @IdBook, тобто ідентифікатор книги:
CREATE PROCEDURE Bookstore.CheckBookPrice
```

```
    @IdBook INT
AS
BEGIN
    SET NOCOUNT ON;

   -- Створюється змінна для збереження вартості книги:
    DECLARE @Price DECIMAL(10,2);

   -- Пошук ціни книги: якщо книга є в таблиці Books, її
   --  запишеться в @Price.
    SELECT TOP(1) @Price = Price
    FROM Bookstore.Books
    WHERE IdBook = @IdBook;

   -- Перевірка на відсутність книги: якщо книга не знайдена —
   -- повертається код 2.
    IF @Price IS NULL
        RETURN 2; -- книга не знайдена

 -- Перевірка ціни: якщо книга дорожча за 300  -- повертається 1;
  -- якщо дешевша або рівно 300 — повертається 0.
    IF @Price > 300
        RETURN 1; -- книга дорога
    ELSE
        RETURN 0; -- книга дешевша або дорівнює 300
END;
GO


   -- Виклик процедури: код завершення зберігається у змінну
   -- @Code. PRINT виводить результат
DECLARE @Code INT;
EXEC @Code = Bookstore.CheckBookPrice @IdBook = 5;
PRINT N'Код завершення: ' + CAST(@Code AS NVARCHAR);
```

У результат виконання, якщо книги з *IdBook* = 5 немає:

```
Код завершення: 2
```

Якщо книга є і коштує 500:

```
Код завершення: 1
```

Якщо книга є і коштує 250:

Код завершення: 0

Оператор + *CAST* використовується для поєднання тексту та числового значення, яке повернула процедура через оператор *RETURN*. Змінна *@Code* має тип *INT*, оскільки в *SQL Server* оператор *RETURN* може повертати лише цілі числа. Якщо спробувати додати число до рядка без приведення типів, результат може бути непередбачуваним або призвести до помилки. У *T-SQL* оператор + використовується і для арифметичного додавання, і для конкатенації рядків; коли один із операндів має числовий тип, *SQL Server* застосовує правила неявного приведення типів за пріоритетом, і тому може відрізнятися залежно від конкретних типів даних. Щоб уникнути непорозумінь, завжди явно приводьте число до рядкового типу за допомогою *CAST* або *CONVERT*, наприклад:

Саме тому застосовується функція CAST, яка змінює тип даних. У наведеному прикладі конструкція *CAST(@Code AS NVARCHAR)* перетворює число у символьний рядок. Після цього оператор + може успішно скласти два рядки: текстове повідомлення "Код завершення: " та значення змінної *@Code*, яке вже подане у вигляді тексту.

У результаті команда *PRINT* виведе зрозуміле повідомлення, наприклад:

Код завершення: 1

Таким чином, *CAST* у цьому випадку необхідний для правильного поєднання числового та текстового значення у єдиний рядок.

Поєднання вихідних параметрів і *RETURN* забезпечує гнучкий механізм взаємодії між збереженими процедурами та викликаним середовищем. Вихідні параметри дозволяють повертати конкретні значення, а *RETURN* – узагальнений статус виконання.

Таблиця 8.1. Параметри та ключові механізми у збережених процедурах SQL Server

Тип	***Призначення***	***Використання***	***Приклад***
INPUT	Передає значення в процедуру для використання всередині її тіла	Фільтрація даних, вставка, обчислення	*@IdЖанру INT* у процедурі *GetGenreById*
OUTPUT	Дозволяє процедурі	Повернення результату	*@RowCount INT OUTPUT* для

Тип	***Призначення***	***Використання***	***Приклад***
	повертати значення через параметр	виконання або обчислених значень без SELECT	кількості оброблених рядків
Значення за замовчуванням	Дозволяє викликати процедуру без явного вказання параметра	Використовується стандартне значення при відсутності аргументу	*@ТипЖанру NVARCHAR(100) = N'Невизначено'*
Result set	SELECT всередині процедури повертає таблицю рядків клієнту	Отримання набору даних через SELECT	*SELECT IdЖанру, НазваЖанру FROM Bookstore.Genres*
RETURN	Повертає код завершення процедури	0 — успішне виконання, інше — код помилки	*RETURN 0* або *RETURN -1*
DECLARE	Оголошує локальні змінні всередині процедури	Тимчасове зберігання проміжних значень, підготовка даних для OUTPUT або SELECT	*DECLARE @TempCount INT;*
SET / SELECT	Присвоєння значень локальним змінним	Проміжні обчислення, формування OUTPUT	*SET @TempCount = (SELECT COUNT(*) FROM Bookstore.Genres);*
SET NOCOUNT ON / OFF	Контролює повідомлення про кількість оброблених рядків	Зменшення зайвого трафіку клієнту, оптимізація виконання	*SET NOCOUNT ON;*
BEGIN...END	Об’єднує кілька	Логічне групування операторів	*BEGIN ... END*

Тип	*Призначення*	*Використання*	*Приклад*
	операторів у блок	всередині процедури	
IF...ELSE	Умовна логіка всередині процедури	Виконання різних блоків коду залежно від параметрів або змінних	*IF @IdЖанру IS NULL BEGIN ... END ELSE BEGIN ... END*
WHILE / LOOP	Циклічна обробка даних	Повторне виконання обчислень або формування проміжних результатів	*WHILE @i < 10 BEGIN ... SET @i = @i + 1; END*
BEGIN TRY...END TRY / BEGIN CATCH...END CATCH	Обробка помилок	Ловлення винятків, отримання повідомлень про помилки	*BEGIN TRY ... END TRY BEGIN CATCH ... END CATCH*
RAISERROR / THROW	Генерація власних повідомлень про помилки	Інформування клієнта про проблеми або нестандартні ситуації	*RAISERROR ('Помилка: %s', 16, 1, @ErrMsg);*
EXEC / EXECUTE	Виклик збереженої процедури	Передача значень параметрів та отримання результатів	*EXEC GenreSummary @IdЖанру = 101;*

8.5. Управління збереженими процедурами

Управління збереженими процедурами передбачає комплекс дій зі створення, модифікації, перегляду та видалення процедур, а також підтримання їхньої ефективності та безпеки. Збережені процедури, як структуровані об’єкти бази даних, здатні автоматизувати виконання *SQL*-запитів та бізнес-логіки, тому належне управління ними є необхідною умовою стабільної роботи інформаційної системи.

Одним із перших етапів управління процедурами є їхнє переглядання та аналіз існуючих об’єктів. У СКБД, таких як *Microsoft SQL Server*, для цього використовуються системні представлення та каталоги, наприклад, *sys.procedures*, яке містить детальну інформацію про наявні процедури в базі даних, та стандартне представлення *INFORMATION_SCHEMA.ROUTINES,* що дозволяє отримати загальні відомості про назви процедур, їхні типи, дати створення та модифікації. Цей крок дозволяє адміністраторам та розробникам оцінити поточний стан процедур і визначити необхідність внесення змін або оптимізації (синтаусис ключових аспектів управління збереженими процедурами представлено у табл. 8.2.).

Процес модифікації існуючих процедур передбачає внесення змін у їхній код без видалення об’єкта, що забезпечує збереження залежностей та наданих прав доступу. У *SQL Server* для цього застосовується оператор *ALTER PROCEDURE,* який дозволяє додавати нові параметри, змінювати логіку виконання або оптимізувати запити. Використання цього підходу є пріоритетним у порівнянні з видаленням та повторним створенням процедури, оскільки забезпечує збереження стабільності системи та запобігає можливим помилкам у залежних об’єктах.

У разі, коли процедура більше не потрібна або її необхідно замінити, застосовується видалення процедур за допомогою команди *DROP PROCEDURE.* Перед видаленням важливо перевірити залежності, оскільки існування інших процедур, представлень або тригерів, що використовують видаляємий об’єкт, може призвести до помилок у виконанні. Це підкреслює необхідність системного підходу до адміністрування та аналізу взаємозв’язків між об’єктами бази даних.

Для забезпечення ефективного управління процедурами важливо вести документацію та контроль версій. Це включає фіксацію автора змін, дати модифікації та опису призначення процедури. Ведення коментарів у коді та використання систем контролю версій (наприклад, *Git* або *SVN*) дозволяють відстежувати історію змін, полегшують співпрацю команди розробників та забезпечують можливість відновлення попередніх версій процедур у разі виникнення помилок.

Особливу увагу слід приділяти аналізу залежностей процедур від інших об’єктів бази даних. Збережені процедури часто використовують таблиці, інші процедури або функції, тому для безпечного внесення змін необхідно визначати їхні взаємозв’язки. У *SQL Server* для цього можна використовувати представлення *sys.sql_expression_dependencies*, яке відображає об’єкти, від яких залежить процедура, що дозволяє уникнути порушення логіки роботи системи при модифікації або видаленні залежних об’єктів.

У практичній діяльності часто використовують тимчасові та глобальні тимчасові процедури, які існують протягом обмеженого сеансу або для всіх користувачів. Локальні тимчасові процедури позначаються символом # на початку імені, а глобальні – ##. Їхнє управління аналогічне звичайним процедурам, проте вони автоматично видаляються після завершення сеансу або коли всі користувачі завершують використання відповідного об’єкта.

Управління процедурами також включає планування та автоматизацію їх виконання, що реалізується через планувальники завдань, наприклад *SQL Server Agent*. Це дозволяє автоматизувати рутинні операції, такі як регулярна генерація звітів, оновлення або архівація даних, що підвищує продуктивність роботи бази даних і зменшує навантаження на адміністратора.

Нарешті, важливим аспектом є резервне копіювання та відновлення процедур. Хоча процедури зберігаються всередині бази даних, рекомендується також зберігати їх у вигляді *SQL*-скриптів, що дозволяє швидко відновлювати їх у разі помилок, переносити між БД або серверами та забезпечувати додатковий рівень безпеки для критично важливих об’єктів.

Таблиця 8.2. Ключові аспекти управління збереженими процедурами

Аспект управління	***Загальний синтаксис***	***Приклад виклику***
Перегляд наявних процедур	*SELECT name, create_date, modify_date FROM sys.procedures; або sql SELECT ROUTINE_NAME, ROUTINE_TYPE, CREATED, LAST_ALTERED FROM INFORMATION_SCHEMA. ROUTINES WHERE ROUTINE_TYPE = 'PROCEDURE';*	–
Створення процедури	*CREATE PROCEDURE [schema_name.]procedure_ name @Параметр1 <тип>, @Параметр2 <тип> = <значення_за_замовчуванням > AS BEGIN <SQL_оператори> END;*	*EXEC [schema_name.]procedure _name @Параметр1 = 10, @Параметр2 = 'Тест';*

Аспект управління	***Загальний синтаксис***	***Приклад виклику***
Модифікація процедури	*ALTER PROCEDURE [schema_name.]procedure_name @Параметр1 <тип>, @Параметр2 <тип> AS BEGIN <нове_тіло_процедури> END;*	*EXEC [schema_name.]procedure_name @Параметр1 = 5;*
Видалення процедури	*DROP PROCEDURE [schema_name.]procedure_name;*	–
Використання вихідних параметрів та RETURN	*CREATE PROCEDURE [schema_name.]procedure_name @Параметр <тип>, @ВихіднийПараметр <тип> OUTPUT AS BEGIN IF <умова> SET @ВихіднийПараметр = <значення>; RETURN 0; ELSE SET @ВихіднийПараметр = <інше_значення>; RETURN 1; END;*	*DECLARE @ResultCode INT, @OutputValue INT; EXEC @ResultCode = [schema_name.]procedure_name @Параметр = 10, @ВихіднийПараметр = @OutputValue OUTPUT;*
Тимчасові процедури	Локальна: *CREATE PROCEDURE #TempProcedure AS BEGIN <SQL_оператори> END;* Глобальна: *CREATE PROCEDURE ##GlobalTempProcedure AS BEGIN*	*EXEC #TempProcedure; sql EXEC ##GlobalTempProcedure;*

Аспект управління	*Загальний синтаксис*	*Приклад виклику*
	<SQL_оператори> *END;*	
Плануван-ня та автомати-зація	Виклик через T-SQL: *EXEC [schema_name.]procedure_name @Параметр = <значення>;* Планування: SQL Server Agent *Job (через GUI або T-SQL)*	*EXEC dbo.MyProcedure @Param1 = 10;*
Резервне копіювання та відновлення	Отримання тексту процедури: *EXEC sp_helptext '[schema_name.]procedure_name';*	–

8.6. Оптимізація та продуктивність збережених процедур

Оптимізація та продуктивність збережених процедур є базовим аспектом для розробки баз даних, оскільки від їхньої ефективності залежить швидкість виконання запитів, навантаження на сервер і стабільність роботи системи. Збережені процедури часто обробляють великі обсяги даних і виконують складну бізнес-логіку, тому їхнє належне проектування та оптимізація забезпечують економію ресурсів і покращують користувацький досвід.

1. Використання індексів для оптимізації доступу до даних

Одним із найефективніших методів оптимізації є використання індексів на таблицях, що беруть участь у процедурах. Індекси дозволяють СКБД швидше знаходити необхідні рядки без повного сканування таблиці. Це особливо важливо для колонок, які використовуються у *WHERE, JOIN, GROUP BY* або *ORDER BY*. Надмірне індексування може уповільнити операції вставки, оновлення і видалення, тому необхідно балансувати між швидкістю читання і запису.

ПРИКЛАД 8.6. Для прискорення вибірки даних у таблицях *PriceList, Sales та SalesItems* можна створювати індекси на колонках, що часто використовуються у *WHERE* або *JOIN*.

ЛИСТИНГ 8.6. Синтаксис створення індексу

```
-- Індекс для швидкого пошуку книг за автором
CREATE INDEX IX_PriceList_IdАвтора
```

```
ON Bookstore.PriceList(IdАвтора);

-- Індекс для швидкого пошуку замовлень клієнта
CREATE INDEX IX_Sales_IdКлієнта
ON Bookstore.Sales(IdКлієнта);
```

2. Використання планів виконання

Збережені процедури дозволяють СКБД зберігати план виконання запиту (*Execution Plan*), що зменшує час компіляції при повторних викликах. Аналіз планів виконання допомагає виявляти «важкі» запити та вузькі місця продуктивності. Для цього використовуються інструменти профілювання, такі як *SET STATISTICS IO* та *SET STATISTICS TIME*, які показують кількість операцій вводу/виводу та витрати часу на виконання.

Для аналізу продуктивності можна увімкнути статистику вводу/виводу та часу для конкретної процедури.

Синтаксис операторів профілювання:

```
-- показує кількість операцій вводу/виводу
SET STATISTICS IO ON;

-- показує час компіляції та виконання запиту
SET STATISTICS TIME ON;

-- виклик процедури
EXEC [schema_name.]procedure_name @Параметр = <значення>;
```

ПРИКЛАД 8.7. Необхідно оцінити продуктивність процедури, яка отримує всі замовлення конкретного клієнта, та визначити витрати ресурсів (I/O та час виконання) для оптимізації запитів.

ЛИСТИНГ 8.7. Використання планів виконання дл БД Bookstore

```
-- Включення статистики вводу/виводу та часу
SET STATISTICS IO ON;
SET STATISTICS TIME ON;

-- Виклик процедури для клієнта з IdКлієнта = 1
EXEC Bookstore.GetClientOrders @IdКлієнта = 1;
```

Пояснення:

- *SET STATISTICS IO ON* дозволяє побачити кількість сканувань таблиць та використання індексів під час виконання процедури;
- *SET STATISTICS TIME ON* показує час компіляції та виконання процедури у мілісекундах;
- виклик *EXEC Bookstore;GetClientOrders @IdКлієнта = 1* повертає список замовлень конкретного клієнта;
- на основі отриманих даних можна виявити вузькі місця у виконанні процедури та оптимізувати її запити.

3. Мінімізація кількості викликів та використання масових операцій

Для підвищення продуктивності важливо мінімізувати кількість звернень до бази даних та уникати вкладених циклів. Замість виконання запитів у циклі, краще застосовувати масові операції (*Set-Based Operations*), які дозволяють обробляти всі дані за один виклик, а це, в свою чергу, значно зменшує витрати та час виконання. Тобто, уникаємо вкладених циклів, застосовуючи *Set-Based* операції для вставки або оновлення даних.

ПРИКЛАД 8.8. Необхідно вставити декілька книг у таблицю *SalesItems* для конкретного чеку. Замість циклічного додавання кожної книги окремим запитом слід використати масову операцію, що підвищує продуктивність та зменшує навантаження на сервер.

Синтаксис загальної операції масової вставки

```
INSERT INTO <TargetTable>(<Колонка1>, <Колонка2>, ...)
SELECT <Колонка1>, <Колонка2>, ...
FROM <SourceTable>
WHERE <умова>;
```

ЛИСТИНГ 8.8. Використання масової операції для підвищення продуктивності

```
   -- Масова вставка всіх книг замовлення в чек №1
INSERT INTO Bookstore.SalesItems (IdЧеку, IdКниги, Кількість)
SELECT 1, IdКниги, 1
FROM Bookstore.PriceList
   -- наприклад, вставляємо всі книги жанру "Фантастика"
WHERE IdЖанру = 2;
```

Пояснення**:**

- масова вставка обробляє всі рядки одночасно, замість циклу по кожній книзі;
- *SELECT* визначає набір рядків для вставки з таблиці *PriceList* за певною умовою (*IdЖанру = 2*);

- *INSERT INTO* додає всі результати одразу в таблицю *SalesItems*.
- такий підхід зменшує час виконання та витрати СКБД, особливо при великій кількості рядків.

4. Використання оптимальних типів даних

Вибір типів даних оптимального розміру впливає на продуктивність процедури. Використання великих типів (наприклад, *NVARCHAR(4000)* для коротких текстових полів) збільшує обсяг пам'яті, час передачі даних і навантаження на сервер. Для підвищення продуктивності слід використовувати типи даних, які відповідають реальному обсягу інформації, що зберігається.

В таблицях *Authors, Clients, PriceList* вже використано типи *INT, NVARCHAR* і *DECIMAL*. Для оптимізації треба перевіряти, чи немає перевитрат пам'яті, наприклад:

- замість *NVARCHAR(200)* для невеликих назв можна застосовувати *NVARCHAR(100);*
- замість *INT* для маленького діапазону чисел використовувати *SMALLINT* або *TINYINT*.

5. Використання параметрів і фільтрації

Збережені процедури повинні обробляти тільки необхідні дані через вхідні параметри. Правильне використання вхідних параметрів дозволяє СКУБД ефективно застосовувати індекси і уникати повного сканування таблиць. Фільтрація даних на рівні *SQL*-запиту зменшує обсяг оброблюваних рядків і підвищує швидкість виконання процедури.

ПРИКЛАД 8.9. Необхідно отримати всі замовлення конкретного клієнта з деталізацією книг, використовуючи параметри для фільтрації даних. Це дозволяє обробляти лише потрібні записи та підвищує продуктивність процедури.

Синтаксис загальної процедури з параметром

```
INSERT INTO <TargetTable>(<Колонка1>, <Колонка2>, ...)
REATE PROCEDURE [schema_name.]procedure_name
  @ВхіднийПараметр <тип>
AS
BEGIN
  SELECT <стовпці>
  FROM <таблиця1> t1
  JOIN <таблиця2> t2 ON t1.<ключ> = t2.<ключ>
  WHERE <стовпець> = @ВхіднийПараметр;
END;
```

```
-- Виклик процедури
EXEC [schema_name.]procedure_name @ВхіднийПараметр = <значення>;
```

ЛИСТИНГ 8.9. Використання загальної процедури з параметром для БД Bookstore

```
-- Створення процедури для отримання замовлень конкретного
-- клієнта
CREATE PROCEDURE Bookstore.GetClientOrders
   @IdКлієнта INT
AS
BEGIN
   SELECT s.IdЧеку, s.ДатаЗамовлення, si.IdКниги, si.Кількість,
     p.НазваКниги
   FROM Bookstore.Sales s
   JOIN Bookstore.SalesItems si ON s.IdЧеку = si.IdЧеку
   JOIN Bookstore.PriceList p ON si.IdКниги = p.IdКниги
   WHERE s.IdКлієнта = @IdКлієнта;
END;

-- Виклик процедури для клієнта з IdКлієнта = 1
EXEC Bookstore.GetClientOrders @IdКлієнта = 1;
```

Пояснення:

- параметр *@IdКлієнта* дозволяє обробляти лише ті замовлення, які належать конкретному клієнту;
- використання *WHERE s.IdКлієнта = @IdКлієнта* забезпечує фільтрацію на рівні *SQL*, зменшуючи обсяг оброблюваних даних;
- *JOIN* між таблицями *Sales, SalesItems* та *PriceList* дозволяє отримати повну інформацію про замовлення та книги;
- завдяки параметру і фільтрації процедура виконується швидше і економніше використовує ресурси сервера.

6. Оптимізація транзакцій

Тривалі транзакції блокують ресурси та можуть призводити до конфліктів доступу. Для підвищення продуктивності рекомендується мінімізувати час виконання транзакцій, обмежуючи кількість операцій всередині *BEGIN TRANSACTION ... COMMIT/ROLLBACK*. Короткі транзакції зменшують ризик блокувань і забезпечують стабільну роботу бази даних.

ПРИКЛАД 8.10. Необхідно створити процедуру додавання нового замовлення та його позицій у таблиці *Sales* і *SalesItems* так, щоб транзакція була короткою, зменшуючи блокування та підвищуючи продуктивність бази даних.

Синтаксис оператора транзакції

```
BEGIN TRANSACTION;  -- початок транзакції
   <SQL_оператори вставки/оновлення>
COMMIT TRANSACTION; -- фіксація змін
-- або
ROLLBACK TRANSACTION; -- відкат у разі помилки
```

ЛИСТИНГ 8.10. Використання оператора транзакції для БД Bookstore

```
CREATE PROCEDURE Bookstore.AddSale
   @IdКлієнта INT,
   @IdКниги INT,
   @Кількість INT = 1
AS
BEGIN
   BEGIN TRANSACTION;

   -- Додавання нового замовлення
   DECLARE @NewCheckID INT;
   INSERT INTO Bookstore.Sales(IdКлієнта)
   VALUES (@IdКлієнта);
   SET @NewCheckID = SCOPE_IDENTITY();

   -- Додавання позиції книги у замовлення
   INSERT INTO Bookstore.SalesItems(IdЧеку, IdКниги, Кількість)
   VALUES (@NewCheckID, @IdКниги, @Кількість);

   COMMIT TRANSACTION;
END;

-- Виклик процедури
EXEC Bookstore.AddSale @IdКлієнта = 1, @IdКниги = 101,
@Кількість = 2;
```

Пояснення:

- транзакція починається оператором *BEGIN TRANSACTION* і закінчується *COMMIT*, що гарантує атомарність операцій;
- всі зміни (створення замовлення і додавання позиції книги) виконуються одночасно, зменшуючи ризик блокувань;
- *SCOPE_IDENTITY*() використовується для отримання *ID* щойно доданого замовлення;
- коротка тривалість транзакції забезпечує високу продуктивність та стабільну роботу бази даних.

7. Профілювання та моніторинг продуктивності

Регулярне профілювання процедур дозволяє виявляти повільні запити та «вузькі місця». Для цього використовуються динамічні представлення *sys.dm_exec_procedure_stats* та *sys.dm_exec_query_stats*, які показують час виконання, кількість викликів та ресурси, витрачені на запит.

ПРИКЛАД 8.11. Необхідно визначити найповільніші збережені процедури в базі *Bookstore* та оцінити їхнє навантаження на сервер, щоб виявити процедури, які потребують оптимізації.

Синтаксис загального запиту для моніторингу

```
SELECT TOP <кількість>
   -- загальний час процесора, витрачений на запит:
   qs.total_worker_time,

    -- кількість виконань:
   qs.execution_count,

  -- план виконання запиту:
   qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE OBJECT_NAME(qs.object_id) = '<ім'я_процедури>'
ORDER BY qs.total_worker_time DESC;
```

ЛИСТИНГ 8.11. Використання моніторингового запиту для БД Bookstore

```
-- Профілювання продуктивності процедури GetClientOrders
SELECT TOP 10

-- загальний час CPU, витрачений на виконання
   qs.total_worker_time,
```

```
    -- кількість викликів процедури
    qs.execution_count,
        -- XML-план виконання запиту
    qp.query_plan
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE OBJECT_NAME(qs.object_id) = 'GetClientOrders'
ORDER BY qs.total_worker_time DESC;
```

Пояснення:

- *qs.total_worker_time* показує, скільки часу витрачається на виконання запиту в CPU;
- *qs.execution_count* показує, скільки разів процедура була викликана;
- *qp.query_plan* повертає план виконання запиту, який допомагає визначити «вузькі місця»;
- використовуючи ці дані, можна ідентифікувати процедури, що навантажують систему, та оптимізувати їхній код або індекси.

8.7. Безпека і керування доступом до процедур

Безпека та керування доступом до збережених процедур є критично важливим аспектом адміністрування баз даних. Збережені процедури можуть містити бізнес-логіку та конфіденційну інформацію, тому контроль доступу забезпечує захист даних, запобігає несанкціонованим змінам і підвищує загальну надійність системи.

1. Ролі і дозволи користувачів

Контроль доступу до збережених процедур здійснюється через ролі та права користувачів. Кожен користувач може отримувати права безпосередньо або через роль, що спрощує адміністрування великих груп користувачів. Завдяки цьому адміністратор бази даних може централізовано керувати доступом і швидко змінювати права для групи користувачів без необхідності змінювати дозволи для кожного користувача окремо.

Надання дозволів обмежує дії, які користувач може виконувати над процедурою, зокрема запуск, зміну або видалення. Це забезпечує принцип мінімальних привілеїв, коли користувач отримує лише ті права, які необхідні для виконання його функцій.

Додатково, використання ролей дозволяє логічно розділити користувачів за рівнем відповідальності та функціоналом. Наприклад, роль «Аналітик» може мати лише право виконувати процедури, тоді як роль «Адміністратор» –

право змінювати їх. Такий підхід підвищує безпеку та зменшує ризик несанкціонованих змін у базі даних.

У *SQL Server* доступ до процедур керується через ролі та дозволи (*Permissions*). Можна призначати права окремим користувачам або групам користувачів. Основні види дозволів:

EXECUTE – право виконувати процедуру;

ALTER – право змінювати код процедури;

CONTROL – повний контроль над процедурою, включно з видаленням;

GRANT, DENY, REVOKE – механізми для надання або відміни дозволів.

Синтаксис надання права виконання процедури

```
-- Надання права виконання процедури конкретному користувачу
GRANT EXECUTE ON Bookstore.GetClientOrders TO [UserName];

-- Відміна права
REVOKE EXECUTE ON Bookstore.GetClientOrders FROM [UserName];
```

Це дозволяє точно визначити, хто може запускати процедуру, а хто – ні.

2. Рівні доступу

Безпека процедур можна організувати на різних рівнях:

- Рівень бази даних – контроль доступу до всіх процедур у схемі;
- Рівень схеми – контроль доступу до групи процедур у схемі (наприклад, Bookstore);
- Рівень окремої процедури – найдетальніший рівень контролю.

Рівень бази даних забезпечує загальний контроль доступу до всіх процедур у системі і використовується для глобального адміністрування і дозволяє швидко надавати або віднімати права великій кількості об’єктів одночасно, що особливо корисно у великих організаціях.

Рівень схеми дозволяє керувати доступом до групи процедур, об’єднаних за логічним або функціональним принципом, забезпечує більш гнучкий підхід, оскільки адміністратор може визначити права на весь підсистемний блок без необхідності налаштовувати кожен об’єкт окремо.

Рівень окремої процедури забезпечує детальний контроль, коли права надаються або відміняються тільки для конкретного об’єкта і дозволяє точно регламентувати дії користувачів над критично важливими процедурами та зменшує ймовірність випадкових помилок чи несанкціонованих змін.

Синтаксис надання доступу до всіх процедур схеми

```
-- Надання права EXECUTE для всіх процедур у схемі Bookstore
```

GRANT EXECUTE ON SCHEMA::Bookstore TO [UserName];

3. Безпека через власника схеми

Власник схеми є елементом управління безпекою, оскільки він контролює всі об'єкти всередині схеми, дозволяє централізовано адмініструвати права доступу та спрощує керування дозволами, зменшуючи необхідність окремо налаштовувати доступ до кожного об'єкта.

Користувачі, які мають права власника схеми, автоматично отримують контроль над усіма процедурами, таблицями та іншими об'єктами всередині схеми. Така модель зменшує ризик пропуску критичних об'єктів при налаштуванні безпеки та забезпечує єдиний центр управління доступом.

Цей підхід також дозволяє реалізувати розподілену відповідальність, коли адміністратори різних схем контролюють тільки свої об'єкти, не впливаючи на чужі. Таким чином підвищується безпека та знижується ризик внутрішніх порушень політики доступу.

Якщо процедура створена в схемі *Bookstore*, то користувачі з правами власника схеми можуть змінювати або видаляти всі процедури без додаткових дозволів.

Синтаксис створення схеми з авторизацією:

CREATE SCHEMA <ім'я_схеми> AUTHORIZATION <власник>;

Це дозволяє централізовано контролювати права доступу для всіх об'єктів схеми.

4. Використання *EXECUTE AS* для контролю контексту виконання

Механізм *EXECUTE AS* дозволяє запускати процедуру від імені іншого користувача або ролі, ізолюючи права користувача, який викликає процедуру, від фактичного доступу до таблиць, цим самим підвищуючи безпеку, оскільки кінцевий користувач не отримує прямий доступ до даних і об'єктів, на які він не має прав. Такий підхід дозволяє централізовано контролювати доступ до конфіденційних даних і бізнес-логіки, забезпечуючи виконання процедур у безпечному контексті. Крім того, він спрощує адміністрування, оскільки адміністратори можуть визначити, під яким контекстом виконуються критичні процедури.

Використання *EXECUTE AS* також допомагає запобігти випадковим або навмисним порушенням безпеки, ізолюючи права користувачів та забезпечуючи централізовану точку контролю доступу до процедур.

Механізм *EXECUTE AS* дозволяє виконувати процедуру під іншим користувачем або роллю, що спрощує керування безпекою та підвищує захист даних.

Синтаксис використання EXECUTE AS

```
CREATE PROCEDURE Bookstore.GetClientOrders
-- процедура виконується від імені AppUser
 WITH EXECUTE AS 'AppUser'
AS
BEGIN
   SELECT * FROM Bookstore.Sales
   WHERE IdКлієнта = @IdКлієнта;
END;
```

Це дозволяє ізолювати права користувача, який викликає процедуру, від фактичного доступу до таблиць.

5. Аудит і контроль доступу

Аудит виконання процедур елемент безпеки, оскільки дозволяє відстежувати дії користувачів та виявляти потенційні загрози, забезпечує прозорість роботи системи і дозволяє аналізувати, хто і коли виконував процедури, які зміни вносив і які дії були спробами несанкціонованого доступу. Завдяки аудиту можна своєчасно виявляти порушення політик безпеки та реагувати на них, що значно знижує ризик витоку конфіденційної інформації або пошкодження даних. Аудит також важливий для дотримання нормативних вимог та внутрішніх стандартів безпеки організації.

Регулярне використання механізмів аудиту дозволяє підтримувати високий рівень контролю за діяльністю користувачів, підвищує дисципліну роботи з даними та сприяє своєчасному вдосконаленню політик доступу до процедур.

Для забезпечення безпеки рекомендується вести аудит виконання процедур, що дозволяє відстежувати дії користувачів та ідентифікувати потенційні загрози. *SQL Server* надає механізми аудиту (*SQL Server Audit*) для запису всіх викликів процедур, змін дозволів і спроб несанкціонованого доступу.

Приклад налаштування аудиту (загальна концепція):

```
-- Створення аудиту
CREATE SERVER AUDIT BookstoreAudit
TO FILE (FILEPATH = 'C:\AuditLogs\');

-- Створення специфікації для процедур
CREATE DATABASE AUDIT SPECIFICATION
BookstoreProcedureAudit
FOR SERVER AUDIT BookstoreAudit
```

ADD (EXECUTE ON OBJECT::Bookstore.GetClientOrders BY [public]);

Це забезпечує прозорість дій користувачів і дозволяє своєчасно реагувати на порушення.

6. Найкращі практики безпеки

Впровадження найкращих практик забезпечує комплексний захист процедур та бази даних. Принцип мінімальних привілеїв гарантує, що користувач отримує лише ті права, які необхідні для виконання конкретних завдань, що значно знижує ймовірність несанкціонованого доступу.

Використання схем і ролей дозволяє централізовано управляти доступом, спрощує адміністрування та підвищує гнучкість у керуванні дозволами. Регулярна перевірка прав користувачів та аудит виконання процедур дозволяє своєчасно виявляти потенційні проблеми та контролювати дотримання політик безпеки.

Ізоляція контексту виконання процедур за допомогою *EXECUTE AS* і ведення аудиту доступу до критичних процедур забезпечують додатковий рівень захисту та мінімізують ризики втрати або викрадення даних. Використання цих практик створює надійну систему безпеки та підтримує стабільну роботу бази даних.

Питання для самоперевірки

1. *Що таке збережена процедура у SQL Server і яке її основне призначення?*
2. *Які переваги надає використання збережених процедур порівняно з безпосереднім виконанням SQL-запитів?*
3. *У чому полягає роль збережених процедур у підвищенні продуктивності роботи системи?*
4. *Як збережені процедури сприяють підвищенню безпеки доступу до даних?*
5. *Чому використання збережених процедур зменшує мережевий трафік між клієнтом і сервером?*
6. *Які команди застосовуються для створення, модифікації, видалення та виконання збережених процедур?*

7. *Які види ідентифікаторів застосовуються у Transact-SQL при роботі зі змінними та процедурами?*
8. *Яка загальна структура (синтаксис) оголошення збереженої процедури?*
9. *Які ключові слова та параметри визначають сигнатуру збереженої процедури?*
10. *Які існують типи параметрів у збережених процедурах? Наведіть приклади.*
11. *Чим відрізняється вихідний параметр OUTPUT від звичайного вхідного параметра?*
12. *Яким чином процедура може повертати результати виконання? Назвіть три основні способи.*
13. *Яке призначення має оператор RETURN у збережених процедурах?*
14. *Для чого використовується конструкція SET NOCOUNT ON у збережених процедурах?*
15. *Як реалізується обробка виняткових ситуацій у збережених процедурах?*
16. *Які основні аспекти охоплює управління збереженими процедурами?*
17. *Чим відрізняється використання команди ALTER PROCEDURE від DROP PROCEDURE?*
18. *Які можливості забезпечує механізм EXECUTE AS при виконанні процедур?*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 8

на тему: «Збережені процедури в Transact-SQL»

МЕТА: *закріпити знання та практичні навички роботи із збереженими процедурами в SQL Server. Навчитися створювати та модифікувати процедури для автоматизації обробки даних у базі, використовувати вхідні та вихідні параметри, оператори RETURN, а також формувати запити з урахуванням логіки бізнес-процесів*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. Створити просту збережену процедуру для вибірки даних з таблиці.
2. Розробити збережену процедуру для додавання нового запису в таблицю.
3. Створити процедуру з вхідними параметрами для пошуку даних за заданими критеріями.
4. Реалізувати процедуру з вихідними параметрами для отримання підсумкової або агрегованої інформації.
5. Розробити процедуру, яка виконує кілька взаємопов’язаних операцій із використанням транзакції.
6. Продемонструвати використання оператора RETURN для повернення результату виконання процедури.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

У SQL Server збережені процедури застосовуються для багаторазового виконання однакових операцій над даними. Вони інкапсулюють логіку запиту в окремий об’єкт бази даних, що дозволяє не лише скоротити дублювання коду, але й підвищити зручність та безпеку роботи з інформацією. Простим прикладом використання збереженої процедури є створення запиту для отримання даних з однієї таблиці без будь-яких додаткових параметрів.

У структурі бази даних «Букіністичний магазин» є таблиця PriceList, яка містить основні відомості про книги: унікальний ідентифікатор (IdКниги), назву книги (НазваКниги), рік видання, посилання на автора, жанр,

видавництво, а також ціну (Ціна). У рамках даного завдання необхідно створити процедуру, яка повертатиме базовий перелік книг із зазначенням їх ідентифікаторів, назв і цін.

Для цього використовується оператор *CREATE PROCEDURE*, за допомогою якого визначається нова процедура. Ім'я процедури формується з урахуванням схеми, у нашому випадку *Bookstore.GetBooksList*. Основна логіка процедури описується у блоці *BEGIN ... END*, усередині якого розташовується інструкція *SELECT*, що обирає необхідні поля з таблиці.

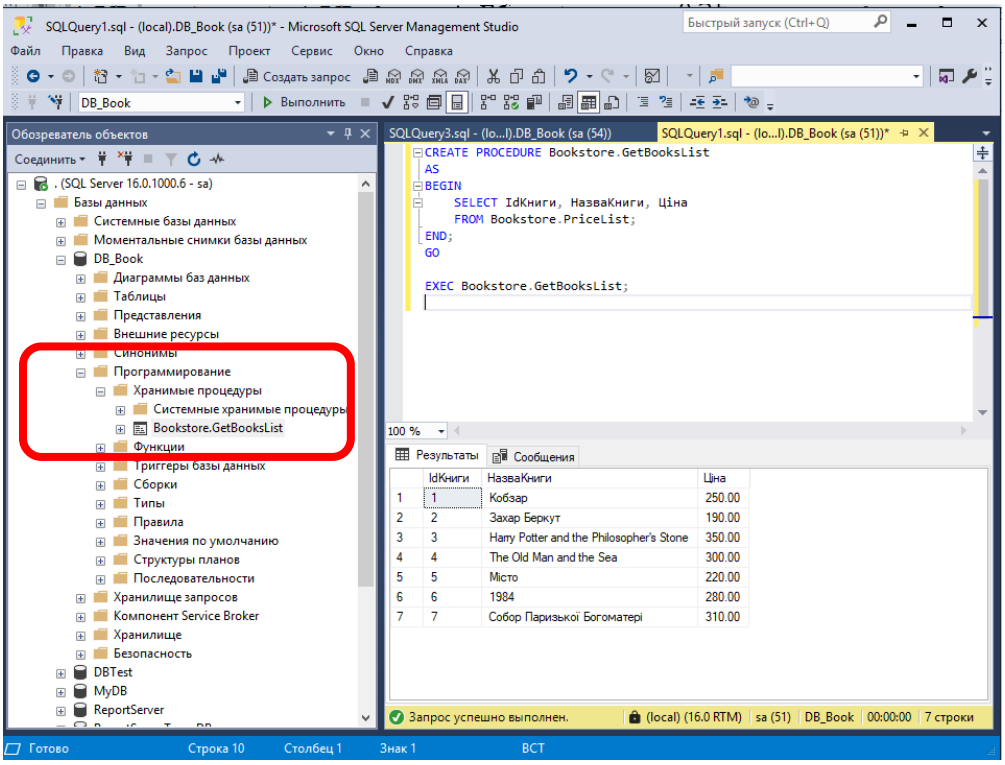
Лістинг 1. Створення простої збереженої процедури

```
CREATE PROCEDURE Bookstore.GetBooksList
AS
BEGIN
    SELECT IdКниги, НазваКниги, Ціна
    FROM Bookstore.PriceList;
END;
```

Виклик процедури здійснюється командою *EXEC* (або *EXECUTE*). У нашому випадку це:

```
EXEC Bookstore.GetBooksList;
```

У результаті виконання процедури буде такий результат:



Збережена процедуру для додавання нового запису в таблицю

У багатьох бізнес-процесах роботи з базами даних виникає потреба не лише переглядати вже наявну інформацію, але й додавати нові записи. Для цього в SQL Server також можуть використовуватися збережені процедури. На відміну від звичайного виконання команди *INSERT INTO* безпосередньо в SQL, збережена процедура дозволяє централізовано управляти логікою додавання даних, забезпечуючи повторне використання коду, підвищену безпеку та зменшення ймовірності помилок при введенні даних. Такі процедури приймають вхідні параметри, що відповідають полям таблиці, та виконують вставку нового запису на основі цих параметрів.

У таблиці *Bookstore.Clients* поле *IdКлієнта* є первинним ключем (*PRIMARY KEY*) і не допускає *NULL*. Це означає, що кожен новий запис повинен мати унікальне значення ключа, і його неможливо залишити порожнім. У нашому випадку поле не має автоматичної генерації значень (не є *IDENTITY*), тому значення *IdКлієнта* повинно визначатися безпосередньо в процедурі.

Процедура *Bookstore.AddNewClient* вирішує цю проблему, забезпечуючи автоматичне присвоєння нового унікального *IdКлієнта*, який обчислюється як *MAX(IdКлієнта) + 1*. Таким чином гарантується унікальність первинного ключа та послідовне додавання клієнтів у таблицю, без конфліктів із уже наявними записами.

Процедура приймає як параметри характеристики нового клієнта: назву, адресу, телефон, реквізити банку та ЄДРПОУ. При цьому обов'язковим є тільки поле *НайменуванняКлієнта*. Якщо воно не передане, процедура генерує помилку через *RAISERROR*, і вставка не відбувається.

Основні етапи виконання процедури:

1. Перевірка обов'язкового поля *НайменуванняКлієнта*. Якщо значення *NULL*, процедура зупиняється та видає повідомлення про помилку.
2. Автоматичне визначення нового *IdКлієнта*. Використовується функція *MAX(IdКлієнта)* для пошуку найбільшого існуючого значення ключа, після чого до нього додається одиниця. Якщо таблиця порожня, новий *Id* буде 1.
3. Вставка нового запису у таблицю за допомогою оператора *INSERT INTO*. Новий запис отримує обчислений *IdКлієнта* і всі передані параметри.
4. Завершення процедури без повернення значення, так як вставка відбувається всередині процедури, і користувач бачить результат у таблиці після виконання.

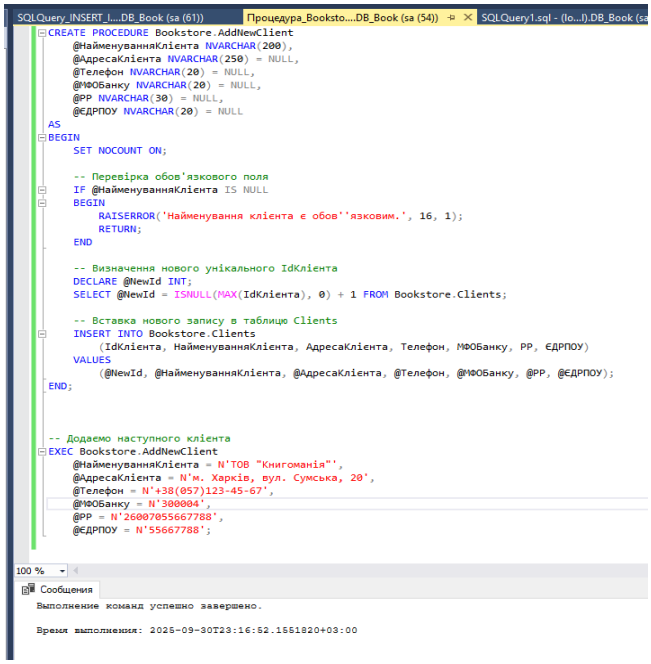
Цей підхід забезпечує цілісність даних, уникає помилок первинного ключа та дозволяє зручно додавати нових клієнтів без ручного введення Id.

Лістинг 2. Створення процедури процедуру для додавання нового клієнта в таблицю Bookstore.Clients.

```
CREATE PROCEDURE Bookstore.AddNewClient
    @НайменуванняКлієнта NVARCHAR(200),
    @АдресаКлієнта NVARCHAR(250) = NULL,
    @Телефон NVARCHAR(20) = NULL,
    @МФОБанку NVARCHAR(20) = NULL,
    @РР NVARCHAR(30) = NULL,
    @ЄДРПОУ NVARCHAR(20) = NULL
AS
BEGIN
    SET NOCOUNT ON;
    -- Перевірка обов'язкового поля
    IF @НайменуванняКлієнта IS NULL
    BEGIN
        RAISERROR('Найменування клієнта є обов''язковим.', 16, 1);
        RETURN;
    END
    -- Визначення нового унікального IdКлієнта
    DECLARE @NewId INT;
    SELECT @NewId = ISNULL(MAX(IdКлієнта), 0) + 1 FROM Bookstore.Clients;
    -- Вставка нового запису в таблицю Clients
    INSERT INTO Bookstore.Clients
        (IdКлієнта, НайменуванняКлієнта, АдресаКлієнта, Телефон, МФОБанку,
РР, ЄДРПОУ)
    VALUES
        (@NewId, @НайменуванняКлієнта, @АдресаКлієнта, @Телефон, @МФОБанку,
@РР, @ЄДРПОУ);
     END;
```

Приклад виклику процедури

```
-- Додаємо наступного клієнта
EXEC Bookstore.AddNewClient
    @НайменуванняКлієнта = N'ТОВ
"Книгоманія"',
    @АдресаКлієнта = N'м. Харків,
вул. Сумська, 20',
    @Телефон = N'+38(057)123-45-
67',
    @МФОБанку = N'300004',
    @РР = N'26007055667788',
    @ЄДРПОУ = N'55667788';
```



Створення процедури з вхідними параметрами для пошуку даних за заданими критеріями

У практичній роботі з базою даних часто виникає потреба отримувати не всі записи, а лише ті, що відповідають певним умовам. Для цього до збережених процедур можна додавати вхідні параметри, які дозволяють динамічно змінювати умови відбору. Таким чином, одна й та сама процедура може працювати з різними наборами даних без необхідності переписувати код.

У базі даних «Букіністичний магазин» прикладом такої задачі може бути пошук книг за певним автором. У таблиці *PriceList* зберігається інформація про книги, а в таблиці *Authors* – дані про авторів. Для виконання пошуку достатньо створити процедуру, яка прийматиме вхідний параметр (наприклад, прізвище автора) та повертатиме список книг, що належать даному авторові.

Створення процедури здійснюється за допомогою оператора *CREATE PROCEDURE*, який створює нову збережену процедуру з іменем *Bookstore.GetBooksByAuthor*. У дужках оголошується параметр *@AuthorLastName* типу *NVARCHAR*(*30*), який слугуватиме умовою відбору записів у запиті: користувач під час виклику процедури передає значення цього параметра (наприклад, «Франко»), і воно підставляється в умову *WHERE*.

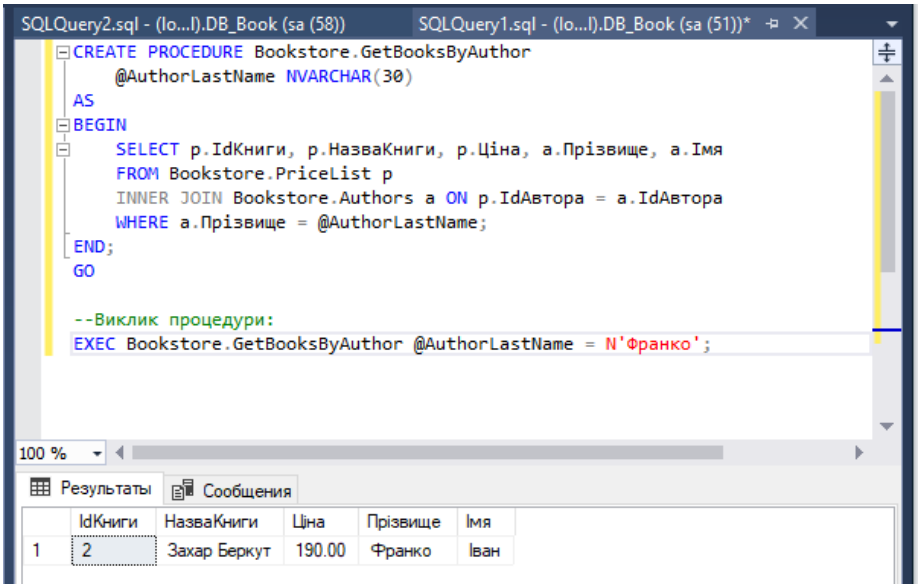
Основна логіка процедури описана в блоці *BEGIN ... END*. Усередині розміщено оператор *SELECT*, який повертає набір полів: ідентифікатор книги (*IdКниги*), назву книги (*НазваКниги*), ціну (*Ціна*), а також прізвище і ім'я автора (*Прізвище, Ім'я*). Для доступу до цих даних здійснюється об'єднання таблиці *PriceList* (що містить інформацію про книги) та таблиці *Authors* (що містить дані про авторів). Оператор *INNER JOIN* встановлює зв'язок між таблицями через ключове поле *IdАвтора*.

Умова *WHERE a.Прізвище = @AuthorLastName* забезпечує вибір тільки тих рядків, де прізвище автора збігається зі значенням параметра, переданим користувачем. Завдяки цьому одна й та сама процедура може виконувати пошук для будь-якого автора.

Для виклику процедури використовується команда *EXEC* (або *EXECUTE*). Наприклад:

```
--Виклик процедури:
    EXEC Bookstore.GetBooksByAuthor @AuthorLastName = N'Франко';
```

У цьому випадку в параметр *@AuthorLastName* передається значення *N'Франко'*. Як результат, процедура поверне список усіх книг автора Франка із зазначенням їх ідентифікаторів, назв і цін. Якщо такого автора немає в базі, буде виведено порожній результат, що також є допустимим.



У даному прикладі продемонстровано базовий принцип використання вхідних параметрів у збережених процедурах, що дозволяє робити процедури більш універсальними, адаптуючи їх під різні запити користувачів. Крім того, параметри сприяють підвищенню безпеки, адже вони зменшують ризик виконання шкідливого коду, характерного для SQL-ін’єкцій, якщо значення параметрів задається правильно.

Процедура з вихідними параметрами для отримання підсумкової або агрегованої інформації.

У попередньому пункті ми розглядали роботу зі збереженими процедурами, які приймають вхідні параметри. Однак у *SQL Server* параметри можуть бути і вихідними. Це означає, що процедура після виконання не лише формує набір результатів через оператор *SELECT*, а й може повертати окремі значення у вигляді змінних. Такий підхід зручний у випадках, коли необхідно обчислити й передати певний підсумковий або агрегований показник.

Вихідні параметри оголошуються під час створення процедури з ключовим словом *OUTPUT*. Усередині процедури їм присвоюються значення за допомогою інструкції *SELECT* або *SET*. Під час виклику процедура може повернути ці значення у змінні, які створює користувач на стороні запиту.

У базі даних «Букіністичний магазин» корисним прикладом є визначення мінімальної та максимальної ціни книг у таблиці *PriceList*. Це завдання демонструє використання агрегатних функцій *MIN* та *MAX*. Для реалізації

створимо процедуру з двома вихідними параметрами: один повертатиме мінімальну ціну, а інший – максимальну.

Процедура *Bookstore.GetPriceRange* визначає найнижчу та найвищу ціну серед усіх книг у таблиці *PriceList*. Вихідні параметри *@MinPrice* і *@MaxPrice* оголошені в заголовку процедури й заповнюються всередині за допомогою агрегатних функцій. Під час виклику процедури користувач створює власні змінні (*@Min*, *@Max*), куди збережені значення автоматично передаються завдяки ключовому слову *OUTPUT*.

Результат виконання можна переглянути через оператор *PRINT* або використати у подальших обчисленнях. У даному випадку буде виведено повідомлення з мінімальною та максимальною ціною книги. Якщо таблиця *PriceList* порожня, обидва параметри матимуть значення *NULL*.

Лістинг 3. Створення процедури з вихідними параметрами для визначення цінового діапазону книг

```
CREATE PROCEDURE Bookstore.GetPriceRange
    @MinPrice DECIMAL(10,2) OUTPUT,
    @MaxPrice DECIMAL(10,2) OUTPUT
AS
BEGIN
    SELECT
        @MinPrice = MIN(Ціна),
        @MaxPrice = MAX(Ціна)
    FROM Bookstore.PriceList;
END;
GO
```

У цьому прикладі після створення процедури *Bookstore.GetPriceRange* необхідно її викликати таким чином, щоб отримані значення мінімальної та максимальної ціни книг можна було зберегти у змінних для подальшого використання. Для цього у *SQL Server* застосовуються локальні змінні, які оголошуються інструкцією *DECLARE*.

Змінні *@Min* та *@Max* визначені як числа з десятковою точністю (2 знаки після коми), що відповідає формату збереження цін у таблиці *PriceList*. Вони поки що не мають значення і будуть заповнені під час виконання процедури. Далі виконується виклик збереженої процедури: викликається процедура *Bookstore.GetPriceRange*, яка має два вихідні параметри: *@MinPrice* і *@MaxPrice*. Їх значення, обчислені всередині процедури, передаються у змінні *@Min* та *@Max*. Важливою є наявність ключового слова *OUTPUT*, яке вказує, що даний параметр є вихідним і має повернути результат у зовнішню змінну. Після виконання цієї інструкції у змінних *@Min* та *@Max*

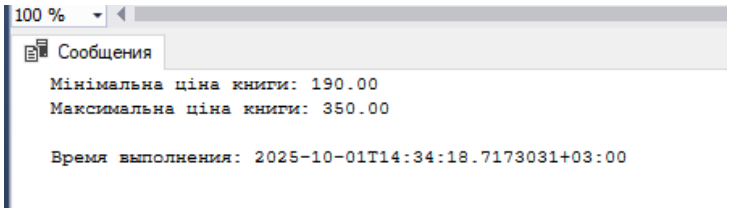
будуть збережені відповідно мінімальна та максимальна ціна книги з таблиці *PriceList*.

Щоб переконатися у правильності виконання процедури, можна вивести значення змінних на екран за допомогою інструкцій *PRINT*: оператор *CAST* використовується для перетворення числових значень у символьний формат, оскільки команда *PRINT* працює тільки з рядковими даними.

Виклик процедури з використанням змінних:

```
DECLARE @Min DECIMAL(10,2), @Max DECIMAL(10,2);
EXEC Bookstore.GetPriceRange @MinPrice = @Min OUTPUT,
                             @MaxPrice = @Max OUTPUT;

PRINT 'Мінімальна ціна книги: ' + CAST(@Min AS NVARCHAR(20));
PRINT 'Максимальна ціна книги: ' + CAST(@Max AS NVARCHAR(20));
```

У результаті користувач побачить текстові повідомлення на кшталт:



Наведений приклад показує, як за допомогою вихідних параметрів можна отримувати агреговану інформацію з бази даних і зберігати її у змінних для подальшої обробки.

Розробка процедури, яка виконує кілька взаємопов'язаних операцій із використанням транзакції

У реальних інформаційних системах дуже часто виникає ситуація, коли потрібно виконати не одну операцію над даними, а цілу групу взаємопов'язаних дій. При цьому важливо, щоб усі дії були виконані успішно як єдине ціле, або ж у випадку помилки всі зміни скасовувалися. Для досягнення такої узгодженості застосовується механізм транзакцій.

Транзакція дозволяє об'єднати кілька *SQL*-операторів в один логічний блок. У *SQL Server* транзакція починається ключовим словом *BEGIN TRANSACTION*, успішно завершується інструкцією *COMMIT TRANSACTION* або ж у разі помилки скасовується командою *ROLLBACK TRANSACTION*. Це гарантує атомарність операцій: або всі зміни зберігаються, або жодна з них не набуває чинності.

У нашій базі «Букіністичний магазин» прикладом такої ситуації є додавання нового продажу. Продаж відображається у двох таблицях:

- у таблиці *Sales* зберігається інформація про чек (номер чеку, клієнт, дата замовлення, дата сплати);
- у таблиці *SalesItems* зберігається перелік книг, які входять до чеку, з вказанням їх кількості.

Для правильного відображення продажу необхідно внести записи в обидві таблиці одночасно. Якщо одна з операцій не виконується, то й інша має бути скасована, щоб уникнути порушення цілісності даних.

Для цього створюємо збереженої процедури *Bookstore.AddSale*, яка реалізує внесення нового продажу в базу даних «Букіністичний магазин». Особливістю цієї процедури є те, що вона виконує одразу дві взаємопов'язані операції: додавання нового запису у таблицю *Sales*, що зберігає загальні відомості про чек, та вставку рядка у таблицю *SalesItems*, де міститься перелік книг, придбаних у цьому чеку. Для забезпечення узгодженості цих операцій використовується механізм транзакцій.

На початку процедури оголошуються чотири параметри: *@IdЧеку*, *@IdКлієнта*, *@IdКниги*, *@Кількість* та *@ДатаСплати* типу *DATE*, який зроблено необов'язковим шляхом присвоєння значення за замовчуванням *NULL*. Це дозволяє записувати продажі як з відомою датою оплати, так і з відкладеною оплатою (коли дата сплати буде заповнена пізніше). Вони задаються користувачем при виклику процедури і дозволяють гнучко формувати новий запис. Наприклад, параметр *@IdЧеку* визначає унікальний номер чеку, *@IdКлієнта* ідентифікує покупця, *@IdКниги* вказує на конкретну книгу, а *@Кількість* – кількість замовленних примірників тощо.

Основна логіка процедури вписана в конструкцію *BEGIN TRY ... END TRY* та *BEGIN CATCH ... END CATCH*. Це стандартний механізм *SQL Server* для обробки виняткових ситуацій. У блоці *TRY* виконується транзакція: вона починається інструкцією *BEGIN TRANSACTION*, після чого виконуються два оператори *INSERT*.

Для забезпечення узгодженості даних і цілісності операцій використано транзакцію, тобто вставка заголовка чеку до таблиці *Sales* і вставка рядка в *SalesItems* виконуються в межах одного логічного блоку, що гарантує атомарність.

Перед початком вставок у блоці *TRY* присвоюється локальна змінна *@OrderDate*, якій присвоюється поточна дата (отримана через *GETDATE*() і приведена до типу *DATE*). Використання єдиного значення *@OrderDate* для обох операцій гарантує узгодженість дат у записах, навіть якщо виконання включає кілька операторів, і запобігає випадковим розбіжностям у межах однієї транзакції.

Перед виконанням вставки введено базову бізнес-валидацію: якщо користувач передав значення *@ДатаСплати*, то воно перевіряється на предмет того, що дата сплати не передує даті замовлення: умова *IF @ДатаСплати IS NOT NULL AND @ДатаСплати < @OrderDate* викликає помилку (*RAISERROR*) у випадку невідповідності. Це просте правило допомагає зберегти логічну послідовність подій (спочатку оформлення замовлення, потім оплата) і запобігає збереженню недостовірних записів.

У разі виникнення будь-якої помилки під час виконання інструкцій *SQL* механізм *TRY...CATCH* перехоплює виняток. Блок *CATCH* перевіряє стан транзакції за допомогою *XACT_STATE*() і, якщо транзакція ще активна або частково готова до відкату, виконує *ROLLBACK TRANSACTION*, щоб повернути базу даних у попередній стан, і цілісність даних не порушується. Після відкату процедура виводить на інформаційний канал текст помилки (*ERROR_MESSAGE*()) і код помилки (*ERROR_NUMBER*()), що полегшує діагностику. Такий підхід є типовим для промислових рішень: він поєднує безпеку зручний зворотний зв'язок для користувача або адміністратора.

Лістинг 4. Створення процедури для додавання інформації про продаж книг з використанням транзакції

```
CREATE PROCEDURE Bookstore.AddSale
    @IdЧеку INT,
    @IdКлієнта INT,
    @IdКниги INT,
    @Кількість INT,
    @ДатаСплати DATE = NULL    -- необов'язковий параметр: дата сплати,
заповнюється користувачем
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        BEGIN TRANSACTION;

        -- Фіксуємо дату замовлення одним значенням для всіх вставок у межах
транзакції
        DECLARE @OrderDate DATE = CONVERT(DATE, GETDATE());

        -- Просте бізнес-правило: дата сплати не може передувати даті
замовлення (якщо задана)
        IF @ДатаСплати IS NOT NULL AND @ДатаСплати < @OrderDate
        BEGIN
            RAISERROR('Дата сплати не може бути ранішою за дату замовлення.',
16, 1);
```

```
        END

        -- Додавання запису в таблицю Sales (включно з датою сплати, якщо
вона задана)
        INSERT INTO Bookstore.Sales (IdЧеку, IdКлієнта, ДатаЗамовлення,
ДатаСплати)
        VALUES (@IdЧеку, @IdКлієнта, @OrderDate, @ДатаСплати);

        -- Додавання запису у вміст продажу
        INSERT INTO Bookstore.SalesItems (IdЧеку, IdКниги, Кількість)
        VALUES (@IdЧеку, @IdКниги, @Кількість);

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        -- Скасовуємо транзакцію тільки якщо вона ще активна
        IF XACT_STATE() <> 0
        BEGIN
            ROLLBACK TRANSACTION;
        END

        -- Повертаємо детальну інформацію про помилку для діагностики
        DECLARE @ErrMsg NVARCHAR(4000) = ERROR_MESSAGE();
        DECLARE @ErrNum INT = ERROR_NUMBER();
        PRINT 'Помилка: транзакцію скасовано. Код помилки: ' + CAST(@ErrNum
AS NVARCHAR(10));
        PRINT 'Текст помилки: ' + ISNULL(@ErrMsg, N'(без опису)');
        RETURN;
    END CATCH
END;
GO
```

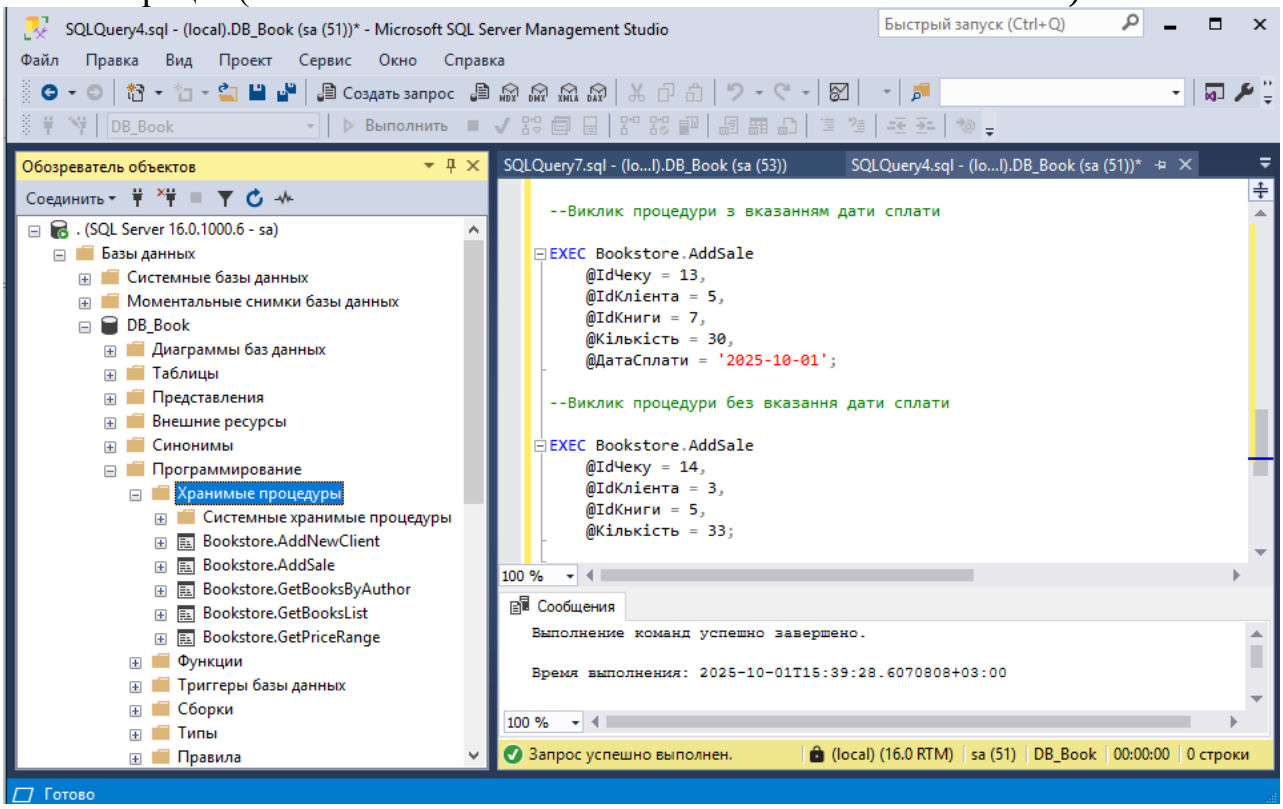
Для виклику показано два сценарії: з передачею дати сплати та без неї. У першому випадку значення *@ДатаСплати* зберігається в полі *ДатаСплати* таблиці *Sales*; у другому – поле буде *NULL*, що вказує на відсутність інформації про оплату на момент реєстрації замовлення.

Виклик процедури:

```
--Виклик процедури з вказанням дати сплати
EXEC Bookstore.AddSale
    @IdЧеку = 13,
    @IdКлієнта = 5,
    @IdКниги = 7,
    @Кількість = 30,
    @ДатаСплати = '2025-10-01';

--Виклик процедури без вказання дати сплати
```

```
EXEC Bookstore.AddSale
    @IdЧеку = 14,
    @IdКлієнта = 5,
    @IdКниги = 3,
    @Кількість = 33;
```

У результаті користувач отримає повідомлення про успішне виконання обох операцій (*INSERT INTO Sales* та *INSERT INTO SalesItems*).



Використання оператора RETURN для повернення результату виконання процедури

Оператор *RETURN* у *SQL Server* дозволяє повернути ціле число після виконання збереженої процедури. Це число зазвичай використовують для індикації результату виконання процедури:

- 0 – успішне виконання процедури;
- 1 – помилка або невиконання умов;
- інші значення – додаткові коди станів (наприклад, кількість рядків, які не задовольняють умову).

На відміну від вихідних параметрів *OUTPUT*, *RETURN* повертає тільки одне числове значення, а не набір даних.

Припустимо, ми хочемо створити процедуру, яка перевіряє наявність книги у прайс-листі і повертає код стану. Збережена процедура *Bookstore.CheckBookExists* призначена для перевірки наявності книги в прайс-листі за її ідентифікатором. Вона приймає один вхідний параметр *@IdКниги*, який визначає конкретну книгу, яку потрібно перевірити. Це дозволяє користувачу або іншій процедурі динамічно передавати значення ідентифікатора без необхідності змінювати сам *SQL*-код.

У тілі процедури спочатку виконується перевірка наявності запису в таблиці *PriceList* за допомогою конструкції *IF EXISTS*. Підзапит *SELECT 1 FROM Bookstore.PriceList WHERE IdКниги = @IdКниги* шукає хоча б один рядок із заданим *IdКниги*. Якщо такий запис існує, умова *IF EXISTS* повертає істину, і процедура переходить до виконання першого блоку команд, при цьому користувачу виводиться повідомлення «Книга знайдена» за допомогою команди *PRINT*. Після цього оператор *RETURN 0* завершує виконання процедури і повертає код 0, який традиційно означає успішне виконання операції. Це дозволяє зовнішнім додаткам або скриптам, що викликають процедуру, швидко визначити, що перевірка пройшла успішно.

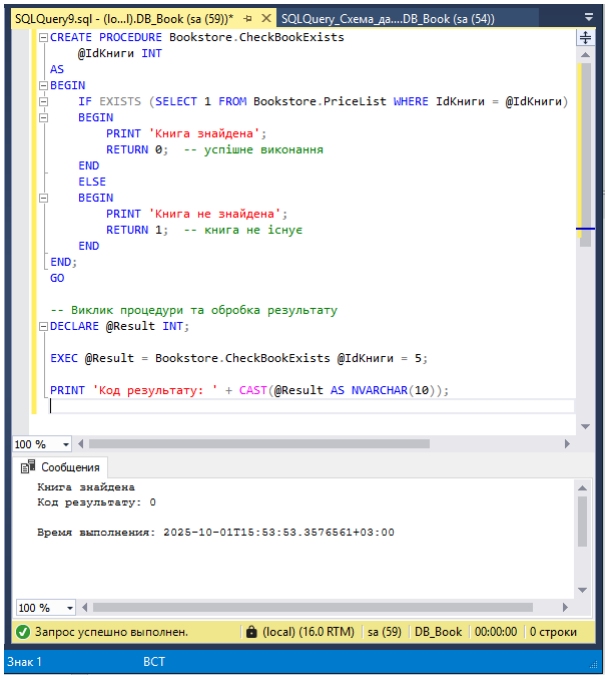
У випадку, якщо книга не знайдена, виконується блок *ELSE*. Процедура виводить повідомлення «Книга не знайдена» і завершує роботу з кодом *RETURN 1*, що сигналізує про відсутність запису.

Лістинг 5. Створення процедури, яка перевіряє наявність книг

```
CREATE PROCEDURE Bookstore.CheckBookExists
    @IdКниги INT
AS
BEGIN
    IF EXISTS (SELECT 1 FROM Bookstore.PriceList WHERE IdКниги = @IdКниги)
    BEGIN
        PRINT 'Книга знайдена';
        RETURN 0;  -- успішне виконання
    END
    ELSE
    BEGIN
        PRINT 'Книга не знайдена';
        RETURN 1;  -- книга не існує
    END
END;
     GO
```

Виклик процедури та обробка результату

```
DECLARE @Result INT;

EXEC @Result =
Bookstore.CheckBookExists @IdКниги =
5;

PRINT 'Код результату: ' + CAST(@Result
AS NVARCHAR(10));
```



Цей механізм корисний для контролю виконання процедур у складних сценаріях, коли потрібно визначати подальші дії в залежності від стану виконання.

Такий підхід дозволяє ефективно обробляти помилки або нестандартні ситуації, не зупиняючи роботу всієї системи, а лише повертаючи числовий код стану.

Інший приклад: перевірка можливості додавання продажу, тобто можна використати *RETURN* для перевірки, чи існує клієнт перед додаванням запису у таблицю *Sales,* а якщо існує чи вже має історію покупок, чм – ні.

Збережена процедура *Bookstore.AddSaleClirnts* призначена для додавання нового запису про продаж у таблицю *Sales*, але перед додаванням виконується перевірка наявності клієнта з заданим ідентифікатором *@IdКлієнта*. Процедура приймає два вхідні параметри: *@IdЧеку*, який задає унікальний номер чеку продажу, та *@IdКлієнта*, що визначає клієнта, для якого здійснюється продаж Мета процедури полягає у тому, щоб гарантувати, що новий запис про продаж буде доданий лише для реального клієнта, а також надати додаткову інформацію про стан його історії покупок.

Процедура *Bookstore.AddSale* починає свою роботу з перевірки, чи існує клієнт із заданим ідентифікатором *@IdКлієнта* у таблиці *Clients*. Для цього використовується конструкція :

IF NOT EXISTS (SELECT 1 FROM Bookstore.Clients
WHERE IdКлієнта = @IdКлієнта)

Цей блок дозволяє виявити, що клієнта взагалі немає у базі. Якщо умова виконується, тобто клієнт не знайдений, процедура одразу виводить повідомлення «Клієнт не знайдений» і завершує свою роботу з кодом *RETURN 1*. Таким чином, новий запис у таблиці *Sales* не додається, і система

сигналізує зовнішньому виклику, що операція не виконана через відсутність клієнта.

Якщо клієнт існує, процедура переходить до наступної перевірки, яка реалізована через *IF ... ELSE*. Тут перевіряється, чи робив клієнт хоча б одну покупку раніше за допомогою

IF NOT EXISTS (SELECT 1 FROM Bookstore.Sales
WHERE IdКлієнта = @IdКлієнта)

Якщо умова виконується, тобто клієнт ще не має жодних продажів, процедура виводить повідомлення «Клієнт існує, але ще не робив жодної покупки. Створюється перший продаж». Це дозволяє системі фіксувати перший продаж клієнта як особливу подію.

Якщо ж клієнт вже мав попередні покупки, спрацьовує блок *ELSE*. У цьому випадку процедура виводить повідомлення «Клієнт знайдений. Додається новий продаж». Таким чином, *IF ... ELSE* дозволяє відрізнити перший продаж клієнта від повторних, забезпечуючи більш детальну інформацію про стан даних для користувача або адміністратора.

Після всіх перевірок процедура виконує вставку нового запису в таблицю *Sales* за допомогою команди

INSERT INTO
Bookstore.Sales (IdЧеку, IdКлієнта, ДатаЗамовлення, ДатаСплати)
VALUES (@IdЧеку, @IdКлієнта, GETDATE(), @ДатаСплати)

Після успішного додавання запису процедура повертає код *RETURN 0*, що сигналізує про успішне виконання операції. Таким чином, комбінація *IF NOT EXISTS* і *IF ... ELSE* забезпечує перевірку наявності клієнта, аналіз його історії покупок та контроль правильності внесення нового продажу, а оператор *RETURN* дозволяє зовнішньому коду отримати інформацію про результат виконання процедури.

Лістинг 6. Створення процедури, яка перевіряє наявність клієнта та втводить інформацію про історію його покурок

```
CREATE PROCEDURE Bookstore.AddSaleClirnts
    @IdЧеку INT,
    @IdКлієнта INT,
    @ДатаСплати DATE
AS
BEGIN
    -- Перевірка існування клієнта
    IF NOT EXISTS (SELECT 1 FROM Bookstore.Clients WHERE IdКлієнта =
@IdКлієнта)
    BEGIN
        PRINT 'Клієнт не знайдений';
        RETURN 1;  -- код помилки: клієнта не існує
    END
```

```
    -- Перевірка історії покупок
    IF NOT EXISTS (SELECT 1 FROM Bookstore.Sales WHERE IdКлієнта =
@IdКлієнта)
    BEGIN
        PRINT 'Клієнт існує, але ще не робив жодної покупки. Створюється
перший продаж.';
    END
    ELSE
    BEGIN
        PRINT 'Клієнт знайдений. Додається новий продаж.';
    END

    -- Додавання нового запису в Sales
    INSERT INTO Bookstore.Sales (IdЧеку, IdКлієнта, ДатаЗамовлення,
ДатаСплати)
    VALUES (@IdЧеку, @IdКлієнта, GETDATE(), @ДатаСплати);

    RETURN 0;  -- код успіху
END;
GO
```

Приклад виклику процедури

```
DECLARE @Result INT;

-- Сценарій 1: клієнт не існує
EXEC @Result = Bookstore.AddSaleClirnts
    @IdЧеку = 15,
    @IdКлієнта = 999,  -- неіснуючий клієнт
    @ДатаСплати = '2025-10-01';
PRINT 'Код виконання процедури: ' + CAST(@Result AS NVARCHAR(10));
-- Очікувано: "Клієнт не знайдений", RETURN = 1

-- Сценарій 2: клієнт існує, але ще не робив покупок
EXEC @Result = Bookstore.AddSaleClirnts
    @IdЧеку = 16,
    @IdКлієнта = 7,   -- існуючий клієнт без попередніх продажів
    @ДатаСплати = '2025-10-01';
PRINT 'Код виконання процедури: ' + CAST(@Result AS NVARCHAR(10));
-- Очікувано: "Клієнт існує, але ще не робив жодної покупки. Створюється
перший продаж.", RETURN = 0

-- Сценарій 3: клієнт існує і вже робив покупки
EXEC @Result = Bookstore.AddSaleClirnts
    @IdЧеку = 17,
    @IdКлієнта = 7,   -- той самий клієнт, тепер він має покупки
    @ДатаСплати = '2025-10-02';
PRINT 'Код виконання процедури: ' + CAST(@Result AS NVARCHAR(10));
     -- Очікувано: "Клієнт знайдений. Додається новий продаж.", RETURN = 0
```

У результаті виконання процедури користувач користувач бачить:

Сценарій 1:

Клієнт з *IdКлієнта = 999* відсутній у таблиці.

Повідомлення: *"Клієнт не знайдений".*

Код *RETURN: 1.*

Новий запис не додається.

Сценарій 2:

Клієнт існує, але ще не має жодного продажу.

Повідомлення: *"Клієнт існує, але ще не робив жодної покупки. Створюється перший продаж."*

Код *RETURN: 0.*

Створюється перший запис у таблиці *Sales*.

Сценарій 3:

Клієнт існує і вже має хоча б один продаж.

Повідомлення: "*Клієнт знайдений. Додається новий продаж."*

Код *RETURN: 0*.

Додається новий запис у таблицю *Sales*.

Створена процедура демонструє поєднання перевірки даних перед додаванням, додавання нового запису та використання оператора *RETURN* для повернення коду стану. Такий підхід забезпечує надійність і цілісність бази даних, дозволяє обробляти помилки на рівні процедур і спрощує інтеграцію у більші бізнес-процеси або системи обліку продажів.

ТЕМА 9. ЗОВНІШНЄ ОБ'ЄДНАННЯ ТАБЛИЦЬ У РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ

ПЛАН:

9.1. Призначення та логічна сутність зовнішнього об'єднання таблиць.
9.2. Види зовнішнього об'єднання: LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN.
9.3. Відмінності зовнішнього об'єднання від внутрішнього (INNER JOIN).
9.4. Особливості використання зовнішніх об'єднань під час аналізу неповних або асиметричних даних.

9.1. Призначення та логічна сутність зовнішнього об'єднання таблиць

У процесі побудови реляційних баз даних часто виникає необхідність поєднувати дані з різних таблиць, які пов'язані між собою спільними атрибутами. Така потреба зумовлена тим, що в добре спроєктованій базі даних інформація нормалізується, тобто розподіляється між кількома таблицями для уникнення дублювання даних та забезпечення їхньої цілісності. Для об'єднання таких таблиць використовується оператор *JOIN*, який формує єдиний набір даних на основі логічної умови зв'язку між ними.

Внутрішнє об'єднання *INNER JOIN* повертає лише ті рядки, які мають відповідність в обох таблицях. Однак на практиці часто виникають ситуації, коли потрібно відобразити всі записи з однієї таблиці, навіть якщо для деяких із них відсутні відповідники в іншій. У такому випадку застосовується зовнішнє об'єднання *OUTER JOIN*, яке забезпечує розширену вибірку даних. На відміну від внутрішнього об'єднання, зовнішнє дозволяє зберегти всі записи з головної таблиці (основного набору даних), додаючи до них відповідні рядки з пов'язаної таблиці, якщо вони існують. За відсутності відповідності значення атрибутів з підлеглої таблиці заповнюються *NULL*-значеннями.

Зовнішнє об'єднання виконує важливу аналітичну функцію, адже дозволяє оцінити повноту даних, виявити відсутні зв'язки між об'єктами, а також створити запити, які відображають повний контекст взаємодії між таблицями. Такий підхід особливо актуальний для задач аудиту, контролю якості даних, а також під час розроблення звітів, де важливо виявити записи

без зв’язків. Наприклад, при аналізі замовлень у торговельній системі часто потрібно отримати перелік усіх клієнтів, включно з тими, хто ще не зробив жодної покупки. Аналогічно, у бібліотечній базі даних може бути необхідним показати всі книги, навіть якщо жодна з них не була взята читачами.

Крім того, зовнішні об’єднання мають аналітичне значення під час дослідження асиметричних даних, коли обсяги інформації у зв’язаних таблицях суттєво різняться. В таких випадках застосування лише внутрішнього об’єднання може спотворити статистичну картину, оскільки з результату буде виключено всі записи без відповідників. Зовнішнє об’єднання дозволяє побудувати більш репрезентативний запит, який відображає як пов’язані, так і непов’язані дані, забезпечуючи повноту інформаційної вибірки.

У середовищі *MS SQL Management Studio* при створенні запитів за допомогою конструктора користувач має можливість у будь-який момент змінити тип запиту з внутрішнього *INNER JOIN*, який встановлюється за замовчуванням, на зовнішній *OUTER JOIN*. Для цього необхідно скористатися контекстним меню і обрати команду *Design Query in Editor* (*Запит у конструкторі редактора*).

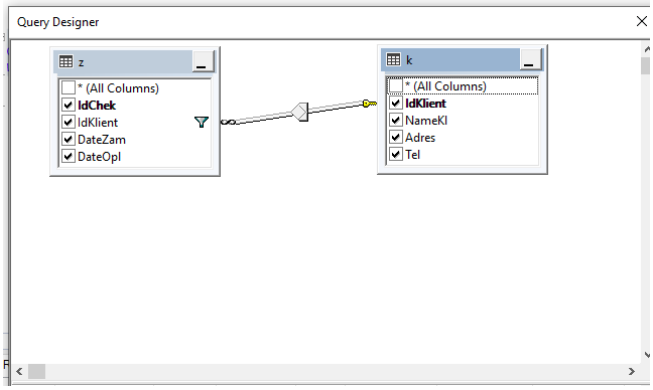


Рис. 9.1. Приклад правого зовнішнього об’єднання у вікні *DesingQwery*

Така функціональність дозволяє гнучко налаштовувати тип з’єднання таблиць відповідно до потреб конкретного запиту.

9.2. Типи зовнішнього об'єднання

Зовнішнє об'єднання *OUTER JOIN* є розширенням внутрішнього об'єднання й застосовується тоді, коли необхідно отримати повну сукупність даних з однієї або обох таблиць, включно з тими записами, які не мають відповідників у суміжній таблиці. Якщо внутрішнє об'єднання *INNER JOIN* формує результат лише для записів, що мають збіг у полях зв'язку, то зовнішнє об'єднання дозволяє зберегти неповні дані, підставляючи значення *NULL* у ті стовпці, де відсутні відповідності.

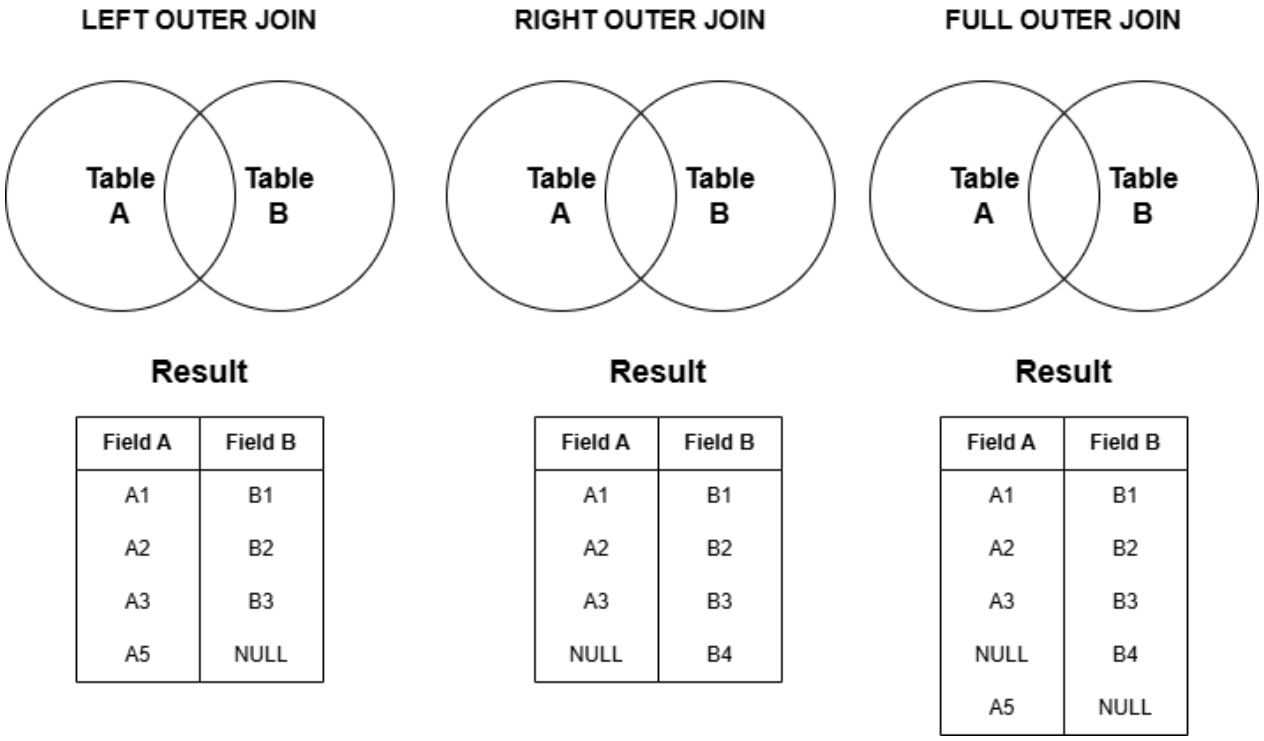


Рис. 9.2. Типи зовнішніх об'єднань

Такий механізм забезпечує можливість аналітичного перегляду інформації, яка ще не має логічних зв'язків або вимагає перевірки на повноту.

Синтаксис зовнішнього об'єднання

```
SELECT <поля>
FROM Таблиця1
   [LEFT | RIGHT | FULL] OUTER JOIN Таблиця2
   ON Таблиця1.Ключ = Таблиця2.Ключ
```

Ключове слово *OUTER* є необов'язковим, однак його вказівка підвищує читаність коду і чітко вказує на тип об'єднання. У *SQL Server* реалізовано три основні види зовнішнього об'єднання: *LEFT OUTER JOIN, RIGHT OUTER JOIN* та *FULL OUTER JOIN*.

✍ Ліве зовнішнє об'єднання *LEFT OUTER JOIN*

Ліве зовнішнє об'єднання *LEFT OUTER JOIN* дозволяє поєднати всі рядки з лівої таблиці з відповідними рядками правої таблиці. Якщо у правій таблиці немає відповідного рядка, поля цієї таблиці у результаті будуть заповнені значенням *NULL*, що забезпечує збереження повного набору даних з головної (лівої) таблиці.

Такий тип об'єднання особливо корисний у випадках, коли важливо відобразити всі записи основної таблиці, незалежно від наявності даних у зв'язаній таблиці. Наприклад, можна формувати звіти про всіх клієнтів магазину, навіть якщо вони ще не робили замовлень.

Ліве об'єднання також застосовується для аналітики та перевірки відсутніх даних. Використовуючи *LEFT OUTER JOIN*, можна швидко визначити, які елементи з лівої таблиці не мають зв'язку з правою таблицею, що допомагає у контролі цілісності даних.

У контексті реляційних баз даних важливо розуміти логіку роботи *LEFT OUTER JOIN*. Об'єднання здійснюється за заданою умовою (наприклад, рівність ключів), а рядки, які не мають співпадінь, не відкидаються, а доповнюються *NULL*. Ліве зовнішнє об'єднання також полегшує інтеграцію даних з різних таблиць. Наприклад, у випадку, коли одна таблиця містить повний список елементів (авторів, клієнтів, товарів), а інша – часткові дані (замовлення, продажі), *LEFT JOIN* гарантує, що всі записи з основної таблиці будуть представлені у результаті.

ПРИКЛАД 9.1. Необхідно вивести інформацію про всіх авторів із назвами їхніх книг і цінами. Для авторів без книг поля *НазваКниги* та *Ціна* будуть містити *NULL*:

Лістинг 9.1. Використання лівого зовнішнього об'єднання

```
SELECT a.Прізвище, a.Ім'я, p.НазваКниги, p.Ціна
FROM Bookstore.Authors a
LEFT OUTER JOIN Bookstore.PriceList p
ON a.IdАвтора = p.IdАвтора;
```

✍ Праве зовнішнє об’єднання *RIGHT OUTER JOIN*

Праве зовнішнє об’єднання *RIGHT OUTER JOIN* працює аналогічно лівому, але фокус робиться на правій таблиці. Усі рядки правої таблиці включаються в результат, а рядки з лівої таблиці доповнюються лише тоді, коли вони відповідають умові з’єднання. Якщо відповідності немає, поля лівої таблиці будуть *NULL*.

RIGHT OUTER JOIN зазвичай використовується, коли головна увага у запиті повинна бути приділена правій таблиці. Наприклад, якщо потрібно відобразити всі видавництва, включно з тими, що ще не видали жодної книги, *RIGHT JOIN* дозволяє зробити це без втрати інформації. Даний тип об’єднання корисний для звітності та контролю даних, коли потрібно зберегти повний набір правої таблиці. Він дозволяє ідентифікувати відсутні зв’язки між таблицями та оцінювати неповні дані.

ПРИКЛАД 9.2. Необхідно вивести інформацію про всі видавництва із назвами виданих книг і цінами. Для видавництв без книг поля *НазваКниги* та *Ціна* будуть містити *NULL*:

Лістинг 9.2. Використання правого зовнішнього об’єднання

```
SELECT p.НазваВидавництва, pr.НазваКниги, pr.Ціна
FROM Bookstore.PriceList pr
RIGHT OUTER JOIN Bookstore.Publishers p
ON pr.IdВидавництва = p.IdВидавництва;
```

RIGHT JOIN часто застосовується в поєднанні з *LEFT JOIN* для побудови складних аналітичних запитів. Наприклад, можна поєднати таблиці продажів і клієнтів таким чином, щоб бачити всі замовлення і водночас всі клієнтські записи, навіть якщо деякі замовлення ще не створені.

У реляційних базах важливо пам’ятати, що *RIGHT OUTER JOIN* є симетричною альтернативою *LEFT JOIN*. Рядки, які не мають співпадінь, не відкидаються, а доповнюються *NULL* у лівій таблиці, що гарантує повноту даних у результаті.

✍ Повне зовнішнє об’єднання *FULL OUTER JOIN*

Повне зовнішнє об’єднання *FULL OUTER JOIN* поєднує результати лівого та правого зовнішньіх об’єднань. На відміну від *INNER JOIN*, який повертає лише ті записи, що мають відповідність у обох таблицях, *FULL OUTER JOIN* повертає усі записи з обох таблиць, навіть якщо для деяких з них не існує відповідних пар у пов’язаній таблиці. Таким чином, цей тип об’єднання дозволяє отримати повну множину даних, де відсутні відповідності позначаються значеннями *NULL*.

FULL OUTER JOIN зазвичай застосовується в аналітичних завданнях, коли необхідно об’єднати всі наявні дані без втрати інформації, навіть якщо частина записів не має відповідних зв’язків. Такий підхід є корисним під час аудиту бази даних, пошуку неузгоджених записів, перевірки цілісності даних або формування повних звітів, які охоплюють як пов’язані, так і ізольовані об’єкти. Наприклад, при інтеграції даних із різних систем можна побачити, які сутності відсутні в одній з них.

СКБД під час виконання *FULL OUTER JOIN* спочатку формує результат лівого об’єднання *LEFT OUTER JOIN*, потім правого *RIGHT OUTER JOIN*, після чого об’єднує їх у єдину вибірку, уникаючи дублювання рядків. Якщо певний запис з лівої таблиці не має відповідного запису у правій, то поля правої таблиці заповнюються значеннями NULL. Аналогічно, якщо запис із правої таблиці не має пари у лівій, поля лівої таблиці також отримують *NULL*. Таким чином, жоден запис не втрачається під час об’єднання.

Основна перевага *FULL OUTER JOIN* полягає у можливості отримати повну картину взаємозв’язків між двома таблицями, включно з усіма записами, що не мають відповідності. Це робить його незамінним при діагностиці структури бази даних і забезпеченні цілісності інформації. Однак варто враховувати, що повне зовнішнє об’єднання є більш ресурсоємною операцією, оскільки потребує обробки повних множин обох таблиць. Тому його слід застосовувати переважно для аналітичних або звітних запитів, а не для операцій із великими обсягами транзакційних даних.

Не всі системи керування базами даних підтримують *FULL OUTER JOIN* у повному обсязі або реалізують його однаково. У деяких випадках доводиться використовувати комбінацію *LEFT JOIN* і *RIGHT JOIN* із подальшим об’єднанням результатів через *UNION*. Також важливо пам’ятати, що через наявність значень *NULL* у результаті такого запиту можуть виникати труднощі під час подальшого фільтрування чи агрегування даних, тому часто застосовуються функція *IS NULL*() для заміни відсутніх значень.

ПРИКЛАД 9.3. Необхідно вивести інформацію про всі видавництва із назвами виданих книг і цінами. Для видавництв, які ще не мають опублікованих книг, поля *НазваКниги* та *Ціна* повинні містити значення *NULL*. Також у результат повинні бути включені книги, для яких інформація про видавництво ще не внесена в базу:

Лістинг 9.3. Використання повного зовнішнього об'єднання

```
SELECT
   P.IdВидавництва,
   P.НазваВидавництва,
   B.IdКниги,
   B.НазваКниги,
   B.Ціна
FROM Bookstore.Publishers P
FULL OUTER JOIN Bookstore.PriceList B
   ON P.IdВидавництва = B.IdВидавництва;
```

У цьому запиті використовується оператор *FULL OUTER JOIN*, який поєднує дані з таблиць *Видавництва* та *Книги* за полем *IdВидавництва*. Результуюча таблиця містить усі видавництва з відповідними назвами книг і цінами, якщо такі є. Якщо видавництво ще не випустило жодної книги, то в рядку, який відповідає цьому видавництву, стовпці *НазваКниги* та *Ціна* отримають значення *NULL*.

Крім того, у вибірку потрапляють і ті книги, що не мають прив'язки до жодного видавництва (наприклад, щойно додані записи або ті, у яких не заповнене поле *IdВидавництва*). Для таких рядків поля, що належать таблиці Видавництва, будуть заповнені *NULL*.

9.4. Особливості використання зовнішніх об'єднань під час аналізу неповних або асиметричних даних

У процесі аналітичної обробки інформації в реляційних базах даних досить часто виникають ситуації, коли дані, що підлягають дослідженню, є неповними, частково узгодженими або асиметричними. Це означає, що між окремими сутностями (таблицями) відсутні відповідності у ключових полях, або ж обсяг даних у взаємопов'язаних таблицях є нерівномірним.

Наприклад, у сисхемі *Bookstore* може існувати видавництво, у якого ще немає зареєстрованих книг; автор, який не має жодної публікації; або клієнт,

який не здійснив жодного замовлення. У традиційних запитах з внутрішнім об'єднанням *INNER JOIN* такі записи не потрапляють до результату, що може призвести до втрати важливої інформації під час звітності чи аналітичного контролю.

Саме для таких ситуацій застосовуються зовнішні об'єднання *OUTER JOIN* як інструмент комплексного аналізу, що дозволяє зберегти у результатах усі сутності з однієї або обох таблиць, незалежно від наявності відповідників. Використання зовнішніх об'єднань особливо актуальне при:

- аудиті повноти даних;
- виявленні неузгоджених записів («orphan records»);
- формуванні звітів, у яких важливо відображати всі елементи категорії, навіть якщо частина з них не має пов'язаних фактів;
- статистичних розрахунках, де відсутність даних є суттєвим показником.

Таким чином, зовнішнє об'єднання є аналітичним механізмом виявлення інформаційних прогалин, що забезпечує цілісне представлення бази знань та підвищує достовірність аналітичних висновків.

Нижче наведено основні особливості використання зовнішніх об'єднань у контексті аналізу неповних або асиметричних даних, доповнені прикладами на основі бази даних *Bookstore*.

✍ Виявлення «сирих» або орфанних записів (anti-join патерн)

Найпоширеніше завдання під час аналізу неповних даних – це пошук записів, які не мають жодного відповідника в пов'язаній таблиці. Такі записи ще називають *«orphan records»* (сирі дані). Вони можуть свідчити про логічні помилки, порушення референтної цілісності або невиконання бізнес-процесу.

Для цього застосовується *LEFT OUTER JOIN* у поєднанні з умовою *WHERE ... IS NULL*, що реалізує так званий *anti-join*.

ПРИКЛАД 9.4. Визначити клієнтів, які не здійснили жодного замовлення:

Лістинг 9.4. Виявлення «сирих» записів

```
SELECT c.IdКлієнта, c.НайменуванняКлієнта
FROM Bookstore.Clients AS c
LEFT JOIN Bookstore.Sales AS s
  ON c.IdКлієнта = s.IdКлієнта
WHERE s.IdЧеку IS NULL;
```

Запит формує повний перелік клієнтів із таблиці *Clients* і додає до нього дані з таблиці *Sales*, якщо такі існують. Клієнти, для яких не знайдено жодного

продажу, залишаються у результаті з *NULL* у полі *IdЧеку*. Це дозволяє визначити неактивних клієнтів або виявити можливі збої у процесі реєстрації замовлень.

✍ Збереження повноти під час агрегування (включення нульових значень)

При побудові аналітичних звітів важливо відображати не лише наявні результати, а й випадки, де активність відсутня. Зовнішні об'єднання дозволяють сформувати агреговані дані з урахуванням «нульових» показників, що є суттєвим для управлінської аналітики.

ПРИКЛАД 9.5. Підрахувати кількість проданих примірників для кожної книги, включно з тими, що не мали продажів:

Лістинг 9.5. Виявлення та включення нульових значень

```
SELECT
  pL.IdКниги,
  pL.НазваКниги,
  COALESCE(SUM(si.Кількість), 0) AS
ЗагальнаКількістьПродажів
 FROM Bookstore.PriceList AS pL
 LEFT JOIN Bookstore.SalesItems AS si
  ON pL.IdКниги = si.IdКниги
 GROUP BY pL.IdКниги, pL.НазваКниги;
```

Операція *LEFT JOIN* гарантує, що у результат потраплять усі книги, навіть ті, які не були продані. Функція *COALESCE* замінює *NULL* на 0, тим самим забезпечуючи коректне відображення кількості проданих примірників.

Такий підхід використовується при побудові управлінських дашбордів, де відсутність продажів також є показником, що впливає на процес прийняття рішень.

✍ Виявлення неузгодженостей між таблицями (Reconciliation)

Повне зовнішнє об'єднання *FULL OUTER JOIN* дозволяє одночасно виявити записи, відсутні в обох напрямах зв'язку. Це особливо корисно для

аудиту цілісності даних та перевірки синхронізації між таблицями після імпорту або інтеграції інформаційних джерел.

ПРИКЛАД 9.6. Знайти книги без автора та авторів без жодної книги:

Лістинг 9.6. Виявлення неузгодженостей між таблицями

```
SELECT
  a.IdАвтора, a.Прізвище, pL.IdКниги, pL.НазваКниги
FROM Bookstore.Authors AS a
FULL OUTER JOIN Bookstore.PriceList AS pL
  ON a.IdАвтора = pL.IdАвтора
WHERE a.IdАвтора IS NULL OR pL.IdКниги IS NULL;
```

У результаті відображаються або автори без книг, або книги без зазначеного автора. Такі ситуації є типовими прикладами порушення інформаційної узгодженості, які слід усунути під час підтримки бази даних у належному стані.

Зовнішні об’єднання дозволяють формувати повну аналітичну картину навіть тоді, коли взаємозв’язки між сутностями неповні або нерівномірні. Вони забезпечують:

- збереження інформації про всі елементи категорії (навіть без зв’язків);
- можливість виявлення пропущених або некоректних записів;
- формування коректних статистичних звітів і контрольних вибірок;
- підтримку аудиту бізнес-процесів (виявлення неактивних клієнтів, невиконаних замовлень тощо).

Зовнішні об’єднання забезпечують повноту та достовірність інформаційно-аналітичних запитів, особливо у тих предметних областях, де дані зберігаються в різнорідних або частково заповнених структурах.

Питання для самоперевірки

1. *У чому полягає основне призначення оператора JOIN у реляційних базах даних?*
2. *Чим відрізняється внутрішнє об'єднання INNER JOIN від зовнішнього OUTER JOIN з точки зору формування вибірки даних?*
3. *Які завдання вирішує зовнішнє об'єднання під час побудови аналітичних запитів?*
4. *У яких випадках доцільно використовувати зовнішнє об'єднання замість внутрішнього?*
5. *Яке значення мають NULL-поля у результатах виконання зовнішнього об'єднання?*
6. *Назвіть основні типи зовнішніх об'єднань, реалізовані у SQL Server, та коротко охарактеризуйте кожен із них.*
7. *У чому полягає логічна різниця між операторами LEFT OUTER JOIN та RIGHT OUTER JOIN?*
8. *Яку роль відіграє оператор FULL OUTER JOIN у забезпеченні повноти даних і які його особливості реалізації?*
9. *Чому ключове слово OUTER у синтаксисі об'єднання вважається необов'язковим, але бажаним для читаності коду?*
10. *Наведіть приклад ситуації, у якій використання LEFT JOIN допоможе виявити «сирі» або орфанні записи.*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 9

на тему: «Зовнішнє об'єднання таблиць»

МЕТА: *набуття практичних навичок використання зовнішніх об'єднань таблиць у середовиші MS SQL Server Management Studio із застосуванням операторів LEFT OUTER JOIN, RIGHT OUTER JOIN та FULL OUTER JOIN для отримання повних вибірок даних із пов'язаних таблиць, аналізу неповних або асиметричних даних.*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. Побудувати запити за допомогою оператора *LEFT OUTER JOIN*.
2. Створити вибірку із використанням оператора *RIGHT OUTER JOIN*.
3. Сформувати повний набір даних із двох таблиць за допомогою *FULL OUTER JOIN*.
4. Поєднати зовнішні об'єднання з умовами фільтрації та сортування для формування аналітичних звітів.
5. Використати зовнішні об'єднання для підрахунку агрегованих показників із врахуванням відсутніх даних.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

У процесі аналітичної роботи з базою даних часто виникає потреба відобразити всі записи головної сутності, навіть якщо для деяких із них відсутні відповідні дані у зв'язаній таблиці. У таких випадках використовується лівий зовнішній об'єднувальний оператор *LEFT OUTER JOIN*, який гарантує включення до результату всіх рядків із лівої таблиці незалежно від наявності зв'язків у правій.

На практиці це дозволяє отримати, наприклад, повний перелік авторів літературних творів, навіть якщо частина з них ще не має опублікованих книг у таблиці цінового переліку. Таке завдання є типовим прикладом перевірки повноти даних і контролю цілісності інформаційної моделі.

Приклад 1. Отримати повний перелік клієнтів магазину разом із даними про їхні замовлення. Для клієнтів, які ще не здійснили жодної покупки, значення атрибутів, що належать таблиці продажів, повинні містити *NULL*.

```
SELECT c.IdКлієнта, c.НайменуванняКлієнта,
s.IdЧеку, s.ДатаЗамовлення
FROM Bookstore.Clients AS c
LEFT OUTER JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта;
```

	IdКлієнта	НайменуванняКлієнта	IdЧеку	ДатаЗамовлення
1	1	ТОВ Книгарня Є	1	2024-01-10
2	1	ТОВ Книгарня Є	7	2024-07-08
3	2	ТОВ Літера	2	2024-02-15
4	2	ТОВ Літера	8	2024-08-14
5	3	ФОП Петренко О.О.	3	2024-03-05
6	3	ФОП Петренко О.О.	9	2024-09-19
7	3	ФОП Петренко О.О.	14	2025-10-01
8	4	ТОВ Книголенд	4	2024-04-20
9	4	ТОВ Книголенд	10	2024-10-03
10	5	ТОВ БукМаркет	5	2024-05-11
11	5	ТОВ БукМаркет	11	2024-11-17
12	5	ТОВ БукМаркет	13	2025-10-01
13	6	ФОП Сидоренко Н.П.	6	2024-06-25
14	6	ФОП Сидоренко Н.П.	12	2024-12-28
15	7	ТОВ "Книгоманія"	16	2025-10-01
16	7	ТОВ "Книгоманія"	17	2025-10-01
17	8	ФОП Іваненко О.П.	NULL	NULL

У даному запиті таблиця *Clients* виступає головною (лівою), а таблиця *Sales* – підлеглою (правою). Оператор *LEFT OUTER JOIN* поєднує обидві таблиці за спільним атрибутом *IdКлієнта*, але, на відміну від внутрішнього об'єднання, не відкидає клієнтів, для яких відсутні записи про продажі.

Якщо клієнт здійснював покупки, то у результаті будуть відображені дані про номер чеку (*IdЧеку*) та дату продажу (*ДатаЗамовлення*). Якщо ж клієнт не має жодного замовлення, відповідні поля приймають значення *NULL*.

У процесі аналізу бази даних часто необхідно зберегти повноту інформації з правої таблиці, навіть якщо для деяких записів не існує відповідників у таблиці, з якою вона поєднується. У таких випадках застосовується праве зовнішнє об'єднання *RIGHT OUTER JOIN*, яке гарантує включення всіх рядків правої таблиці до результату запиту. Якщо для певного запису з правої таблиці не знайдено відповідного запису з лівої, значення полів лівої таблиці замінюються *NULL*.

Приклад 2. Отримати повний перелік видавництв із назвами книг і цінами. Для тих видавництв, у яких ще не зареєстровано жодної книги, поля, що стосуються книг, повинні містити значення NULL.

```
SELECT p.НазваВидавництва, pr.НазваКниги, pr.Ціна
FROM Bookstore.PriceList AS pr
RIGHT OUTER JOIN Bookstore.Publishers AS p
    ON pr.IdВидавництва = p.IdВидавництва;
```

У цьому запиті таблиця *Publishers* є правою, тобто основною, а таблиця *PriceList* – лівою, тобто підлеглою. Завдяки використанню оператора *RIGHT OUTER JOIN* у результат потрапляють усі видавництва, незалежно від того, чи мають вони надруковані книги в таблиці *PriceList*.

Якщо видавництво має видані книги, то відображаються їхні назви та ціни. Якщо ж книги у видавництві відсутні, поля НазваКниги та Ціна набувають значення *NULL*.

	НазваВидавництва	НазваКниги	Ціна
1	Видавництво Старий Лев	Кобзар	250.00
2	Ранок	Захар Беркут	270.00
3	Фоліо	The Old Man and the Sea	345.00
4	Фоліо	1984	322.00
5	А-Ба-Ба-Га-Ла-Ма-Га	Harry Potter and the Philosopher's Stone	350.00
6	Клуб Сімейного Дозвілля	Собор Паризької Богоматері	310.00
7	Кальварія	Місто	220.00
8	Світ Книг	NULL	NULL
9	Глорія Прес	NULL	NULL

Сформувати повний набір даних із двох таблиць за допомогою *FULL OUTER JOIN*

У складних інформаційно-аналітичних запитах часто виникає потреба отримати повну сукупність даних із двох таблиць, включно з тими записами, які не мають відповідних даних у пов'язаній таблиці. У таких випадках використовується повне зовнішнє об'єднання *FULL OUTER JOIN*.

Даний тип об'єднання повертає всі рядки з обох таблиць, які беруть участь в операції, незалежно від наявності збігів за умовою з'єднання. Якщо для певного рядка з однієї таблиці не знайдено пов'язаного запису в іншій, у результаті цей рядок все одно включається до вибірки, а відсутні значення з другої таблиці заповнюються *NULL*. Таким чином, *FULL OUTER JOIN* забезпечує відображення як пов'язаних, так і непов'язаних даних, формуючи повну сукупність інформації з обох таблиць.

Приклад 3. Отримати повний перелік видавництв і книг, що зберігаються в базі даних. У результат повинні потрапити: усі видавництва, включно з тими, які ще не мають надрукованих книг; усі книги, для яких відсутня інформація про видавництво.

```
SELECT p.IdВидавництва, p.НазваВидавництва, pr.IdКниги, pr.НазваКниги,
pr.Ціна
FROM Bookstore.Publishers AS p
FULL OUTER JOIN Bookstore.PriceList AS pr
    ON p.IdВидавництва = pr.IdВидавництва;
```

У цьому запиті об'єднуються таблиці *Publishers* та *PriceList* за спільним атрибутом *IdВидавництва*. На відміну від *INNER JOIN*, який відображає лише пари записів, що мають збіг, оператор *FULL OUTER JOIN* включає усі записи з обох таблиць: якщо видавництво не має книг у стовпцях *НазваКниги* та *Ціна* стоїть *NULL*; якщо книга не має вказаного видавництва – у полях таблиці Publishers буде *NULL*.

	IdВидавництва	НазваВидавництва	IdКниги	НазваКниги	Ціна
1	1	Видавництво Старий Лев	1	Кобзар	250.00
2	2	Ранок	2	Захар Беркут	270.00
3	3	Фоліо	4	The Old Man and the Sea	345.00
4	3	Фоліо	6	1984	322.00
5	4	А-Ба-Ба-Га-Ла-Ма-Га	3	Harry Potter and the Philosopher's Stone	350.00
6	5	Клуб Сімейного Дозвілля	7	Собор Паризької Богоматері	310.00
7	6	Кальварія	5	Місто	220.00
8	7	Світ Книг	NULL	NULL	NULL
9	8	Глорія Прес	NULL	NULL	NULL
10	NULL	NULL	8	Борислав сміється	210.00

Поєднання зовнішніх об’єднань із фільтрацією, та сортуванням

У реальних аналітичних запитах зовнішні об’єднання майже завжди використовуються в поєднанні з іншими механізмами обробки даних, такими як фільтрація *WHERE, HAVING*, сортування *ORDER BY* та агрегування *GROUP BY*. Таке поєднання дозволяє перетворювати сирі дані у змістовну аналітичну інформацію, зберігаючи при цьому записи, для яких відсутні пов’язані елементи.

На відміну від внутрішніх об’єднань, де результат формується лише за наявності збігів, зовнішні об’єднання дають змогу зберегти повноту вибірки і одночасно виконати аналітичні операції над повним набором даних, що особливо важливо при складанні звітів, контролі якості інформації або оцінюванні ефективності бізнес-процесів.

Фільтрація результатів у запитах із зовнішнім об’єднанням потребує особливої уваги, оскільки неправильне розміщення умов відбору може змінити логіку об’єднання. Якщо умову фільтрації застосувати у секції *WHERE* після виконання об’єднання, це може призвести до виключення рядків з *NULL*, фактично перетворюючи *LEFT JOIN* або *RIGHT JOIN* на *INNER JOIN*. Тому, коли потрібно зберегти всі записи з базової таблиці, фільтри, що стосуються пов’язаної таблиці, слід розміщувати в секції *ON*.

Приклад 4. Вивести всі видавництва та відповідні їм книги, які були видані у 2024 році, якщо такі існують.

```
SELECT pub.НазваВидавництва,
    COALESCE(pL.НазваКниги, N'Немає книг у 2024 році') AS НазваКниги,
    pL.РікВидання
FROM Bookstore.Publishers pub
LEFT JOIN Bookstore.PriceList pL
    ON pub.IdВидавництва = pL.IdВидавництва
    AND pL.РікВидання = 2024
ORDER BY pub.НазваВидавництва;
```

	НазваВидавництва	НазваКниги	РікВидання
1	А-Ба-Ба-Га-Ла-Ма-Га	Немає книг у 2024 році	NULL
2	Видавництво Старий Лев	Кобзар 2024	2024
3	Глорія Прес	Немає книг у 2024 році	NULL
4	Кальварія	Немає книг у 2024 році	NULL
5	Клуб Сімейного Дозвілля	Немає книг у 2024 році	NULL
6	Ранок	Немає книг у 2024 році	NULL
7	Світ Книг	Немає книг у 2024 році	NULL
8	Фоліо	Немає книг у 2024 році	NULL

У цьому прикладі демонструється застосування зовнішніх об’єднань (*LEFT JOIN*) із фільтрацією та сортуванням для аналізу неповних даних. Фільтр *pL.РікВидання = 2024* розміщено у предикаті *ON*. *COALESCE* замінює *NULL* на зрозуміле повідомлення *'Немає книг у 2024 році'* тільки там, де реально немає книги 2024 року. *ORDER BY*

pub.НазваВидавництва сортує результат за назвою видавництва для зручності перегляду.

Зовнішні об’єднання для підрахунку агрегованих показників

Агрегування у поєднанні із зовнішніми об’єднаннями дає змогу отримати узагальнені результати з урахуванням відсутніх даних, що корисно при побудові зведених звітів, де важливо показати всі сутності (наприклад, авторів, видавництва, жанри) навіть у тому випадку, коли вони не мають відповідних записів у пов’язаних таблицях.

Приклад 5. Підрахувати кількість проданих книг кожного видавництва, включно з тими, що не мали продажів

```
SELECT pub.НазваВидавництва,
    COALESCE(SUM(si.Кількість), 0) AS КількістьПроданихПримірників
FROM Bookstore.Publishers pub
LEFT JOIN Bookstore.PriceList pL
    ON pub.IdВидавництва = pL.IdВидавництва
LEFT JOIN Bookstore.SalesItems si
    ON pL.IdКниги = si.IdКниги
GROUP BY pub.НазваВидавництва
ORDER BY pub.НазваВидавництва;
```

Результаты | Сообщения

	НазваВидавництва	КількістьПроданихПримірників
1	А-Ба-Ба-Га-Ла-Ма-Га	55
2	Видавництво Старий Лев	40
3	Глорія Прес	0
4	Кальварія	88
5	Клуб Сімейного Дозвілля	80
6	Ранок	55
7	Світ Книг	0
8	Фоліо	105

Запит гарантує, що всі видавництва потраплять до результату незалежно від наявності продажів. Функція *COALESCE* замінює відсутні значення (*NULL*) на нуль, що дає змогу уникнути викривлення аналітики. Таким чином формується повний аналітичний звіт, де значення 0 вказує не на помилку, а на факт відсутності реалізації книг.

Іноді після виконання групування необхідно відфільтрувати результати за певними умовами. Для цього використовується секція *HAVING*, яка працює вже з агрегованими значеннями.

Приклад 6. Вивести перелік усіх авторів і назв їхніх книг, відсортований за країною автора (у тому числі для авторів без країни).

```
SELECT a.Прізвище, a.Імя, c.НайменуванняКраїни, pL.НазваКниги
FROM Bookstore.Authors a
LEFT JOIN Bookstore.Countries c
```

```
    ON a.IdКраїни = c.IdКраїни
LEFT JOIN Bookstore.PriceList pL
    ON a.IdАвтора = pL.IdАвтора
ORDER BY a.Прізвище;
```

Перший *LEFT JOIN* із таблицею *Countries* забезпечує приєднання інформації про країну автора. Завдяки використанню *LEFT JOIN* усі автори включаються у результат, навіть якщо для них відсутнє значення країни (*IdКраїни = NULL*). Другий *LEFT JOIN* із таблицею *PriceList* дозволяє підключити дані про книги авторів. Якщо автор не має записів у прайс-листі, у полі *НазваКниги* відображатиметься значення *NULL*. Такий підхід дозволяє сформувати повний перелік авторів без втрати інформації про тих, хто ще не має виданих книг. Сортування результату здійснюється за прізвищем автора за допомогою *ORDER BY a.Прізвище*. Для забезпечення коректного відображення авторів із *NULL* у прізвищі можна використовувати функції *ISNULL* або *COALESCE*.

Розглянемо приклад комплексного запиту, який поєднує всі розглянуті механізми.

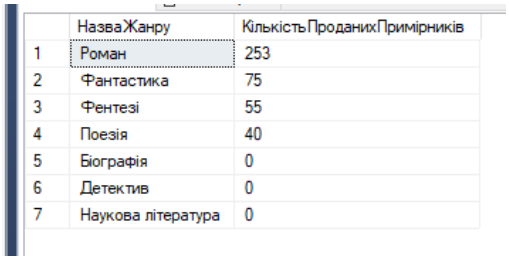
Приклад 7. Сформувати аналітичний звіт про кількість проданих книг кожного жанру у 2024 році, включно з жанрами, для яких продажів не зафіксовано.

```
SELECT g.НазваЖанру,
    COALESCE(SUM(si.Кількість), 0) AS КількістьПроданихПримірників
FROM Bookstore.Genres g
LEFT JOIN Bookstore.PriceList pL
    ON g.IdЖанру = pL.IdЖанру
LEFT JOIN Bookstore.SalesItems si
    ON pL.IdКниги = si.IdКниги
LEFT JOIN Bookstore.Sales s
    ON si.IdЧеку = s.IdЧеку
    AND YEAR(s.ДатаЗамовлення) = 2024
GROUP BY g.НазваЖанру
ORDER BY КількістьПроданихПримірників DESC;
```

У запиті використовується лівий зовнішній *JOIN*, щоб забезпечити збереження всіх жанрів, навіть якщо для них не було продажів у вказаний період. Фільтрація за роком *YEAR(s.ДатаЗамовлення) = 2024* розміщена у секції *ON*, що дозволяє включити у результат усі жанри без продажів і не виключати їх із підсумкового звіту.

Функція *COALESCE* використовується для коректного відображення нульових продажів, замінюючи значення *NULL* на нуль.

Ключова частина аналітики забезпечується через *GROUP BY*, яка підсумовує продажі за жанрами, а *ORDER BY* впорядковує результати за обсягом реалізації.



	НазваЖанру	КількістьПроданихПримірників
1	Роман	253
2	Фантастика	75
3	Фентезі	55
4	Поезія	40
5	Біографія	0
6	Детектив	0
7	Наукова література	0

У результаті формується повний аналітичний звіт: жанри, які продавалися найбільше, розташовуються на початку, а ті, що не мали продажів у 2024 році – наприкінці списку.

ТЕМА 10. РЕЛЯЦІЙНА АЛГЕБРА

ПЛАН:

10.1. Основні поняття, сутність і призначення реляційної алгебри.
10.2. Основні операції реляційної алгебри (за Коддом) та принцип їх виконання.
10.3. Реалізація операцій реляційної алгебри засобами SQL.
10.4. Додаткові операції реляційної алгебри (за Дейтом).
10.5. Узагальнення та практичне значення реляційної алгебри.

10.1. Основні поняття, сутність і призначення реляційної алгебри

Реляційна алгебра є формальною математичною системою для опрацювання інформації у реляційних базах даних. Вона ґрунтується на теорії множин, логіці предикатів та булевій алгебрі, що дозволяє строго визначати операції над даними та способи їх комбінування. Основна ідея реляційної алгебри полягає у представленні інформації у вигляді відношень, які математично визначаються як множини кортежів з фіксованою структурою атрибутів, де кожен атрибут має визначений тип даних і відображає конкретну властивість об’єкта предметної області. Кожне відношення складається з кортежів, які описують окремі об’єкти предметної області. Атрибути кортежів відображають властивості цих об’єктів і мають визначені типи даних. Така структура забезпечує строгість і однозначність представлення інформації, що є ключовою вимогою для коректного зберігання та обробки даних у реляційних базах.

Сутність реляційної алгебри полягає у визначенні операцій, які дозволяють отримувати нові відношення на основі наявних. До базових операцій належать об’єднання, перетин, різниця, проєкція та селекція. Кожна з цих операцій формально визначена і завжди повертає відношення як результат, що забезпечує замкненість системи. Замкненість реляційної алгебри є важливим принципом, оскільки дозволяє комбінувати операції між собою, формуючи складні запити без втрати структурної узгодженості даних. Це означає, що результат будь-якої операції можна використовувати як вхідні дані для наступної операції, що значно підвищує гнучкість і потужність запитів.

Призначення реляційної алгебри полягає у формальному описі процесів

маніпулювання даними. Вона забезпечує методологічну основу для розробки запитів у системах керування базами даних і слугує теоретичною моделлю, на якій будуються практичні мови запитів, такі як SQL. Крім того, реляційна алгебра дозволяє оптимізувати запити, оскільки формальні правила виконання операцій дають змогу планувати ефективну послідовність обробки даних. Наприклад, операції селекції та проєкції можна виконувати на ранніх етапах обробки, щоб зменшити обсяг проміжних результатів і підвищити продуктивність.

Важливою особливістю реляційної алгебри є її незалежність від фізичної організації даних. Вона оперує логічними структурами (відношеннями), що дозволяє користувачам і розробникам зосередитися на змісті та структурі інформації, не турбуючись про фізичне розміщення даних на носіях. Реляційна алгебра також забезпечує коректність і точність запитів, оскільки формальні визначення операцій виключають неоднозначності, що є особливо важливим у великих інформаційних системах, де будь-яка помилка у визначенні запиту може призвести до некоректного отримання або втрати даних.

Реляційна алгебра використовується у практичних завданнях аналізу даних, проектування баз даних, формування звітності та автоматизації інформаційних процесів. Знання принципів реляційної алгебри є необхідною базою для будь-якого фахівця з баз даних, аналітика чи розробника програмного забезпечення. Отже, реляційна алгебра поєднує в собі математичну строгость, логічну послідовність операцій та практичну застосовність у побудові ефективних систем обробки даних.

10.2. Основні операції реляційної алгебри (за Коддом) та принцип їх виконання

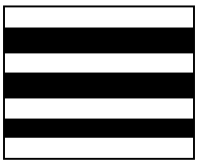
Основні операції реляційної алгебри були визначені засновником реляційної моделі даних Едгаром Френком Коддом у 1970 році. Він запропонував набір формально визначених операцій, які дозволяють здійснювати пошук, відбір, об’єднання та перетворення даних, що зберігаються у відношеннях. Кожна з цих операцій має чітку математичну інтерпретацію, ґрунтується на законах теорії множин і завжди повертає нове відношення, що забезпечує замкненість системи.

До базових операцій реляційної алгебри за Коддом належать: селекція (*SELECT*), проєкція (*PROJECT*), об’єднання (*UNION*), перетин (*INTERSECTION*), різниця (*DIFFERENCE*), декартовий добуток (*CARTESIAN PRODUCT*) та перейменування (*RENAME*). Ці операції утворюють мінімальний функціональний набір, за допомогою якого можна

сформулювати будь-який запит до реляційної бази даних. Вони визначають, як з наявних відношень побудувати нові, що відповідають певним умовам або логічним запитам користувача.

Операція селекції (σ) використовується для вибору підмножини кортежів, які задовольняють певну умову, що відповідає логічному фільтруванню рядків за заданим предикатом.

Селекція/вибірка



Формально селекція визначається як

σ_{умова}(R),

де R – вихідне відношення, а умова – булевий вираз, що задає критерії вибору. У результаті формується нове відношення, яке містить лише ті кортежі, що задовольняють цю умову.

Операція проєкції (π) виконує вибір певних атрибутів із відношення і дозволяє відобразити лише необхідні стовпці таблиці, усуваючи дублікати результатів.

Проекція

Формально операція задається як

π_{список_атрибутів}(R),

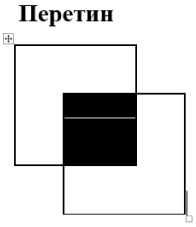
де R – вихідне відношення. Проєкція зменшує розмірність відношення, але зберігає його сутність, даючи змогу виділяти потрібні характеристики даних без зміни логічної природи таблиці.

Об'єднання (∪) бінарна операція, яка поєднує два відношення з однаковими наборами атрибутів. Результатом є відношення, що містить усі кортежі, які належать хоча б одному з об'єднуваних відношень.

Об'єднання

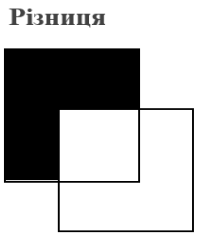
Операція відповідає об'єднанню множин і є корисною для злиття даних, що мають спільну структуру, наприклад, при формуванні загальних списків клієнтів, товарів чи транзакцій.

Операція перетину (∩) виділяє спільні кортежі двох відношень і дозволяє отримати множину даних, що одночасно належать двом таблицям з однаковою схемою.



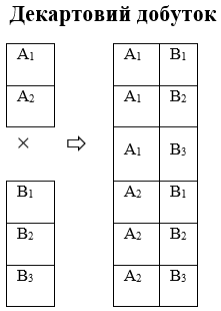
Використовується для визначення спільних записів, наприклад, клієнтів, які є у двох різних базах даних. Дана операція є протилежною за змістом до операції різниці й, подібно до неї, базується на властивостях множинного перетину.

Операція різниці (−) визначає множину кортежів, які належать одному відношенню, але відсутні в іншому. Її результатом є відношення, що відображає відмінності між двома множинами даних.



У практичних завданнях операція різниці використовується для визначення записів, які є у першій таблиці, але відсутні у другій, наприклад, для пошуку клієнтів, які зробили замовлення минулого місяця, але не цього.

Декартовий добуток (×) результатом операції є відношення, у якому кожен кортеж однієї таблиці поєднується з кожним кортежем іншої. Хоча така операція часто породжує велику кількість комбінацій, вона є основою для реалізації зв'язків типу *JOIN* у SQL.



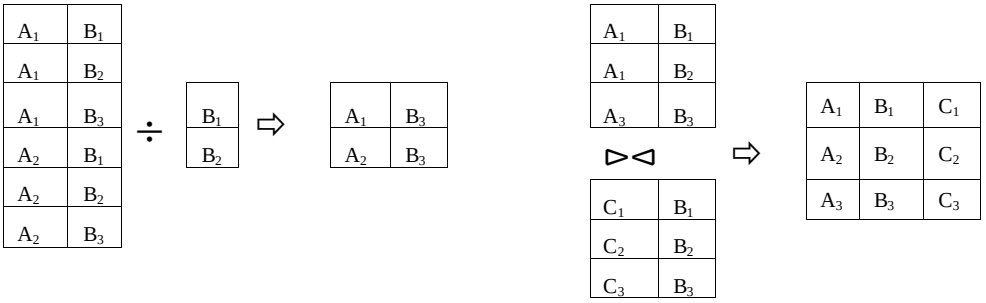
Декартовий добуток є однією з найважливіших операцій реляційної алгебри, оскільки саме на його основі будуються складні запити з об'єднанням таблиць.

Операція ділення (÷) використовується для вибірки таких кортежів з однієї таблиці (відношення), які пов'язані з *усіма* кортежами іншої таблиці. Вона є однією з найскладніших за своєю семантикою, але має важливе практичне значення при роботі з базами даних, зокрема для формулювання запитів типу *«знайти всіх авторів, які написали всі книги певного жанру»*.

Якщо маємо два відношення $R(A, B)$ і $S(B)$, то результат операції ділення $R \div S$ є новим відношенням, яке містить лише ті значення атрибутів A із

таблиці *R*, для яких існують записи у *R*, що відповідають усім значенням атрибутів *B* з таблиці *S*. Іншими словами, для кожного елемента з *A* перевіряється, чи співіснують у *R* всі пари з множиною *B*, визначеною у *S*.

Ділення **З'єднання**



Операція з'єднання (⋈) об'єднує інформацію з двох або більше таблиць на основі спільних атрибутів. Семантично ця операція є узагальненням добутку декартового типу (*CARTESIAN PRODUCT*), до якого додається умова відбору кортежів, що задовольняють певне співвідношення між атрибутами.

Формально, якщо маємо два відношення *R(A, B)* і *S(B,C),* то результат операції з'єднання $R \bowtie S$ – це підмножина декартового добутку $R \times S$, у якій значення атрибутів *B* у обох таблицях збігаються. У результаті формується нове відношення з атрибутами *A, B, C.*

Реляційна алгебра розрізняє кілька видів з'єднань:

- θ-з'єднання (*θ-JOIN*) – коли об'єднання виконується за умовою порівняння (наприклад, *R.A > S.B*);
- еквіз'єднання (*EQUI-JOIN*) – частковий випадок θ-з'єднання, де використовується рівність (=);
- природне з'єднання (*NATURAL JOIN*) – автоматично поєднує таблиці за атрибутами з однаковими назвами;
- зовнішні з'єднання (*OUTER JOIN*) – дозволяють зберігати записи з однієї або обох таблиць, навіть якщо для них немає відповідності (*LEFT, RIGHT, FULL OUTER JOIN*).

10.3. Реалізація операцій реляційної алгебри засобами SQL

Реляційна алгебра є теоретичною моделлю, на основі якої побудована

мова структурованих запитів *SQL*. Вона визначає логічний апарат для виконання операцій над відношеннями, тоді як *SQL* реалізує ці операції практично у середовищі систем укрування базами даних. Таким чином, між реляційною алгеброю та *SQL* існує глибокий зв'язок: кожна операція алгебри має прямий або опосередкований аналог у синтаксисі *SQL*.

✍ Операція селекції (σ)

Операція селекції у реляційній алгебрі відповідає відбору кортежів за заданим предикатом. У *SQL* вона реалізується через конструкцію *WHERE*.

ПРИКЛАД 10.1. Необхідно вибрати всі книги художньої літератури з таблиці *Bookstore.PriceList*:

Лістинг 10.1. Використання операції селекції

```
SELECT НазваКниги, Ціна
FROM Bookstore.PriceList
WHERE IdЖанру IN (SELECT IdЖанру
FROM Bookstore.Genres
WHERE ТипЖанру = N'Художня література');
```

Цей запит реалізує селекцію на основі вкладеного підзапиту, який фільтрує лише ті кортежі, що належать до певного типу жанру. Аналогічно, у реляційній алгебрі така операція позначається як:
σ_{ТипЖанру='Художня література'}(PriceList ⋈ Genres).

✍ Операція проєкції (π)

Операція проєкції у реляційній алгебрі передбачає вибір певних атрибутів відношення, тобто певних атрибутів (стовпців) таблиці. В *SQL* її еквівалентом є вибір стовпців у частині *SELECT*.

ПРИКЛАД 10.2. Отримати лише назви книг та роки їх видання:

Лістинг 10.2. Виконання операції проєкції

```
SELECT НазваКниги, РікВидання
FROM Bookstore.PriceList;
```

Така проєкція зменшує кількість стовпців, зберігаючи лише потрібну інформацію. На рівні реляційної алгебри це відповідає виразу:

π_{НазваКниги, РікВидання}(PriceList).

✍ Операція об’єднання (UNION)

Операція об’єднання реалізує множинне додавання кортежів двох відношень, що мають однакову структуру. У *SQL* реалізується оператором *UNION* (або *UNION ALL*, якщо потрібно зберегти дублікати).

ПРИКЛАД 10.3. Сформувати загальний перелік назв видавництв, які зберігаються у двох різних таблицях (основній і додатковій):

Лістинг 10.3. Реалізація операції об’єднання

```
SELECT НазваВидавництва
FROM Bookstore.Publishers
UNION
SELECT НазваВидавництва
FROM Bookstore.Publishers
WHERE IdВидавництва > 6;
```

У результаті утворюється нове логічне відношення, що містить усі назви видавництв без дублікатів, що є аналогом операції об’єднання множин у реляційній алгебрі.

✍ Операція різниці (−)

Операція різниці повертає ті кортежі, які є в одному відношенні, але відсутні в іншому. Має відповідник у *SQL* оператор *EXCEPT*.

ПРИКЛАД 10.4. Визначити книги, які **не належать** до навчальної літератури:

Лістинг 10.4. Застосування операції різниці

```
SELECT НазваКниги
FROM Bookstore.PriceList
EXCEPT
SELECT НазваКниги
FROM Bookstore.PriceList
```

```
WHERE IdЖанру IN
 (SELECT IdЖанру
FROM Bookstore.Genres
WHERE ТипЖанру = N'Навчальна');
```

Цей запит поверне всі книги, окрім тих, які віднесені до навчальної літератури. У реляційній алгебрі аналогічна операція записується як *π_{НазваКниги}(PriceList) – π_{НазваКниги}(σ_{ТипЖанру = 'Навчальна'}(PriceList ⋈ Genres)).*

✍ Операція перетину (∩)

Операція перетину повертає лише ті кортежі, які одночасно належать двом відношенням, у SQL виражається через *INTERSECT*.

ПРИКЛАД 10.5. Знайти авторів, книги яких видані в українських або британських видавництвах:

Лістинг 10.5. Використання операції перетину

```
SELECT Прізвище, Імя
FROM Bookstore.Authors
WHERE IdКраїни
IN
(SELECT IdКраїни
FROM Bookstore.Countries
WHERE НайменуванняКраїни = N'Україна')
INTERSECT
SELECT Прізвище, Імя
FROM Bookstore.Authors
WHERE IdКраїни
IN
(SELECT IdКраїни
FROM Bookstore.Countries
WHERE НайменуванняКраїни = N'Велика Британія');
```

Такий запит поверне лише тих авторів, які належать до обох множин одночасно, тобто відповідає множинному перетину у реляційній алгебрі.

✍ Операція декартового добутку (×)

Операція декартового добутку утворює всі можливі комбінації записів із двох таблиць, незалежно від того, чи пов'язані вони логічно. Кожен рядок першого відношення поєднується з кожним рядком другого, у результаті чого формується нова таблиця, кількість записів у якій дорівнює добутку кількості рядків у вихідних таблицях.

У мові *SQL* декартовий добуток реалізується за допомогою команди *CROSS JOIN*. Ця операція часто використовується для створення всіх можливих комбінацій елементів або для подальшої побудови більш складних запитів (наприклад, при тестуванні або комбінуванні довідників).

ПРИКЛАД 10.6. Отримати всі можливі комбінації авторів і видавництв — тобто кожен автор поєднується з кожним видавництвом, незалежно від того, чи публікував він у ньому книги:

Лістинг 10.6. Виконання операції декартового добутку

```
SELECT a.Прізвище, a.Ім'я, p.НазваВидавництва
FROM Bookstore.Authors AS a
CROSS JOIN Bookstore.Publishers AS p;
```

Команда *CROSS JOIN* створює таблицю, у якій кожен автор поєднаний із кожним видавництвом. Наприклад, якщо у таблиці *Authors* є 4 записи, а у таблиці Publishers – 5, результат міститиме 20 рядків. Цей результат не обов'язково має практичний сенс, але він є корисним для аналітичних або тестових завдань, коли необхідно розглянути всі можливі варіанти поєднання елементів двох множин.

ПРИКЛАД 10.7. Отримати всі можливі комбінації авторів і лише тих видавництв, у назві яких міститься слово *Press*. Такий запит дозволяє сформувати добуток множини авторів і підмножини видавництв, що відповідають певному критерію відбору:

Лістинг 10.7. Застосування операції декартового добутку з умовою відбору

```
SELECT a.Прізвище, a.Ім'я, p.НазваВидавництва
FROM Bookstore.Authors AS a
CROSS JOIN Bookstore.Publishers AS p
WHERE p.НазваВидавництва LIKE N'%Press%';
```

Декартовий добуток є операцією реляційної алгебри, на основі якої реалізуються більш складні форми об'єднання таблиць зокрема, внутрішнє та зовнішнє з'єднання.

✍ Операція з'єднання (⋈)

Операція з'єднання дозволяє об'єднувати дані з різних таблиць на основі спільного атрибута. У реляційній алгебрі позначається символом ⋈, а в мові SQL реалізується через ключове слово *JOIN*.

ПРИКЛАД 10.8. Вивести назви книг разом із прізвищами їхніх авторів:

Лістинг 10.8. Реалізація операції з'єднання

```
SELECT p.НазваКниги, a.Прізвище
FROM Bookstore.PriceList p
JOIN Bookstore.Authors a
ON p.IdАвтора = a.IdАвтора;
```

У реляційній алгебрі операція з'єднання еквівалентна такому запису:

PriceList ⋈_{IdАвтора} Authors.

✍ Операція ділення (Division, ÷)

Операція ділення у реляційній алгебрі використовується для вибірки таких кортежів з одного відношення, які пов'язані з усіма елементами іншого відношення. Інакше кажучи, ділення дозволяє знайти об'єкти, що задовольняють умову «для всіх», а не лише «для деяких».

У SQL ця операція не має прямого аналога, проте її можна реалізувати за допомогою вкладених запитів (*Subqueries*), групування (*GROUP BY*) та підрахунку кількості збігів (*HAVING COUNT*).

ПРИКЛАД 10.9. Необхідно знайти авторів, книги яких видавалися у всіх видавництвах, присутніх у базі даних.

Для цього скористаємося таблицями *Bookstore.PriceList* (прайс-лист) та *Bookstore.Publishers* (видавництва). Тобто шукаємо таких авторів, для яких у прайс-листі є принаймні по одній книзі, опублікованій у кожному видавництві, зазначеному в таблиці *Publishers*.

Лістинг 10.9. Імітація операції ділення

```
SELECT a.IdАвтора, a.Прізвище, a.Імя
```

```
FROM Bookstore.Authors AS a
WHERE NOT EXISTS (
  SELECT p.IdВидавництва
  FROM Bookstore.Publishers AS p
  WHERE NOT EXISTS (
    SELECT pl.IdКниги
    FROM Bookstore.PriceList AS pl
    WHERE pl.IdАвтора = a.IdАвтора
     AND pl.IdВидавництва = p.IdВидавництва)
);
```

Логіка запиту полягає у наступному: зовнішній запит обирає всіх авторів із бази даних; перший підзапит, використовуючи конструкцію *NOT EXISTS*, перевіряє для кожного автора наявність видавництв, у яких він не має жодної книги, якщо таких видавництв не існує, тобто автор представлений у всіх видавництвах, він потрапляє до результатної множини. Таким чином, у фінальному результаті будуть лише ті автори, чиї книги представлені у всіх видавництвах, що містяться в таблиці *Publishers*.

У реляційній алгебрі операція ділення для наведеного прикладу представлена у такому вигляді:

$$Authors \div Publishers = \{ a \mid \forall p \in Publishers, (a, p) \in PriceList$$

Тобто результатом операції $T \div S$ є множина всіх авторів, для яких у таблиці *PriceList* наявні пари (*IdАвтора, IdВидавництва*) для кожного значення *IdВидавництва*, що входить до множини *Publishers*.

Незважаючи на те, що SQL є декларативною мовою, усі її запити базуються на формальних принципах реляційної алгебри. Кожен запит у процесі виконання перетворюється системою управління базами даних у дерево реляційних операцій, після чого проходить етапи логічної та фізичної оптимізації. Це забезпечує узгодженість між теоретичним описом обробки даних і практичною реалізацією в обчислювальних системах.

10.4. Додаткові операції реляційної алгебри (за Дейтом).

Реляційна алгебра є формальною системою операцій, що дозволяє описувати процеси вибірки, об’єднання та перетворення даних у реляційній моделі. Базовий набір операцій (селекція, проекція, об’єднання, різниця, декартовий добуток) забезпечує виконання фундаментальних запитів, але не охоплює всіх практичних потреб користувачів і розробників баз даних.

З цієї причини К. Дейт запропонував набір додаткових операцій реляційної алгебри, які розширюють можливості маніпуляцій із відношеннями. Такі операції не замінюють базові, а логічно їх доповнюють, забезпечуючи повноцінну підтримку операцій редагування, узагальнення, оновлення та порівняння даних.

Додаткові операції дозволяють працювати не лише з вибірками, а й із модифікацією структур відношень, обчисленням похідних атрибутів, створенням агрегованих показників, оновленням записів і логічним порівнянням результатів. Саме завдяки їм реляційна алгебра стає основою не лише для запитів типу SELECT, але й для повного набору DML-операцій (INSERT, UPDATE, DELETE) у сучасних СКБД.

✍ Перейменування (RENAME)

Операція перейменування (ρ) у реляційній алгебрі дозволяє надати відношенню або його атрибутам нові імена, не змінюючи фактичного змісту даних. Теоретично вона слугує для збереження семантичної узгодженості при виконанні операцій, у яких беруть участь декілька таблиць із подібною структурою. Наприклад, якщо дві таблиці містять поле *Id*, то без перейменування виникла б неоднозначність при їх об'єднанні.

У реляційних системах перейменування є формальним актом зміни заголовка відношення, тобто воно створює новий вигляд таблиці, в якій змінено лише назви атрибутів. У мові *SQL* ця ідея реалізується оператором *AS*, що визначає *псевдонім* (*Alias*) для стовпця або самої таблиці. Завдяки цьому забезпечується зрозуміліша структура запитів, підвищується читабельність і логічна незалежність від фізичної моделі.

З теоретичного погляду, перейменування є нейтральною операцією: вона не впливає на кількість і вміст кортежів, але може змінювати спосіб звертання до них у подальших операціях. Саме тому К. Дейт вважав її ключовою у створенні виразних запитів, що узгоджують рівні представлення даних користувацький і внутрішній.

ПРИКЛАД 10.10. Отримати назву книги, автора та видавництво, перейменувавши атрибути для зручності подальшого використання у звітах:

Лістинг 10.10. Виконання операції перейменування

```
SELECT b.НазваКниги AS [Книга], a.Прізвище AS [Автор],
    p.НазваВидавництва AS [Видавництво]
```

```
FROM Bookstore.PriceList b
JOIN Bookstore.Authors a
ON b.IdАвтора = a.IdАвтора
JOIN Bookstore.Publishers p
ON b.IdВидавництва = p.IdВидавництва;
```

Запит демонструє логічну функцію перейменування, тобто створюється нове «віртуальне» відношення з іншими назвами стовпців, яке можна використовувати як базове у складніших запитах (наприклад, при створенні представлень даних у бізнес-звітах).

✍ Операція розширення (EXTEND)

Операція розширення використовується для створення в реляційному виразі нового атрибута, значення якого обчислюється на основі існуючих атрибутів відношення. Тобто *EXTEND* дозволяє розширити схему відношення додатковими стовпцями, що формуються за допомогою арифметичних, логічних або строкових виразів. Така операція є ключовим засобом побудови нових аналітичних ознак і часто застосовується під час попередньої обробки даних.

У реляційній алгебрі операція розширення представлена у такому вигляді: *EXTEND R ADD (новий_атрибут := вираз)*

де *R* – вихідне відношення, а *вираз* визначає, як формується значення нового атрибута.

ПРИКЛАД 10.11. Обчислити загальну суму реалізованих книг для кожного продажу:

Для кожного запису у таблиці *Sales* створюється новий атрибут *СумаПродажу*, який дорівнює добутку кількості та ціни.

Лістинг 10.11. Застосування операції розширення

```
SELECT IdПродажу, IdКлієнта, IdКниги, Кількість, Ціна,
    (Кількість * Ціна) AS СумаПродажу
FROM Bookstore.Sales;
```

EXTEND часто використовується у поєднанні з операціями проєкції, селекції та групування. Таким чином, ця операція є функціональним аналогом

конструкції *AS* у *SQL*, яка застосовується для створення обчислюваних полів у запиті (*SELECT Ціна*Кількість AS Сума FROM Sales*).

Перевагою *EXTEND* є те, що ця операція не змінює вихідні дані, а лише доповнює їх, створюючи похідні характеристики, що робить її зручною для формування проміжних результатів при складних запитах, коли необхідно додати обчислювані колонки, наприклад, «прибуток», «податок», «знижка», «рейтинг» тощо.

🖉 Операція підведення підсумків (SUMMARIZE)

Операція підведення підсумків використовується для обчислення агрегованих значень за групами або за всією множиною кортежів і дозволяє скоротити великий набір даних до компактнішої форми, де зберігається лише інформація про узагальнені характеристики (наприклад, середні значення, максимуми, мінімуми чи суми).

У реляційній алгебрі операція підведення підсумків представлена у такому вигляді:

SUMMARIZE R ADD (новий_атрибут :=
агрегатна_функція(атрибут)).

Функція може бути *SUM, AVG, COUNT, MAX, MIN* тощо.

ПРИКЛАД 10.12. Обчислити податок на додану вартість у вигляді 20% від суми продажу:

Лістинг 10.12. Використання операції підведення підсумків

```
SELECT IdПродажу, IdКлієнта, IdКниги, Кількість, Ціна,
    (Ціна * Кількість * 0.2) AS ПДВ
FROM Bookstore.Sales;
```

Операція *SUMMARIZE* має важливе аналітичне значення, оскільки дозволяє здійснювати агрегування безпосередньо в межах реляційної моделі, не звертаючись до сторонніх обчислювальних засобів. У системах управління базами даних ця операція відповідає оператору *GROUP BY* у *SQL* у поєднанні з агрегатними функціями. У практичному використанні *SUMMARIZE* допомагає отримати ключові показники діяльності, такі як загальний дохід, середню зарплату працівників, сумарний обсяг замовлень тощо. Її застосовують для побудови аналітичних звітів, статистичних моделей тощо.

✍ Операція множинного підведення підсумків (GROUP SUMMARIZE)

Операція множинного підведення підсумків є узагальненням попередньої. Вона дозволяє обчислювати агрегати не для всієї множини кортежів, а для кожної групи, утвореної за певними атрибутами. Тобто відношення розбивається на підмножини згідно з унікальними значеннями вказаних атрибутів, після чого до кожної підмножини застосовуються агрегатні функції.

У реляційній алгебрі операція множинного підведення підсумків представлена у такому вигляді:

SUMMARIZE R PER (атрибути_групування)
ADD (новий_атрибут := агрегатна_функція(атрибут)).

ПРИКЛАД 10.13. Визначити середню вартість продажу книг для кожного міста, де мешкають клієнти:

Лістинг 10.13. Реалізація операції множинного підведення підсумків

```
SELECT C.Місто,
    ROUND(AVG(S.Ціна * S.Кількість), 2) AS СереднійПродаж
FROM Bookstore.Sales AS S
JOIN Bookstore.Clients AS C
  ON S.IdКлієнта = C.IdКлієнта
GROUP BY C.Місто;
```

У наведеному прикладі демонструється застосування операції множинного підведення підсумків, яка дозволяє групувати результати за певним атрибутом та виконувати агрегатні обчислення в межах кожної групи. У даному випадку обчислюється середня сума продажу для кожного міста, у якому зареєстровані клієнти. Таблиці *Sales* та *Clients* об'єднуються за атрибутом IdКлієнта, що дає змогу пов'язати інформацію про продажі з географічними даними клієнтів. Далі, у межах кожного міста, виконується розрахунок середнього значення добутку полів *Ціна* * *Кількість*, що фактично відображає середню суму транзакції.

Такий підхід є коректним із точки зору реляційної алгебри, оскільки операція агрегування виконується після попереднього з'єднання відношень, а групування здійснюється за класифікаційним атрибутом *Місто*.

Використання функції *AVG*() у *SQL* реалізує операцію середнього арифметичного над сукупністю добутків ціни та кількості проданих книг, що відображає середню вартість продажу у межах певного населеного пункту. У результаті формується нове відношення, у якому кожен рядок відповідає окремому місту, а атрибут *СереднійПродаж* містить значення середньої вартості реалізованої продукції. Таким чином, дана операція дає змогу виконувати багатовимірний аналіз даних у реляційній моделі, що має важливе значення для бізнес-аналітики та побудови звітів.

Множинне підведення підсумків широко застосовується в бізнес-аналітиці, фінансовій звітності та інтелектуальному аналізі даних. Наприклад, можна обчислити сумарний обсяг продажів по кожному регіону, середню оцінку студента по дисциплінах або загальний дохід за кожен місяць. Окрім того, *GROUP SUMMARIZE* може використовуватись у комбінації з *EXTEND*, що дає змогу створювати складні звіти з обчисленими полями, коефіцієнтами та динамічними показниками. Таким чином, множинне підведення підсумків реалізує потужну можливість реляційної алгебри в частині аналітичної обробки даних, наближаючи її до функціоналу *OLAP*-систем.

10.5. Узагальнення та практичне значення реляційної алгебри

Головне призначення реляційної алгебри полягає у тому, щоб задати чіткий, логічно обґрунтований спосіб обробки даних незалежно від конкретної системи керування базами даних (СКБД) чи технічних засобів їх реалізації. На відміну від імперативних підходів, реляційна алгебра надає декларативний інструментарій, що визначає, які дані потрібно отримати, а не як саме їх обчислити. Завдяки цьому вона є теоретичною основою мови SQL та всіх інших реляційно-орієнтованих систем.

Узагальнюючи зміст реляційної алгебри, слід підкреслити, що її базові операції вибірка, проєкція, об’єднання, перетин, різниця, з’єднання, ділення утворюють повний набір інструментів для маніпуляцій із даними у будь-якій реляційній моделі. Вони дозволяють виконувати фільтрацію, комбінування та узагальнення інформації, створюючи нові відношення на основі існуючих. Додаткові операції, запропоновані Крісом Дейтом, такі як перейменування, розширення, множинного підведення підсумків тощо, розширюють можливості аналітичної обробки, наближаючи реляційну алгебру до практичних задач сучасних інформаційних систем.

З практичної точки зору, реляційна алгебра є формальною основою для побудови оптимізаторів запитів у СКБД. Кожен SQL-запит під час обробки в системі транслюється у послідовність алгебраїчних операцій, які потім

оптимізуються для досягнення найкращої продуктивності. Таким чином, розуміння структури реляційних операторів дозволяє фахівцям з баз даних краще усвідомлювати, як саме система виконує запит, які перетворення можливі, і яким чином вони впливають на ефективність роботи з великими обсягами інформації.

Узагальнення реляційної алгебри має також методологічне значення у викладанні дисциплін, пов'язаних із проектуванням баз даних. Розуміння принципів алгебри допомагає не лише створювати складні *SQL*-запити, але й розробляти системи, здатні виконувати автоматичний синтез або аналіз таких запитів.

Значення реляційної алгебри виходить далеко за межі академічного підґрунтя. У прикладних інформаційних системах вона лежить в основі модулів аналітики, систем бізнес-інтелекту (BI), обробки транзакцій та прогнозного аналізу. Алгебраїчний підхід дозволяє уніфікувати доступ до даних, формувати багатоетапні обчислення та забезпечувати відтворюваність результатів. Наприклад, у базі даних «Bookstore» операції вибірки, об'єднання, групування та агрегації застосовуються для побудови звітів про продажі, аналізу динаміки реалізації літературних жанрів або визначення ефективності роботи видавництв.

У підсумку, реляційна алгебра виступає не лише теоретичним інструментом, але й практичною основою сучасних технологій керування даними. Вона забезпечує узгодженість, точність і відтворюваність обчислень, створює передумови для автоматизації процесів аналітики та підтримує концептуальну єдність між логічною моделлю даних і їх фізичним представленням. Таким чином, володіння принципами реляційної алгебри є необхідною складовою професійної підготовки фахівців у галузі комп'ютерних наук, інформаційних систем та аналітики даних.

Питання для самоперевірки

1. *Що таке реляційна алгебра і на яких математичних концепціях вона ґрунтується?*
2. *Як визначаються відношення у реляційній алгебрі, і з чого складаються кортежі?*
3. *Який принцип замкненості реляційної алгебри і чому він важливий?*

4. *Яке практичне призначення реляційної алгебри у роботі з базами даних?*
5. *Назвіть базові операції реляційної алгебри за Коддом.*
6. *Як формально визначається операція селекції (σ) і яка її роль?*
7. *Що таке операція проєкції (π) та який результат вона дає?*
8. *Чим відрізняються операції об'єднання (∪), перетину (∩) та різниці (−)?*
9. *Яка роль декартового добутку (×) у реляційній алгебрі і як він реалізується у SQL?*
10. *Що таке операція ділення (÷) і як її можна реалізувати у SQL?*
11. *Назвіть основні види з'єднань (JOIN) у реляційній алгебрі та їх особливості.*
12. *Які додаткові операції реляційної алгебри запропонував Кріс Дейт і для чого вони потрібні?*
13. *Що робить операція EXTEND і як вона відображається у SQL?*
14. *Яка різниця між операцією SUMMARIZE та GROUP SUMMARIZE?*
15. *Чому розуміння реляційної алгебри важливе для оптимізації SQL-запитів та аналітичної роботи з даними?*

ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ДО ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ 10

на тему: «Операції реляційної алгебри та їх реалізація засобами мови SQL»

МЕТА: *засвоїти принципи виконання базових операцій реляційної алгебри за Коддом (селекція, проєкція, об'єднання, різниця, перетин, декартовий добуток, з'єднання, ділення) та навчитися реалізовувати їх засобами мови структурованих запитів SQL.*

ПЛАН ЛАБОРАТОРНОЇ РОБОТИ

1. Виконати базові операції реляційної алгебри (селекція, проєкція, об'єднання, різниця, перетин, декартовий добуток, з'єднання, ділення).
2. Виконати додаткові операції реляційної алгебри (перейменування, розширення, підведення підсумків, множинне підведення підсумків).
3. Проаналізувати отримані результати та описати особливості кожної операції.

ХІД ВИКОНАННЯ ЛАБОРАТОРНОЇ РОБОТИ

Реляційна алгебра є основою роботи з реляційними базами даних і надає формальні математичні методи для опису операцій над відношеннями. Вона дозволяє чітко визначати, як можна відбирати, об'єднувати, трансформувати та аналізувати дані, зберігаючи при цьому логічну узгодженість та структурну цілісність інформації. Основними елементами реляційної алгебри є відношення (таблиці), кортежі (рядки) та атрибути (стовпці), а також набір операцій, які перетворюють одне відношення на інше.

Базові операції реляційної алгебри, запропоновані Едгаром Ф. Коддом, включають селекцію, проєкцію, об'єднання, перетин, різницю, декартовий добуток, з'єднання та ділення. Виконання цих операцій дозволяє формувати запити до бази даних, які відображають різноманітні логічні умови та вимоги користувача. Крім базових, існує також розширений набір операцій, описаний Крісом Дейтом, що включає перейменування, розширення, підведення підсумків, що значно підвищує аналітичні можливості реляційної моделі.

Мова структурованих запитів *SQL* реалізує всі операції реляційної алгебри у практичному середовищі СКБД. Завдяки цьому фахівці з баз даних можуть безпосередньо застосовувати математичні принципи алгебри для виконання практичних завдань, таких як фільтрація даних, об'єднання таблиць, підрахунок агрегатних показників та аналіз великих обсягів інформації. Розглянемо операції реляційної алгебри на прикладах:

Приклад 1. Селекція (σ). Отримати список клієнтів, чия загальна сума покупок перевищує 2000.

```
SELECT c.IdКлієнта, c.НайменуванняКлієнта,
       SUM(si.Кількість * p.Ціна) AS СумаПокупок
FROM Bookstore.Clients AS c
JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
GROUP BY c.IdКлієнта, c.НайменуванняКлієнта
HAVING SUM(si.Кількість * p.Ціна) > 2000;
```

	IdКлієнта	НайменуванняКлієнта	СумаПокупок
1	1	ТОВ Книгарня Є	17000.00
2	2	ТОВ Літера	18650.00
3	3	ФОП Петренко О.О.	19200.00
4	4	ТОВ Книголенд	19200.00
5	5	ТОВ БукМаркет	25740.00
6	6	ФОП Сидоренко Н.П.	22970.00

У цьому запиті об'єднані таблиці *Clients, Sales, SalesItems* та *PriceList*, щоб порахувати загальну суму покупок кожного клієнта. Вибірка (селекція) відбувається за умовою *SUM(si.Кількість * p.Ціна) > 2000*. Агрегація по клієнту виконується через *GROUP BY*, а *HAVING* дозволяє застосувати умову до агрегованого результату. Таким чином, запит повертає тільки тих клієнтів, чия загальна витрачена сума перевищує 2000.

Приклад 2. Проєкція (вибір певних атрибутів). Отримати список назв книг та їхніх цін.

```
SELECT НазваКниги, Ціна
FROM Bookstore.PriceList;
```

Тут використовуємо операцію проєкція, тобто вибираємо лише окремі атрибути таблиці *PriceList*. У результаті будуть відображені всі книги з цінами, без зайвої інформації про авторів, жанри чи видавництва. Проєкція

дозволяє виділити потрібні колонки з таблиці для подальшого аналізу або звітності.

Приклад 3. Об'єднання (UNION). Отримати єдину множину географічних назв (країн з таблиці *Countries* і міст клієнтів з таблиці *Clients*).

```
SELECT НайменуванняКраїни AS Географія
FROM Bookstore.Countries
UNION
SELECT SUBSTRING(АдресаКлієнта, 1, CHARINDEX(',', АдресаКлієнта)-1) AS
Географія
FROM Bookstore.Clients;
```

	Географія
1	Велика Британія
2	м. Дніпро
3	м. Київ
4	м. Львів
5	м. Одеса
6	м. Харків
7	м. Чернігів
8	Німеччина
9	США
10	Україна
11	Франція

Операція *UNION* об'єднує результати двох або більше запитів у єдину множину, автоматично видаляючи дублікати. У нашому випадку ми отримали колонку *Географія*, яка містить як усі країни з таблиці *Countries*, так і всі міста клієнтів з таблиці *Clients*. Це дозволяє сформувати єдиний перелік унікальних значень незалежно від того, з якої таблиці вони походять. Такий підхід корисний, коли необхідно об'єднати різнорідні дані, зберігаючи повноту інформації та уникаючи повторів, наприклад, для подальшого аналізу або відображення у звіті.

Приклад 4. Різниця (EXCEPT). Отримати клієнтів, які ще не купували книгу з жанру "Фантастика".

```
SELECT DISTINCT c.НайменуванняКлієнта
FROM Bookstore.Clients AS c
EXCEPT
SELECT DISTINCT c.НайменуванняКлієнта
FROM Bookstore.Clients AS c
JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
JOIN Bookstore.Genres AS g
    ON p.IdЖанру = g.IdЖанру
WHERE g.НазваЖанру = N'Фантастика'
```

Тут ми використовуємо операцію різниця. Перша частина запиту вибирає всіх клієнтів, а друга для тих, хто купував книги жанру “Фантастика”. *EXCEPT* видаляє з першої множини тих клієнтів, що зустрічаються у другій. У результаті залишаються тільки клієнти, які не купували книги цього жанру. Ця операція корисна для аналітики непокритих сегментів ринку.

	НайменуванняКлієнта
1	ТОВ "Книгоманія"
2	ТОВ Книгарня Є
3	ТОВ Книголенд
4	ФОП Іваненко О.П.

Приклад 5. Перетин (INTERSECT). Отримати клієнтів, які купували книги жанру «Роман» і «Фантастика».

```
SELECT DISTINCT c.IdКлієнта, c.НайменуванняКлієнта
FROM Bookstore.Clients AS c
JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
JOIN Bookstore.Genres AS g
    ON p.IdЖанру = g.IdЖанру
WHERE g.НазваЖанру = N'Роман'

INTERSECT

SELECT DISTINCT c.IdКлієнта, c.НайменуванняКлієнта
FROM Bookstore.Clients AS c
JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
JOIN Bookstore.Genres AS g
    ON p.IdЖанру = g.IdЖанру
WHERE g.НазваЖанру = N'Фантастика';
```

У запиті використана операція перетин. Кожний підзапит вибирає клієнтів, які купували книги певного жанру. Команда *INTERSECT* повертає тільки ті рядки, які присутні в обох підзапитах, тобто клієнтів, які купували і «Роман» і «Фантастика». Дана операція дозволяє виділяти спільні підмножини у базі даних.

	IdКлієнта	НайменуванняКлієнта
1	2	ТОВ Літера
2	3	ФОП Петренко О.О.
3	5	ТОВ БукМаркет
4	6	ФОП Сидоренко Н.П.

Приклад 6. Декартовий добуток (CROSS JOIN). Отримати всі можливі комбінації клієнтів та жанрів книг.

```
SELECT c.НайменуванняКлієнта, g.НазваЖанру
FROM Bookstore.Clients AS c
CROSS JOIN Bookstore.Genres AS g;
```

Кожен клієнт об'єднується з кожним жанром книг, створюючи повну комбінацію. Декартовий добуток використовується для створення всіх можливих поєднань елементів двох таблиць, що може бути корисним для маркетингового аналізу або прогнозування попиту.

Приклад 7. З'єднання (INNER JOIN). Отримати список клієнтів разом з назвами куплених книг.

```
SELECT c.НайменуванняКлієнта, p.НазваКниги, si.Кількість
FROM Bookstore.Clients AS c
INNER JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
INNER JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
INNER JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги;
```

	НайменуванняКлієнта	НазваКниги	Кількість
1	ТОВ Книгарня Є	Кобзар	20
2	ТОВ Книгарня Є	Harry Potter and the Philosopher's Stone	15
3	ТОВ Літера	The Old Man and the Sea	10
4	ТОВ Літера	1984	25
5	ФОП Петренко О.О.	Місто	25
6	ТОВ Книголенд	Захар Беркут	15
7	ТОВ Книголенд	Собор Паризької Богоматері	10
8	ТОВ БукМаркет	Кобзар	10
9	ТОВ БукМаркет	1984	20
10	ФОП Сидоренко Н.П.	Harry Potter and the Philosopher's Stone	15
11	ФОП Сидоренко Н.П.	The Old Man and the Sea	10
12	ФОП Сидоренко Н.П.	Місто	15
13	ТОВ Книгарня Є	Захар Беркут	25
14	ТОВ Літера	Кобзар	10
15	ТОВ Літера	Собор Паризької Богоматері	15
16	ФОП Петренко О.О.	1984	20
17	ТОВ Книголенд	Harry Potter and the Philosopher's Stone	25
18	ТОВ Книголенд	Місто	15
19	ТОВ БукМаркет	Захар Беркут	15
20	ТОВ БукМаркет	The Old Man and the Sea	10
21	ФОП Сидоренко Н.П.	1984	10
22	ФОП Сидоренко Н.П.	Собор Паризької Богоматері	25
23	ТОВ БукМаркет	Собор Паризької Богоматері	30
24	ФОП Петренко О.О.	Місто	33

У цьому прикладі використано внутрішнє з'єднання *INNER JOIN*. Воно повертає тільки ті записи, для яких є відповідність у всіх таблицях: клієнт, продаж, елемент продажу та книга. Якщо клієнт не купував жодної книги, він у результаті не відобразиться. Такий підхід використовується для аналізу реальних продажів і створення детальних звітів по клієнтах.

Приклад 8. Ділення (Division). Знайти всіх клієнтів, які купили всі книги, що коштують менше 300 грн**.**

```
SELECT c.НайменуванняКлієнта FROM Bookstore.Clients AS c
WHERE NOT EXISTS (
    SELECT p.IdКниги FROM Bookstore.PriceList AS p
    WHERE p.Ціна < 300
    AND NOT EXISTS (
        SELECT si.IdКниги FROM Bookstore.Sales AS s
        JOIN Bookstore.SalesItems AS si
```

```
            ON s.IdЧеку = si.IdЧеку
        WHERE s.IdКлієнта = c.IdКлієнта AND si.IdКниги = p.IdКниги     )
);
```

Даний запит реалізує операцію ділення. Зовнішній запит обирає всіх клієнтів. Перший підзапит формує множину всіх книг з ціною менше 500 грн. Другий підзапит перевіряє, чи купив клієнт кожну книгу з цієї множини. Якщо є книга з цієї множини, яку клієнт не купив, *NOT EXISTS* стає *FALSE*, і клієнт не потрапляє у результат. У результаті залишаються лише ті клієнти, які купили усі книги з цієї множини, що і демонструє принцип ділення.

	НайменуванняКлієнта	НазваКниги	Ціна
1	ТОВ Книгарня Є	Кобзар	250.00
2	ФОП Петренко О.О.	Місто	220.00
3	ТОВ Книголенд	Захар Беркут	270.00
4	ТОВ БукМаркет	Кобзар	250.00
5	ФОП Сидоренко Н.П.	Місто	220.00
6	ТОВ Книгарня Є	Захар Беркут	270.00
7	ТОВ Літера	Кобзар	250.00
8	ТОВ Книголенд	Місто	220.00
9	ТОВ БукМаркет	Захар Беркут	270.00
10	ФОП Петренко О.О.	Місто	220.00

Приклад 9. Перейменування (RENAME). Отримати список клієнтів із новою колонкою Клієнт.

```
SELECT НайменуванняКлієнта AS Клієнт, АдресаКлієнта
FROM Bookstore.Clients;
```

Тут реалізована операція перейменування. Колонка *НайменуванняКлієнта* у результаті отримала нове ім'я *Клієнт*. Це допомагає зробити звіт або запит більш зрозумілим для користувача, особливо коли вихідні назви колонок складні або довгі.

	Клієнт	АдресаКлієнта
1	ТОВ Книгарня Є	м. Київ, вул. Хрещатик, 22
2	ТОВ Літера	м. Львів, вул. Дорошенка, 11
3	ФОП Петренко О.О.	м. Одеса, вул. Дерибасівська, 5
4	ТОВ Книголенд	м. Харків, вул. Сумська, 19
5	ТОВ БукМаркет	м. Дніпро, просп. Яворницького, 10
6	ФОП Сидоренко Н.П.	м. Чернігів, вул. Шевченка, 7
7	ТОВ "Книгоманія"	м. Харків, вул. Сумська, 20
8	ФОП Іваненко О.П.	м. Львів, вул. Грушевського, 10

Приклад 10. Розширення (Extension). Отримати список всіх чеків клієнтів разом із новим обчислюваним атрибутом *ЗагальнаСума*, який відображає загальну вартість кожного чеку.

```
SELECT s.IdЧеку, s.IdКлієнта, s.ДатаЗамовлення,
       SUM(si.Кількість * p.Ціна) AS ЗагальнаСума
FROM Bookstore.Sales AS s
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
GROUP BY s.IdЧеку, s.IdКлієнта, s.ДатаЗамовлення;
```

У даному прикладі використовуємо розширення, додаючи новий обчислюваний атрибут *ЗагальнаСума*, обчислюється шляхом множення

кількості книг у кожному чеку на їхню ціну та підсумовування результату для всіх позицій у чеку. Таким чином, результатом запиту є таблиця з усіма чеками та їхньою загальною вартістю. Це дозволяє аналізувати фінансові показники без зміни первинних даних у базі.

	IdЧеку	IdКлієнта	ДатаЗамовлення	ЗагальнаСума
1	1	1	2024-01-10	10250.00
2	2	2	2024-02-15	11500.00
3	3	3	2024-03-05	5500.00
4	4	4	2024-04-20	7150.00
5	5	5	2024-05-11	8940.00
6	6	6	2024-06-25	12000.00
7	7	1	2024-07-08	6750.00
8	8	2	2024-08-14	7150.00
9	9	3	2024-09-19	6440.00
10	10	4	2024-10-03	12050.00
11	11	5	2024-11-17	7500.00
12	12	6	2024-12-28	10970.00
13	13	5	2025-10-01	9300.00
14	14	3	2025-10-01	7260.00

Приклад 11. Підведення підсумків (Aggregation). Порахувати загальну суму продажів для кожного клієнта за рік.

```
SELECT c.НайменуванняКлієнта, SUM(si.Кількість * p.Ціна) AS
ЗагальнаСумаПродажів
FROM Bookstore.Clients AS c
JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
GROUP BY c.НайменуванняКлієнта;
```

Тут застосовується підведення підсумків (агрегація) для обчислення загальної суми продажів кожного клієнта. Оператор *SUM* підсумовує вартість всіх книг, які купив клієнт, з'єднуючи дані з таблиць клієнтів, продажів, позицій продажів і прайс-листу. Результат показує, скільки грошей приніс кожен клієнт магазину протягом усього періоду.

	НайменуванняКлієнта	ЗагальнаСумаПродажів
1	ТОВ БукМаркет	25740.00
2	ТОВ Книгарня Є	17000.00
3	ТОВ Книголенд	19200.00
4	ТОВ Літера	18650.00
5	ФОП Петренко О.О.	19200.00
6	ФОП Сидоренко Н.П.	22970.00

Приклад 12. Множинне підведення підсумків (Grouped Aggregation). Порахувати загальну суму продажів та кількість чеків для кожного клієнта.

```
SELECT c.НайменуванняКлієнта,
       COUNT(s.IdЧеку) AS КількістьЧеків,
       SUM(si.Кількість * p.Ціна) AS ЗагальнаСумаПродажів
FROM Bookstore.Clients AS c
JOIN Bookstore.Sales AS s
    ON c.IdКлієнта = s.IdКлієнта
JOIN Bookstore.SalesItems AS si
    ON s.IdЧеку = si.IdЧеку
JOIN Bookstore.PriceList AS p
    ON si.IdКниги = p.IdКниги
GROUP BY c.НайменуванняКлієнта;
```

	НайменуванняКлієнта	КількістьЧеків	ЗагальнаСумаПродажів
1	ТОВ БукМаркет	5	25740.00
2	ТОВ Книгарня Є	3	17000.00
3	ТОВ Книголенд	4	19200.00
4	ТОВ Літера	4	18650.00
5	ФОП Петренко О.О.	3	19200.00
6	ФОП Сидоренко Н.П.	5	22970.00

У цьому прикладі використовується множинне підведення підсумків, коли для кожного клієнта обчислюються два показники одночасно: кількість чеків *COUNT* та загальна сума продажів *SUM*. Завдяки групуванню *GROUP BY* всі позиції продажів одного клієнта об’єднуються, що дозволяє отримати повну статистику по кожному клієнту в одному результаті. Ця операція допомагає аналізувати одночасно кілька показників для різних груп даних.

СПИСОК РКЕОМЕНДОВАНИХ ДЖЕРЕЛ

1. Рзаєва, С. Л., & Харченко, О. А. (2021). *Бази даних: Навчальний посібник*. КНТЕУ.
2. Abramov, V., Astafieva, M., Boiko, M., Bodnenko, D., Bushma, A., Vember, V., Hlushak, O., Zhyltsov, O., Ilich, L., Kobets, N., Kovaliuk, T., Kuchakovska, H., Lytvyn, O., Lytvyn, P., Mashkina, I., Morze, N., Nosenko, T., Proshkin, V., Radchenko, S., & Yaskevych, V. (2021). *Theoretical and practical aspects of the use of mathematical methods and information technology in education and science*. https://doi.org/10.28925/9720213284km
3. Берко, А. Ю., Верес, О. М., & Пасічник, В. В. (2024). *Системи баз даних та знань. Книга 1. Організація баз даних та знань* (2-ге вид.). Магнолія 2006.
4. Beaulieu, A. (2025). *Learning SQL: Master SQL Fundamentals*. O'Reilly.
5. Forta, B. (2025). *SQL in 10 Minutes, Sams Teach Yourself* (5th ed.). Sams Publishing.
6. West, R., Assaf, W., & Noble, E. (2023). *SQL Server 2022 Administration Inside Out*. Microsoft Press.
7. Доценко, С. І. (2023). *Організація та системи керування базами даних: Навчальний посібник*. УкрДУЗТ.
8. Blokdyk, G. (2020). *Administration of databases. The complete guide*. 5STARCooks.
9. Трофименко, О. Г., Прокоп, Ю. В., Логінова, Н. І., & Копитчук, І. М. (2019). *Організація баз даних* (2-ге вид., виправ. і доповн.). Фенікс.
10. Rainer, K., & Prince, B. (2022). *Introduction to Information Systems* (9th ed.). Wiley.
11. Mashkina, I., Rzaieva, S., Kostiuk, Y., Mazur, N., & Brzhevska, Z. (2025). Cybersecurity in intelligent transport systems: Current challenges and solutions. In *Cybersecurity Providing in Information and Telecommunication Systems 2025* (pp. 1–13). CEUR-WS. https://ceur-ws.org/Vol-3991/
12. Rzaieva, S., Rzaiev, D., Mykytenko, N., Dreis, Y., & Grechaninov, V. (2025). Methods of personal data protection in retail: Practical solutions. In *Cybersecurity Providing in Information and Telecommunication Systems 2025* (pp. 492–506). CEUR-WS. https://ceur-ws.org/Vol-3991/
13. Складанний, П. М., Костюк, Ю. В., Рзаєва, С. Л., Самойленко, Ю. О., & Савченко, Т. В. (2025). Розробка модульних нейронних мереж для виявлення різних класів мережевих атак. *Кібербезпека: освіта, наука, техніка*, 3(27), 534–548. https://doi.org/10.28925/2663-4023.2025.27.772
14. Костюк, Ю. В., Бебешко, Б. Т., Складанний, П. М., Рзаєва, С. Л., & Хорольська, К. В. (2024). Оптимізація буфера та пріоритетів для забезпечення безпеки у Bluetooth-мережах. *Безпека інформаційних систем і технологій*, 2(8), 5–16. https://doi.org/10.17721/ISTS.2024.8
15. Костюк, Ю. В., Довженко, Н. М., Мазур, Н. П., Складанний, П. М., & Рзаєва, С. Л. (2025). Методика захисту GRID-середовища від шкідливого коду під час виконання обчислювальних завдань. *Кібербезпека: освіта, наука, техніка*, 3(27), 22–40. https://doi.org/10.28925/2663-4023.2025.27.710
16. Складанний, П. М., Машкіна, І. В., Рзаєва, С. Л., & Костюк, Ю. В. (2025). Методи GDPR для забезпечення безпеки сховищ даних від витоків та загроз. *Телекомунікаційні та інформаційні технології*, 2, 59–76. https://doi.org/10.31673/2412-4338.2025.027860
17. Savchuk, K., Rzaieva, S., Savchenko, T., & Rzaiev, D. (2024). Data protection strategies and technologies for ensuring national financial security. *In Springer Proceedings* (pp. 431–440). https://doi.org/10.1007/978-3-031-70399-7_32

18. Костюк, Ю. В., Бебешко, Б. Т., Гулак, Г. М., Складанний, П. М., Рзаєва, С. Л., & Хорольська, К. В. (2024). Забезпечення кібербезпеки та швидкодії передачі даних у безпровідних мережах. *Безпека інформації*, 30(3), 365–375. https://doi.org/10.18372/2225-5036.30.20357
19. Rzaieva, S., Rzaiev, D., Kostyuk, Y., Hulak, H., & Shcheblanin, O. (2024). Methods of modeling database system security. *CEUR Workshop Proceedings*, 3654, 384–390. https://ceur-ws.org/Vol-3654/
20. Рзаєва, С., & Чернишова, Д. (2024). Інформаційно-аналітична система для гендерно-нейтрального відбору кадрів закладу вищої освіти. *Технічні науки та технології*, 2(36), 126–136. https://doi.org/10.25140/2411-5363-2024-2(36)-126-136
21. Довженко, Н. М., Мазур, Н. П., Костюк, Ю. В., & Рзаєва, С. Л. (2024). Інтеграція IoT та штучного інтелекту в інтелектуальні транспортні системи. *Кібербезпека: освіта, наука, техніка*, 2(26), 430–444. https://doi.org/10.28925/2663-4023.2024.26.708
22. Костюк, Ю. В., Складанний, П. М., Бебешко, Б. Т., Хорольська, К. В., Рзаєва, С. Л., & Ворохоб, М. В. (2025). *Безпека інформаційно-комунікаційних систем*. Україна.
23. Костюк, Ю. В., Складанний, П. М., Гулак, Г. М., Бебешко, Б. Т., Хорольська, К. В., & Рзаєва, С. Л. (2025). *Системи захисту інформації*. Україна.

Internet-ресурси

1. Управляючі Конструкції sql. [Електронний ресурс] - Режим доступу до ресурсу: https://studfiles.net/preview/5210288/page:2
2. Адміністрування бази даних – режим доступу: http://ua-referat.com/%D0%90%D0%B4%D0%BC%D1%96%D0%BD%D1%96%D1%81%D1%82%D1%80%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85
3. Системи баз даних та знань – режим доступу: http://ism.lp.edu.ua/uk/content/systemy-baz-danyh-ta-znan-knyga-1-organizaciya-baz-danyh-ta-znan-0.
4. Технологія доступу, зберігання та адміністрування даних – режим доступу: http://posibniki.com.ua/post-tehnologiya-dostupu-zberigannya-ta-administruvannya-danih-u-kis.
5. Microsoft SQL Server – режим доступу: https://learn.microsoft.com/en-us/sql/relational-databases/databases/create-a-database?view=sql-server-ver16

Навчальне видання

РЗАЄВА Світлана Леонідівна,
МАШКІНА Ірина Вікторівна,
СКЛАДАННИЙ Павло Миколайович,
КОСТЮК Юлія Володимирівна,
РЗАЄВ Дмитро Олександрович,
КРАСЮК Юлія Миколаївна

ОСНОВИ БАЗ ДАНИХ

Посібник