[Головні поняття реляційних баз даних 8](#_Toc57748893)

[SQL – Structured Query Language 21](#_Toc57748894)

[Подальша нормалізація БД 24](#_Toc57748895)

[Перша нормальна форма 24](#_Toc57748896)

[Мова визначення даних DDL 25](#_Toc57748897)

[Створення та зміна елементів схеми даних 25](#_Toc57748898)

[Створення БД 25](#_Toc57748899)

[Створення таблиць 26](#_Toc57748900)

[Створення таблиці з використанням іншої таблиці 26](#_Toc57748901)

[Створення нових таблиць 26](#_Toc57748902)

[Створення та вилучення тимчасових таблиць 27](#_Toc57748903)

[Важливі факти про тимчасові таблиці: 27](#_Toc57748904)

[Приклад тимчасової таблиці 28](#_Toc57748905)

[Створення тимчасової таблиці і додання данних 28](#_Toc57748906)

[Створення тимчасової таблиці на підставі оператору select 29](#_Toc57748907)

[Вилучення тимчасових таблиць 29](#_Toc57748908)

[Створення подання View 29](#_Toc57748909)

[Зміна стовпчика таблиці 30](#_Toc57748910)

[Зміна таблиці ALTER TABLE 30](#_Toc57748911)

[Вилучення всіх даних з таблиці TRUNCATE TABLE 30](#_Toc57748912)

[Вилучення елементів схеми даних 31](#_Toc57748913)

[Вилучення стовпчику з таблиці 31](#_Toc57748914)

[Вилучення таблиці 31](#_Toc57748915)

[Вилучення подання (VIEW) 31](#_Toc57748916)

[Вилучення БД 31](#_Toc57748917)

[SQL обмеження 31](#_Toc57748918)

[Обмеження NOT NULL 32](#_Toc57748919)

[Визначення обмеження NOT NULL під час CREATE TABLE 32](#_Toc57748920)

[Зміна обмеження NOT NULL за допомогою ALTER TABLE 33](#_Toc57748921)

[Обмеження унікальності UNIQUE 33](#_Toc57748922)

[Визначення обмеження UNIQUE під час CREATE TABLE 33](#_Toc57748923)

[Зміна обмеження UNIQUE за допомогою ALTER TABLE 33](#_Toc57748924)

[Вилучення обмеження UNIQUE 34](#_Toc57748925)

[Обмеження первинного ключа PRIMARY KEY Constraint 34](#_Toc57748926)

[Визначення обмеження PRIMARY KEY під час CREATE TABLE 34](#_Toc57748927)

[Зміна обмеження PRIMARY KEY за допомогою ALTER TABLE 34](#_Toc57748928)

[Вилучення обмеження PRIMARY KEY 35](#_Toc57748929)

[Обмеження зовнішнього ключа FOREIGN KEY 35](#_Toc57748930)

[Визначення обмеження FOREIGN KEY під час CREATE TABLE 36](#_Toc57748931)

[Зміна обмеження FOREIGN KEY за допомогою ALTER TABLE 36](#_Toc57748932)

[Вилучення обмеження завнішнього ключа FOREIGN KEY 36](#_Toc57748933)

[Обмеження додаткової умови CHECK 37](#_Toc57748934)

[Визначення обмеження CHECK під час CREATE TABLE 37](#_Toc57748935)

[Зміна обмеження CHECK за допомогою ALTER TABLE 37](#_Toc57748936)

[Вилучення обмеження додаткової умови CHECK 37](#_Toc57748937)

[Обмеження DEFAULT 37](#_Toc57748938)

[Визначення обмеження DEFAULTпід час CREATE TABLE 37](#_Toc57748939)

[Зміна обмеження DEFAULT за допомогою ALTER TABLE 38](#_Toc57748940)

[Вилучення обмеження DEFAULT 38](#_Toc57748941)

[Створення індексу CREATE INDEX 38](#_Toc57748942)

[Створення звичайного індексу 38](#_Toc57748943)

[Створення унікального індексу 38](#_Toc57748944)

[Два способи створення індексу 39](#_Toc57748945)

[Унікальний індекс (Unique Index) 40](#_Toc57748946)

[Вилучення індексу (Drop Index) 40](#_Toc57748947)

[Перейменування індексу 41](#_Toc57748948)

[Типи даних MySQL Data Types (Version 8.0) 41](#_Toc57748949)

[Визначення тексту мовою DDL 41](#_Toc57748950)

[Визначення чисел мовою DDL 42](#_Toc57748951)

[Визначення дат мовою DDL 43](#_Toc57748952)

[Тип даних для визначення Дати та/або Часу: 43](#_Toc57748953)

[Формат дати (Date) 44](#_Toc57748954)

[MySQL формат 44](#_Toc57748955)

[MS SQL формат 44](#_Toc57748956)

[Подальша нормалізація БД 2 46](#_Toc57748957)

[Зміна стану БД 46](#_Toc57748958)

[Внесення даних 46](#_Toc57748959)

[Модифікація даних 47](#_Toc57748960)

[Видалення записів з таблиці оператором Delete 47](#_Toc57748961)

[Цілісність БД 48](#_Toc57748962)

[Друга нормальна форма 51](#_Toc57748963)

[Третая нормальна форма 53](#_Toc57748964)

[Нормальна форма Бойса-Кодда (БКНФ) 55](#_Toc57748965)

[Четверта нормальна форма 55](#_Toc57748966)

[П'ята нормальна форма 57](#_Toc57748967)

[Доменно-ключова нормальна форма (ДКНФ) 58](#_Toc57748968)

[Шоста нормальна форма 58](#_Toc57748969)

[Мова маніпулювання даними DML 58](#_Toc57748970)

[Таблиці бази даних, використані як приклад 58](#_Toc57748971)

[Оператор INSERT 59](#_Toc57748972)

[Оператор INSERT INTO 60](#_Toc57748973)

[Оператор INSERT комбінований з SELECT 60](#_Toc57748974)

[Оператор UPDATE 61](#_Toc57748975)

[Оператор DELETE 62](#_Toc57748976)

[Оператор SELECT 64](#_Toc57748977)

[Джерело даних FROM 64](#_Toc57748978)

[Опції TOP, LIMIT ROWNUM 65](#_Toc57748979)

[Опція OFFSET-FETCH (ORACLE) 66](#_Toc57748980)

[Опція DISTINCT 67](#_Toc57748981)

[NULL функції IFNULL(),COALESCE() 67](#_Toc57748982)

[Функції MIN, MAX 68](#_Toc57748983)

[Функції COUNT, SUM, AVG у запитах 68](#_Toc57748984)

[Аліаси у запитах 69](#_Toc57748985)

[Аліаси стовпчиків Alias for Columns 70](#_Toc57748986)

[Аліаси таблиць Alias for Tables 71](#_Toc57748987)

[Опція CASE 72](#_Toc57748988)

[WHERE у запиті на вибір данних 73](#_Toc57748989)

[Предикат WHERE BETWEEN 74](#_Toc57748990)

[NOT BETWEEN 75](#_Toc57748991)

[BETWEEN with IN 75](#_Toc57748992)

[BETWEEN Text Values 75](#_Toc57748993)

[NOT BETWEEN Text Values 75](#_Toc57748994)

[BETWEEN Dates 75](#_Toc57748995)

[Операції WHERE AND, OR, NOT 76](#_Toc57748996)

[Предикати WHERE ANY and ALL 78](#_Toc57748997)

[Предикат WHERE EXISTS 80](#_Toc57748998)

[Предикат WHERE IN 81](#_Toc57748999)

[Предикат IS NULL 82](#_Toc57749000)

[Предикат IS NOT NULL 83](#_Toc57749001)

[Предикат WHERE LIKE 84](#_Toc57749002)

[Пошук рядків за шаблоном Wildcard 86](#_Toc57749003)

[MS Access 86](#_Toc57749004)

[MS SQL 86](#_Toc57749005)

[Використання %у шаблоні пошуку % 87](#_Toc57749006)

[Використання \_ у шаблоні пошуку 87](#_Toc57749007)

[Використання [charlist] у шаблоні пошуку 87](#_Toc57749008)

[Використання [!charlist] у шаблоні пошуку 88](#_Toc57749009)

[Сортування результатів запиту ORDER BY 88](#_Toc57749010)

[The SQL ORDER BY syntax 88](#_Toc57749011)

[Пряме сортування ASC 88](#_Toc57749012)

[Зворотнє сортування DESC 88](#_Toc57749013)

[Групування даних GROUP BY 91](#_Toc57749014)

[Просте групування даних 91](#_Toc57749015)

[Предикат HAVING 92](#_Toc57749016)

[GROUP BY With JOIN 94](#_Toc57749017)

[Комбінування записів кількох таблиць JOIN 94](#_Toc57749018)

[Different Types of SQL JOINs 95](#_Toc57749019)

[INNER JOIN Keyword 97](#_Toc57749020)

[JOIN Three Tables 99](#_Toc57749021)

[SQL LEFT JOIN 99](#_Toc57749022)

[SQL RIGHT JOIN 100](#_Toc57749023)

[Оператори FULL JOIN FULL OUTER JOIN 102](#_Toc57749024)

[Оператор Self JOIN 104](#_Toc57749025)

[Об’єднання результатів запитів UNION 106](#_Toc57749026)

[UNION ALL Syntax 107](#_Toc57749027)

[UNION With WHERE 108](#_Toc57749028)

[UNION ALL With WHERE 108](#_Toc57749029)

[Another UNION Example 108](#_Toc57749030)

[Вкладені запити - Subqueries 108](#_Toc57749031)

[Некорельовані підзапити 109](#_Toc57749032)

[Корельовані підзапити 111](#_Toc57749033)

[Копіювання таблиць оператором SELECT INTO 112](#_Toc57749034)

[Підготовлені оператори Prepared Statements 114](#_Toc57749035)

[Prepared Statements in Application Programs 114](#_Toc57749036)

[Prepared Statements in SQL Scripts 114](#_Toc57749037)

[PREPARE, EXECUTE, DEALLOCATE PREPARE 115](#_Toc57749038)

[MySQL PREPARE Statement 118](#_Toc57749039)

[EXECUTE Statement 121](#_Toc57749040)

[DEALLOCATE PREPARE Statement 121](#_Toc57749041)

[PL/SQL 122](#_Toc57749042)

[Типи процедур і функцій 122](#_Toc57749043)

[Створення збережених процедур і функцій 122](#_Toc57749044)

[Параметри процедур і функцій 123](#_Toc57749045)

[Програми і залежності. Перекомпіляція програм 124](#_Toc57749046)

[Отримання інформації про процедури і функції 125](#_Toc57749047)

[Виконання функції 127](#_Toc57749048)

[Вилучення функції з БД 128](#_Toc57749049)

[MySQL процедури користувача MySQL 128](#_Toc57749050)

[Виконання процедури EXEC 128](#_Toc57749051)

[Вилученя процедури з серверу БД 129](#_Toc57749052)

[Елементи мови PL/SQL 129](#_Toc57749053)

[Оператор IF-THEN-ELSE 129](#_Toc57749054)

[Оператор WHILE 129](#_Toc57749055)

[Оператор LEAVE 130](#_Toc57749056)

[Оператор ITERATE 130](#_Toc57749057)

[Оператор RETURN 131](#_Toc57749058)

[Оператор LOOP 132](#_Toc57749059)

[Оператор REPEAT 132](#_Toc57749060)

[Курсор MySQL: Декларування курсору 135](#_Toc57749061)

[Відкриття курсору OPEN 135](#_Toc57749062)

[Оператор FETCH – отримання даних 136](#_Toc57749063)

[Закриття курсору CLOSE 136](#_Toc57749064)

[Маніпулятор курсора Handler for Cursor's NOT FOUND condition 137](#_Toc57749065)

[Тригери MySQL: 137](#_Toc57749066)

[Створення і включення тригерів 138](#_Toc57749067)

[Класифікація тригерів. 139](#_Toc57749068)

[Порядок активації тригерів. 139](#_Toc57749069)

[Тригерні предикати. 139](#_Toc57749070)

[Псевдозаписи OLD І NEW. 139](#_Toc57749071)

[Вмикання, вимивання та видалення тригерів. 140](#_Toc57749072)

[Приклади створення тригерів 140](#_Toc57749073)

[BEFORE INSERT Trigger 142](#_Toc57749074)

[MySQL тригер AFTER INSERT 143](#_Toc57749075)

[Тригер MySQL: BEFORE UPDATE 144](#_Toc57749076)

[MySQL тригер: AFTER UPDATE 145](#_Toc57749077)

[MySQL тригер : BEFORE DELETE 146](#_Toc57749078)

[MySQL тригер : AFTER DELETE 147](#_Toc57749079)

[Вилучення тригера з сервера: DROP TRIGGER 147](#_Toc57749080)

[Висновки 148](#_Toc57749081)

[TCL (MySQL Transaction Control Language) 150](#_Toc57749082)

[Introduction on Transaction 150](#_Toc57749083)

[Understand the concept of a transaction 150](#_Toc57749084)

[MySQL and the ACID Model 151](#_Toc57749085)

[MySQL Transaction 152](#_Toc57749086)

[SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT 155](#_Toc57749087)

[LOCK and UNLOCK Tables 156](#_Toc57749088)

[SET TRANSACTION Syntax 156](#_Toc57749089)

[Transactional and Locking Statements 157](#_Toc57749090)

[START TRANSACTION, COMMIT, and ROLLBACK Statements 157](#_Toc57749091)

[Statements That Cannot Be Rolled Back 159](#_Toc57749092)

[Statements That Cause an Implicit Commit 160](#_Toc57749093)

[SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements 160](#_Toc57749094)

[LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE Statements 161](#_Toc57749095)

[LOCK TABLES and UNLOCK TABLES Statements 162](#_Toc57749096)

[*Table Lock Acquisition* 162](#_Toc57749097)

[*Table Lock Release* 164](#_Toc57749098)

[*Interaction of Table Locking and Transactions* 164](#_Toc57749099)

[*LOCK TABLES and Triggers* 165](#_Toc57749100)

[*Table-Locking Restrictions and Conditions* 165](#_Toc57749101)

[SET TRANSACTION Statement 166](#_Toc57749102)

[*Transaction Isolation Levels* 167](#_Toc57749103)

[*Transaction Access Mode* 167](#_Toc57749104)

[*Transaction Characteristic Scope* 167](#_Toc57749105)

[DCL data control language 169](#_Toc57749106)

[MySQL DCL 171](#_Toc57749107)

[SQL GRANT Command 171](#_Toc57749108)

[SQL REVOKE Command: 172](#_Toc57749109)

[Privileges and Roles: 172](#_Toc57749110)

[Creating Roles: 173](#_Toc57749111)

[Drop role 174](#_Toc57749112)

[SQL Server: Grant/Revoke Privileges 174](#_Toc57749113)

[Grant Privileges on Table 174](#_Toc57749114)

[Example 175](#_Toc57749115)

[Revoke Privileges on Table 175](#_Toc57749116)

[Syntax 175](#_Toc57749117)

[Example 176](#_Toc57749118)

[SQL ін’єкція 182](#_Toc57749119)

[SQL у веб сторінках 182](#_Toc57749120)

[SQL ін’єкція , що використовує завжди вірну умову 1=1 182](#_Toc57749121)

[Ін’єкція, що базується на завжди вірній умові ""="" 182](#_Toc57749122)

[SQL ін’єкція, що базується на кількох операторах SQL 184](#_Toc57749123)

[Використання SQL параметрів для захисту БД 184](#_Toc57749124)

[Приклади захисту у середовищі ASP.NET 184](#_Toc57749125)

[Ключові слова SQL 186](#_Toc57749126)

[Синтаксис операторів SQL 189](#_Toc57749127)

[SQL Statement 189](#_Toc57749128)

[Syntax 189](#_Toc57749129)

[Інші типи баз даних - DBMS Tutorials 193](#_Toc57749130)

**Реляційні бази даних**

# Головні поняття реляційних баз даних

В основі РБД лежить поняття множин та сутностей з їх відношеннями.

Наприклад, для університету можливо вибрати такі сутності:

Студент, Викладач, Група, Предмет, Факультет

Кожна з цих сутностей у реальному світі має багато примірників:

Різні студенти, викладачі, групи, предмети, факультети

Якщо сутність має тальки один об’єкт, то ми її не розглядаємо

Найкраще побудову моделі даних проводити у сучасних засобах, наприклад ERWIN. Для цього запустимо засіб та виберемо створення логічної та фізичної моделей даних. При цьому ми маємо визначити нашу майбутню СУБД, наприклад MySQL.

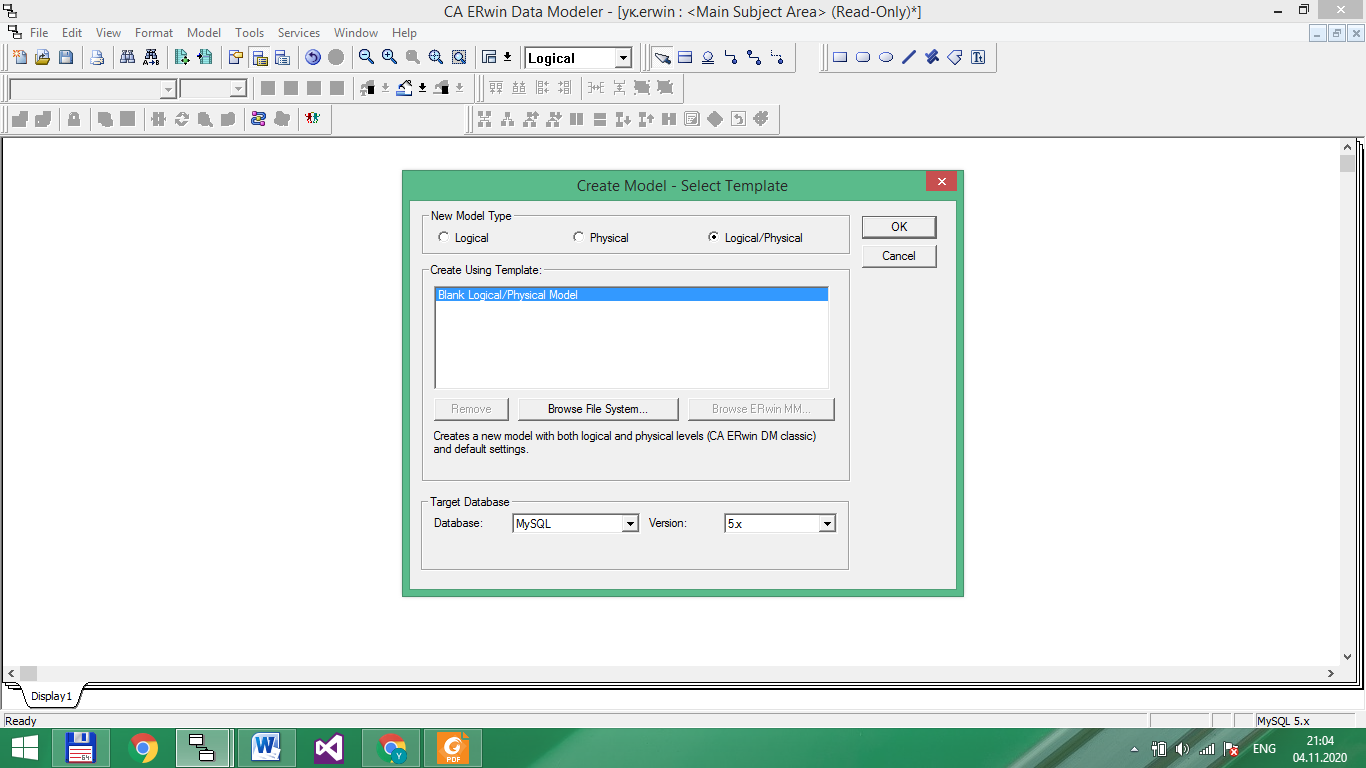


Рисунок 1 Середовище розроблення БД

Після обирання СУБД створимо кілька сутностей та зв’язків між ними:



Рисунок 2 Сутності моделі даних

На наступному етапі маємо показати зв’язки між сутностями. Для цього треба дослідити предметну область, та визначити, що:

один студент навчається у одній групі; в одній групі є багато студентів;

група вчиться на факультеті; факультет складається з багатьох груп;

студенти вивчають багато предметів; один предмет вивчає багато студентів;

Викладач навчає студентів багатьом предметам;

Один предмет викладає тільки один викладач.

Ці правила дозволяють побудувати концептуальну модель даних (КМД):



Рисунок 3 Концептуальна модель даних

КМД визначає тільки головні сутності предметної області. Чорна цяточка визначає , той бік відношення, де є багато об’єктів.

Надалі нам треба визначити сутності більш детально. Для цього використовують математичне поняття множину.

Множина - математичний об’єкт, що містить сукупність елементів

Задати множину означає визначити два правила:

1. Правило, що дозволяє зрозуміти : чи належить об’єкт множині. Тобто всі об’єкти множини мають якусь певну ознаку,що їх об’єднує. Наприклад: множина студентів, множина викладачів, множина аудиторій тощо.
2. Правило, що дозволяє розрізняти окремі об’єкти множини. Тобто всі елементи мають якісь додаткові ознаки. Наприклад, студенти мають різні прізвища, імена, дати народження тощо

Висновок: будь-яка пара елементів однієї множини не може бути однаковою.

Над множинами можливі певні операції , напр., об’єднання(U), перетин (∩):

X={1,2,3,4,5} Y={4,5,6,7,8} – дві множини цілих чисел

X U Y ={1,2,3,4,5,6,7,8} - об'єднання множин

X ∩ Y ={4,5} - перетин множин

Декартовий добуток дозволяє на підставі кількох множин утворювати множину складних елементів.

Наприклад, якщо яснують дві множини цілих чисел

X={1,2,3,4,5} Y={1,2,3,4,5}

то декартовий добуток буде таким:

XхY ={(1,1),(1,2),(1,3),(1,4),(1,5),

(2,1),(2,2),(2,3),(2,4),(2,5),

(3,1),(3,2),(3,3),(3,4),(3,5),

(4,1),(4,2),(4,3),(4,4),(4,5),

(5,1),(5,2),(5,3),(5,4),(5,5)}

Тобто у резальтаті декартового добутку двох множин, які мають по п'ять елементів, ми матимемо двадцять п'ять упорядкованих пар. Прикладом дуже вам відомого декартового добутку є декартова площина.

Висновок: загальна кількість результуючої множини визначається добутком кількості елементів двох множин. Тобто декартовий добуток трьох множин, кожна з яких має 10 елементів, складає 1000 упорядкованих трійок.

Надалі декартовий добуток буде використовуватися дуже часто.

Таким чином, РБД визначає, що у нас є певні множини значень якихось ознак. Наприклад, множина прізвищ, множина імен, дат народжень тощо.

Це дозволяє зберігати певну інформацію. Зрозуміло, що ці множини визначаються задачею, яку треба вирішувати за допомогою РБД.

Сукупність таких множин називають відношеннями, які мають свої назви, а множини – атрибутами відношень.

Далі подано три відношення: Студент, Пацієнт, Викладач, що мають певні атрибути. Відношення використовують у РБД для моделювання сутностей

|  |  |  |  |
| --- | --- | --- | --- |
| Назва відношення | Студент | Пацієнт | Викладач |
| Атрибути відношення | Прізвище | Прізвище | Прізвище |
| Ім’я | Ім’я | Ім’я |
| По-батькові | По-батькові | По-батькові |
| ДатаНародження | ДатаНародження | ДатаНародження |
| Стать | Стать | Освіта |
|  | Захворювання | Науковий ступінь |
|  | Відділення | Наукове звання |
|  |  | Лікар | кафедра |

Слід сказати, що кількість атрибутів визначається під час проектування БД, а рівень деталізації тільки вимогами користувача майбутньої БД

Атрибути мають бути константами, або такими, що змінюються дуже рідко.

Наприклад, ми не можемо написати атрибут ВІК, бо його значення буде змінюватися кожного року. Замість такого атрибуту краще використати дату народження, яка є постійною, а кожного разу, коли потрібен вік, ми його маємо підрахувати самостійно: ВІК=ДАТА\_ПОТОЧНА-ДАТА\_НАРОДЖЕННЯ

Атрибути відношення можуть бути простими(атомарними): число, рядок, дата, час.

Існують також складні атрибути, наприклад , пацієнт може мати одразу кілька захворювань.

Надалі, розуміється, що атрибут відношення має тільки одне значення.

Приклад відношення Студент

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Прізвище | Ім’я | По-батькові | ДатаНародження | Стать |
| Студент1 | Шевченко | Ольга | Петрівна | 14.05.2002 | ж |
| Студент2 | Петренко | Івга | Семенівна | 14.04.2001 | ж |
| Студент3 | Тесленко | Іван | Іванович | 13.02.2003 | ч |
| Студент4 | Ляшенко | Петро | Петрович | 13.07.2003 | ч |

У цій таблиці рядки відповідають різним студентам, а стовпчики – атрибутам

Треба зауважити, що значення відношення залежить тільки від даних і не залежать від порядку рядків, або стовпчиків. У звичайних таблицях значення залежить від порядку рядків чи стовпчиків.

Інваріантність відношення до порядку рядків(записів) чи стовпчиків(атрибутів) обумовлює можливість роботи СУБД з оптимізації запитів до неї.

Біль детально розглянемо атрибути відношення.

Як ми вже казали, атрибути мають бути атомарними

Серед атрибутів є такі, що мають унікальне значення, такі атрибути є потенційними ключами.

У свою чергу потенційні ключі бувають простими чи складними. Ключ є простим, якщо він складається тільки з одного атрибуту. Якщо ключ складається з кількох атрибутів, то він є складним

Для студента унікальним атрибутом є :

номер студ. квитка - простий потенційний ключ

ідентиф. код - простий потенційний ключ

номер паспорта - простий потенційний ключ

прізвище, ім’я, по-батькові, дата народження – складний потенційний ключ

Один з потенційних ключів оголошується головним, або первинним (primary) ключем.

Такий ключ використовується для подальшого проектування БД

Слід сказати, що процес проектування БД називається нормалізацією

Головний принцип нормалізації полягає у тому, що один факт має залишатися тільки у одному місці. Для використання факту застосовують посилання на факт

Розглянемо нашу КМД та спробуємо визначити атрибути сутностей (відношень)



Рисунок 4 Розширена концептуальна модель даних

У нашій моделі як первинні ключі були створені додаткові атрибути, що гарантовано мають унікальні значення та не залежать від впливу зовнішнього світу. Такі штучні потенційні називають сурогатними. У нас: кодФакультету, кодГрупи… кодВикладача

Через те, що назва групи кожного року змінюється, як первинний ключ було використано сурогатний ключ – кодГрупи.

Слід сказати, що назва сутності пишеться у іменному відмінку й однині.

Назви атрибутів та сутностей можливо писати у сучасних БД будь-якою зручною мовою.

Подальше проектування полягає в уточненні атрибутів та спрощенню зв’язків.

Зв’язки між відношеннями утворюються шляхом міграції первинного ключа.

При цьому утворюються пари «головна-підлегла сутність». У нашому випадку таких зв’язків п’ять. Відповідно п’ять пар сутностей. РБД припускає такі кардинальності зв’язків «один-до одного», «один до багатьох», «багато до багатьох». Останній зв'язок у процесі проектування має перетворитися на два зв’язки і додаткову сутність:

«один до багатьох» ;

асоціативна сутність;

«багато до одного».

Крім того існують також зв’язки: «тип-підтип» та «ієрархічний», які ми розглянемо далі.

Зв’язком «багато до багатьох» у нашому випадку є зв'язок між предметом та студентом, який має кардинальність «багато до багатьох». Для цього на зв’язку натиснемо праву кнопку та в меню оберемо пункт:Create Association Entity

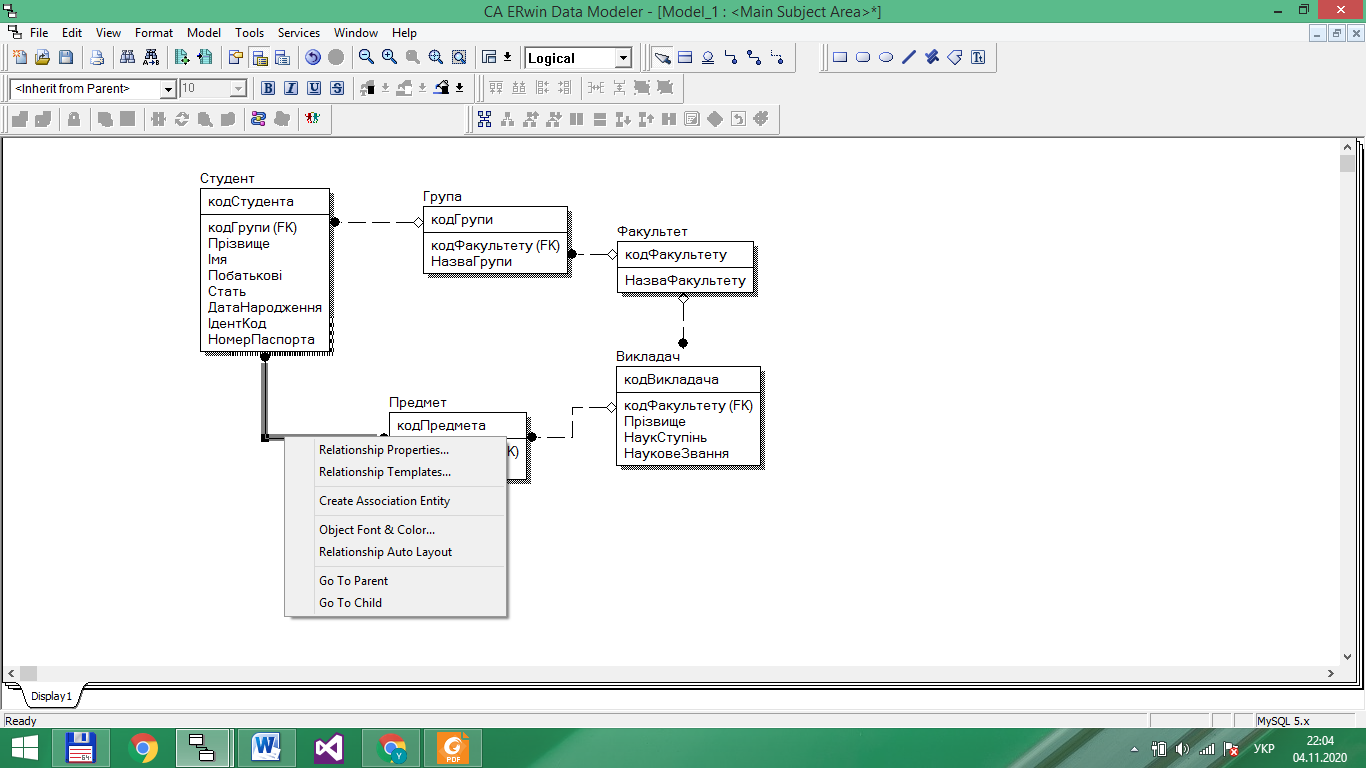


Рисунок 5 Розширена концептуальна модель даних 2

Це трішки змінить модель даних



Рисунок 6 Розширена концептуальна модель даних 3

У створеній сутності додамо атрибути: бал, який отримав студент; дата, коли студент отримав бал. Для цього уточнимо атрибути нового відношення

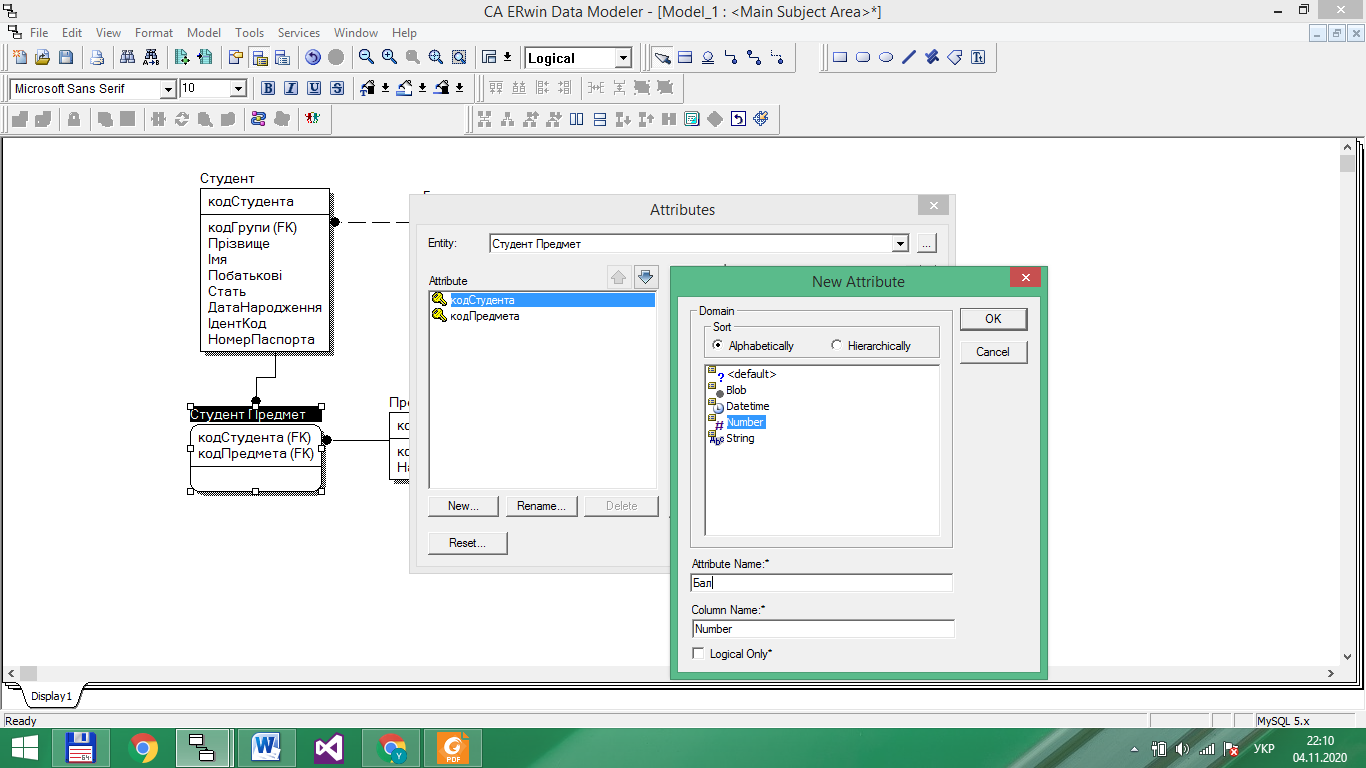


Рисунок 7 Додавання атрибутів



Рисунок 8 Додавання атрибутів до КМД

Таким чином маємо модель даних, що дозволяє зберігати дані про студентів, поточну успішність, а також викладачів та факультети.

Ця модель даних називається логічною моделлю даних (ЛМД)

ЛМД (рис. 8) має два типи звязків – пунктирний (неідентифікуючий) та

Сплошний (ідентифікуючий).

Неідентифікуючий звязок поєднує дві сутності. При цьому міграція первинного ключа головної сутності у підлеглу сутність призводить до додавання іще одного атрибуту у підлеглої сутності. Цей новий атрибут не є ключовим і не дозволяє ідентифікувати підлеглу сутність Тому такий зв'язок є неідентифікуючим.

Ідентифікуючий зв'язок між відношеннями Предмет <-> СтудентПредмет і Студент <-> СтудентПредмет створено завдяки міграціям первинного ключа до підлеглого відношення СтудентПредмет.

Ці атрибути КодСтудента та КодПредмета є зовнішніми ключами, що дозволяють визначити хто зі студентів який бал, коли і за який предмет отримав, ось чому у свою чергу пара зовнішніх ключей є складним певинним ключом.

Подальший етап – побудова фізичної моделі даних. Тобто ми маємо визначити типи даних атрибутів та довжину рядків.

Переходимо до фіз. моделі даних (ФМД)у засобі

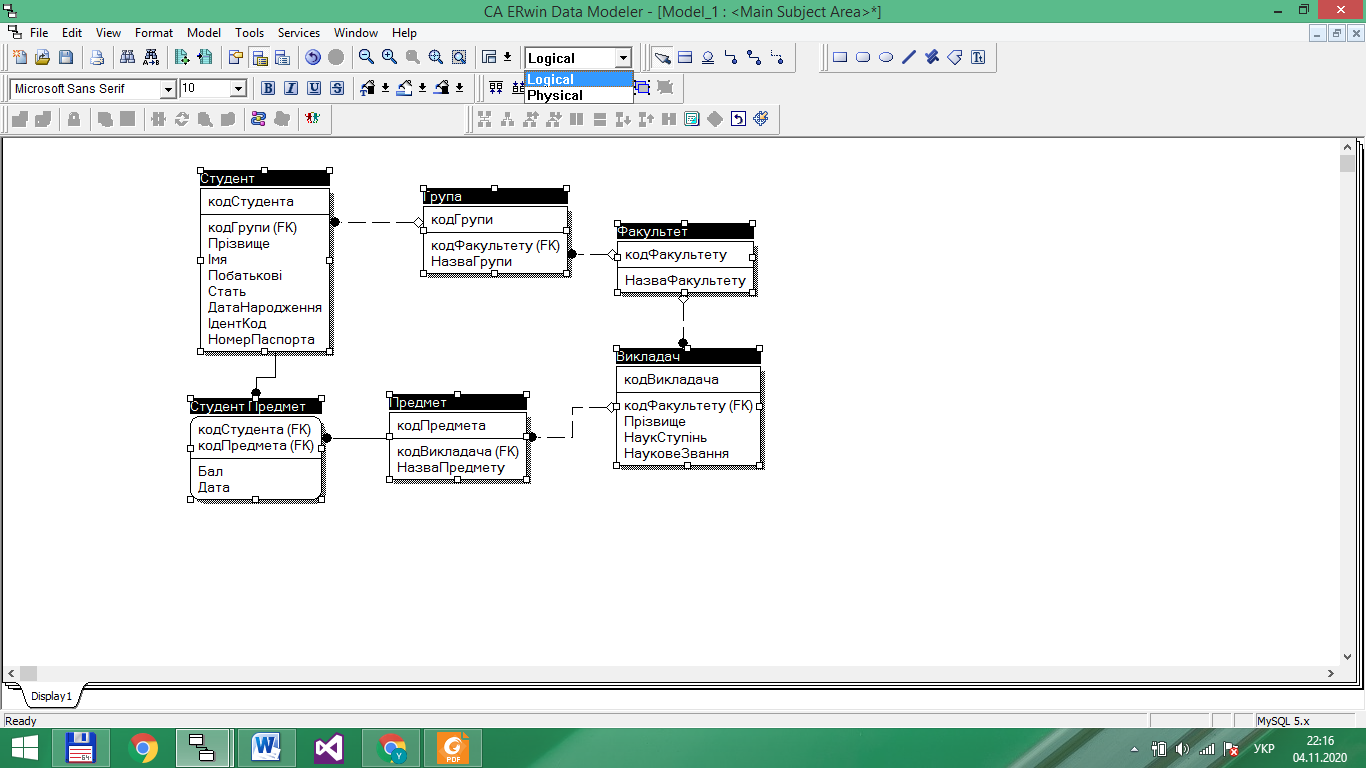


Рисунок 9 Перехід до ФМД

Сляд сказати, що ФМД оперує таблицями, стовпчики яких маєть атрибути певних типі даних

Визначамо відображення типів даних

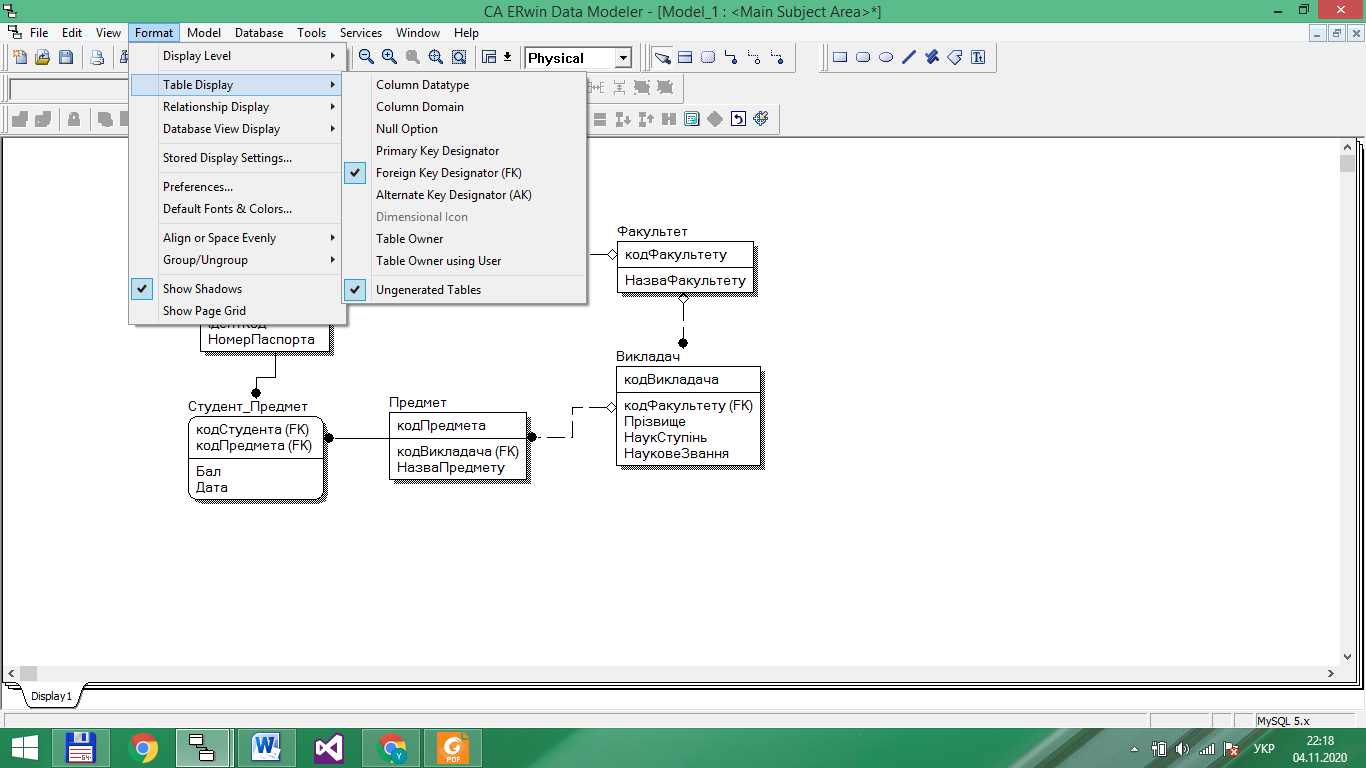


Рисунок 10 Визначення типів ФМД

Та покажемо типи стовпчиків таблиць



Рисунок 11 Типи атрибутів таблиць ФМД

Тут слід одразу сказати, що де-які атрибути у нас є не зовсім коректними.

Наприклад, навіщо стать використовує тип Integer, а назва групи складається з 20 символів - VARCHAR(20) ?

Для цього зміними типи даних атрибутів для того, що БД займала менше пам'яті.

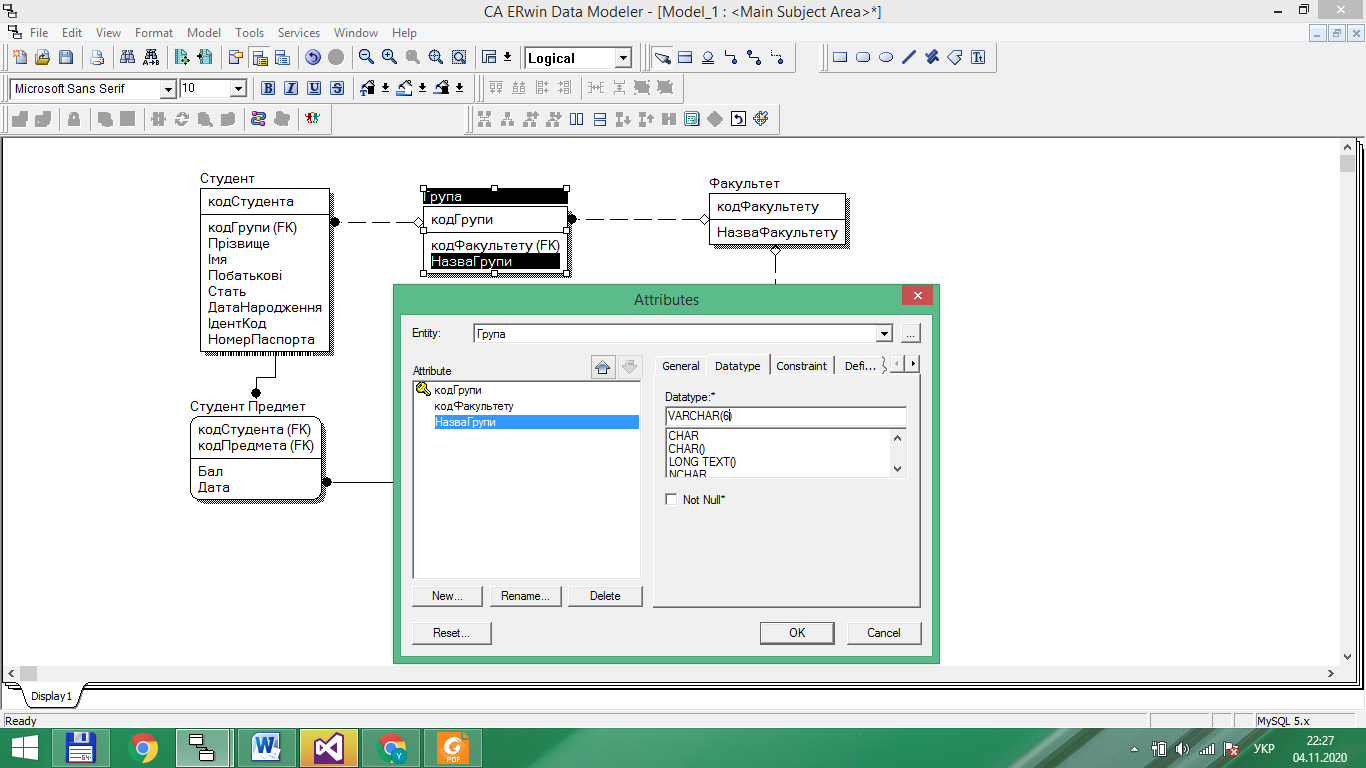


Рисунок 12 Зміна типів стовпчиків таблиць



Рисунок 13 Змінені типи стовпчиків таблиць ФМД

Подальше проектування полягає у перегляді коду для створення БД

Для цього у ФМД обираємо таке:

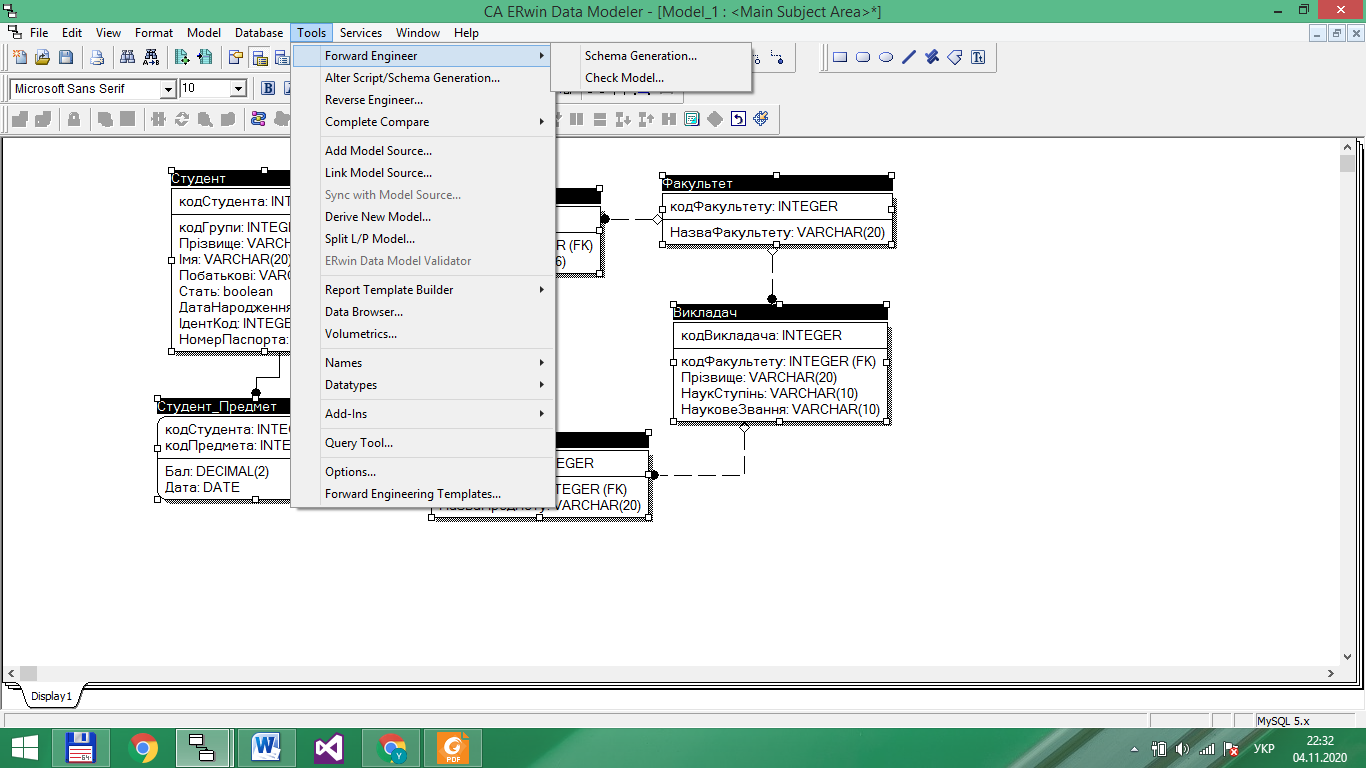


Рисунок 14 Перегляд коду генерації ФМД

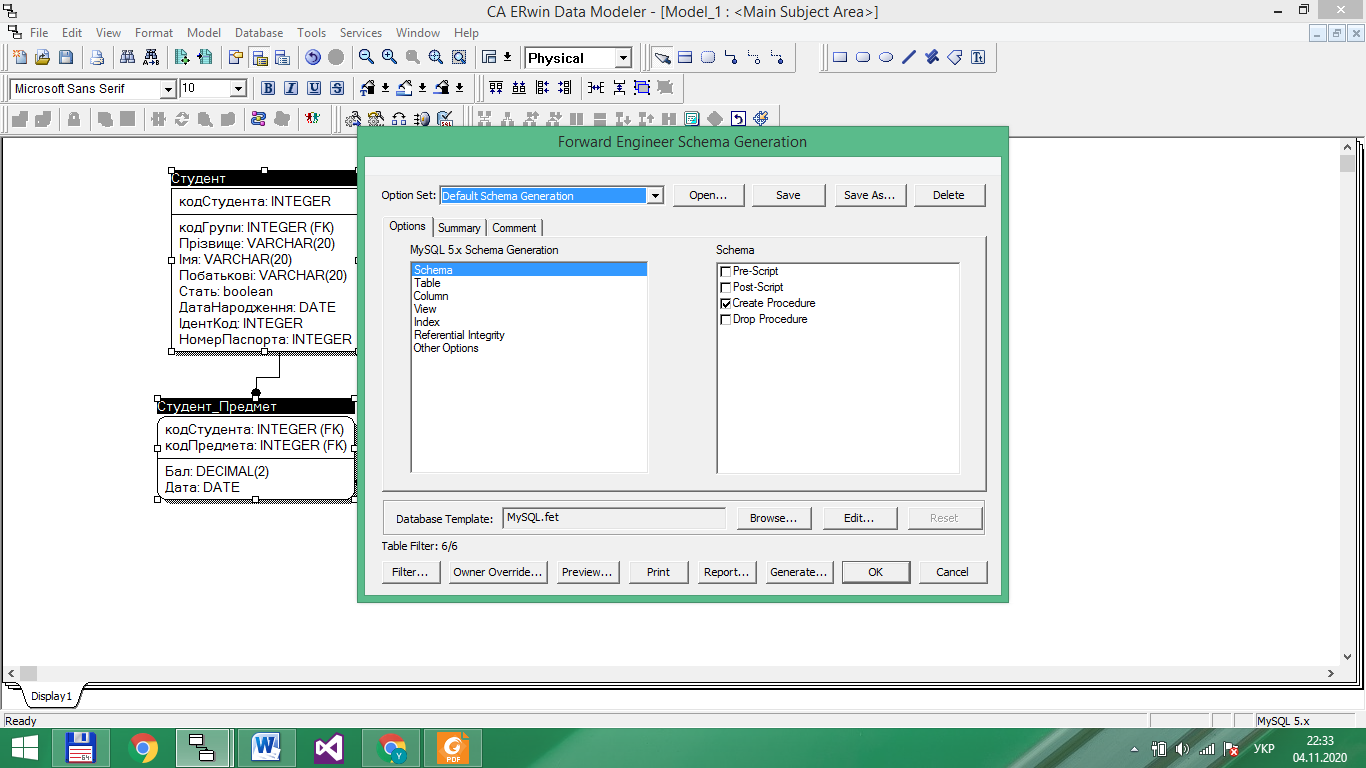


Рисунок 15 Генерація коду ФМД

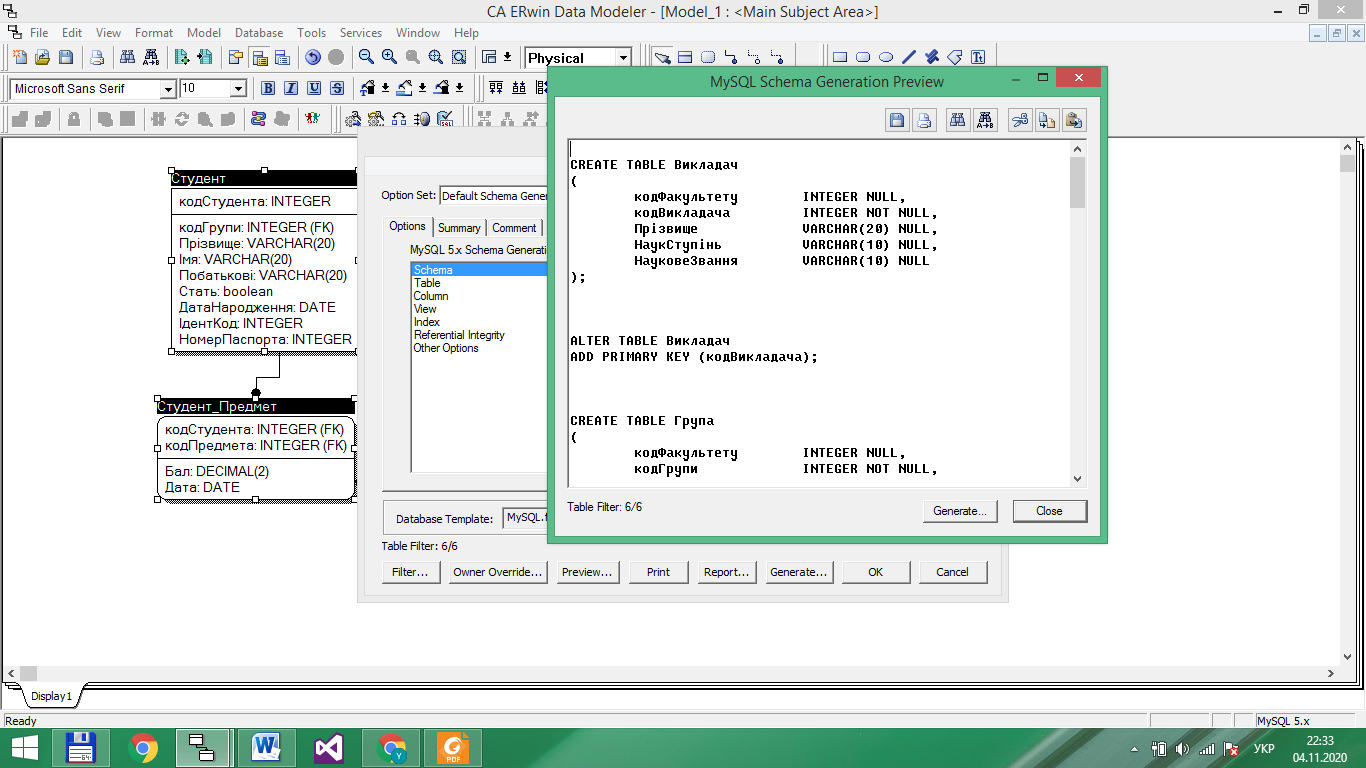


Рисунок 16 Генерація коду ФМД 2

Згенерований у ER WIN код завантажуємо на сервер БД:

CREATE TABLE Викладач

(

кодФакультету INTEGER NULL,

кодВикладача INTEGER NOT NULL,

Прізвище VARCHAR(20) NULL,

НаукСтупінь VARCHAR(10) NULL,

НауковеЗвання VARCHAR(10) NULL

);

ALTER TABLE Викладач

ADD PRIMARY KEY (кодВикладача);

CREATE TABLE Група

(

кодФакультету INTEGER NULL,

кодГрупи INTEGER NOT NULL,

НазваГрупи VARCHAR(6) NULL

);

ALTER TABLE Група

ADD PRIMARY KEY (кодГрупи);

CREATE TABLE Предмет

(

кодВикладача INTEGER NULL,

кодПредмета INTEGER NOT NULL,

НазваПредмету VARCHAR(20) NULL

);

ALTER TABLE Предмет

ADD PRIMARY KEY (кодПредмета);

CREATE TABLE Студент

(

кодГрупи INTEGER NULL,

кодСтудента INTEGER NOT NULL,

Прізвище VARCHAR(20) NULL,

Імя VARCHAR(20) NULL,

Побатькові VARCHAR(20) NULL,

Стать boolean NULL,

ДатаНародження DATE NULL,

ІдентКод INTEGER NULL,

НомерПаспорта INTEGER NULL

);

ALTER TABLE Студент

ADD PRIMARY KEY (кодСтудента);

CREATE TABLE Студент\_Предмет

(

кодСтудента INTEGER NOT NULL,

кодПредмета INTEGER NOT NULL,

Бал DECIMAL(2) NULL,

Дата DATE NULL

);

ALTER TABLE Студент\_Предмет

ADD PRIMARY KEY (кодСтудента,кодПредмета);

CREATE TABLE Факультет

(

кодФакультету INTEGER NOT NULL,

НазваФакультету VARCHAR(20) NULL

);

ALTER TABLE Факультет

ADD PRIMARY KEY (кодФакультету);

ALTER TABLE Викладач

ADD FOREIGN KEY R\_4 (кодФакультету) REFERENCES Факультет (кодФакультету);

ALTER TABLE Група

ADD FOREIGN KEY R\_3 (кодФакультету) REFERENCES Факультет (кодФакультету);

ALTER TABLE Предмет

ADD FOREIGN KEY R\_7 (кодВикладача) REFERENCES Викладач (кодВикладача);

ALTER TABLE Студент

ADD FOREIGN KEY R\_2 (кодГрупи) REFERENCES Група (кодГрупи);

ALTER TABLE Студент\_Предмет

ADD FOREIGN KEY R\_5 (кодСтудента) REFERENCES Студент (кодСтудента);

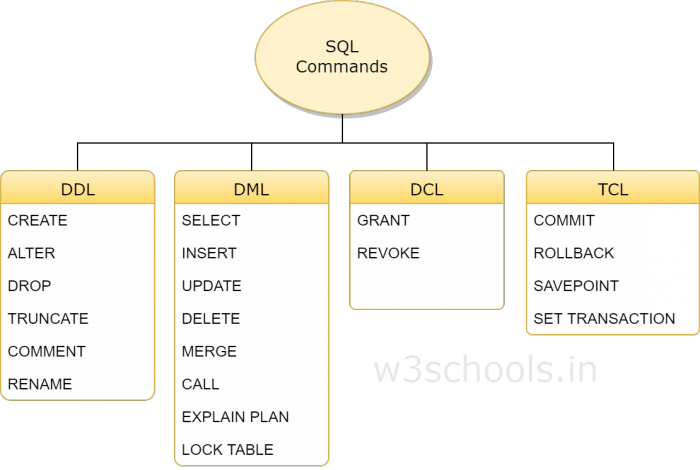
ALTER TABLE Студент\_Предмет

ADD FOREIGN KEY R\_9 (кодПредмета) REFERENCES Предмет (кодПредмета);

Рисунок 17 Згенерований за допомогою середовища ERWIN SQL код

# SQL – Structured Query Language

**Structured Query Language (SQL)** — мова структурованих запитів, що дозволяє створювати спеціальні запити (SQL інструкції) до БД для створення різноманітних об’єктів або отримання даних з цих обєктів та маніпулювання даними.



SQL – це сукупність операторів, що розподіляються на групи: DDL, DML, DCL, TCL.

**Мова визначення даних DDL – Data Definition Language**

**Data Definition Language (DDL)** – це підмножина, що дозволяє визначити структуру БД, та стоврювати, змінювати або вилучати елементи БД: схема, таблиця, ключ, індекс, послідовність (Autoincrement), користувач тощо.

В цю групу входять такі оператори:

* [CREATE](https://www.w3schools.in/mysql/php-mysql-create/) - to create a database and its objects like (table, index, views, store procedure, function, and triggers)
* ALTER - alters the structure of the existing database
* DROP - delete objects from the database
* TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
* COMMENT - add comments to the data dictionary
* RENAME - rename an object

CREATE – створити об’єкт БД;

Створення БД: CREATE DATABASE databasename;

Створення таблиці БД:

CREATE TABLE table\_name ( column1 datatype, column2 datatype, column3 datatype,....);

CREATE TABLE Persons (PersonID int, LastName varchar(255), FirstName varchar(255),Address varchar(255),City varchar(255) );

Створення таблиці на підставі іншої таблиці:

CREATE TABLE new\_table\_name AS SELECT column1, column2,... FROM existing\_table\_name WHERE ....;

* ALTER – змінити об’єкт БД;
* DROP – вилучити об’єкт БД.

Вилучити об’єкт БД

DROP DATABASE databasename;

DROP TABLE table\_name;

**Мова маніпулювання данними DML – Data Manipulation Language**

**Data Manipulation Language (DML)** – підмножина операторів для маніпулювання даними, що дозволяє: додавати (INSERT), змінювати (UPDATE), вилучати (DELETE), вибірати (SELECT) дані, тобто маніпулювати ними.

DML is short name of **Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

* [SELECT](https://www.w3schools.in/mysql/php-mysql-select/) - retrieve data from a database
* [INSERT](https://www.w3schools.in/mysql/php-mysql-insert/) - insert data into a table
* [UPDATE](https://www.w3schools.in/mysql/php-mysql-update/) - updates existing data within a table
* [DELETE](https://www.w3schools.in/mysql/php-mysql-delete/) - Delete all records from a database table
* MERGE - UPSERT operation (insert or update)
* CALL - call a PL/SQL or Java subprogram
* EXPLAIN PLAN - interpretation of the data access path
* LOCK TABLE - concurrency Control

**Мова керування доступом до даних DCL – Data Control Language**

**Data Control Language (DCL)** – підмножина операторів, що визначає доступ до даних: керування дозволами, що дозволяють або забороняють виконання певних операцій над об’єктами БД.

* GRANT – надає користувачу або групі користувачів дозвіл на певні операції з об’єктом БД;
* REVOKE – вилучає дозвіл , який було створено раніше;
* DENY– створення заборони, що має пріоритет над дозволами.

TCL – Transaction Control Language

**Мова керування транзакціями**

**Transaction Control Language (TCL)** – підмножина операторів, що керують транзакціями. Транзакція – це команда чи блок команд (інструкцій), які успішно завершуються як єдине ціле, при цьому БД фіксує всі внесені зміни, або відкидує їх (якщо зміна стану БД завершилася некоректно), тобто всі зміни, спричинені однією командою, яка входила до транзакції відміняються.

* BEGIN TRANSACTION – визначає початок транзакції;
* COMMIT TRANSACTION – завершує транзакцію та приймає всі зміни стану БД;
* ROLLBACK TRANSACTION – повертає назад всі зміни, що були виконані у поточні транзакції;
* SAVE TRANSACTION – створює проміжну точку збереження стану БД посеред виконання транзакції.

# Подальша нормалізація БД

Таким чином, ми маємо модель БД, основу якого складають відношення та їх зв’язки.

Нормалізація – процес побудови БД, у якому відношення мають відповідати певним правилам.

Спочатку треба всі відношення привести до найпростішої, першої нормальної форми.

# Перша нормальна форма

**Перша нормальна форма** (1НФ, 1NF) – найпростіша форма це властивість відношення у [реляційній баз даних](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D1%96%D0%B9%D0%BD%D0%B0_%D0%B1%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85). Відношення знаходиться в першій нормальній формі тоді і тільки тоді, коли атрибути відношення мають атомарний тип, тобто лише нероздільні значення, а значення кожного атрибута містить лише одне значення з цього домену. Крім того є певний набір атрибутів( один атрибут або кілька атрибутів), що приймає унікальне значення. Такий набор називають потенційним ключем. Один з потенційним ключем оголошується головним, або первинним (primary)

Такими чином, кожне відношення має первинний ключ.

Самі атрибути можуть належати різним доменам. Так існують: числовий, текстовий, дата тощо.

У свою чергу є типи даних: ціле число, текстовий рядок фіксованої довжини, рядок змінної довжини тощо.

Тип даних разом за накладеними на нього обмеженнями і є доменом.

Наприклад, домен температура людини: число з точністю 0.1 С, що має діапазон від 32С до 42 С

Зріст людини у см – ціле число, що має діапазон від 50 до 250 см тощо

Кожного разу, коли створюється відношення (а потім таблиця) треба уважно обирати певний домен. У першу чергу намагатися зробити так, щоб він вимагав ресурсів(пам’яті чи процесору).

А через необхідність зв’язування відношень шляхом міграції первинного ключа до підлеглого відношення (таблиці) треба обирати ключовими найпростіші та незмінні типи даних.

Так у відношенні студент можливо обрати первинний ключ(прізвище, ім’я. по-батькові, день народження), але це призведе до гальмування БД через значний зріст необхідних ресурсів для фіксації поточних результатів навчання у відношенні **студент-предмет** . Обирання первинним ключем ід.коду або номеру студентського квитка призведе до залежності БД від зовнішнього світу. Наприклад, хтось загубить квиток і це призведе до необхідності значної модифікації даних. Ось кому у нашому випадку було використано як первинний ключ штучний атрибут –сурогатний ключ – кодСтудента

Цей код слід зробити унікальним або вручну, або додати під час створення таблиці слово Unique або використати спеціальний тип даних Autoincrement. Цей тип дозволяє автоматично генерувати відповідне значення у запис про нового студента. Крім того атрибут займає чотири байти і дозволяє мінімізувати додаткові витрати на зв'язок сутностей студент та студентПредмет.

# Мова визначення даних DDL

## Створення та зміна елементів схеми даних

### Створення БД

The CREATE DATABASE command is used is to create a new SQL database.

The following SQL creates a database called "testDB":

CREATE DATABASE testDB;

**Tip:** Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases with the following SQL command: SHOW DATABASES;

### Створення таблиць

#### Створення таблиці з використанням іншої таблиці

A copy of an existing table can also be created using CREATE TABLE.

The following SQL creates a new table called "TestTables" (which is a copy of the "Customers" table):

CREATE TABLE TestTable AS SELECT customername, contactname FROM customers

#### Створення нових таблиць

CREATE TABLE Persons

( ID int NOT NULL UNIQUE,

LastName varchar(255) NOT NULL,

FirstName varchar(255), Age int

);

NOT NULL – означає, що поле не може бути пустим

UNIQUE - означає, що поле має уникальні значення , тобто такі, що не повторюються для різних персон

varchar(255) – текст довжиноб до 255 символів

Або так :

CREATE TABLE Persons

(

Personid int NOT NULL AUTO\_INCREMENT,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

PRIMARY KEY (Personid)

);

AUTO\_INCREMENT – означає, що це поле автоматично генерується як ціле число (4 байти). Максимальне значення близько 4 000 000 000. При вилученні певного запису, його номер вже не використовується. Тобто ця таблиця може жити не більше 4000 000 000 додавань записів.

Якщо є певні обмеження на атрибут відношення, то їх слід перевіряти. Наприклад:

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

CHECK (Age>=18)

);

Кожного разу коли буде додаватися запис, або змінюватися значення, вік перевірятиметься на досягнення 18

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255), Age int, City varchar(255),

CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes')

);

Було створено обмеження- CONSTRAINT CHK\_Person, що дозволяє первіряти не тільки вік, але й місто

Цей оператор задає значення полю у таблиці. Це поле автоматично приймає таке значення

ALTER TABLE Persons ALTER City SET DEFAULT 'Sandnes';

The following SQL creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

CREATE TABLE Persons (

PersonID int,

LastName varchar(255),

FirstName varchar(255),

Address varchar(255),

City varchar(255)

);

### Створення та вилучення тимчасових таблиць

Тимчасова таблиця визначається на період сесії. Коли сесія завершується то тимчасова таблиця автоматично ліквідується. Таблиця, створена в однії сесії недоступна до інших сесій. Пізні сесії можуть мати однойменні тимчасові таблиці.

Такі таблиці слід використовувати, якщо багато разів у збережених процедурах використовується мала підмножина великої таблиці. В таких випадках замість багаторазової фільтрації слід зберегти відфільтровані дані у тимчасовій таблиці, яку й використовувати у процедурах та функціях.

#### Важливі факти про тимчасові таблиці:

1. MySQL Temporary table has a decoupled relationship with the database schema, which means that when we drop the MySQL database, it does not necessarily drop the temporary tables within the database
2. The InnoDB database engine does not support the compressed, temporary table. If you had enabled the **InnoDB\_strict\_mode** parameter, then it will return an error. If you disabled the **InnoDB\_strict\_mode** parameter, MySQL will issue a warning and create a table with a non-compressed row format
3. To create a temporary table within the stored procedure, the user must have **create temporary table** privilege. Once it is granted, then to update, delete, or insert the data within that temporary table does not require any special privileges. There is one implication of this behavior. Suppose the stored procedure that uses a temporary table and a **USER 1**who had developed the procedure has the **create temporary table** privilege. Now, **USER 2** tries to execute it, but it does not have the **create temporary table** permission. In such cases, MySQL uses the privileges of the **USER 1** to create the temporary table. Once the procedure completes successfully, the privileges will be reverted
4. You cannot create a temporary table based on the definition that the table already has, and it resides on MySQL tablespace, a general tablespace, or InnoDB tablespace. To create a temporary table based on the definition of another table, you must use the following syntax

|  |  |
| --- | --- |
| 1 | Select temporary table.. select \* from |

1. Unlike regular create table statements, the Create temporary table statement does not cause the implicit commit
2. The temporary table can have the same name as the MySQL regular table has. For example, even though a table named employee exists on the database, you can create a temporary table named employee in the database. In such cases, the MySQL table becomes inaccessible. Once the temporary table is dropped, the physical table can be accessed again. So it is always a good practice to create a temporary table with some unique name or make sure that the application drops the temporary table immediately after its purpose is completed

#### Приклад тимчасової таблиці

The syntax to create a temp table is as follows:

create temporary table tblemployee

    (

    Column\_1 datatype,

    Column\_2 datatype,

    Column\_3 datatype,

    …

    )

As you can see, the syntax to create a temporary table is the same as the syntax of creating a MySQL table. The only difference is that you must specify the **temporary** keyword between **create** and **table** keywords. This is very important because if you forget to add the keyword, it creates a regular MySQL table.

Now, let us walk through some examples of the temporary table. I am going to demonstrate the

following use cases:

1. Create a temp table and insert data in the table
2. Create a temp table using the output of the select statement
3. Drop the temp table
4. Create a temp table using the structure of another table

#### Створення тимчасової таблиці і додання данних

To create a new temp table named tblemployee on the MySQL database, execute the following query:

Сreate temporary table tblemployee

(

id int auto\_increment Primary key,

emp\_name varchar(500),

emp\_address varchar(500),

emp\_dept\_id int

)

Execute the following query to insert the data in the temp table.

insert into tblemployee(emp\_name, emp\_address,emp\_dept\_id)

values

('Nisarg Upadhyay','Mehsana',10),

('Nirali U','Mehsana',20),

('Milan J','Mehsana',5);

Once data is inserted in the temp table, execute the following query to verify that data have been inserted.

|  |  |
| --- | --- |
| 1 | Select \* from tblemployee; |

Now, to verify the scope of the tblemployee table, let us run the same query from the MySQL command-line client. To do that, open it, provide a password to connect to the MySQL Server, and execute the following queries:

|  |  |
| --- | --- |
| 1  2 | Use sakila;  Select \* from tblemployee; |

#### Створення тимчасової таблиці на підставі оператору select

Suppose I want to create a temp table using the output of the select query. For example, I want to insert the details of the films table with its category in a temp table named **temp\_film\_details**. To do that, execute the following query.

Сreate temporary table tbl\_film\_details

select title, description,release\_year,rental\_rate ,rental\_duration,special\_features,replacement\_cost,c.name from film a

inner join film\_category b on a.film\_id=b.film\_id

inner join category c on b.category\_id=c.category\_id

Limit 10;

#### Вилучення тимчасових таблиць

To drop the temp table, we can use the following syntax:

Drop temporary table tbl\_film\_details

As I mentioned that the on a database a temporary table and an InnoDB table can have same name so make sure you specify the temporary keyword; otherwise, if a table with the same will be dropped

### Створення подання View

Іноді доцільно створити спеціальний елемент подання View. Це елемент зберігає завасно відібрані з інших таблиць. Якщо дані в таблицях змінюється, то автоматично змінюється і подання. Це зменшує час оброблення даних і дозволяє спростити код.

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

CREATE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

**Note:** A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

The following SQL creates a view that shows all customers from Brazil:

CREATE VIEW [Brazil Customers] AS SELECT CustomerName, ContactName  
FROM Customers WHERE Country = 'Brazil';

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

CREATE VIEW [Products Above Average Price] AS SELECT ProductName, Price FROM Products WHERE Price > (SELECT AVG(Price) FROM Products);

Створення або оновлення подання

CREATE OR REPLACE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

The following SQL adds the "City" column to the "Brazil Customers" view:

CREATE OR REPLACE VIEW [Brazil Customers] AS  
SELECT CustomerName, ContactName, City  
FROM Customers  
WHERE Country = 'Brazil';

### Зміна стовпчика таблиці

The ALTER COLUMN command is used to change the data type of a column in a table.

The following SQL changes the data type of the column named "BirthDate" in the "Employees" table to type year:

ALTER TABLE Employees ALTER COLUMN BirthDate year;

### Зміна таблиці ALTER TABLE

The ALTER TABLE command adds, deletes, or modifies columns in a table.

The ALTER TABLE command also adds and deletes various constraints in a table.

The following SQL adds an "Email" column to the "Customers" table:

ALTER TABLE Customers ADD Email varchar(255);

The following SQL deletes the "Email" column from the "Customers" table:

ALTER TABLE Customers DROP COLUMN Email;

## Вилучення всіх даних з таблиці TRUNCATE TABLE

The TRUNCATE TABLE command deletes the data inside a table, but not the table itself. The following SQL truncates the table "Categories":

TRUNCATE TABLE Categories;

## Вилучення елементів схеми даних

Якщо Ви випадково створили помилково базу даних, таблицю, чи її стовпчик, то БД дозволяє Вам вилучити ці елементи

### Вилучення стовпчику з таблиці

The DROP COLUMN command is used to delete a column in an existing table.

The following SQL deletes the "ContactName" column from the "Customers" table:

ALTER TABLE Customers DROP COLUMN ContactName;

### Вилучення таблиці

The DROP TABLE command deletes a table in the database.

The following SQL deletes the table "Shippers":

DROP TABLE Shippers;

**Note:** Be careful before deleting a table. Deleting a table results in loss of all information stored in the table!

### Вилучення подання (VIEW)

The DROP VIEW command deletes a view.

The following SQL drops the "Brazil Customers" view:

DROP VIEW [Brazil Customers];

### Вилучення БД

The DROP DATABASE command is used is to delete an existing SQL database.

The following SQL drops a database named "testDB":

DROP DATABASE testDB;

**Note:** Be careful before dropping a database. Deleting a database will result in loss of complete information stored in the database!

## SQL обмеження

Constraints (обмеження) можуть бути визначені під час створення таблиці оператором CREATE TABLE

Або після створення таблиці за допомого оператору ALTER TABLE .

CREATE TABLE table\_name (

column1 datatype *constraint*,

column2 datatype *constraint*,

column3 datatype *constraint*,

....  
);

SQL constraints використовують для визначення правил перевірки даних, що є у таблиці.

Constraints гарантують надійність та достовірність даних у таблиці. Якщо є якесь протиріччя між обмеженнями та даними, то дія оператору припиняється.

Існують такі види обмежень :

* [NOT NULL](https://www.w3schools.com/sql/sql_notnull.asp) – гарантує, що дані не можуть мати значення NULL, тобто дані обов’язково слід внести у стовпчик
* [UNIQUE](https://www.w3schools.com/sql/sql_unique.asp) – гарантує, шо всі значення цього стовпчика або стовпчиків таблиці мають бути різними
* [PRIMARY KEY](https://www.w3schools.com/sql/sql_primarykey.asp) – Є комбінацією NOT NULL та UNIQUE. Визначає, що кожний рядок таблиці є унікальним
* [FOREIGN KEY](https://www.w3schools.com/sql/sql_foreignkey.asp) – Унікальний ідентифікатор рядку(запису) головної(іншої) таблиці відповідає кільком рядкам цієї(підлеглої) таблиці
* [CHECK](https://www.w3schools.com/sql/sql_check.asp) –Гарантує, що всі стовпчика таблиці задовольняють певному правилу - умові
* [DEFAULT](https://www.w3schools.com/sql/sql_default.asp) – Визначає значення за правилом замовчування, за яким дані приймають певне значення
* [INDEX](https://www.w3schools.com/sql/sql_create_index.asp) - використовується для пришвидшення бази даних

### Обмеження NOT NULL

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

#### Визначення обмеження NOT NULL під час CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

CREATE TABLE Persons (ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255) NOT NULL,

Age int

);

#### Зміна обмеження NOT NULL за допомогою ALTER TABLE

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

ALTER TABLE Persons MODIFY Age int NOT NULL

### Обмеження унікальності UNIQUE

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

#### Визначення обмеження UNIQUE під час CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

UNIQUE (ID));

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

CONSTRAINT UC\_Person UNIQUE (ID,LastName)

);

#### Зміна обмеження UNIQUE за допомогою ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

ALTER TABLE Persons ADD UNIQUE (ID);

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons ADD CONSTRAINT UC\_Person UNIQUE (ID,LastName);

#### Вилучення обмеження UNIQUE

To drop a UNIQUE constraint, use the following SQL:

ALTER TABLE Persons DROP INDEX UC\_Person;

### Обмеження первинного ключа PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

#### Визначення обмеження PRIMARY KEY під час CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

CREATE TABLE Persons (ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),Age int,PRIMARY KEY (ID) );

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

CREATE TABLE Persons (

¶ID int NOT NULL,

¶LastName varchar(255) NOT NULL,

¶FirstName varchar(255),

¶Age int,¶CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName)  
);

**Note:** In the example above there is only ONE PRIMARY KEY (PK\_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

#### Зміна обмеження PRIMARY KEY за допомогою ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

ALTER TABLE Persons ADD PRIMARY KEY (ID);

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

**Note:** If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

#### Вилучення обмеження PRIMARY KEY

To drop a PRIMARY KEY constraint, use the following SQL:

ALTER TABLE Persons DROP PRIMARY KEY;

### Обмеження зовнішнього ключа FOREIGN KEY

Зовнішній ключ використовується для зв’язку двох таблиць.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Look at the following two tables:

"Persons" table:

|  |  |  |  |
| --- | --- | --- | --- |
| PersonID | LastName | FirstName | Age |
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

"Orders" table:

|  |  |  |
| --- | --- | --- |
| OrderID | OrderNumber | PersonID |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

#### Визначення обмеження FOREIGN KEY під час CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

CREATE TABLE Orders (

¶OrderID int NOT NULL,

¶OrderNumber int NOT NULL,

¶PersonID int,

¶PRIMARY KEY (OrderID),

¶FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)  
);

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

CREATE TABLE Orders (

¶OrderID int NOT NULL,

¶OrderNumber int NOT NULL,

¶PersonID int,

¶PRIMARY KEY (OrderID),

¶CONSTRAINT FK\_PersonOrder FOREIGN KEY (PersonID)

¶REFERENCES Persons(PersonID)

);

#### Зміна обмеження FOREIGN KEY за допомогою ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

ALTER TABLE Orders

ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Orders

ADD CONSTRAINT FK\_PersonOrder FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

#### Вилучення обмеження завнішнього ключа FOREIGN KEY

To drop a FOREIGN KEY constraint, use the following SQL:

ALTER TABLE Orders DROP FOREIGN KEY FK\_PersonOrder;

### Обмеження додаткової умови CHECK

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

#### Визначення обмеження CHECK під час CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

CREATE TABLE Persons ( ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int, CHECK (Age>=18));

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

CREATE TABLE Persons ( ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255),

CONSTRAINT CHK\_Person CHECK (Age>=18 AND City='Sandnes')

);

#### Зміна обмеження CHECK за допомогою ALTER TABLE

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

ALTER TABLE Persons ADD CHECK (Age>=18);

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

ALTER TABLE Persons ADD CONSTRAINT CHK\_PersonAge CHECK (Age>=18 AND City='Sandnes');

#### Вилучення обмеження додаткової умови CHECK

To drop a CHECK constraint, use the following SQL:

ALTER TABLE Persons DROP CHECK CHK\_PersonAge;

### Обмеження DEFAULT

The DEFAULT constraint is used to provide a default value for a column.

The default value will be added to all new records IF no other value is specified.

#### Визначення обмеження DEFAULTпід час CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255) DEFAULT 'Sandnes' );

Обмеження DEFAULT може також використано під час внесення даних, наприклад за допомогою функції GETDATE():

CREATE TABLE Orders (ID int NOT NULL,

OrderNumber int NOT NULL,

OrderDate date DEFAULT GETDATE()

);

#### Зміна обмеження DEFAULT за допомогою ALTER TABLE

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

ALTER TABLE Persons ALTER City SET DEFAULT 'Sandnes';

#### Вилучення обмеження DEFAULT

To drop a DEFAULT constraint, use the following SQL:

ALTER TABLE Persons ALTER City DROP DEFAULT;

### Створення індексу CREATE INDEX

Індексування це метод, що пришвидшує доступ до записів. Індекс створює окремо для кожного атрибуту або групи атрибутів.

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

#### Створення звичайного індексу

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name ON table\_name (column1, column2, ...);

#### Створення унікального індексу

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name ON table\_name (column1, column2, ...);

**Note:** The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx\_pname ON Persons (LastName, FirstName);

#### Два способи створення індексу

Існує два способи створення індексу: або під час створення таблиці оператором CREATE TABLE або післі використання оператору CREATE INDEX.

Синтаксис створення індексу у MySQL:

CREATE TABLE table\_name

(

column1 datatype [ NULL | NOT NULL ],

column2 datatype [ NULL | NOT NULL ],

...

column\_n datatype [ NULL | NOT NULL ],

INDEX index\_name [ USING BTREE | HASH ]

(index\_col1 [(length)] [ASC | DESC],

index\_col2 [(length)] [ASC | DESC],

...

index\_col\_n [(length)] [ASC | DESC])

);

або

CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index\_name

[ USING BTREE | HASH ]

ON table\_name

(index\_col1 [(length)] [ASC | DESC],

index\_col2 [(length)] [ASC | DESC],

...

index\_col\_n [(length)] [ASC | DESC]);

UNIQUE – необов’язково . Цей модифікатор означає , що значення у індексованому стовпчику є унікальними

**FULLTEXT – необов’язково .** FULLTEXT модифікатор індексує стовпчик та не дозволяє префіксування. InnoDB та MyISAM таблиці підтримують цю опцію.

**SPATIAL необов’язково**. SPATIAL модифікатор індексує стовпчик, що не має NULL значення. InnoDB та MyISAM таблиці підтримують цю опцію

**index\_name** назва індексу.

**table\_name** назва таблиці у якій створюється індекс

index\_col1, index\_col2, ... index\_col\_n стовпчики, використані для індексу.

**length - необов’язково**.Визначає, що тільки префікс стовпчика проіндексовано, а не його значення. Для не бінарних рядків визначає кількість символів, що будуть проіндексовані. Для бінарних – кількість проіндексовинх байт.

ASC - необов’язково. Індексація у порядку а-я

DESC необов’язково. Індексація у порядку я-а

Приклад створення індексованої таблиці

CREATE TABLE contact

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id),

INDEX contacts\_idx (last\_name, first\_name)

);

Створюється таблиця contact, що індексована індексом contacts\_idx, який складається з двох полів last\_name та first\_name.

Створення не індексованої таблиці , а потім створення індексу

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

CREATE INDEX contacts\_idx

ON contacts (last\_name, first\_name);

#### Унікальний індекс (Unique Index)

Для створення унікального ундексу у таблиці, необхідно використати слово UNIQUE.

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id),

UNIQUE INDEX contacts\_idx (last\_name, first\_name)

);

або

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

CREATE UNIQUE INDEX contacts\_idx ON contacts (last\_name, first\_name);

contacts\_idx приклади створюють той самий індекс, але комбінація last\_name та first\_name є унікальною і не дублюється. Це значно підвищує цілісність ДБ, якщо індексована комбінація не є часткою первинного

#### Вилучення індексу (Drop Index)

DROP INDEX index\_name ON table\_name;

Приклад:

DROP INDEX contacts\_idx ON contact;

Вилучається індекс contacts\_idx з таблиці contact.

#### Перейменування індексу

ALTER TABLE table\_name RENAME INDEX index\_name TO new\_index\_name;

table\_name назва індексованої таблиці

index\_name назва індексу, що буде перейменовано

new\_index\_name нова назва індексу

ALTER TABLE contact RENAME INDEX contacts\_idx TO contacts\_new\_index;

## Типи даних MySQL Data Types (Version 8.0)

MySQL має три головних типи даних: string - текст, numeric - число, date and time – дата час.

### Визначення тексту мовою DDL

|  |  |
| --- | --- |
| CHAR(size) | Текстовий рядок фіксованої довжини. size визначає довжину рядка –від 0 до 255. По замовчуванню довжина 1 |
| VARCHAR(size) | Текстовий рядок змінної довжини. size визначає максимальну довжину рядка –від 0 до 65535 символів |
| BINARY(size) | Цей тип еквівалентний CHAR(), але зберігає бінарні байти рядку. size визначає довжину рядка у байтах. По замовчуванню довжина 1 |
| VARBINARY(size) | Цей тип еквівалентний VARCHAR(),але зберігає бінарні байти рядку. size визначає максимальну довжину рядка у байтах.. |
| TINYBLOB | Використовується для BLOBs (Binary Large OBjects). Maксимальна довжина: 255 байт |
| TINYTEXT | Містить рядок максимальною довжиною до 255 символів |
| TEXT(size) | Містить рядок максимальною довжиною до 65,535 байт |
| BLOB(size) | Використовується для BLOBs (Binary Large OBjects). Містить до 65,535 байт даних |
| MEDIUMTEXT | Містить рядок з максимальною довжиною 16,777,215 символів |
| MEDIUMBLOB | Використовується для BLOBs (Binary Large OBjects). Містить до 16,777,215 байт даних |
| LONGTEXT | Містить рядок з максимальною довжиною 4,294,967,295 символів |
| LONGBLOB | Використовується для BLOBs (Binary Large OBjects). Містить до 4,294,967,295 байт даних |
| ENUM(val1, val2, val3, ...) | Текст може мати тільки одне значення, що обирається з можливих значень. Список може мати до 65535 значень. Якщо значення відсутнє у списку, додається у таблицю пусте значення. Значення у списку відсортовані у тому порядку, як їх було внесено |
| SET(val1, val2, val3, ...) | Текст може мати від 1 до 64 значень, що обираються серед списку можливих. |

### Визначення чисел мовою DDL

|  |  |
| --- | --- |
| BIT(size) | Бітове поле. size визначає кількість біт на одне значення від1 до 64. По замовчуванню size = 1. |
| TINYINT | Дуже мале ціле число -128 127. Беззнакове 0 255. |
| BOOL | Нуль відповідає false, ненуль - true. |
| BOOLEAN | Еквівалентно BOOL |
| SMALLINT | Мале ціле. Знакове -32768 .. 32767. Беззнакове 0 to 65535. |
| MEDIUMINT | Середнє ціле. Знакове -8388608 8388607. Беззнакове 0 .. 16777215. |
| INT | Знакове ціле -2147483648 до 2147483647. Беззнакове ціле 0 4294967295. |
| INTEGER | Еквівалентне INT |
| BIGINT | Велике ціле число. Знакове -9223372036854775808 до 9223372036854775807. Беззнакове 0 до 18446744073709551615. |
| FLOAT(size, d) | Число з плаваючою комою. Загальна кількість цифр визначається size. Кількість цифр після коми - dr. Цей синтаксис застарілий і буде змінено |
| FLOAT(p) | Число з плаваючою комою. Якщо р 0 24 то тип стає FLOAT(). Якщо p 25 до 53, тип даних стає DOUBLE() |
| DOUBLE(size, d) | Число з плаваючою комою. Нормального розміру. Size визначає кількість цифр. Кількість цифр після коми d |
| DECIMAL(size, d) | Число з фіксованою комою. Загальна кількість цифр визначається як size. Кількість цифр після коми - d. Максимальна кількість цифр 65. Максимальне значення d 30. За правилами замовчування size 10, а d 0. |
| DEC(size, d) | Еквівалентно DECIMAL(size,d) |

**Зауваження:** Всі числові дані можуть мати додаткові опції: UNSIGNED (беззнакове число) або ZEROFILL (заповнюються старші нулі) .

Якщо ви додаєте UNSIGNED , то MySQL забороняє вносити значення менше нуля у цей атрибут.

Якщо ви додаєте ZEROFILL, то MySQL автоматично додає також UNSIGNED до цього атрибуту.

Характеристики цілих чисел

| Tип | Пам'ять (байт) | Мінімальне значення  знакове | Мінімальне значення  безнакове | Максімальне значення  знакове | Максімальне значення  безнакове |
| --- | --- | --- | --- | --- | --- |
| TINYINT | 1 | -128 | 0 | 127 | 255 |
| SMALLINT | 2 | -32768 | 0 | 32767 | 65535 |
| MEDIUMINT | 3 | -8388608 | 0 | 8388607 | 16777215 |
| INT | 4 | -2147483648 | 0 | 2147483647 | 4294967295 |
| BIGINT | 8 | -263 | 0 | 263-1 | 264-1 |

### Визначення дат мовою DDL

#### Тип даних для визначення Дати та/або Часу:

|  |  |
| --- | --- |
| DATE | Дата, що має формат: YYYY-MM-DD. Підтримує значення '1000-01-01' до '9999-12-31' |
| DATETIME(fsp) | Комбінація дати та часу, що має формат: YYYY-MM-DD hh:mm:ss.  Діапазон '1000-01-01 00:00:00' до '9999-12-31 23:59:59'.  Додавання слова DEFAULT та ON UPDATE у стовпчик визначає автоматичну ініціалізацію поточною датою та часом |
| TIMESTAMP(fsp) | Часовий штамп. TIMESTAMP це значення запомятовує як число кількість секунд з початку епохи Unix ('1970-01-01 00:00:00' UTC). Формат: YYYY-MM-DD hh:mm:ss.  Діапазон '1970-01-01 00:00:01' UTC до '2038-01-09 03:14:07' UTC.Автоматична інціалізація та оновлення поточною датою та часом може бути визначено та: using DEFAULT CURRENT\_TIMESTAMP та ON UPDATE CURRENT\_TIMESTAMP під час визначення стовпчику |
| TIME(fsp) | Час. Формат: hh:mm:ss. Підтримується діапазон '-838:59:59' дo '838:59:59' |
| YEAR | Рік у форматі чотирьох цифр.  Діапазон: 1901 від 2155 та 0000. MySQL 8.0 не підтримує формат року у двох цифрах. |

найскладнішим у роботі з датами є необхідність збігу форматів дати, що було прийнято під час створення стовпчика таблиці та формату дати, яку вносить користувач.

Доки ваша дані містять лише дату ваш запит буде діяти так, як ви очікували. Коли ж ви додасте до даних час зявляться труднощі.

#### Формат дати (Date)

##### MySQL формат

MySQL використовує такі типи даних для збереження дати або дати/часу :

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MI:SS
* TIMESTAMP - format: YYYY-MM-DD HH:MI:SS
* YEAR - format YYYY or YY

##### MS SQL формат

SQL Server має такі формати:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MI:SS
* SMALLDATETIME - format: YYYY-MM-DD HH:MI:SS
* TIMESTAMP - format: a unique number

**Зауваження:** Тип формату дати обирається під час створення стовпчика таблиці. Ви можете легко порівнювати дати, якщо вони не містять часової компоненти!

Припустимо , що ви маєте таку таблицю «Замовлення»:

|  |  |  |
| --- | --- | --- |
| кодЗамовлення | НазваПродукту | ДатаЗамовлення |
| 1 | Юшка | 2020-10-03 |
| 2 | Холодець | 2020-10-09 |
| 3 | Шашлик | 2020-10-03 |
| 4 | Плов | 2020-10-29 |

І хотіли б надрукувати записи про замовлення від "2020-10-03".

Для цього слід використати такий оператор SELECT:

SELECT \* FROM Замовлення WHERE ДатаЗамовлення='2020-10-03'

Результат роботи такого оператора буде таким:

|  |  |  |
| --- | --- | --- |
| кодЗамовлення | НазваПродукту | ДатаЗамовлення |
| 1 | Юшка | 2020-10-03 |
| 3 | Шашлик | 2020-10-03 |

Тепер припустимо, таблиця «Замовлення» виглядає так:

|  |  |  |
| --- | --- | --- |
| кодЗамовлення | НазваПродукту | ДатаЗамовлення |
| 1 | Юшка | 2020-10-03 13:23:44 |
| 2 | Холодець | 2020-10-09 15:45:21 |
| 3 | Шашлик | 2020-10-03 11:12:01 |
| 4 | Плов | 2020-10-29 14:56:59 |

Якщо ми використаємо той же самий оператор, то він не створить нам результат, тому, що запит шукатиме дати без часу

**Висновок:** Для підтримки ваших запитів простими і легкими у використанні, не дозволяйте використовувати часову компоненту у ваших датах!

# Подальша нормалізація БД 2

Перша нормальна форма є істотною властивістю відношення у реляційній базі даних. Взагалі, нормалізація даних - це процес представлення бази даних з точки зору відносин у стандартних нормальних формах, де перша нормальна є мінімальною вимогою. Критерії першої нормальної форми:

* Кожна таблиця повинна мати первинний тип: мінімальний набір стовпчиків, які ідентифікують запис.
* Атомарність: кожен атрибут повинен мати лише одне значення, а не множину значень.

Як ми вже казали, **відношення перебуває у 1NF, якщо всі атрибути відношення є простими ( атомарні),** тобто не мають компонентів. Іншими словами, домен атрибута повинен складатися з неподільних значень і не може містити в собі безліч значень із більше елементарних доменів**, а також має первинний ключ**. Таким чином, ми можемо сказати, що існує функціональна залежність між первинним ключем, та атрибутами, що не ввійшли у його склад. Такі атрибути називають не ключовими. У свою чергу кажуть, що відношення має свій заголовок – перелік атрибутів. Дані дадаються як кортежі – упорядковані множини значень атрибутів. У нашому випадку інформація, наприклад, про студентів міститься у кортежах, а у таблицях БД вона міститься у записах.

## Зміна стану БД

Зміна стану БД має місце тільки після зміни даних у таблицях, тобто виконання операторів: Insert, Update, Delete

Окремі записи до таблиці додаються оператором Insert, оновлюються оператором Update, а вилучаються з таблиці оператором Delete. Якщо БД складається з одного відношення( таблиці), то всі ці оператори ми легко можемо виконати. Але коли є зв’язки, то занесення/модифікація даних можливі тільки починаючи з головної таблиці (Група), а потім – підлеглої (Студент). Вилучати дані навпаки треба починати з підлеглої (студент), а потім з головної таблиці (Група)

### Внесення даних

Для внесення даних використовується оператор INSERT INTO, який дозволяє внести новий запис у таблицю. Для цього слід набрати такий текст

INSERT INTO table\_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);

Якщо ж ви додаєте дані до всіх стовпчиків таблиці, то вам не треба писати їх назви у запиті, але необхідно гарантувати той самий порядок, що збігається з порядком стовпчиків у таблиці. Такий оператор має синтаксис:

INSERT INTO table\_name VALUES (value1, value2, value3, ...);

Для перевірки внесення даних слід скористатися запитом, що повертає всі стовпчики таблиці

SELECT \* FROM table\_name

Для прикладу внесення даних можно розглянути цей запит:

INSERT INTO Customers

(CustomerName, ContactName, Address, City, PostalCode, Country) VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

Якщо таблиця має поле, що значення якого автоматично генерується, то таке поле просто не слід вказувати у переліку

Приклад

Ми маємо таблицю, що створюється оператором:

CREATE TABLE Person (

Personid int NOT NULL AUTO\_INCREMENT,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int, PRIMARY KEY (Personid) );

Для додавання запису у цю таблицю поле Personid не слід перелічувати, тому, що його значення генерується автоматично:

INSERT INTO Persons (FirstName,LastName) VALUES ('Lars','Monsen');

### Модифікація даних

Модифікація даних виконується за допомогою оператора Update

UPDATE table\_name SET column1 = value1, column2 = value2, ... WHERE condition;

**Зауваження: Треба обережно оновлювати записи у таблиці**! Особливо , якщо є умова оновлення, яка записується після слова. Якщо вине визначите мову оновлення, то всі записи у таблиці будуть змінені!

Приклад оновлення одного запису Customer таблиці:

UPDATE Customer SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

Оновлення багатьох записів таблиці

Якщо визначену умова оновлення ( за допомогою слова WHERE), це може призвести до оновлення багатьох записів.

UPDATE Customers SET ContactName='Juan' WHERE Country='Mexico';

Цей оператор змінює Контактне ім’я всім жителям Мексики

### Видалення записів з таблиці оператором Delete

Оператор DELETE використовують для видалення існуючих записів у таблиці.

DELETE FROM table\_name WHERE condition;

**Зауваження:** Треба бути обережним з видаленням Слово WHERE з його умовою визначає ті записи, які будуть видалені. Якщо ви пропустите умову, то будуть видалені всі записти таблиці!

Видалення одного запису у таблиці:

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

Видалення всіх записів таблиці

DELETE FROM table\_name;

Цей приклад видаляє всі записи у таблиці "Customer":

DELETE FROM Customer;

# Цілісність БД

Іноді під час додавання, видалення чи модифікації записів виникають проблеми. Найчастіше це має місце, якщо маємо кілька зв’язаних таблиць, наприклад дивись логічну модель даних:



Рисунок 20 Логічна схема даних

Тут слід сказати, що одна з головних задач СУБД – підтримання **цілісності БД**

У свою чергу є цілісність сутностей, та цілісність посилань.

Підтримка цілісності сутностей полягає у тому, що СУБД кожного разу, коли змінюються дані таблиці, перевіряє виконання різноманітних обмежень – у першу чергу унікальність первинного ключа, а потім інші обмеження – умови на значення певних атрибутів. Якщо обмеження не виконується, то оператори не виконуються і і СУБД повертає код помилки, який ми зможемо використати для додаткового оброблення помилково внесених оператором даних.

У нашому випадку всі прості первинні ключі мають, для спрощення, генеруватися автоматично. Це забезпечує унікальність значень, що генеруються Autoincrement’ом. Але ми маємо і складний первинний ключ у відношенні Студент-Предмет. У цьому випадку ситуація складніша, бо частини цього ключа є у свою чергу зовнішніми ключами. Система має перевірити, що у цьому відношенні комбінація «кодСтудента-кодПредмета» є унікальною і не повторюється у нашому відношенні. Повторення значення первинного ключа у будь-якому випадку призведе до помилки, обумовленої порушенням цілісності сутності.

Цілісність посилань перевіряється СУБД таким чином: значення зовнішнього ключа(foreign key) підлеглої таблиці перевіряється на приналежність множини значень первинного (primary) ключа головної таблиці. Якщо зовнішній ключ не належить цій множині, то формується помилкова ситуація і система не дозволяє внести або змінити дані.

Порушення цілісності БД можливо тільки під час виконання :Insert Update Delete.

Розглянемо використання цих операторів до головної (master) та підлеглої (slave) таблиць

|  |  |  |
| --- | --- | --- |
|  | master | slave |
| Insert | Може призвести до порушення сутностної цілісності головної таблиці через порушення унікальності первинного ключа або інших обмежень. | Може призвести до порушення сутностної цілісності підлеглої таблиці через порушення унікальності первинного ключа або інших обмежень. Крім того, може порушитися цілісність посилань, якщо вставляються дані з некоректним значення зовнішнього ключа |
| Update | Може призвести до порушення сутностної цілісності головної таблиці через порушення унікальності первинного ключа або інших обмежень.  Може призвести до порушення цілісності посилань, якщо модифікується первинний ключ. Це призводить до того, що у підлеглій таблиці можуть з’явитися записи ніяк не зв’язані з головною таблицею. Це «сміття» залишається у БД назавжди і призводить до помилок у роботі. | Може призвести до порушення сутностної цілісності підлеглої таблиці через порушення унікальності первинного ключа або інших обмежень. Крім того, може порушитися цілісність посилань, якщо після оновлення маємо дані з некоректним значення зовнішнього ключа |
| Delete | Може призвести до порушення цілісності посилань БД. У підлеглій таблиці можуть з’явитися записи ніяк не зв’язані з головною таблицею. Це «сміття» залишається у БД назавжди і призводить до помилок у роботі. | Не призводить до порушення цілісності БД |

Ось чому треба уважно підходити до внесення/оновлення/вилучення даних з таблиць .



У нашому випадку треба розділити ЛМД на пари і вносити дані так: спочатку головну таблицю, а потім підлеглу таблицю.

Наприклад,

Факультет –Група-Студент потім Викладач-Предмет і СтудентПредмет.

Вилучення даних має виконуватися у зворотному порядку.

Оновлення даних можливо у будь-якому порядку, якщо оновлення не стосується первинних та зовнішніх ключем.

Розглянемо більш детально відношення СтудентПредмет

Може статися так, що студенти отримують результат з певного предмету у певну дату. Наприклад, всі здають математику в один день. У цьому випадку, дата отримання результату визначається кодом предмета, або, як кажуть, є функціональна залежність між датою та кодом предмета. Може статися й так, що певний студент має один бал зі всіх предметів, а інший студент студент – інший бал. Тобто існує функціональна залежність між кодом студента та атрибутом Бал.

В цьому випадку говорять, що відношення має надлишковість і слід виконати декомпозицію відношення на кілька відношень.

Розглянемо це більш детально на прикладі математичних функцій. Маємо відношення зі складним первинним ключем X Y та неключовими атрібутами XY – добуток атрибутів, sin(X) cos(XY)



Якщо ми будемо зберігати дані і ця таблиця матиме 1000- значень X, що відповідають 1000- значенням Y, то ми матимемо 1 000 000 записів у таблиці. Кожен запис буде мати 5 (атрибуті) \* (8 байт – довжина кожного double атрибуту). Загалом 40 МБ на таблицю.

Подивимося на відношення уважно. Тільки один атрибут залежить від X Y напряму. У той же час sin(X) залежить тільки від одного атрибуту. У той же час cos(XY) залежить від добутку двох атрибутів первинного ключа, а сам добуток повторює свої значення( через комутативність операції множення).

Висновок: треба винести sin(X) та X у окреме відношення, бо серед 1000 000 записів таблиці ми маємо тільки 1000 різних X та sin(X).



Загальний обсяг даних 1000 000 \*(4 \* 8) = 32МБ та 1000 \*2\*8=16 КБ

Таким чином, завдяки ліквідації надлишковості, ми маємо значне зменшення обсягу.

Звідси витікає необхідність приведення до другої нормальної форми

## Друга нормальна форма

Відношення знаходиться у **другій нормальній формі,** якщо воно вже знаходиться у першій нормальній формі та всі неключові атрибуті цілком залежать від первинного ключа.

Зауваження:

1 якщо відношення має простий первинний ключ, то друга форма досягається автоматично з першою нормальною формою

**2**. Дуже уважно слід перевіряти функціональні залежності між атрибутами, бо їх наявність може призвести до значної надлишковості у БД і , як наслідок, до її розпухання та гальмування і витрати зайвих ресурсів процесора та п’амяті.

Розглянемо подальшу нормалізацію БД.

У цій логічні моделі існує надлишковість, обумовлена функціональною залежністю неключових (ті, що не ввійшли до складу первинного ключа) атрибутів.



Ця залежність призводить до надлишковості у нашій БД. Скільки разів повторюється добуток XY стільки ж разів повторюється значення cos(XY). Для цього слід згадати про комутативність операції множення ( операнди можливо міняти місцями і це не призводить до зміни результату ). Це дозволяє деком озувати перше відношення на два таким чином:



Загальний обсяг даних буде не менше за цей:  
 1 000 000 \*(3 \* 8)=24 МБ + 500 000(2\*8)=8 МБ + 1000 \*2\*8=16 КБ або 32 МБ.

У випадку, якщо X та Y лежать в інтервалі 0…1 з кроком 0.001, кількість різних добутків лише 286 918, як було визначен за допомогою програми

#include "stdafx.h"

#include <iostream>

#include <map>

using namespace std;

int main()

{

map<double, double> m;

double dx = 0.001, dy = 0.001, r;

int k;

for (double x = 0; x < 1; x += dx)

{

for (double y = 0; y < 1; y += dy){r = x\*y; m[r] = r;}

}

std::cout << m.size();

std::cin >> k;

return 0;

}

тому у цьому випадку загальний обсяг приблизно:

1 000 000 \*(3 \* 8)=24 МБ + 300 000(2\*8)=4,8 МБ + 1000 \*2\*8=16 КБ

або 28,8 МБ.

Як бачимо, має місце зменшення загального обсягу даних

Таким чином, подальша нормалізація полягає у приведенні відношення до третьої нормальної форми:

## Третая нормальна форма

Відношення знаходиться у **третій нормальній формі**, якщо воно знаходиться у другій нормальній формі, та відсутні функціональні залежності між неключовими атрибутами.

У нашому випадку ми маємо три відношення з одним неключовим атрибутом.

Висновок: якщо відношення має тільки один неключовий атрибут, то після приведення до другої нормальної форми воно автоматично має і третю нормальну форму.

**Зауваження:** треба дуже ретельно дослідити не ключові атрибути та їх зв’язки.є

Ще раз повернемося до відношень:



Як було сказано раніше, слід уникати атрибутів, значення яких змінюється у часі, або можливо підрахувати. Сучасні РБД мають багату бібліотеку функцій. Тому ці відношення зовсім не слід переводити у таблиці даних. Їх треба просто ліквідувати з моделі, а неключові атрибути підраховувати за допомогою процедур.

На жаль існують також нечислові функції, які утворюють функціональні залежності.

Розглянемо таке відношення



Існує функціональна залежність між «Серією Паспорта» та «Установою, що видала паспорт». Тобто одна установа може видавати паспорта кількох серій, а інші установи не можуть видавати паспорта цих серій, або

установа= f(серія)

У нашому випадку доцільно це відношення розділити на два:



Це дає можливість зменшити загальний обсяг даних, тому що атрибут установа – це текстовий рядок більшої довжини ніж атрибут «серіяПаспорту». Крім того існує також залежність між містом народження та установою, що видала паспорт. В одному місті є кілька установ, що видають паспорт. Тобто одному місту відповідає багато серій паспортів. Для зменшення загального обсягу можливо створити таку схему:



Доцільність такої схеми також у можливості розділення інформації загального доступу та конфіденційної за допомогою механізму розділення прав.

У відношення Студент існує також атрибут «Ідентифікаціний код» (цей атрибут є унікальним), який створює функціональний зв'язок до групи атрибутів (прізвище, ім’я, по-батькові, день народження), які утворюють потенційний ключ з великою ймовірністю, бо малоймовірно існування двох однакових людей.

Можливо сказати таке: група\_атрибутів= функція ( ідент.код), але аргумент функціональної залежності є унікальним, тобто є потенційним ключем. Тому винесення цієї залежності у окреме відношення не зменшить обсяг нашої БД. Відношення, що має таку властивість відповідає нормальній формі Бойса-Кодда.

## Нормальна форма Бойса-Кодда (БКНФ)

Відношення знаходиться у **нормальній формі Бойса-Кодда (БКНФ)**, якщо воно знаходиться у другій нормальній формі, та серед його неключових атрибутів є тільки такі функціональні залежності, детермінанти (аргументи) яких є потенційними ключами.

Подальша нормалізація полягає у приведенні відношень до четвертої нормальної форми.

## Четверта нормальна форма

Відношення знаходиться у **четвертій нормальній формі,** якщо воно відповідає БКНФ та всі нетривіальні багатозначні залежності фактично є функціональними залежностями від її потенційних ключей.

*Четверта нормальна форма* стосується відношень, в яких є повторювані набори даних. Декомпозиція, заснована на функціональних залежностях, не призводить до виключення такої надмірності. У цьому випадку використовують декомпозицію, засновану на багатозначних залежностях.

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

Приведення відношення до 4НФ дозволяє виключити тип аномалій оновлення. Для приведення відношення з БКНФ до 4НФ слід виконати проекції вихідного відношення на пари атрибутів, створюючих багатозначні залежності.

**Приклад**

Ресторани готують різні страви, та можуть доставляти страви у певні райони міста. Складний первинний ключ має три атрибути:



Це відношення не відповідає 4НФ, тому, що існує багатозначні залежності:

* {Ресторан} ↠ {Страва}
* {Ресторан} ↠ {Район }

Тобто додавання певної страви обумовлює внесення по одному новому кортежу для кожного району. Можлива логічна аномалія, коли певній страві будуть відповідати лише де-які райони доставки зі всіх можливих районів міста.

Для унеможливлення аномалії слід декомпозувати відношення, та записати незалежні факти у різних відношеннях.

Наприклад, виконати декомпозицію:



Але, якщо до відношення додати атрибут, що функціонально залежить від потенційного ключа, наприклад ціну з урахуванням вартості доставки:



тоді це відношення знаходиться у 4НФ і його неможливо декомпозувати без втрат інформації

Приведені вище багатозначні залежності називають *вбудованими*.

## П'ята нормальна форма

***П'ята нормальна форма*** (5НФ) - одна з можливих нормальних форм відношення реляційної бази даних

Відношення знаходиться в п'ятій нормальній формі (інакше - в проекційно-сполучній нормальній формі) тоді і тільки тоді, коли кожна нетривіальна залежність з'єднання в ньому визначається потенційним ключем цих відносини. При цьому:

* Залежність з'єднання \*{A, B ,..., Z} є тривіальним тоді і тільки тоді, коли по крайній мірі одна з підмножин A, B ,..., Z множина [атрибутів](http://uk.wikipedia.org/wiki/%D0%90%D1%82%D1%80%D0%B8%D0%B1%D1%83%D1%82) є множиною всіх атрибутів відносини;
* Залежність з'єднання \*{A, B ,..., Z} визначається потенційним ключем (ключами) тоді і тільки тоді, коли кожна з підмножин A, B ,..., Z множина атрибутів є суперключем відносин.

Дуже рідко відношення, яка перебуває у 4НФ, не відповідає 5НФ. Це ті ситуації, в яких реальні правила, що обмежують можливі комбінації атрибутів, ніяк не виражені в структурі таблиці (наприклад, правила певного бізнесу). У такому випадку, якщо таблиця не приведена до 5НФ, тягар забезпечення логічної цілісності даних частково наляже на додаток, що відповідає за додавання, видалення і зміни даних у таблицях, що відповідають відношенням. У цьому випадку існує ризик виникнення аномалій даних. П'ята нормальна форма виключає виникнення таких аномалій.

Припустимо, що продавець може торгувати продукцією кількох фірм, асортимент у фірм різний, причому продавець може пропонувати тільки частину товарів конкретної фірми.

Відношення



відповідає 4НФ, проте не відображає обмеження, пов'язаного з асортиментом продукції фірм.

Може виникнути кортеж, в якому фірмі буде відповідати вид товару, який вона не випускає.

У даному випадку (для приведення до 5НФ) відношення слід декомпозувати на три:



Відношення у 5НФ автоматично знаходиться також у 4НФ та інших нормальних формах.

5НФ 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.

## Доменно-ключова нормальна форма (ДКНФ)

Ця нормальна форма вимагає, аби в схемі не було інших обмежень окрім ключів та доменів.

**Доменно-ключова нормальна форма** - це нормальна форма що використовується під час нормалізації БД і вимагає щоб БД не містила жодних інших обмежень крім обмежень доменів і обмежень ключів.

Обмеження домену - обмеження, яке наказує використовувати для певного атрибуту значення тільки з деякого заданого домену. Обмеження по своїй суті є заданням переліку (або логічного еквіваленту переліку) допустимих значень типу і оголошенням про те, що вказаний атрибут має даний тип.

Обмеження ключа – обмеження, яке стверджує, що деякий атрибут чи комбінація атрибутів є потенційним ключем.

Будь-яке відношення, що знаходиться в ДКНФ, обов'язково знаходиться у 5НФ. Але не будь-яку змінну відношення можна привести до ДКНФ.

## Шоста нормальна форма

Таблиця знаходиться у 6NF, якщо вона знаходиться у 5NF та задовольняє вимозі відсутності нетривіальних залежностей. Зазвичай 6NF ототожнюють з DKNF.

# Мова маніпулювання даними DML

SQL statements are somewhat like simple English sentences.  
Keywords include SELECT, UPDATE, WHERE, ORDER BY, etc.

## Таблиці бази даних, використані як приклад

All examples based on the simple tables:

|  |
| --- |
| PRODUCT |
| Id |
| ProductName |
| SupplierId |
| UnitPrice |
| Package |
| IsDiscontinued |

|  |
| --- |
| ORDERITEM |
| Id |
| OrderId |
| ProductId |
| UnitPrice |
| Quantity |

|  |
| --- |
| ORDER |
| Id |
| OrderDate |
| OrderNumber |
| CustomerId |
| TotalAmount |

|  |
| --- |
| CUSTOMER |
| Id |
| FirstName |
| LastName |
| City |
| Country |
| Phone |

|  |
| --- |
| SUPPLIER |
| Id |
| CompanyName |
| ContactName |
| City |
| Country |
| Phone |
| Fax |

Існує чотири фундаментальні оператори, які надалі будуть детально розглядатися

**SELECT** -- Read the data

**INSERT** -- Insert new data

**UPDATE** -- Update existing data

**DELETE** -- Remove data

Загалом ці оператори називають **CRUD** (Create, Read, Update, Delete).

## Оператор INSERT

**INSERT table-name (column-names) VALUES (column-values)**

**INSERT Supplier (Name, ContactName, City, Country)**

**VALUES ('Oxford Trading', 'Ian Smith', 'Oxford', 'UK')**

### Оператор INSERT INTO

The INSERT INTO statement is used to add new data to a database.  
INSERT INTO adds a new record to a table.

INSERT INTO can contain values for some or all of its columns.  
INSERT INTO can be combined with a SELECT to insert records.

The general syntax is:

**INSERT INTO table-name (column-names) VALUES (values)**

**Problem**: Add a new customer to the database

1. **INSERT INTO Customer (FirstName, LastName, City, Country, Phone)**
2. **VALUES ('Craig', 'Smith', 'New York', 'USA', 1-01-993 2800)**

**Result:** 1 new record inserted

**Problem**: Add a new customer named Anita Coats to the database

**INSERT INTO Customer (FirstName, LastName) VALUES ('Anita', 'Coats')**

**Result:** 1 new record inserted

### Оператор INSERT комбінований з SELECT

The general syntax is

1. **INSERT INTO table-name (column-names)**
2. **SELECT column-names FROM table-name WHERE condition**

**Problem**: The Bigfoot Brewery supplier has also become a customer. Add a customer record with values from the supplier table.

1. **INSERT INTO Customer (FirstName, LastName, City, Country, Phone)**
2. **SELECT LEFT(ContactName, CHARINDEX(' ',ContactName) - 1),**
3. **SUBSTRING(ContactName, CHARINDEX(' ',ContactName) + 1, 100),**
4. **City, Country, Phone**
5. **FROM Supplier**
6. **WHERE CompanyName = 'Bigfoot Breweries'**

Note: ContactName is parsed into FirstName and LastName.  
Parsing takes place with built-in functions: LEFT, SUBSTRING, and CHARINDEX.

**Result:** 1 new record inserted

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

* INSERT INTO SELECT requires that data types in source and target tables match
* The existing records in the target table are unaffected

Copy all columns from one table to another table:

INSERT INTO table2 SELECT \* FROM table1 WHERE condition;

Copy only some columns from one table into another table:

INSERT INTO table2 (column1, column2, column3, ...)

SELECT column1, column2, column3, ...

FROM table1

WHERE condition;

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

INSERT INTO Customers (CustomerName, City, Country)

SELECT SupplierName, City, Country FROM Suppliers;

The following SQL statement copies "Suppliers" into "Customers" (fill all columns):

INSERT INTO

Customers (CustomerName, ContactName, Address, City, PostalCode, Country)

SELECT SupplierName, ContactName, Address, City, PostalCode, Country

FROM Suppliers;

The following SQL statement copies only the German suppliers into "Customers":

INSERT INTO Customers (CustomerName, City, Country)

SELECT SupplierName, City, Country

FROM Suppliers WHERE Country='Germany';

## Оператор UPDATE

1. **UPDATE table-name SET column-name = column-value WHERE condition**
2. **UPDATE OrderItem SET Quantity = 2 WHERE Id = 388**

The SET command is used with UPDATE to specify which columns and values that should be updated in a table.

The following SQL updates the first customer (CustomerID = 1) with a new ContactName and a new City:

UPDATE Customers SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

A SQL WHERE clause filters for rows that meet certain criteria.  
It is a way to limit the rows to the ones you're interested in.

**UPDATE table-name SET column-name = value WHERE condition**

Finally, a DELETE statement with a WHERE clause:

Note: Referential integrity may prevent this deletion.  
A better approach may be to discontinue the product, that is, set the column IsDiscontinued to true.

The UPDATE statement updates data values in a database.  
UPDATE can update one or more records in a table.  
Use the WHERE clause to UPDATE only specific records.

**UPDATE table-name SET column-name1 = value1, column-name2 = value2, ..**

To limit the number of records to UPDATE append a WHERE clause:

1. **UPDATE table-name SET column-name1 = value1, column-name2 = value2, ... WHERE condition**

**Problem**: Discontinue all products in the database

1. **UPDATE Product SET IsDiscontinued = 1**

Note: the value 1 denotes true.

**Result:** 77 records updated.

**Problem**: Discontinue products over $50.

**UPDATE Product SET IsDiscontinued = 1 WHERE UnitPrice > 50**

Note: the value 1 denotes true.

**Result:** 7 records updated.

**Problem**: Discontinue product with Id = 46.

1. **UPDATE Product SET IsDiscontinued = 1 WHERE Id = 46**

This is a more common scenario in which a single record is updated.  
Note: the value 1 denotes true.

**Result:** 1 record updated.

**Problem**: Supplier Norske Meierier (Id = 15) has moved: update their city, phone and fax.

1. **UPDATE Supplier SET City = 'Oslo', Phone = '(0)1-953530', Fax = '(0)1-953555' WHERE Id = 15**

This is a common scenario in which a single record is updated.

## Оператор DELETE

The DELETE command is used to delete existing records in a table.

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

**Note:** Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

The following SQL statement deletes all rows in the "Customers" table, without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM Customers;

**DELETE table-name WHERE condition**

**DELETE Customer WHERE Email = 'alex@gmail.com'**

DELETE permanently removes records from a table.

DELETE can delete one or more records in a table.

Use the WHERE clause to DELETE only specific records.

The general syntax is

1. **DELETE table-name**

To delete specific records append a WHERE clause:

1. **DELETE table-name WHERE condition**

**Problem**: Delete all products.

1. **DELETE Product**

**Result:** 77 records deleted.

**Problem**: Delete products over $50.

1. **DELETE Product WHERE UnitPrice > 50**

**Result:** 7 records deleted.

**Problem**: Delete customer with Id = 21.

1. **DELETE Customer WHERE Id = 21**

This is a common scenario in which a single record is deleted.

**Result:** 1 record deleted.

A SQL WHERE clause filters for rows that meet certain criteria.  
It is a way to limit the rows to the ones you're interested in.

a DELETE statement with a WHERE clause:

**DELETE table-name WHERE condition**

**Problem:** Delete all products with unit price higher than $50.

**DELETE FROM Product WHERE UnitPrice > 50**

**Result:** 7 records deleted.

Note: Referential integrity may prevent this deletion.  
A better approach may be to discontinue the product, that is, set the column IsDiscontinued to true.

## 

## Оператор SELECT

1. **SELECT column-names**
2. **FROM table-name**
3. **WHERE condition**
4. **ORDER BY sort-order**
5. **SELECT FirstName, LastName, City, Country**
6. **FROM Customer**
7. **WHERE City = 'Paris'**
8. **ORDER BY LastName**

### Джерело даних FROM

The FROM command is used to specify which table to select or delete data from.

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

SELECT CustomerName, City FROM Customers;

The following SQL statement selects all the columns from the "Customers" table:

SELECT \* FROM Customers;

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

To select all columns use \* **SELECT \* FROM table-name**

**Problem**: List all customers

**SELECT \* FROM Customer** Result: 91 records

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id | FirstName | LastName | City | Country | Phone |
| 1 | Maria | Anders | Berlin | Germany | 030-0074321 |
| 2 | Ana | Trujillo | México D.F. | Mexico | (5) 555-4729 |
| 3 | Antonio | Moreno | México D.F. | Mexico | (5) 555-3932 |
| 4 | Thomas | Hardy | London | UK | (171) 555-7788 |
| 5 | Christina | Berglund | Luleå | Sweden | 0921-12 34 65 |
| https://www.dofactory.com/img/ellipsis.png | | | | | |

**Problem**: List the first name, last name, and city of all customers

**SELECT FirstName, LastName, City FROM Customer**

**Result:** 91 records

|  |  |  |
| --- | --- | --- |
| FirstName | LastName | City |
| Maria | Anders | Berlin |
| Ana | Trujillo | México D.F. |
| Antonio | Moreno | México D.F. |
| Thomas | Hardy | London |
| Christina | Berglund | Luleå |
| https://www.dofactory.com/img/ellipsis.png | | |

### Опції TOP, LIMIT ROWNUM

The SELECT TOP statement returns a specified number of records.  
SELECT TOP is useful when working with very large datasets.  
Non SQL Server databases use keywords like LIMIT, OFFSET, and ROWNUM.

**SELECT TOP n column-names FROM table-name**

**Problem**: List the top 10 most expensive products ordered by price

1. **SELECT TOP 10 Id, ProductName, UnitPrice, Package**
2. **FROM Product**
3. **ORDER BY UnitPrice DESC**

**Result:** 10 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | ProductName | UnitPrice | Package |
| 38 | Côte de Blaye | 263.50 | 12 - 75 cl bottles |
| 29 | Thüringer Rostbratwurst | 123.79 | 50 bags x 30 sausgs. |
| 9 | Mishi Kobe Niku | 97.00 | 18 - 500 g pkgs. |
| 20 | Sir Rodney's Marmalade | 81.00 | 30 gift boxes |
| 18 | Carnarvon Tigers | 62.50 | 16 kg pkg. |
| 59 | Raclette Courdavault | 55.00 | 5 kg pkg. |
| 51 | Manjimup Dried Apples | 53.00 | 50 - 300 g pkgs. |
| 62 | Tarte au sucre | 49.30 | 48 pies |
| 43 | Ipoh Coffee | 46.00 | 16 - 500 g tins |
| 28 | Rössle Sauerkraut | 45.60 | 25 - 825 g cans |

The SELECT TOP command is used to specify the number of records to return.

**Note:** Not all database systems support SELECT TOP. MySQL uses LIMIT, and Oracle uses ROWNUM.

The following SQL statement selects the first three records from the "Customers" table:

SELECT TOP 3 \* FROM Customers;

The following SQL statement shows the equivalent example using the LIMIT clause:

SELECT \* FROM Customers LIMIT 3;

The following SQL statement shows the equivalent example using ROWNUM:

SELECT \* FROM Customers WHERE ROWNUM <= 3;

### Опція OFFSET-FETCH (ORACLE)

How do I implement pagination in SQL?

OFFSET excludes the first set of records.  
This keyword can only be used with an ORDER BY clause.

OFFSET with FETCH NEXT returns a defined window of records.  
OFFSET with FETCH NEXT is wonderful for building pagination support.

**SELECT column-names FROM table-name**

**ORDER BY column-names OFFSET n ROWS**

To exclude the first n records and return only the next m records:

**SELECT column-names FROM table-name**

**ORDER BY column-names OFFSET n ROWS**

**FETCH NEXT m ROWS ONLY**

This will return only record (n + 1) to (n + m). See example below.

**Problem**: Get all but the 10 most expensive products sorted by price

1. **SELECT Id, ProductName, UnitPrice, Package**
2. **FROM Product ORDER BY UnitPrice DESC**
3. **OFFSET 10 ROWS**

**Result:** 68 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | ProductName | UnitPrice | Package |
| 27 | Schoggi Schokolade | 43.90 | 100 - 100 g pieces |
| 63 | Vegie-spread | 43.90 | 15 - 625 g jars |
| 8 | Northwoods Cranberry Sauce | 40.00 | 12 - 12 oz jars |
| 17 | Alice Mutton | 39.00 | 20 - 1 kg tins |
| 12 | Queso Manchego La Pastora | 38.00 | 10 - 500 g pkgs. |

**Problem**: Get the 10th to 15th most expensive products sorted by price

1. **SELECT Id, ProductName, UnitPrice, Package**
2. **FROM Product**
3. **ORDER BY UnitPrice DESC**
4. **OFFSET 10 ROWS**
5. **FETCH NEXT 5 ROWS ONLY**

**Result:** 5 records

|  |  |  |  |
| --- | --- | --- | --- |
| Id | ProductName | UnitPrice | Package |
| 27 | Schoggi Schokolade | 43.90 | 100 - 100 g pieces |
| 63 | Vegie-spread | 43.90 | 15 - 625 g jars |
| 8 | Northwoods Cranberry Sauce | 40.00 | 12 - 12 oz jars |
| 17 | Alice Mutton | 39.00 | 20 - 1 kg tins |
| 12 | Queso Manchego La Pastora | 38.00 | 10 - 500 g pkgs. |

### Опція DISTINCT

SELECT DISTINCT returns only distinct (i.e. different) values.  
The DISTINCT keyword eliminates duplicate records from the results.

DISTINCT can be used with aggregates: COUNT, AVG, MAX, etc.  
It operates on a single column. DISTINCT for multiple columns is not supported.

The general syntax is

1. **SELECT DISTINCT column-name FROM table-name**

Can be used with COUNT and other aggregates

1. **SELECT COUNT (DISTINCT column-name) FROM table-name**

**Problem**: List all unique supplier countries in alphabetical order.

1. **SELECT DISTINCT Country FROM Supplier ORDER BY COUNTRY**

**Result:** 16 rows

|  |
| --- |
| Country |
| Australia |
| Brazil |
| Canada |
| Denmark |
| https://www.dofactory.com/img/ellipsis.png |

**Problem**: List the number of unique supplier countries

**SELECT COUNT (DISTINCT Country) FROM Supplier**

Result:

|  |
| --- |
| Count |
| 16 |

### NULL функції IFNULL(),COALESCE()

Look at the following "Products" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P\_Id | ProductName | UnitPrice | UnitsInStock | UnitsOnOrder |
| 1 | Jarlsberg | 10.45 | 16 | 15 |
| 2 | Mascarpone | 32.56 | 23 |  |
| 3 | Gorgonzola | 15.67 | 9 | 20 |

Suppose that the "UnitsOnOrder" column is optional, and may contain NULL values.

Look at the following SELECT statement:

SELECT ProductName, UnitPrice \* (UnitsInStock + UnitsOnOrder)

FROM Products;

In the example above, if any of the "UnitsOnOrder" values are NULL, the result will be NULL.

The MySQL [IFNULL()](https://www.w3schools.com/sql/func_mysql_ifnull.asp) function lets you return an alternative value if an expression is NULL:

SELECT ProductName, UnitPrice \* (UnitsInStock + IFNULL(UnitsOnOrder, 0))

FROM Products;

or we can use the [COALESCE()](https://www.w3schools.com/sql/func_mysql_coalesce.asp) function, like this:

SELECT ProductName, UnitPrice\*(UnitsInStock + COALESCE(UnitsOnOrder, 0))

FROM Products;

### Функції MIN, MAX

SELECT MIN returns the minimum value for a column.  
And SELECT MAX returns the maximum value for a column.

The general MIN|MAX syntax is:

**SELECT MIN(column-name) FROM table-name**

**SELECT MAX(column-name) FROM table-name**

**Problem**: Find the cheapest product

1. **SELECT MIN(UnitPrice) FROM Product**

Result:

|  |
| --- |
| UnitPrice |
| 2.50 |

**Problem**: Find the largest order placed in 2020

1. **SELECT MAX(TotalAmount) FROM [Order] WHERE YEAR(OrderDate) = 2020**

Result:

|  |
| --- |
| TotalAmount |
| 17250.00 |

**Problem**: Find the last order date in 2020

1. **SELECT MAX(OrderDate) FROM [Order] WHERE YEAR(OrderDate) = 2020**

Note: MIN and MAX can be used with numeric, string, and date values.

Result:

|  |
| --- |
| OrderDate |
| 2020-10-31 00:00:00.000 |

### Функції COUNT, SUM, AVG у запитах

The COUNT() function returns the number of rows that matches a specified criterion.

The AVG() function returns the average value of a numeric column.

The SUM() function returns the total sum of a numeric column.

The general COUNT SUM AVG syntax is:

SELECT COUNT(column\_name) FROM table\_name WHERE condition;

SELECT AVG(column\_name) FROM table\_name WHERE condition;

SELECT SUM(column\_name) FROM table\_name WHERE condition;

**Problem**: Find the total number of customers.

1. **SELECT COUNT(Id) FROM Customer**

Result:

|  |
| --- |
| Count |
| 91 |

**Problem**: Compute the total amount sold in 2020.

**SELECT SUM(TotalAmount) FROM [Order] WHERE YEAR(OrderDate) = 2020**

Result:

|  |
| --- |
| Sum |
| 658388.75 |

**Problem**: Compute the average size of all orders.

**SELECT AVG(TotalAmount) FROM [Order]**

Result:

|  |
| --- |
| Average |
| 1631.877819 |

### Аліаси у запитах

An Alias is a shorthand for a table or column name.  
Aliases reduce the amount of typing required to enter a query.

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of the query.

SELECT column\_name AS alias\_name FROM table\_name;

SELECT column\_name(s) FROM table\_name AS alias\_name;

**SELECT column-name AS alias-name FROM table-name alias-name**

**WHERE condition**

Complex queries with Aliases are generally easier to read.  
Aliases are useful with JOINs and aggregates: SUM, COUNT, etc.  
An Alias only exists for the duration of the query.

**Problem:** List total customers in each country.Display results with easy to understand column headers.

1. **SELECT COUNT(C.Id) AS TotalCustomers, C.Country AS Nation**
2. **FROM Customer C GROUP BY C.Country**

TotalCustomers and Nation are column Aliases. The table Alias (C) in this example is not particularly useful.

**Result:** 21 records

|  |  |
| --- | --- |
| TotalCustomers | Nation |
| 3 | Argentina |
| 2 | Austria |
| 2 | Belgium |
| 9 | Brazil |
| 3 | Canada |

**Problem:** List the total amount ordered by customer with easy to read column headers

1. **SELECT C.Id AS Identifier, C.LastName + ', ' + C.FirstName AS CustomerName,**
2. **SUM(O.TotalAmount) AS TotalSpent**
3. **FROM [Order] O JOIN Customer C ON O.CustomerId = C.Id**
4. **GROUP BY C.Id, C.LastName + ', ' + C.FirstName**
5. **ORDER BY TotalSpent DESC**

The Aliases significantly simplify writing the JOIN and ORDER BY clauses.  
The C Alias in C.Id helps identify the Customer Id rather then the Order Id.

**Results:** 89 records

|  |  |  |
| --- | --- | --- |
| Identifier | CustomerName | TotalSpent |
| 63 | Kloss, Horst | 117483.39 |
| 71 | Pavarotti, Jose | 115673.39 |
| 20 | Mendel, Roland | 113236.68 |
| 37 | McKenna, Patricia | 57317.39 |
| 65 | Wilson, Paula | 52245.90 |
| https://www.dofactory.com/img/ellipsis.png | | |

#### Аліаси стовпчиків Alias for Columns

The AS command is used to rename a column or table with an alias.

An alias only exists for the duration of the query.

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

SELECT CustomerID AS ID, CustomerName AS Customer FROM Customers;

The following SQL statement creates two aliases. Notice that it requires double quotation marks or square brackets if the alias name contains spaces:

SELECT CustomerName AS Customer, ContactName AS [Contact Person]

FROM Customers;

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address FROM Customers;

**Note:** To get the SQL statement above to work in MySQL use the following:

SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address FROM Customers;

#### Аліаси таблиць Alias for Tables

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

SELECT o.OrderID, o.OrderDate, c.CustomerName  
FROM Customers AS c, Orders AS o WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

SELECT CustomerID AS ID, CustomerName AS Customer FROM Customers;

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. **Note:** It requires double quotation marks or square brackets if the alias name contains spaces:

SELECT CustomerName AS Customer, ContactName AS [Contact Person]  
FROM Customers;

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS Address  
FROM Customers;

**Note:** To get the SQL statement above to work in MySQL use the following:

SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS Address FROM Customers;

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

SELECT o.OrderID, o.OrderDate, c.CustomerName  
FROM Customers AS c, Orders AS o  
WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;

The following SQL statement is the same as above, but without aliases:

SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName  
FROM Customers, Orders  
WHERE Customers.CustomerName='Around the Horn' AND Customers.CustomerID=Orders.CustomerID;

Aliases can be useful when:

* There are more than one table involved in a query
* Functions are used in the query
* Column names are big or not very readable
* Two or more columns are combined together

### Опція CASE

The CASE command is used is to create different output based on conditions.

The following SQL goes through several conditions and returns a value when the specified condition is met:

SELECT OrderID, Quantity,

CASE WHEN Quantity > 30 THEN "The quantity is greater than 30"

WHEN Quantity = 30 THEN "The quantity is 30"

ELSE "The quantity is under 30"

END  
FROM OrderDetails;

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

SELECT CustomerName, City, Country

FROM Customers

ORDER BY

(CASE WHEN City IS NULL THEN Country

ELSE City

END);

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

CASE WHEN condition1 THEN result1

WHEN condition2 THEN result2¶

WHEN conditionN THEN result

ELSE result

END;

### WHERE у запиті на вибір данних

A SQL WHERE clause filters for rows that meet certain criteria.  
It is a way to limit the rows to the ones you're interested in.

WHERE is followed by a condition that returns either true or false.  
The WHERE clause is used with SELECT, UPDATE, and DELETE.

Here is a SELECT statement with a WHERE clause:

**SELECT column-names FROM table-name WHERE condition**

**Problem:** List all customers in Sweden

1. **SELECT Id, FirstName, LastName, City, Country, Phone**
2. **FROM Customer WHERE Country = 'Sweden'**

Result: 2 records

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id | FirstName | LastName | City | Country | Phone |
| 5 | Christina | Berglund | Luleå | Sweden | 0921-12 34 65 |
| 24 | Maria | Larsson | Bräcke | Sweden | 0695-34 67 21 |

The WHERE command filters a result set to include only records that fulfill a specified condition.

The following SQL statement selects all the customers from "Mexico" in the "Customers" table:

SELECT \* FROM Customers WHERE Country='Mexico';

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

SELECT \* FROM Customers WHERE CustomerID=1;

**Note:** The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

The following operators can be used in the WHERE clause:

|  |  |
| --- | --- |
| Operator | Description |
| = | Equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| BETWEEN | Between a certain range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |

#### Предикат WHERE BETWEEN

The BETWEEN command is used to select values within a given range. The values can be numbers, text, or dates.

The BETWEEN command is inclusive: begin and end values are included.

The following SQL statement selects all products with a price BETWEEN 10 and 20:

SELECT \* FROM Products WHERE Price BETWEEN 10 AND 20;

To display the products outside the range of the previous example, use NOT BETWEEN:

SELECT \* FROM Products WHERE Price NOT BETWEEN 10 AND 20;

The following SQL statement selects all products with a ProductName BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

SELECT \* FROM Products WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'  
ORDER BY ProductName;

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

SELECT column\_name(s) FROM table\_name WHERE column\_name BET

The following SQL statement selects all products with a price BETWEEN 10 and 20:

SELECT \* FROM Products WHERE Price BETWEEN 10 AND 20;

##### NOT BETWEEN

SELECT \* FROM Products WHERE Price NOT BETWEEN 10 AND 20;

##### BETWEEN with IN

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

SELECT \* FROM Products WHERE Price BETWEEN 10 AND 20 AND CategoryID NOT IN (1,2,3);

##### BETWEEN Text Values

The following SQL statement selects all products with a ProductName BETWEEN Carnarvon Tigers and Mozzarella di Giovanni:

SELECT \* FROM Products  
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'  
ORDER BY ProductName;

The following SQL statement selects all products with a ProductName BETWEEN Carnarvon Tigers and Chef Anton's Cajun Seasoning:

SELECT \* FROM Products  
WHERE ProductName BETWEEN "Carnarvon Tigers" AND "Chef Anton's Cajun Seasoning"  
ORDER BY ProductName;

##### NOT BETWEEN Text Values

The following SQL statement selects all products with a ProductName NOT BETWEEN Carnarvon Tigers and Mozzarella di Giovanni:

SELECT \* FROM Products  
WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni' ORDER BY ProductName;

##### BETWEEN Dates

The following SQL statement selects all orders with an OrderDate BETWEEN '01-July-1996' and '31-July-1996':

SELECT \* FROM Orders WHERE OrderDate BETWEEN #01/07/1996# AND #31/07/1996#;

OR:

SELECT \* FROM OrdersWHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';

WHERE BETWEEN returns values that fall within a given range. This clause is a shorthand for >= AND <=. The BETWEEN operator is inclusive, that is, begin and end values are included.

The general syntax is

1. **SELECT column-names FROM table-name**
2. **WHERE column-name BETWEEN value1 AND value2**

**Problem**: List all products between $10 and $20

1. **SELECT Id, ProductName, UnitPrice**
2. **FROM Product**
3. **WHERE UnitPrice BETWEEN 10 AND 20**
4. **ORDER BY UnitPrice**

**Result:** 29 records.

|  |  |  |
| --- | --- | --- |
| Id | ProductName | UnitPrice |
| 3 | Aniseed Syrup | 10.00 |
| 21 | Sir Rodney's Scones | 10.00 |
| 74 | Longlife Tofu | 10.00 |
| 46 | Spegesild | 12.00 |
| 31 | Gorgonzola Telino | 12.50 |
| https://www.dofactory.com/img/ellipsis.png | | |

**Problem**: List all products not between $10 and $100 sorted by price.

1. **SELECT Id, ProductName, UnitPrice**
2. **FROM Product**
3. **WHERE UnitPrice NOT BETWEEN 5 AND 100**
4. **ORDER BY UnitPrice**

**Result:** 4 records.

|  |  |  |
| --- | --- | --- |
| Id | ProductName | UnitPrice |
| 33 | Geitost | 2.50 |
| 24 | Guaraná Fantástica | 4.50 |
| 29 | Thüringer Rostbratwurst | 123.79 |
| 38 | Côte de Blaye | 263.50 |

**Problem**: Get the number of orders and amount sold  
between Jan 1, 2013 and Jan 31, 2013.

1. **SELECT COUNT(Id), SUM(TotalAmount)**
2. **FROM [Order]**
3. **WHERE OrderDate BETWEEN '1/1/2013' AND '1/31/2013'**

Result:

|  |  |
| --- | --- |
| Count | TotalAmount |
| 33 | 66692.80 |

#### Операції WHERE AND, OR, NOT

WHERE conditions can be combined with AND, OR, and NOT.  
A WHERE clause with AND requires that two conditions are true.

The AND command is used with WHERE to only include rows where both conditions is true.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

SELECT \* FROM Customers WHERE Country='Germany' AND City='Berlin';

A WHERE clause with OR requires that one of two conditions is true.  
And a WHERE clause with NOT negates the specified condition.

A WHERE clause with AND OR NOT:

**SELECT column-names FROM table-name WHERE condition1 AND condition2**

**UPDATE table-name SET column-name = value WHERE condition1 OR condition2**

**DELETE table-name WHERE NOT condition**

**Problem:** Get the customer named Thomas Hardy.

1. **SELECT Id, FirstName, LastName, City, Country**
2. **FROM Customer**
3. **WHERE FirstName = 'Thomas' AND LastName = 'Hardy'**

**Result:** 1 record.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | FirstName | LastName | City | Country |
| 4 | Thomas | Hardy | London | UK |

**Problem**: List all customers from Spain or France

1. **SELECT Id, FirstName, LastName, City, Country**
2. **FROM Customer**
3. **WHERE Country = 'Spain' OR Country = 'France'**

**Result:** 16 records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | FirstName | LastName | City | Country |
| 7 | Frédérique | Citeaux | Strasbourg | France |
| 8 | Martín | Sommer | Madrid | Spain |
| 9 | Laurence | Lebihan | Marseille | France |
| 18 | Janine | Labrune | Nantes | France |
| 22 | Diego | Roel | Madrid | Spain |
| 23 | Martine | Rancé | Lille | France |
| https://www.dofactory.com/img/ellipsis.png | | | | |

**Problem**: List all customers that are not from the USA

1. **SELECT Id, FirstName, LastName, City, Country**
2. **FROM Customer**
3. **WHERE NOT Country = 'USA'**

**Result:** 78 records.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | FirstName | LastName | City | Country |
| 1 | Maria | Anders | Berlin | Germany |
| 2 | Ana | Trujillo | México D.F. | Mexico |
| 3 | Antonio | Moreno | México D.F. | Mexico |
| 4 | Thomas | Hardy | London | UK |
| 5 | Christina | Berglund | Luleå | Sweden |
| 6 | Hanna | Moos | Mannheim | Germany |
| 7 | Frédérique | Citeaux | Strasbourg | France |
| https://www.dofactory.com/img/ellipsis.png | | | | |

**Problem**: List all orders that are not between $50 and $15000

1. **SELECT Id, OrderDate, CustomerId, TotalAmount**
2. **FROM [Order]**
3. **WHERE NOT (TotalAmount >= 50 AND TotalAmount <= 15000)**
4. **ORDER BY TotalAmount DESC**

**Result:** 16 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | OrderDate | CustomerId | TotalAmount |
| 618 | 2/2/2014 12:00:00 AM | 63 | 17250.00 |
| 783 | 4/17/2014 12:00:00 AM | 71 | 16321.90 |
| 734 | 3/27/2014 12:00:00 AM | 34 | 15810.00 |
| 175 | 1/22/2013 12:00:00 AM | 27 | 49.80 |
| 24 | 8/1/2012 12:00:00 AM | 75 | 48.00 |
| https://www.dofactory.com/img/ellipsis.png | | | |

The OR command is used with WHERE to include rows where either condition is true.

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR city is "München":

SELECT \* FROM Customers

WHERE City='Berlin' OR City='München';

The NOT command is used with WHERE to only include rows where a condition is not true.

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

SELECT \* FROM Customers WHERE NOT Country='Germany';

#### Предикати WHERE ANY and ALL

ANY and ALL keywords are used with WHERE or HAVING.  
They operate on subqueries that return multiple values.

The ANY operator returns true if any of the subquery values meet the condition.

The ALL operator returns true if all of the subquery values meet the condition.

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator ANY

(SELECT column\_name FROM table\_name WHERE condition);

SELECT column\_name(s)

FROM table\_name

WHERE column\_name operator ALL

(SELECT column\_name FROM table\_name WHERE condition);

**Note:** The operator must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

The following SQL statement returns TRUE and lists the product names if it finds ANY records in the OrderDetails table that quantity = 10:

SELECT ProductName

FROM Products

WHERE ProductID = ANY

(SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

The following SQL statement returns TRUE and lists the product names if it finds ANY records in the OrderDetails table that quantity > 99:

SELECT ProductName

FROM Products

WHERE ProductID = ANY

(SELECT ProductID FROM OrderDetails WHERE Quantity > 99);

**Problem**: Which products were sold by the unit, that is, quantity = 1.

1. **SELECT ProductName FROM Product**
2. **WHERE Id = ANY**
3. **(SELECT ProductId FROM OrderItem WHERE Quantity = 1)**

**Result:** 17 records

|  |
| --- |
| ProductName |
| Chef Anton's Cajun Seasoning |
| Grandma's Boysenberry Spread |
| Uncle Bob's Organic Dried Pears |
| Ikura |
| Konbu |
| Tofu |
| Teatime Chocolate Biscuits |
| Sir Rodney's Marmalade |

**Problem**: List customers who placed orders that are larger than the average of each customer order.

1. **SELECT DISTINCT FirstName + ' ' + LastName as CustomerName**
2. **FROM Customer, [Order]**
3. **WHERE Customer.Id = [Order].CustomerId**
4. **AND TotalAmount > ALL**
5. **(SELECT AVG(TotalAmount)**
6. **FROM [Order]**
7. **GROUP BY CustomerId)**

**Result:** 22 records

|  |
| --- |
| CustomerName |
| Art Braunschweiger |
| Christina Berglund |
| Elizabeth Lincoln |
| Frédérique Citeaux |
| Georg Pipps |
| Horst Kloss |
| Howard Snyder |
| https://www.dofactory.com/img/ellipsis.png |

The ALL command returns true if all of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the productnames if ALL the records in the OrderDetails table has quantity = 10:

SELECT ProductName FROM Products WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);

#### Предикат WHERE EXISTS

WHERE EXISTS tests for the existence of any records in a subquery.  
EXISTS returns true if the subquery returns one or more records.  
The EXISTS condition is commonly used with correlated subqueries.

The general syntax is

1. **SELECT column-names FROM table-name**
2. **WHERE EXISTS**
3. **(SELECT column-name FROM table-name WHERE condition)**

**Problem:** Find suppliers with products over $100.

1. **SELECT CompanyName FROM Supplier**
2. **WHERE EXISTS**
3. **(SELECT ProductName FROM Product**
4. **WHERE SupplierId = Supplier.Id AND UnitPrice > 100)**

Note: This is a **correlated subquery** because the subquery references the enclosing query (with Supplier.Id).

**Result:** 2 records

|  |
| --- |
| CompanyName |
| Plutzer Lebensmittelgroßmärkte AG |
| Aux joyeux ecclésiastiques |

SELECT column\_name(s) FROM table\_name WHERE EXISTS (SELECT column\_name FROM table\_name WHERE condition);

Цей оператор повертає TRUE та список постачальників з цінами менше за 20:

SELECT SupplierName FROM Suppliers WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =Suppliers.supplierID AND Price < 20);

Цей оператор повертає TRUE та список постачальників з цінами 22:

SELECT SupplierName FROM Suppliers WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =Suppliers.supplierID AND Price = 22);

#### Предикат WHERE IN

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1, value2, ...);

or:

SELECT column\_name(s) FROM table\_name WHERE column\_name IN (*SELECT* STATEMENT);

WHERE IN returns values that matches values in a list or subquery.  
The WHERE IN clause is shorthand for multiple OR conditions.

The IN command allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

The following SQL selects all customers that are located in "Germany", "France" and "UK":

SELECT \* FROM Customers WHERE Country IN ('Germany', 'France', 'UK');

The following SQL selects all customers that are NOT located in "Germany", "France" or "UK":

SELECT \* FROM Customers WHERE Country NOT IN ('Germany', 'France', 'UK');

The following SQL selects all customers that are from the same countries as the suppliers:

SELECT \* FROM Customers WHERE Country IN (SELECT Country FROM Suppliers);

**Problem**: List all suppliers from the USA, UK, OR Japan

1. **SELECT Id, CompanyName, City, Country**
2. **FROM Supplier WHERE Country IN ('USA', 'UK', 'Japan')**

**Result:** 8 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | CompanyName | City | Country |
| 1 | Exotic Liquids | London | UK |
| 2 | New Orleans Cajun Delights | New Orleans | USA |
| 3 | Grandma Kelly's Homestead | Ann Arbor | USA |
| 4 | Tokyo Traders | Tokyo | Japan |
| 6 | Mayumi's | Osaka | Japan |
| 8 | Specialty Biscuits, Ltd. | Manchester | UK |
| 16 | Bigfoot Breweries | Bend | USA |
| 19 | New England Seafood Cannery | Boston | USA |

**Problem**: List all products that are not exactly $10, $20, $30, $40, or $50

1. **SELECT Id, ProductName, UnitPrice**
2. **FROM Product WHERE UnitPrice NOT IN (10,20,30,40,50)**

**Result:** 72 records.

|  |  |  |
| --- | --- | --- |
| Id | ProductName | UnitPrice |
| 1 | Chai | 18.00 |
| 2 | Chang | 19.00 |
| 4 | Chef Anton's Cajun Seasoning | 22.00 |
| 5 | Chef Anton's Gumbo Mix | 21.35 |
| 6 | Grandma's Boysenberry Spread | 25.00 |
| https://www.dofactory.com/img/ellipsis.png | | |

**Problem**: List all customers that are from the same countries as where the suppliers are.

1. **SELECT Id, FirstName, LastName, Country**
2. **FROM Customer**
3. **WHERE Country IN**
4. **(SELECT Country FROM Supplier)**

**Result:** 91 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | FirstName | LastName | Country |
| 1 | Maria | Anders | Germany |
| 4 | Thomas | Hardy | UK |
| 5 | Christina | Berglund | Sweden |
| 6 | Hanna | Moos | Germany |
| 7 | Frédérique | Citeaux | France |
| https://www.dofactory.com/img/ellipsis.png | | | |

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

SELECT \* FROM Customers WHERE Country IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

SELECT \* FROM Customers WHERE Country NOT IN ('Germany', 'France', 'UK');

The following SQL statement selects all customers that are from the same countries as the suppliers:

SELECT \* FROM Customers WHERE Country IN (SELECT Country FROM Suppliers);

#### Предикат IS NULL

NULL is a special value that signifies 'no value'.  
Comparing a column to NULL using the = operator is *undefined*.  
Instead, use WHERE IS NULL or WHERE IS NOT NULL.

The general IS NULL and IS NOT NULL syntax is

**SELECT column-names FROM table-name WHERE column-name IS NULL**

**SELECT column-names FROM table-name WHERE column-name IS NOT NULL**

**Problem**: List all suppliers that have no fax number

**SELECT Id, CompanyName, Phone, Fax FROM Supplier WHERE Fax IS NULL**

**Result:** 16 records

|  |  |  |  |
| --- | --- | --- | --- |
| Id | CompanyName | Phone | Fax |
| 1 | Exotic Liquids | (171) 555-2222 | NULL |
| 2 | New Orleans Cajun Delights | (100) 555-4822 | NULL |
| 4 | Tokyo Traders | (03) 3555-5011 | NULL |
| 5 | Cooperativa de Quesos 'Las Cabras' | (98) 598 76 54 | NULL |
| 6 | Mayumi's | (06) 431-7877 | NULL |
| https://www.dofactory.com/img/ellipsis.png | | | |

**Problem**: List all suppliers that **do** have a fax number

1. **SELECT Id, CompanyName, Phone, Fax**
2. **FROM Supplier**
3. **WHERE Fax IS NOT NULL**

**Result:** 13 records

|  |  |  |  |
| --- | --- | --- | --- |
| Id | CompanyName | Phone | Fax |
| 3 | Grandma Kelly's Homestead | (313) 555-5735 | (313) 555-3349 |
| 7 | Pavlova, Ltd. | (03) 444-2343 | (03) 444-6588 |
| 9 | PB Knäckebröd AB | 031-987 65 43 | 031-987 65 91 |
| 13 | Nord-Ost-Fisch Handelsgesellschaft mbH | (04721) 8713 | (04721) 8714 |
| 14 | Formaggi Fortini s.r.l. | (0544) 60323 | (0544) 60603 |
| 18 | Aux joyeux ecclésiastiques | (1) 03.83.00.68 | (1) 03.83.00.62 |
| 19 | New England Seafood Cannery | (617) 555-3267 | (617) 555-3389 |
| 21 | Lyngbysild | 43844108 | 43844115 |
| 22 | Zaanse Snoepfabriek | (12345) 1212 | (12345) 1210 |
| 24 | G'day, Mate | (02) 555-5914 | (02) 555-4873 |
| 26 | Pasta Buttini s.r.l. | (089) 6547665 | (089) 6547667 |
| 28 | Gai pâturage | 38.76.98.06 | 38.76.98.58 |
| 29 | Forêts d'érables | (514) 555-2955 | (514) 555-2921 |

The IS NULL command is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

SELECT CustomerName, ContactName, Address

FROM Customers

WHERE Address IS NULL;

#### Предикат IS NOT NULL

The IS NOT NULL command is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

SELECT CustomerName, ContactName, Address

FROM Customers

WHERE Address IS NOT NULL;

#### Предикат WHERE LIKE

Use WHERE LIKE when only a fragment of a text value is known.  
The WHERE LIKE clause determines if a character string matches a pattern.  
WHERE LIKE supports two wildcard match options: % and\_.

The general syntax is

**SELECT column-names FROM table-name WHERE column-name LIKE value**

The LIKE command is used in a WHERE clause to search for a specified pattern in a column.

You can use two wildcards with LIKE:

* % - Represents zero, one, or multiple characters
* \_ - Represents a single character (MS Access uses a question mark (?) instead)

The following SQL selects all customers with a CustomerName starting with "a":

SELECT \* FROM Customers WHERE CustomerName LIKE 'a%';

The following SQL selects all customers with a CustomerName ending with "a":

SELECT \* FROM Customers WHERE CustomerName LIKE '%a';

The following SQL selects all customers with a CustomerName that have "or" in any position:

SELECT \* FROM Customers WHERE CustomerName LIKE '%or%';

The following SQL statement selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

SELECT \* FROM Customers WHERE CustomerName LIKE 'a\_%\_%';

Wildcard characters allowed in 'value' are % (percent) and \_ (underscore).

* % (percent) matches any string with zero or more characters.
* \_ (underscore) matches any single character.

**Problem**: List all products with names that start with 'Ca'

**SELECT Id, ProductName, UnitPrice, Package**

**FROM Product WHERE ProductName LIKE 'Ca%'**

**Result:** 2 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | ProductName | UnitPrice | Package |
| 18 | Carnarvon Tigers | 62.50 | 16 kg pkg. |
| 60 | Camembert Pierrot | 34.00 | 15-300 g rounds |

**Problem**: List all products that start with 'Cha' or 'Chan' and have one more character.

1. **SELECT Id, ProductName, UnitPrice, Package**
2. **FROM Product**
3. **WHERE ProductName LIKE 'Cha\_' OR ProductName LIKE 'Chan\_'**

**Result:** 2 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | ProductName | UnitPrice | Package |
| 1 | Chai | 18.00 | 10 boxes x 20 bags |
| 2 | Chang | 19.00 | 24 - 12 oz bottles |

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the LIKE operator:

* % - The percent sign represents zero, one, or multiple characters
* \_ - The underscore represents a single character

**Note:** MS Access uses an asterisk (\*) instead of the percent sign (%), and a question mark (?) instead of the underscore (\_).

The percent sign and the underscore can also be used in combinations!

SELECT column1, column2, ... FROM table\_name WHERE columnN LIKE pattern;

**Tip:** You can also combine any number of conditions using AND or OR operators.

Here are some examples showing different LIKE operators with '%' and '\_' wildcards:

|  |  |
| --- | --- |
| LIKE оператор | Опис |
| WHERE CustomerName LIKE 'a%' | Знайти рядки, що починаються з "a" |
| WHERE CustomerName LIKE '%a' | Знайти рядки, що закінчуються на "a" |
| WHERE CustomerName LIKE '%or%' | Знайти ряки, що мають "or" |
| WHERE CustomerName LIKE '\_r%' | Знайти рядки з "r" у другій позиції |
| WHERE CustomerName LIKE 'a\_%' | Знайти рядки довжиною не менше 2, та починаються з "a" |
| WHERE CustomerName LIKE 'a\_\_%' | Знайти рядки довжиною не менше 3, та починаються з "a" |
| WHERE ContactName LIKE 'a%o' | Знайти рядки, що починаються з "a" та закінчуються "o" |

##### Пошук рядків за шаблоном Wildcard

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the [SQL LIKE](https://www.w3schools.com/sql/sql_like.asp) operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

###### MS Access

|  |  |  |
| --- | --- | --- |
| Символ | Опис | Приклад |
| \* | Represents zero or more characters | bl\* finds bl, black, blue, and blob |
| ? | Represents a single character | h?t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ! | Represents any character not in the brackets | h[!oa]t finds hit, but not hot and hat |
| - | Represents a range of characters | c[a-b]t finds cat and cbt |
| # | Represents any single numeric character | 2#5 finds 205, 215, 225, 235, 245, 255, 265, 275, 285, and 295 |

###### MS SQL

|  |  |  |
| --- | --- | --- |
| Символ | Опис | Приклад |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents a range of characters | c[a-b]t finds cat and cbt |

All the wildcards can also be used in combinations!

У цій таблиці подано різницю між операторами LIKE щр мають '%' та '\_' :

|  |  |
| --- | --- |
| LIKE Operator | Description |
| WHERE CustomerName LIKE 'a%' | Рядки, що починаються з "a" |
| WHERE CustomerName LIKE '%a' | Рядки, що закінчуються на "a" |
| WHERE CustomerName LIKE '%or%' | Рядки, що мають "or" |
| WHERE CustomerName LIKE '\_r%' | Рядки, що мають "r" у другій позиції |
| WHERE CustomerName LIKE 'a\_%\_%' | Рядки, що починаються з "a" і мають довжину не менше 3 символів |
| WHERE ContactName LIKE 'a%o' | Рядки, що починаються з "a" та закінчуються "o" |

##### Використання %у шаблоні пошуку %

The following SQL statement selects all customers with a City starting with "ber":

SELECT \* FROM Customers WHERE City LIKE 'ber%';

The following SQL statement selects all customers with a City containing the pattern "es":

SELECT \* FROM Customers WHERE City LIKE '%es%';

##### Використання \_ у шаблоні пошуку

The following SQL statement selects all customers with a City starting with any character, followed by "ondon":

SELECT \* FROM Customers WHERE City LIKE '\_ondon';

The following SQL statement selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

SELECT \* FROM Customers WHERE City LIKE 'L\_n\_on';

##### Використання [charlist] у шаблоні пошуку

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

SELECT \* FROM Customers WHERE City LIKE '[bsp]%';

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

SELECT \* FROM Customers WHERE City LIKE '[a-c]%';

##### Використання [!charlist] у шаблоні пошуку

Два оператора обирають всіх замовників у City, що не починаються з "b", "s", or "p":

SELECT \* FROM Customers WHERE City LIKE '[!bsp]%';

або:

SELECT \* FROM Customers WHERE City NOT LIKE '[bsp]%';

### Сортування результатів запиту ORDER BY

SELECT (without ORDER BY) returns records in no particular order.  
To ensure a specific sort order use the ORDER BY clause.

ORDER BY allows sorting by one or more columns.  
Records can be returned in ascending or descending order.

The ORDER BY command is used to sort the result set in ascending or descending order.

The ORDER BY command sorts the result set in ascending order by default. To sort the records in descending order, use the DESC keyword.

The following SQL statement selects all the columns from the "Customers" table, sorted by the "CustomerName" column:

SELECT \* FROM Customers ORDER BY CustomerName;

#### The SQL ORDER BY syntax

**SELECT colnames FROM table-name WHERE condition ORDER BY colnames**

##### Пряме сортування ASC

The ASC command is used to sort the data returned in ascending order.

The following SQL statement selects all the columns from the "Customers" table, sorted by the "CustomerName" column:

SELECT \* FROM Customers ORDER BY CustomerName ASC;

##### Зворотнє сортування DESC

The DESC command is used to sort the data returned in descending order.

The following SQL statement selects all the columns from the "Customers" table, sorted descending by the "CustomerName" column:

SELECT \* FROM Customers ORDER BY CustomerName DESC;

**Problem**: List all suppliers in alphabetical order

1. **SELECT CompanyName, ContactName, City, Country**
2. **FROM Supplier ORDER BY CompanyName**

The default sort order is ascending, that is, low-high or a-z.

**Result:** 29 records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | CompanyName | ContactName | City | Country |
| 18 | Aux joyeux ecclésiastiques | Guylène Nodier | Paris | France |
| 16 | Bigfoot Breweries | Cheryl Saylor | Bend | USA |
| 5 | Cooperativa de Quesos 'Las Cabras' | Antonio del Valle Saavedra | Oviedo | Spain |
| 27 | Escargots Nouveaux | Marie Delamare | Montceau | France |
| 1 | Exotic Liquids | Charlotte Cooper | London | UK |
| https://www.dofactory.com/img/ellipsis.png | | | | |

**Problem**: Get a list of all suppliers in *reverse* alphabetical order

1. **SELECT CompanyName, ContactName, City, Country**
2. **FROM Supplier ORDER BY CompanyName DESC**

The keyword DESC denotes descending, i.e., reverse order.

**Result:** 29 records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | CompanyName | ContactName | City | Country |
| 22 | Zaanse Snoepfabriek | Dirk Luchte | Zaandam | Netherlands |
| 4 | Tokyo Traders | Yoshi Nagase | Tokyo | Japan |
| 17 | Svensk Sjöföda AB | Michael Björn | Stockholm | Sweden |
| 8 | Specialty Biscuits, Ltd. | Peter Wilson | Manchester | UK |
| 10 | Refrescos Americanas LTDA | Carlos Diaz | Sao Paulo | Brazil |
| https://www.dofactory.com/img/ellipsis.png | | | | |

**Problem**: List all customers ordered by country, then by city within each country

Note: Ordering by one or more columns is entirely possible.

1. **SELECT FirstName, LastName, City, Country**
2. **FROM Customer ORDER BY Country, City**

**Result:** 91 records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | FirstName | LastName | City | Country |
| 12 | Patricio | Simpson | Buenos Aires | Argentina |
| 54 | Yvonne | Moncada | Buenos Aires | Argentina |
| 64 | Sergio | Gutiérrez | Buenos Aires | Argentina |
| 20 | Roland | Mendel | Graz | Austria |
| 59 | Georg | Pipps | Salzburg | Austria |
| 50 | Catherine | Dewey | Bruxelles | Belgium |
| 76 | Pascale | Cartrain | Charleroi | Belgium |
| https://www.dofactory.com/img/ellipsis.png | | | | |

**Problem**: List all suppliers in the USA, Japan, and Germany,  
ordered by city, then by company name in reverse order.

1. **SELECT Id, CompanyName, City, Country**
2. **FROM Supplier**
3. **WHERE Country IN ('USA', 'Japan', 'Germany')**
4. **ORDER BY Country ASC, CompanyName DESC**

This shows that you can order by more than one column.  
ASC denotes ascending, but is optional as it is the default order.

**Result:** 9 records

|  |  |  |  |
| --- | --- | --- | --- |
| Id | CompanyName | City | Country |
| 12 | Plutzer Lebensmittelgroßmärkte AG | Frankfurt | Germany |
| 13 | Nord-Ost-Fisch Handelsgesellschaft mbH | Cuxhaven | Germany |
| 11 | Heli Süßwaren GmbH & Co. KG | Berlin | Germany |
| 4 | Tokyo Traders | Tokyo | Japan |
| 6 | Mayumi's | Osaka | Japan |
| 2 | New Orleans Cajun Delights | New Orleans | USA |
| 19 | New England Seafood Cannery | Boston | USA |
| 3 | Grandma Kelly's Homestead | Ann Arbor | USA |
| 16 | Bigfoot Breweries | Bend | USA |

**Problem**: Show all orders, sorted by total amount, the largest first, within each year

1. **SELECT Id, OrderDate, CustomerId, TotalAmount FROM [Order]**
2. **ORDER BY YEAR(OrderDate) ASC, TotalAmount DESC**

Note: DESC means descending.  
Also, [Order] must be bracketed because it is a keywork in SQL.

Result: 830 records.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | OrderDate | CustomerId | TotalAmount |
| 125 | 2012-12-04 00:00:00.000 | 62 | 12281.20 |
| 106 | 2012-11-13 00:00:00.000 | 59 | 10741.60 |
| 113 | 2012-11-22 00:00:00.000 | 7 | 7390.20 |
| https://www.dofactory.com/img/ellipsis.png | | | |
| 144 | 2012-12-23 00:00:00.000 | 17 | 86.40 |
| 24 | 2012-08-01 00:00:00.000 | 75 | 48.00 |
| 177 | 2013-01-23 00:00:00.000 | 51 | 11493.20 |
| 170 | 2013-01-16 00:00:00.000 | 73 | 11283.20 |
| https://www.dofactory.com/img/ellipsis.png | | | |
| 560 | 2013-12-31 00:00:00.000 | 27 | 18.40 |
| 535 | 2013-12-17 00:00:00.000 | 12 | 12.50 |
| 618 | 2014-02-02 00:00:00.000 | 63 | 17250.00 |
| 783 | 2014-04-17 00:00:00.000 | 71 | 16321.90 |
| https://www.dofactory.com/img/ellipsis.png | | | |

Notice the year breakpoints: 2012 - 2013 and 2013 - 2014. Each year starts with the highest TotalAmounts. This shows that other data types, such as numbers, strings, dates, and bits can also be sorted. YEAR is a built-in function that returns the year from a date.

### Групування даних GROUP BY

The GROUP BY command is used to group the result set (used with aggregate functions: COUNT, MAX, MIN, SUM, AVG). The GROUP BY clause groups records into summary rows. It returns one record for each group.

The following SQL lists the number of customers in each country:

SELECT COUNT(CustomerID), Country

FROM Customers GROUP BY Country;

The following SQL lists the number of customers in each country, sorted high to low:

SELECT COUNT(CustomerID), Country

FROM Customers GROUP BY Country

ORDER BY COUNT(CustomerID) DESC;

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

SELECT column\_name(s) FROM table\_name WHERE condition GROUP BY column\_name(s) ORDER BY column\_name(s);

#### Просте групування даних

The following SQL lists the number of customers in each country:

SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country;

The following SQL lists the number of customers in each country, sorted high to low:

SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country ORDER BY COUNT(CustomerID) DESC;

The general syntax is

**SELECT col-names FROM table-name WHERE condition GROUP BY col-names**

The general syntax with ORDER BY is:

**SELECT column-names FROM table-name**

**WHERE condition GROUP BY column-names ORDER BY column-names**

**Problem:** List the number of customers in each country.

1. **SELECT COUNT(Id), Country FROM Customer GROUP BY Country**

**Result:** 21 records.

|  |  |
| --- | --- |
| Count | Country |
| 3 | Argentina |
| 2 | Austria |
| 2 | Belgium |
| 9 | Brazil |
| 3 | Canada |
| https://www.dofactory.com/img/ellipsis.png | |

**Problem:** List the number of customers in each country sorted high to low

**SELECT COUNT(Id), Country FROM Customer**

**GROUP BY Country ORDER BY COUNT(Id) DESC**

**Result:** 21 records.

|  |  |
| --- | --- |
| Count | Country |
| 13 | USA |
| 11 | France |
| 11 | Germany |
| 9 | Brazil |
| 7 | UK |
| https://www.dofactory.com/img/ellipsis.png | |

**Problem:** List the total amount ordered for each customer

1. **SELECT SUM(O.TotalAmount) AS SUM, C.FirstName, C.LastName**
2. **FROM [Order] O JOIN Customer C**
3. **ON O.CustomerId = C.Id**
4. **GROUP BY C.FirstName, C.LastName**
5. **ORDER BY SUM(O.TotalPrice) DESC**

This query JOINs Order with Customer to obtain customer names

**Result:** 89 records.

|  |  |  |
| --- | --- | --- |
| Sum | FirstName | LastName |
| 117483.39 | Horst | Kloss |
| 115673.39 | Jose | Pavarotti |
| 113236.68 | Roland | Mendel |
| 57317.39 | Patricia | McKenna |
| 52245.90 | Paula | Wilson |
| 34101.15 | Mario | Pontes |
| 32555.55 | Maria | Larsson |

#### Предикат HAVING

The HAVING clause is like WHERE but operates on grouped records returned by a GROUP BY. HAVING applies to summarized group records, whereas WHERE applies to individual records. Only the groups that meet the HAVING criteria will be returned.

The HAVING command is used instead of WHERE with aggregate functions.

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

HAVING requires that a GROUP BY clause is present. Both WHERE and HAVING can be used in the same query at the same time.

**Загальний синтаксис**

**SELECT column-names FROM table-name**

**WHERE condition GROUP BY column-names HAVING condition**

The general syntax with ORDER BY is:

**SELECT column-names FROM table-name**

**WHERE condition GROUP BY column-names HAVING condition**

**ORDER BY column-names**

**Problem:** List the number of customers in each country.  
Only include countries with more than 10 customers.

**SELECT COUNT(Id), Country FROM Customer**

**GROUP BY Country HAVING COUNT(Id) > 10**

**Result:** 3 records

|  |  |
| --- | --- |
| Count | Country |
| 11 | France |
| 11 | Germany |
| 13 | USA |

**Problem:** List the number of customers in each country, except the USA, sorted high to low. Only include countries with 9 or more customers.

**SELECT COUNT(Id), Country FROM Customer**

**WHERE Country <> 'USA'**

**GROUP BY Country HAVING COUNT(Id) >= 9 ORDER BY COUNT(Id) DESC**

**Result:** 3 records

|  |  |
| --- | --- |
| Count | Country |
| 11 | France |
| 11 | Germany |
| 9 | Brazil |

**Problem:** List all customer with average orders between $1000 and $1200.

1. **SELECT AVG(TotalAmount), FirstName, LastName**
2. **FROM [Order] O JOIN Customer C ON O.CustomerId = C.Id**
3. **GROUP BY FirstName, LastName**
4. **HAVING AVG(TotalAmount) BETWEEN 1000 AND 1200**

**Result:** 10 records

|  |  |  |
| --- | --- | --- |
| Average | FirstName | LastName |
| 1081.215000 | Miguel | Angel Paolino |
| 1063.420000 | Isabel | de Castro |
| 1008.440000 | Alexander | Feuer |
| 1062.038461 | Thomas | Hardy |
| 1107.806666 | Pirkko | Koskitalo |
| 1174.945454 | Janete | Limeira |
| 1073.621428 | Antonio | Moreno |
| 1065.385000 | Rita | Müller |
| 1183.010000 | José | Pedro Freyre |
| 1057.386666 | Carine | Schmitt |

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5;

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

SELECT COUNT(CustomerID), Country

FROM Customers

GROUP BY Country

HAVING COUNT(CustomerID) > 5

ORDER BY COUNT(CustomerID) DESC;

The following SQL statement lists the employees that have registered more than 10 orders:

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders

FROM (Orders

INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)

GROUP BY LastName

HAVING COUNT(Orders.OrderID) > 10;

The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders

FROM Orders

INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID

WHERE LastName = 'Davolio' OR LastName = 'Fuller'

GROUP BY LastName HAVING COUNT(Orders.OrderID) > 25;

#### GROUP BY With JOIN

The following SQL statement lists the number of orders sent by each shipper:

SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders  
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID  
GROUP BY ShipperName;

### Комбінування записів кількох таблиць JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

|  |  |  |
| --- | --- | --- |
| OrderID | CustomerID | OrderDate |
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Then, look at a selection from the "Customers" table:

|  |  |  |  |
| --- | --- | --- | --- |
| CustomerID | CustomerName | ContactName | Country |
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

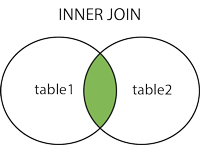
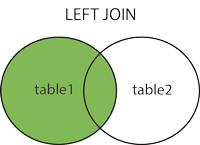
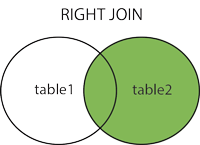
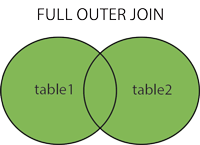
and it will produce something like this:

|  |  |  |
| --- | --- | --- |
| OrderID | CustomerName | OrderDate |
| 10308 | Ana Trujillo Emparedados y helados | 9/18/1996 |
| 10365 | Antonio Moreno Taquería | 11/27/1996 |
| 10383 | Around the Horn | 12/16/1996 |
| 10355 | Around the Horn | 11/15/1996 |
| 10278 | Berglunds snabbköp | 8/12/1996 |

#### Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

* **(INNER) JOIN**: Returns records that have matching values in both tables
* **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table
* **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table
* **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table

A SQL JOIN combines records from two tables.  
A JOIN locates related column values in the two tables.  
A query can contain zero, one, or multiple JOIN operations.  
INNER JOIN is the same as JOIN; the keyword INNER is optional.

Note: All INNER and OUTER keywords are optional.

Details about the different JOINs are available in subsequent tutorial pages.

The general syntax is

**SELECT column-names**

**FROM table-name1 JOIN table-name2 ON column-name1 = column-name2**

**WHERE condition**

The general syntax with INNER is:

**SELECT column-names**

**FROM table-name1 INNER JOIN table-name2 ON column-name1 = column-name2 WHERE condition**

Note: The INNER keyword is optional: it is the default as well as the most commmonly used JOIN operation.

**Problem:** List all orders with customer information

1. **SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country**
2. **FROM [Order] JOIN Customer**
3. **ON [Order].CustomerId = Customer.Id**

In this example using [table Aliases](https://www.dofactory.com/sql/alias) for [Order] and Customer might have been useful.

**Result:** 830 records.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OrderNumber | TotalAmount | FirstName | LastName | City | Country |
| 542378 | 440.00 | Paul | Henriot | Reims | France |
| 542379 | 1863.40 | Karin | Josephs | Münster | Germany |
| 542380 | 1813.00 | Mario | Pontes | Rio de Janeiro | Brazil |
| 542381 | 670.80 | Mary | Saveley | Lyon | France |
| 542382 | 3730.00 | Pascale | Cartrain | Charleroi | Belgium |
| 542383 | 1444.80 | Mario | Pontes | Rio de Janeiro | Brazil |
| 542384 | 625.20 | Yang | Wang | Bern | Switzerland |
| https://www.dofactory.com/img/ellipsis.png | | | | | |

**Problem:** List all orders with product names, quantities, and prices

1. **SELECT O.OrderNumber, CONVERT(date,O.OrderDate) AS Date,**
2. **P.ProductName, I.Quantity, I.UnitPrice**
3. **FROM [Order] O**
4. **JOIN OrderItem I ON O.Id = I.OrderId**
5. **JOIN Product P ON P.Id = I.ProductId**
6. **ORDER BY O.OrderNumber**

This query performs two JOIN operations with 3 tables.

The O, I, and P are [table Aliases](https://www.dofactory.com/sql/alias). Date is a [column Alias](https://www.dofactory.com/sql/alias).

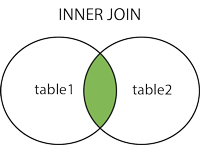
**Result:** 2155 records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OrderNumber | Date | ProductName | Quantity | UnitPrice |
| 542378 | 7/4/2012 12:00:00 AM | Queso Cabrales | 12 | 14.00 |
| 542378 | 7/4/2012 12:00:00 AM | Singaporean Hokkien Fried Mee | 10 | 9.80 |
| 542378 | 7/4/2012 12:00:00 AM | Mozzarella di Giovanni | 5 | 34.80 |
| 542379 | 7/5/2012 12:00:00 AM | Tofu | 9 | 18.60 |
| 542379 | 7/5/2012 12:00:00 AM | Manjimup Dried Apples | 40 | 42.40 |
| 542380 | 7/8/2012 12:00:00 AM | Jack's New England Clam Chowder | 10 | 7.70 |
| 542380 | 7/8/2012 12:00:00 AM | Manjimup Dried Apples | 35 | 42.40 |
| 542380 | 7/8/2012 12:00:00 AM | Louisiana Fiery Hot Pepper Sauce | 15 | 16.80 |
| 542381 | 7/8/2012 12:00:00 AM | Gustaf's Knäckebröd | 6 | 16.80 |
| 542381 | 7/8/2012 12:00:00 AM | Ravioli Angelo | 15 | 15.60 |
| 542381 | 7/8/2012 12:00:00 AM | Louisiana Fiery Hot Pepper Sauce | 20 | 16.80 |
| 542382 | 7/9/2012 12:00:00 AM | Sir Rodney's Marmalade | 40 | 64.80 |
| 542382 | 7/9/2012 12:00:00 AM | Geitost | 25 | 2.00 |
| https://www.dofactory.com/img/ellipsis.png | | | | |

#### INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

SELECT column\_name(s) FROM table1 INNER JOIN table2 ON table1.column\_name = table2.column\_name;



Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And a selection from the "Customers" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

The following SQL statement selects all orders with customer information:

SELECT Orders.OrderID, Customers.CustomerName FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

**Note:** The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

#### JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName

FROM ((Orders INNER JOIN Customers ON Orders.CustomerID =

Customers.CustomerID)

INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);

#### SQL LEFT JOIN

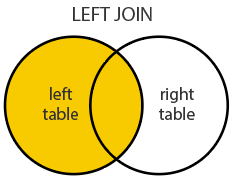
A LEFT JOIN performs a join starting with the first (left-most) table.  
Then, any matched records from the second table (right-most) will be included.  
LEFT JOIN and LEFT OUTER JOIN are the same.

The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.

SELECT column\_name(s) FROM tabl

LEFT JOIN table2 ON table1.column\_name = table2.column\_name;

**Note:** In some databases LEFT JOIN is called LEFT OUTER JOIN.



The general LEFT JOIN syntax is

1. **SELECT column-names**
2. **FROM table-name1 LEFT JOIN table-name2**
3. **ON column-name1 = column-name2**
4. **WHERE condition**

The general LEFT OUTER JOIN syntax is

1. **SELECT column-names**
2. **FROM table-name1 LEFT OUTER JOIN table-name2**
3. **ON column-name1 = column-name2**
4. **WHERE condition**

**Problem:** List all customers and the total amount they spent irrespective  
whether they placed any orders or not.

1. **SELECT OrderNumber, TotalAmount, FirstName, LastName, City, Country**
2. **FROM Customer C LEFT JOIN [Order] O ON O.CustomerId = C.Id**
3. **ORDER BY TotalAmount**

Note: The ORDER BY TotalAmount shows the customers without orders first (i.e. TotalMount is NULL).

**Result:** 832 records

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OrderNumber | TotalAmount | FirstName | LastName | City | Country |
| NULL | NULL | Diego | Roel | Madrid | Spain |
| NULL | NULL | Marie | Bertrand | Paris | France |
| 542912 | 12.50 | Patricio | Simpson | Buenos Aires | Argentina |
| 542937 | 18.40 | Paolo | Accorti | Torino | Italy |
| 542897 | 28.00 | Pascale | Cartrain | Charleroi | Belgium |
| 542716 | 28.00 | Maurizio | Moroni | Reggio Emilia | Italy |
| 543028 | 30.00 | Yvonne | Moncada | Buenos Aires | Argentina |
| 543013 | 36.00 | Fran | Wilson | Portland | US |
|  | | | | | |

The following SQL statement will select all customers, and any orders they might have:

SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID  
ORDER BY Customers.CustomerName;

**Note:** The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

#### SQL RIGHT JOIN

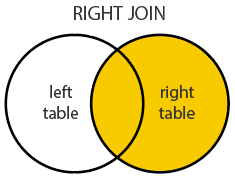
A RIGHT JOIN performs a join starting with the second (right-most) table and then any matching first (left-most) table records. RIGHT JOIN and RIGHT OUTER JOIN are the same.

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

SELECT column\_name(s) FROM table1 RIGHT JOIN table2 ON table1.column\_name = table2.column\_name;

**Note:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.

**Note:** The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).



The general syntax is

1. **SELECT column-names FROM table-name1 RIGHT JOIN table-name2**
2. **ON column-name1 = column-name2**
3. **WHERE condition**

The general RIGHT OUTER JOIN syntax is:

1. **SELECT column-names FROM table-name1 RIGHT OUTER JOIN table-name2**
2. **ON column-name1 = column-name2 WHERE condition**

**Problem:** List customers that have not placed orders

1. **SELECT TotalAmount, FirstName, LastName, City, Country**
2. **FROM [Order] O RIGHT JOIN Customer C**
3. **ON O.CustomerId = C.Id**
4. **WHERE TotalAmount IS NULL**

This returns customers that, when joined, have no matching order.

**Result:** 2 records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TotalAmount | FirstName | LastName | City | Country |
| NULL | Diego | Roel | Madrid | Spain |
| NULL | Marie | Bertrand | Paris | France |

The following SQL statement will return all employees, and any orders they might have placed:

SELECT Orders.OrderID, Employees.LastName, Employees.FirstName  
FROM Orders  
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID  
ORDER BY Orders.OrderID;

**Problem:** List all customers (with or without orders) and a count of the orders that include a *2kg box with Konbu* (product with Id = 13)

1. **SELECT DISTINCT (C.Id), Firstname, LastName, COUNT(O.Id) AS Orders**
2. **FROM [Order] O**
3. **JOIN OrderItem I ON O.Id = I.OrderId AND I.ProductId = 13**
4. **RIGHT JOIN Customer C ON C.Id = O.CustomerId**
5. **GROUP BY C.Id, FirstName, LastName**
6. **ORDER BY COUNT(O.Id)**

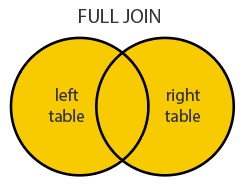
This returns all customers whether they have orders or not. The ones with orders we check if productId = 13 is present.

**Result:** 91 records (all customers)

|  |  |  |  |
| --- | --- | --- | --- |
| Id | FirstName | LastName | Orders |
| 1 | Maria | Anders | 0 |
| 3 | Antonio | Moreno | 0 |
| 6 | Hanna | Moos | 0 |
| https://www.dofactory.com/img/ellipsis.png | | | |
| 36 | Yoshi | Latimer | 1 |
| 88 | Paula | Patenta | 1 |
| 85 | Paul | Henriot | 1 |
| https://www.dofactory.com/img/ellipsis.png | | | |
| 11 | Victoria | Ashworth | 2 |
| 17 | Sven | Ottlieb | 2 |
| 20 | Roland | Mendel | 2 |
| https://www.dofactory.com/img/ellipsis.png | | | |
| 71 | Jose | Pavarotti | 4 |

#### Оператори FULL JOIN FULL OUTER JOIN

FULL JOIN returns all matching records from both tables whether the other table matches or not. Be aware that a FULL JOIN can potentially return very large datasets.  
These two: FULL JOIN and FULL OUTER JOIN are the same.



The general syntax is:

1. **SELECT column-names**
2. **FROM table-name1 FULL JOIN table-name2**
3. **ON column-name1 = column-name2**
4. **WHERE condition**

The general FULL OUTER JOIN syntax is:

1. **SELECT column-names**
2. **FROM table-name1 FULL OUTER JOIN table-name2**
3. **ON column-name1 = column-name2**
4. **WHERE condition**

**Problem:** Match all customers and suppliers by country

1. **SELECT C.FirstName, C.LastName, C.Country AS CustomerCountry,**
2. **S.Country AS SupplierCountry, S.CompanyName**
3. **FROM Customer C FULL JOIN Supplier S ON C.Country = S.Country**
4. **ORDER BY C.Country, S.Country**

This returns suppliers that have no customers in their country,  
and customers that have no suppliers in their country,  
and customers and suppliers that are from the same country.

**Result:** 195 records

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FirstName | LastName | CustomerCountry | SupplierCountry | CompanyName |
| NULL | NULL | NULL | Australia | Pavlova, Ltd. |
| NULL | NULL | NULL | Australia | G'day, Mate |
| NULL | NULL | NULL | Japan | Tokyo Traders |
| NULL | NULL | NULL | Japan | Mayumi's |
| NULL | NULL | NULL | Netherlands | Zaanse Snoepfabriek |
| NULL | NULL | NULL | Singapore | Leka Trading |
| Patricio | Simpson | Argentina | NULL | NULL |
| Yvonne | Moncada | Argentina | NULL | NULL |
| Sergio | Gutiérrez | Argentina | NULL | NULL |
| Georg | Pipps | Austria | NULL | NULL |
| Roland | Mendel | Austria | NULL | NULL |
| Pascale | Cartrain | Belgium | NULL | NULL |
| Catherine | Dewey | Belgium | NULL | NULL |
| Bernardo | Batista | Brazil | Brazil | Refrescos Americanas LTDA |
| Lúcia | Carvalho | Brazil | Brazil | Refrescos Americanas LTDA |
| Janete | Limeira | Brazil | Brazil | Refrescos Americanas LTDA |
| Aria | Cruz | Brazil | Brazil | Refrescos Americanas LTDA |
| André | Fonseca | Brazil | Brazil | Refrescos Americanas LTDA |
| Mario | Pontes | Brazil | Brazil | Refrescos Americanas LTDA |
| Pedro | Afonso | Brazil | Brazil | Refrescos Americanas LTDA |
| Paula | Parente | Brazil | Brazil | Refrescos Americanas LTDA |
| Anabela | Domingues | Brazil | Brazil | Refrescos Americanas LTDA |
| Elizabeth | Lincoln | Canada | Canada | Ma Maison |
| Elizabeth | Lincoln | Canada | Canada | Forêts d'érables |
| Yoshi | Tannamuri | Canada | Canada | Ma Maison |
| Yoshi | Tannamuri | Canada | Canada | Forêts d'érables |
| Jean | Fresnière | Canada | Canada | Ma Maison |
| https://www.dofactory.com/img/ellipsis.png | | | | |

The FULL OUTER JOIN command returns all rows when there is a match in either left table or right table.

The following SQL statement selects all customers, and all orders:

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

FULL OUTER

JOIN Orders ON Customers.CustomerID=Orders.CustomerID

ORDER BY Customers.CustomerName;

**Note:** The FULL OUTER JOIN keyword returns all the rows from the left table (Customers), and all the rows from the right table (Orders). If there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

#### Оператор Self JOIN

A self JOIN occurs when a table takes a 'selfie', that is, it JOINs with itself.  
A self JOIN is a regular join but the table that it joins to is itself.

A self JOIN is a regular join, but the table is joined with itself.

SELECT column\_name(s) FROM table1 T1, table1 T2 WHERE condition;

T1 and T2 are different table aliases for the same table.

The following SQL statement matches customers that are from the same city:

SELECT A.CustomerName AS CustomerName1,

B.CustomerName AS CustomerName2, A.City

FROM Customers A, Customers B

WHERE A.CustomerID <> B.CustomerID AND A.City = B.City

ORDER BY A.City;

This can be useful when modeling hierarchies.

SELF JOINs are also useful for comparisons within a table.

1. **SELECT column-names**
2. **FROM table-name T1 JOIN table-name T2**
3. **WHERE condition**

T1 and T2 are different table Aliases for the same table

**Problem:** Match customers that are from the same city and country

1. **SELECT B.FirstName AS FirstName1, B.LastName AS LastName1,**
2. **A.FirstName AS FirstName2, A.LastName AS LastName2,**
3. **B.City, B.Country**
4. **FROM Customer A JOIN Customer B**
5. **ON A.Id <> B.Id**
6. **AND A.City = B.City**
7. **AND A.Country = B.Country**
8. **ORDER BY A.Country**

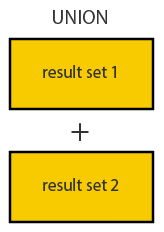
A and B are Aliases for the same Customer table.

**Result:** 88 records

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| FirstName1 | LastName1 | FirstName2 | LastName2 | City | Country |
| Patricio | Simpson | Yvonne | Moncada | Buenos Aires | Argentina |
| Patricio | Simpson | Sergio | Gutiérrez | Buenos Aires | Argentina |
| Yvonne | Moncada | Patricio | Simpson | Buenos Aires | Argentina |
| Yvonne | Moncada | Sergio | Gutiérrez | Buenos Aires | Argentina |
| Sergio | Gutiérrez | Patricio | Simpson | Buenos Aires | Argentina |
| Sergio | Gutiérrez | Yvonne | Moncada | Buenos Aires | Argentina |
| Anabela | Domingues | Lúcia | Carvalho | Sao Paulo | Brazil |
| Anabela | Domingues | Aria | Cruz | Sao Paulo | Brazil |
| Anabela | Domingues | Pedro | Afonso | Sao Paulo | Brazil |
| Bernardo | Batista | Janete | Limeira | Rio de Janeiro | Brazil |
| Bernardo | Batista | Mario | Pontes | Rio de Janeiro | Brazil |
| Lúcia | Carvalho | Anabela | Domingues | Sao Paulo | Brazil |
| Lúcia | Carvalho | Aria | Cruz | Sao Paulo | Brazil |
| Lúcia | Carvalho | Pedro | Afonso | Sao Paulo | Brazil |
| Janete | Limeira | Bernardo | Batista | Rio de Janeiro | Brazil |
| Janete | Limeira | Mario | Pontes | Rio de Janeiro | Brazil |
| Aria | Cruz | Anabela | Domingues | Sao Paulo | Brazil |
| Aria | Cruz | Lúcia | Carvalho | Sao Paulo | Brazil |
| Aria | Cruz | Pedro | Afonso | Sao Paulo | Brazil |
| Mario | Pontes | Bernardo | Batista | Rio de Janeiro | Brazil |
| Mario | Pontes | Janete | Limeira | Rio de Janeiro | Brazil |
| Pedro | Afonso | Anabela | Domingues | Sao Paulo | Brazil |
| Pedro | Afonso | Lúcia | Carvalho | Sao Paulo | Brazil |
| Pedro | Afonso | Aria | Cruz | Sao Paulo | Brazil |
| Dominique | Perrier | Marie | Bertrand | Paris | France |
| Marie | Bertrand | Dominique | Perrier | Paris | France |
| Janine | Labrune | Carine | Schmitt | Nantes | France |
| Carine | Schmitt | Janine | Labrune | Nantes | France |
| **FULL OUTER JOIN**  The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.  **Note:** FULL OUTER JOIN can potentially return very large result-sets!  **Tip:** FULL OUTER JOIN and FULL JOIN are the same.  SELECT column\_name(s) FROM table1 FULL OUTER JOIN table2 ON table1.column\_name = table2.column\_name WHERE condition;  SQL FULL OUTER JOIN  The following SQL statement selects all customers, and all orders:  SELECT Customers.CustomerName, Orders.OrderID  FROM Customers  FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID ORDER BY Customers.CustomerName;  **Note:** The FULL OUTER JOIN keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well. | | | | | |

### Об’єднання результатів запитів UNION

UNION combines the result sets of two different queries.  
Column data types in the two queries must match.  
A UNION combines the results by column position rather than column name.



The general syntax is

1. **SELECT column-names1 FROM table-name1**
2. **UNION**
3. **SELECT column-names2 FROM table-name2**

**Problem:** List all companies, including suppliers and customers.

1. **SELECT 'Customer' As Type,**
2. **FirstName + ' ' + LastName AS ContactName,**
3. **City, Country, Phone**
4. **FROM Customer**
5. **UNION**
6. **SELECT 'Supplier',**
7. **ContactName, City, Country, Phone**
8. **FROM Supplier**

Result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | ContactName | City | Country | Phone |
| Customer | Alejandra Camino | Madrid | Spain | (91) 745 6200 |
| Customer | Alexander Feuer | Leipzig | Germany | 0342-023176 |
| Customer | Ana Trujillo | México D.F. | Mexico | (5) 555-4729 |
| Customer | Anabela Domingues | Sao Paulo | Brazil | (11) 555-2167 |
| https://www.dofactory.com/img/ellipsis.png | | | | |
| Supplier | Anne Heikkonen | Lappeenranta | Finland | (953) 10956 |
| Supplier | Antonio del Valle Saavedra | Oviedo | Spain | (98) 598 76 54 |
| Supplier | Beate Vileid | Sandvika | Norway | (0)2-953010 |
| Supplier | Carlos Diaz | Sao Paulo | Brazil | (11) 555 4640 |
| Supplier | Chandra Leka | Singapore | Singapore | 555-8787 |
| Supplier | Chantal Goulet | Ste-Hyacinthe | Canada | (514) 555-2955 |
| Supplier | Charlotte Cooper | London | UK | (171) 555-2222 |

The UNION operator is used to combine the result-set of two or more SELECT statements.

* Each SELECT statement within UNION must have the same number of columns
* The columns must also have similar data types
* The columns in each SELECT statement must also be in the same order

SELECT col\_name(s) FROM table1 UNION SELECT col\_name(s) FROM table2;

#### UNION ALL Syntax

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

SELECT column\_name(s) FROM table1

UNION ALL

SELECT column\_name(s) FROM table2;

**Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

SELECT City FROM Customers

UNION

SELECT City FROM Suppliers ORDER BY City;

**Note:** If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only distinct values. Use UNION ALL to also select duplicate values!

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

SELECT City FROM Customers

UNION ALL

SELECT City FROM Suppliers ORDER BY City;

#### UNION With WHERE

The following SQL statement returns the German cities (only distinct values) from both the "Customers" and the "Suppliers" table:

SELECT City, Country FROM Customers WHERE Country='Germany'

UNION  
SELECT City, Country FROM Suppliers WHERE Country='Germany'ORDER BY City;

#### UNION ALL With WHERE

The following SQL statement returns the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

SELECT City, Country FROM Customers WHERE Country='Germany'

UNION ALL

SELECT City, Country FROM Suppliers WHERE Country='Germany'

ORDER BY City;

#### Another UNION Example

The following SQL statement lists all customers and suppliers:

SELECT 'Customer' AS Type, ContactName, City, Country FROM Customers  
UNION  
SELECT 'Supplier', ContactName, City, Country FROM Suppliers

### Вкладені запити - Subqueries

A subquery is a SQL query within a query.  
They are nested queries that provide data to the enclosing query.

Subqueries can return individual values or a list of records.  
Note that subquery statements are enclosed between parenthesis.

There is no general syntax; subqueries are regular queries placed inside parenthesis.  
Subqueries can be used in different ways and at different locations inside a query:  
Here is a subquery with the IN operator.

1. **SELECT column-names**
2. **FROM table-name1**
3. **WHERE value IN (SELECT column-name**
4. **FROM table-name2**
5. **WHERE condition)**

Підзапити це вирази SELECT, які вбудовані у інший запит, наприклад.

#### Некорельовані підзапити

Розглянемо складний запит, що формує той самий вектор значень для кожного рядку зовнішнього запиту:

1. **SELECT column1 = (SELECT column-name FROM table-name WHERE condition), column-names**
2. **FROM table-name WHЕRE condition**

**Problem:** List products with order quantities greater than 100.

1. **SELECT ProductName**
2. **FROM Product**
3. **WHERE Id IN (SELECT ProductId**
4. **FROM OrderItem**
5. **WHERE Quantity > 100)**

**Result:** 12 records.

|  |
| --- |
| PoductName |
| Guaraná Fantástica |
| Schoggi Schokolade |
| Chartreuse verte |
| Jack's New England Clam Chowder |
| Rogede sild |
| Manjimup Dried Apples |
| Perth Pasties |
| https://www.dofactory.com/img/ellipsis.png |

Створимо таблиці для товарів та замовлень:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | CREATE TABLE Product  (      Id INT AUTO\_INCREMENT PRIMARY KEY,      ProductName VARCHAR(30) NOT NULL,      Manufacturer VARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price DECIMAL NOT NULL  );  CREATE TABLE Order  (      Id INT AUTO\_INCREMENT PRIMARY KEY,      ProductId INT NOT NULL,      ProductCount INT DEFAULT 1,      CreatedAt DATE NOT NULL,      Price DECIMAL NOT NULL,      FOREIGN KEY (ProductId) REFERENCES Products(Id) ON DELETE CASCADE  ); |

Додамо у таблиці:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)  VALUES ('iPhone X', 'Apple', 2, 76000),  ('iPhone 8', 'Apple', 2, 51000),  ('iPhone 7', 'Apple', 5, 42000),  ('Galaxy S9', 'Samsung', 2, 56000),  ('Galaxy S8', 'Samsung', 1, 46000),  ('Honor 10', 'Huawei', 2, 26000),  ('Nokia 8', 'HMD Global', 6, 38000);    INSERT INTO Orders (ProductId, CreatedAt, ProductCount, Price)  VALUES  (      (SELECT Id FROM Products WHERE ProductName='Galaxy S8'),      '2018-05-21',      2,      (SELECT Price FROM Products WHERE ProductName='Galaxy S8')  ),  (      (SELECT Id FROM Products WHERE ProductName='iPhone X'),      '2018-05-23',      1,      (SELECT Price FROM Products WHERE ProductName='iPhone X')  ),  (      (SELECT Id FROM Products WHERE ProductName='iPhone 8'),      '2018-05-21',      1,      (SELECT Price FROM Products WHERE ProductName='iPhone 8')  ); |

Для додавання нових даних у таблицу Order використовуються підзапити.

Перше замовлення було на товар Galaxy S8. Відповідно у таблиці Orders слід зберігти інформацію по замовлення, де ProductId вказує на Id товару Galaxy S8, поле Price – на його ціну. Але на момент написання запиту ми можемо не знати: Id покупця, Id товара, ціну товару. У цьому випадку слід виконати підзапит у дужках

(SELECT Price FROM Products WHERE ProductName='iPhone 8')

У цьому випадку для додавання одного товару виконується два підзапита. Кожний запит повертає один скалярний вираз, напр., числовий ідентиффкатор.

Підзапити можуть виконуватися і до головної таблиці. Наприклад, знайдемо товари з таблиці Product, що мають мінімальну ціну:

SELECT \* FROM Products WHERE Price = (SELECT MIN(Price) FROM Products);

Знайдемо товари з ціною вищою за середню

SELECT \* FROM Products WHERE Price > (SELECT AVG(Price) FROM Products);

#### Корельовані підзапити

У попередніх випадках результат підзапиту не залежав від рядків, які вибіраються у головному запиті. Такий підзапит виконується тільки один раз для свого зовнішнього запиту, тому цей підзапит називають некорельованим.

Існують також **корельовані підзапити** (correlated subquery), результати яких залежать від рядків, які вибірає зовнішній запит.

**Problem:** List all customers with their total number of orders

1. **SELECT FirstName, LastName,**
2. **OrderCount = (SELECT COUNT(O.Id)**
3. **FROM [Order] O**
4. **WHERE O.CustomerId = C.Id) FROM Customer C**

This is a **correlated subquery** because the subquery references the enclosing query (i.e. the C.Id in the WHERE clause).

**Result:** 91 records

|  |  |  |
| --- | --- | --- |
| FirstName | LastName | OrderCount |
| Maria | Anders | 6 |
| Ana | Trujillo | 4 |
| Antonio | Moreno | 7 |
| Thomas | Hardy | 13 |
| Christina | Berglund | 18 |
| https://www.dofactory.com/img/ellipsis.png | | |

Наприклад, виберемо всі замовлення з Orders та додамо до них інформацію про товар:

SELECT  CreatedAt, Price,

        (SELECT ProductName FROM Products

        WHERE Products.Id = Orders.ProductId) AS Product

FROM Orders;

Для кожного рядка з Orders буде виконано підзапит, результат якого залежить від ProductId. Кожного разу підзапит повертає різні дані.

Корельований підзапит може виконуватися й для таблиці заовнішнього запиту.

Наприклад, виберемо Products ті товари, ціна яких вище за середнб фіню товарів для цього виробника:

SELECT ProductName,

       Manufacturer,

       Price,

        (SELECT AVG(Price) FROM Products AS SubProds

         WHERE SubProds.Manufacturer=Prods.Manufacturer)  AS AvgPrice

FROM Products AS Prods

WHERE Price >

    (SELECT AVG(Price) FROM Products AS SubProds

     WHERE SubProds.Manufacturer=Prods.Manufacturer);

Тут є два корельовани= підзапита. Перший підзапит визначає специфікацію AvgPrice Та виконується для кожного рядка з таблиці Products. У підзапит передаєтся виробник товару й на його підставі обірається середня ціна для товарів саме цьогоо виробника. Виробник товарів може відрізнятися, тому й результат підзапиту у кожному випадку також може бути свій.

Другий підзапит побудований аналогічно, тільки він використовуються для фільтрації даних з таблиці Products. Він виконується кожного разу для кожного рядка окремо.

Для унеможливлення подвійної фільтрації у підзапиті при порівняння виробників (SubProds.Manufacturer=Prods.Manufacturer) для зовнішньої вибірки є аліас (псевдоним) Prods, а для вибірки з підзапитів визначено аліас ( псевдоним) SubProds.

у зв'язку з виконанням корельованих запитів окремо для кожно рядка зовнішнього запиту, швидкість всього запиту значно знижується.

## Копіювання таблиць оператором SELECT INTO

SELECT INTO copies data from one table into a brand new table.  
This operation creates a new table in the default filegroup.

Copy all columns into a new table:

SELECT \* INTO newtable [IN externaldb]

FROM oldtable WHERE condition;

Copy only some columns into a new table:

SELECT column1, column2, column3, ...

INTO newtable [IN externaldb]

FROM oldtable WHERE condition;

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.

The following SQL statement creates a backup copy of Customers:

SELECT \* INTO CustomersBackup2017 FROM Customers;

The following SQL statement uses the IN clause to copy the table into a new table in another database:

SELECT \* INTO CustomersBackup2017 IN 'Backup.mdb'

FROM Customers;

The following SQL statement copies only a few columns into a new table:

SELECT CustomerName, ContactName INTO CustomersBackup2017

FROM Customers;

The following SQL statement copies only the German customers into a new table:

SELECT \* INTO CustomersGermany

FROM Customers WHERE Country = 'Germany';

The following SQL statement copies data from more than one table into a new table:

SELECT Customers.CustomerName, Orders.OrderID

INTO

CustomersOrderBackup2017

FROM Customers LEFT JOIN Orders

ON Customers.CustomerID = Orders.CustomerID;

**Tip:** SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \* INTO newtable FROM oldtable WHERE 1 = 0;

The general syntax is

1. **SELECT column-names**
2. **INTO new-table-name FROM table-name WHERE condition**

The new table will have column names as specified in the query.

**Problem:** Copy all suppliers from the USA to a new SupplierUSA table.

1. **SELECT \* INTO SupplierUSA FROM Supplier WHERE Country = 'USA'**

**Result:** 4 rows inserted

Here are the records in the newly created table SupplierUSA:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Id | CompanyName | ContactName | City | Country | Phone | Fax |
| 2 | New Orleans Cajun Delights | Shelley Burke | New Orleans | USA | (100) 555-4822 | NULL |
| 3 | Grandma Kelly's Homestead | Regina Murphy | Ann Arbor | USA | (313) 555-5735 | (313) 555-3349 |
| 16 | Bigfoot Breweries | Cheryl Saylor | Bend | USA | (100) 555-4822 | NULL |
| 19 | New England Seafood Cannery | Robb Merchant | Boston | USA | (617) 555-3267 | (617) 555-3389 |

# Підготовлені оператори Prepared Statements

MySQL 8.0 provides support for server-side prepared statements. This support takes advantage of the efficient client/server binary protocol. Using prepared statements with placeholders for parameter values has the following benefits:

Less overhead for parsing the statement each time it is executed. Typically, database applications process large volumes of almost-identical statements, with only changes to literal or variable values in clauses such as WHERE for queries and deletes, SET for updates, and VALUES for inserts.

Protection against SQL injection attacks. The parameter values can contain unescaped SQL quote and delimiter characters.

The following sections provide an overview of the characteristics of prepared statements:

## Prepared Statements in Application Programs

You can use server-side prepared statements through client programming interfaces, including the [MySQL C API client library](https://dev.mysql.com/doc/c-api/8.0/en/) for C programs, [MySQL Connector/J](https://dev.mysql.com/doc/connector-j/8.0/en/) for Java programs, and [MySQL Connector/NET](https://dev.mysql.com/doc/connector-net/en/) for programs using .NET technologies. For example, the C API provides a set of function calls that make up its prepared statement API. See [C API Prepared Statements](https://dev.mysql.com/doc/c-api/8.0/en/c-api-prepared-statements.html). Other language interfaces can provide support for prepared statements that use the binary protocol by linking in the C client library, one example being the [mysqli extension](http://php.net/mysqli), available in PHP 5.0 and higher.

## Prepared Statements in SQL Scripts

An alternative SQL interface to prepared statements is available. This interface is not as efficient as using the binary protocol through a prepared statement API, but requires no programming because it is available directly at the SQL level:

You can use it when no programming interface is available to you.

You can use it from any program that can send SQL statements to the server to be executed, such as the [**mysql**](https://dev.mysql.com/doc/refman/8.0/en/mysql.html) client program.

You can use it even if the client is using an old version of the client library.

SQL syntax for prepared statements is intended to be used for situations such as these:

To test how prepared statements work in your application before coding it.

To use prepared statements when you do not have access to a programming API that supports them.

To interactively troubleshoot application issues with prepared statements.

To create a test case that reproduces a problem with prepared statements, so that you can file a bug report.

## PREPARE, EXECUTE, DEALLOCATE PREPARE

SQL syntax for prepared statements is based on three SQL statements:

[EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) executes a prepared.

[DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) releases a prepared statement.

The following examples show two equivalent ways of preparing a statement that computes the hypotenuse of a triangle given the lengths of the two sides.

The first example shows how to create a prepared statement by using a string literal to supply the text of the statement:

mysql> PREPARE stmt1 FROM 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';

mysql> SET @a = 3;

mysql> SET @b = 4;

mysql> EXECUTE stmt1 USING @a, @b;

+------------+

| hypotenuse |

+------------+

| 5 |

+------------+

mysql> DEALLOCATE PREPARE stmt1;

The second example is similar, but supplies the text of the statement as a user variable:

mysql> SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';

mysql> PREPARE stmt2 FROM @s;

mysql> SET @a = 6;

mysql> SET @b = 8;

mysql> EXECUTE stmt2 USING @a, @b;

+------------+

| hypotenuse |

+------------+

| 10 |

+------------+

mysql> DEALLOCATE PREPARE stmt2;

Here is an additional example that demonstrates how to choose the table on which to perform a query at runtime, by storing the name of the table as a user variable:

mysql> USE test;

mysql> CREATE TABLE t1 (a INT NOT NULL);

mysql> INSERT INTO t1 VALUES (4), (8), (11), (32), (80);

mysql> SET @table = 't1';

mysql> SET @s = CONCAT('SELECT \* FROM ', @table);

mysql> PREPARE stmt3 FROM @s;

mysql> EXECUTE stmt3;

+----+

| a |

+----+

| 4 |

| 8 |

| 11 |

| 32 |

| 80 |

+----+

mysql> DEALLOCATE PREPARE stmt3;

A prepared statement is specific to the session in which it was created. If you terminate a session without deallocating a previously prepared statement, the server deallocates it automatically.

A prepared statement is also global to the session. If you create a prepared statement within a stored routine, it is not deallocated when the stored routine ends.

To guard against too many prepared statements being created simultaneously, set the [max\_prepared\_stmt\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_prepared_stmt_count) system variable. To prevent the use of prepared statements, set the value to 0.

SQL Syntax Permitted in Prepared Statements

The following SQL statements can be used as prepared statements:

ALTER TABLE

ALTER USER

ANALYZE TABLE

CACHE INDEX

CALL

CHANGE MASTER

CHECKSUM {TABLE | TABLES}

COMMIT

{CREATE | DROP} INDEX

{CREATE | RENAME | DROP} DATABASE

{CREATE | DROP} TABLE

{CREATE | RENAME | DROP} USER

{CREATE | DROP} VIEW

DELETE

DO

FLUSH {TABLE | TABLES | TABLES WITH READ LOCK | HOSTS | PRIVILEGES

| LOGS | STATUS | MASTER | SLAVE | USER\_RESOURCES}

GRANT

INSERT

INSTALL PLUGIN

KILL

LOAD INDEX INTO CACHE

OPTIMIZE TABLE

RENAME TABLE

REPAIR TABLE

REPLACE

RESET {MASTER | SLAVE}

REVOKE

SELECT

SET

SHOW {WARNINGS | ERRORS}

SHOW BINLOG EVENTS

SHOW CREATE {PROCEDURE | FUNCTION | EVENT | TABLE | VIEW}

SHOW {MASTER | BINARY} LOGS

SHOW {MASTER | SLAVE} STATUS

SLAVE {START | STOP}

TRUNCATE TABLE

UNINSTALL PLUGIN

UPDATE

Other statements are not supported.

For compliance with the SQL standard, which states that diagnostics statements are not preparable, MySQL does not support the following as prepared statements:

SHOW WARNINGS, SHOW COUNT(\*) WARNINGS

SHOW ERRORS, SHOW COUNT(\*) ERRORS

Statements containing any reference to the [warning\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_warning_count) or [error\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_error_count) system variable.

Generally, statements not permitted in SQL prepared statements are also not permitted in stored programs.

Metadata changes to tables or views referred to by prepared statements are detected and cause automatic repreparation of the statement when it is next executed.

Placeholders can be used for the arguments of the LIMIT clause when using prepared statements.

In prepared [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) statements used with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) and [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), placeholder support for OUT and INOUT parameters is available beginning with MySQL 8.0..

SQL syntax for prepared statements cannot be used in nested fashion. That is, a statement passed to [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) cannot itself be a [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), or [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) statement.

SQL syntax for prepared statements is distinct from using prepared statement API calls. For example, you cannot use the [mysql\_stmt\_prepare()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-stmt-prepare.html) C API function to prepare a [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html), or [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) statement.

SQL syntax for prepared statements can be used within stored procedures, but not in stored functions or triggers. However, a cursor cannot be used for a dynamic statement that is prepared and executed with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) and [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html). The statement for a cursor is checked at cursor creation time, so the statement cannot be dynamic.

SQL syntax for prepared statements does not support multi-statements (that is, multiple statements within a single string separated by ; characters).

To write C programs that use the [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) SQL statement to execute stored procedures that contain prepared statements, the CLIENT\_MULTI\_RESULTS flag must be enabled. This is because each [CALL](https://dev.mysql.com/doc/refman/8.0/en/call.html) returns a result to indicate the call status, in addition to any result sets that might be returned by statements executed within the procedure.

CLIENT\_MULTI\_RESULTS can be enabled when you call [mysql\_real\_connect()](https://dev.mysql.com/doc/c-api/8.0/en/mysql-real-connect.html), either explicitly by passing the CLIENT\_MULTI\_RESULTS flag itself, or implicitly by passing CLIENT\_MULTI\_STATEMENTS (which also enables CLIENT\_MULTI\_RESULTS).

## MySQL PREPARE Statement

PREPARE stmt\_name FROM preparable\_stmt

The [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html) statement prepares a SQL statement and assigns it a name, **stmt\_name**, by which to refer to the statement later. The prepared statement is executed with [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) and released with [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html).

Statement names are not case-sensitive. **preparable\_stmt** is either a string literal or a user variable that contains the text of the SQL statement. The text must represent a single statement, not multiple statements. Within the statement, ? characters can be used as parameter markers to indicate where data values are to be bound to the query later when you execute it. The ? characters should not be enclosed within quotation marks, even if you intend to bind them to string values. Parameter markers can be used only where data values should appear, not for SQL keywords, identifiers, and so forth.

If a prepared statement with the given name already exists, it is deallocated implicitly before the new statement is prepared. This means that if the new statement contains an error and cannot be prepared, an error is returned and no statement with the given name exists.

The scope of a prepared statement is the session within which it is created, which as several implications:

A prepared statement created in one session is not available to other sessions.

When a session ends, whether normally or abnormally, its prepared statements no longer exist. If auto-reconnect is enabled, the client is not notified that the connection was lost. For this reason, clients may wish to disable auto-reconnect..

A prepared statement created within a stored program continues to exist after the program finishes executing and can be executed outside the program later.

A statement prepared in stored program context cannot refer to stored procedure or function parameters or local variables because they go out of scope when the program ends and would be unavailable were the statement to be executed later outside the program. As a workaround, refer instead to user-defined variables, which also have session scope;

Beginning with MySQL 8.0.22, a parameter used in a prepared statement has its type determined when the statement is first prepared, and retains this type whenever [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) is invoked for this prepared statement (unless the statement is reprepared, as explained later in this section). Rules for determining a parameter's type are listed here:

A parameter which is an operand of a binary arithmetic operator has the same data type as the other operand.

If both operands of a binary arithmetic operator are parameters, the type of the parameters is decided by the context of the operator.

If a parameter is the operand of a unary arithmetic operator, the parameter's type is decided by the context of the operator.

If an arithmetic operator has no type-determining context, the derived type for any parameters involved is [DOUBLE PRECISION](https://dev.mysql.com/doc/refman/8.0/en/floating-point-types.html). This can happen, for example, when the parameter is a top-level node in a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) list, or when it is part of a comparison operator.

A parameter which is an operand of a character string operator has the same derived type as the aggregated type of the other operands. If all operands of the operator are parameters, the derived type is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html); its collation is determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_collation_connection).

A parameter which is an operand of a temporal operator has type [DATETIME](https://dev.mysql.com/doc/refman/8.0/en/datetime.html) if the operator returns a DATETIME, [TIME](https://dev.mysql.com/doc/refman/8.0/en/time.html) if the operator returns a TIME, and [DATE](https://dev.mysql.com/doc/refman/8.0/en/datetime.html) if the operator returns a DATE.

A parameter which is an operand of a binary comparison operator has the same derived type as the other operand of the comparison.

A parameter that is an operand of a ternary comparison operator such as [BETWEEN](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#operator_between) has the same derived type as the aggregated type of the other operands.

If all operands of a comparison operator are parameters, the derived type for each of them is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), with collation determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_collation_connection).

A parameter that is an output operand of any of [CASE](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#operator_case), [COALESCE](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_coalesce), [IF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_if), [IFNULL](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_ifnull), or [NULLIF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_nullif) has the same derived type as the aggregated type of the operator's other output operands.

If all output operands of any of [CASE](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#operator_case), [COALESCE](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_coalesce), [IF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_if), [IFNULL](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_ifnull), or [NULLIF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_nullif) are parameters, or they are all NULL, the type of the parameter is decided by the context of the operator.

If the parameter is an operand of any of of [CASE](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#operator_case), [COALESCE()](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#function_coalesce), [IF](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_if), or [IFNULL](https://dev.mysql.com/doc/refman/8.0/en/flow-control-functions.html#function_ifnull), and has no type-determining context, the derived type for each of the parameters involved is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), and its collation is determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_collation_connection).

A parameter which is the operand of a [CAST()](https://dev.mysql.com/doc/refman/8.0/en/cast-functions.html#function_cast) has the same type as specified by the CAST().

If a parameter is an immediate member of a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) list that is not part of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, the derived type of the parameter is [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), and its collation is determined by the value of [collation\_connection](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_collation_connection).

If a parameter is an immediate member of a SELECT list that is part of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, the derived type of the parameter is the type of the corresponding column into which the parameter is inserted.

If a parameter is used as source for an assignment in a SET clause of an [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) statement or in the ON DUPLICATE KEY UPDATE clause of an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement, the derived type of the parameter is the type of the corresponding column which is updated by the SET or ON DUPLICATE KEY UPDATE clause.

If a parameter is an argument of a function, the derived type depends on the function's return type.

For some combinations of actual type and derived type, an automatic repreparation of the statement is triggered, to ensure closer compatibility with previous versions of MySQL. Repreparation does not occur if any of the following conditions are true:

NULL is used as the actual parameter value.

A parameter is an operand of a [CAST()](https://dev.mysql.com/doc/refman/8.0/en/cast-functions.html#function_cast). (Instead, a cast to the derived type is attempted, and an exception raised if the cast fails.)

A parameter is a string. (In this case, an implicit CAST(? AS **derived\_type**) is performed.)

The derived type and actual type of the parameter are both [INTEGER](https://dev.mysql.com/doc/refman/8.0/en/integer-types.html) and have the same sign.

The parameter's derived type is [DECIMAL](https://dev.mysql.com/doc/refman/8.0/en/fixed-point-types.html) and its actual type is either DECIMAL or [INTEGER](https://dev.mysql.com/doc/refman/8.0/en/integer-types.html).

The derived type is [DOUBLE](https://dev.mysql.com/doc/refman/8.0/en/floating-point-types.html) and the actual type is any numeric type.

Both the derived type and the actual type are string types.

If the derived type is temporal and the actual type is temporal. Exceptions: The derived type is [TIME](https://dev.mysql.com/doc/refman/8.0/en/time.html) and the actual type is not TIME; the derived type is [DATE](https://dev.mysql.com/doc/refman/8.0/en/datetime.html) and the actual type is not DATE.

The derived type is temporal and the actual type is numeric.

For cases other than those just listed, the statement is reprepared and the actual parameter types are used instead of the derived parameter types.

These rules also apply to a user variable referenced in a prepared statement.

Using a different data type for a given parameter or user variable within a prepared statement for executions of the statement subsequent to the first execution causes the statement to be reprepared. This is less efficient; it may also lead to the paremeter's (or variable's) actual type to vary, and thus for results to be inconsistent, with subsequent executions of the prepared statement. For these reasons, it is advisable to use the same data type for a given parameter when re-executing a prepared statement.

### EXECUTE Statement

EXECUTE stmt\_name [USING @var\_name [, @var\_name] ...]

After preparing a statement with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), you execute it with an [EXECUTE](https://dev.mysql.com/doc/refman/8.0/en/execute.html) statement that refers to the prepared statement name. If the prepared statement contains any parameter markers, you must supply a USING clause that lists user variables containing the values to be bound to the parameters. Parameter values can be supplied only by user variables, and the USING clause must name exactly as many variables as the number of parameter markers in the statement.

You can execute a given prepared statement multiple times, passing different variables to it or setting the variables to different values before each execution.

### DEALLOCATE PREPARE Statement

{DEALLOCATE | DROP} PREPARE stmt\_name

To deallocate a prepared statement produced with [PREPARE](https://dev.mysql.com/doc/refman/8.0/en/prepare.html), use a [DEALLOCATE PREPARE](https://dev.mysql.com/doc/refman/8.0/en/deallocate-prepare.html) statement that refers to the prepared statement name. Attempting to execute a prepared statement after deallocating it results in an error. If too many prepared statements are created and not deallocated by either the DEALLOCATE PREPARE statement or the end of the session, you might encounter the upper limit enforced by the [max\_prepared\_stmt\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_prepared_stmt_count) system variable.

# PL/SQL

Функції та процедури (підпрограми) являють собою оформлені спеціальним чином іменовані блоки PL / SQL, які можуть бути викликані для виконання і яким можуть бути передані параметри. Як правило, процедури і функції реалізують певну закінчену дію над деякими об'єктами БД.

## Типи процедур і функцій

Існують два види процедур і функцій: локальні і збережені. Локальні процедури і функції можуть використовуватися тільки в тих блоках, де вони визначені. Збережені процедури і функції компілюються і зберігаються в БД в скомпільованому вигляді. При необхідності вони можуть бути викликані для виконання анонімними і іменованими блоками PL / SQL, процедурами і функціями обох видів, тригерами, а також з інтерактивного середовища SQL \* PLUS.

Крім цього збережена функція може бути викликана і в операторі SQL.

## Створення збережених процедур і функцій

Для створення збереженої процедури використовується такий синтаксис:

CREATE [OR REPLACE] PROCEDURE ім’я\_процедури

[(параметр1 [, параметр2, ...])] IS

[розділ локальних оголошень]

BEGIN

виконуваний розділ

[EXCEPTION

розділ обробки виключень]

END [ім'я процедури];

створення збереженої функції:

CREATE [OR REPLACE] FUNCTION ім’я\_функції

[(параметр1 [, параметр2, ...]] RETURN тип\_даних IS

[розділ локальних оголошень]

BEGIN

виконуваний розділ

[EXCEPTION

розділ обробки виключень]

END [ім'я функції];

Збережені процедури і функції, що викликаються блоками PL / SQL, процедурами і функціями, тригерами, викликаються завданням імені функції або процедури з вказівкою списку фактичних параметрів. Якщо викликається функція, то вона повинна бути частиною виразу; процедура викликається як окремий оператор.

Для виклику з SQL \* PLUS збереженої процедури:

EXECUTE ім’я\_процедури (список\_фактичних\_параметрів);

Оскільки функція з інтерактивного редактора не може бути безпосередньо викликана, для її виклику необхідно використовувати блок PL / SQL, анонімний або іменований, або оператор SQL.

## Параметри процедур і функцій

Для передачі інформації використовуються параметри. Змінні або вирази, перераховані в списку параметрів в специфікації програми, називаються формальними параметрами, а перераховані в списку параметрів при виклику називаються фактичними аргументами. При виклику фактичні аргументи обчислюються і результуючі значення присвоюються формальним параметрам, причому виконуються необхідні перетворення типів, тому формальні параметри та фактичні аргументи повинні мати сумісні типи.

Список параметрів є перерахування цих параметрів через кому. Кожен формальний параметр може бути описаний таким синтаксисом:

Ім’я\_параметра [вид] тип [{: = │DEFAULT} значення];

Параметр *вид* визначає режим передачі параметра. Є три режими передачі параметрів: IN (за замовчуванням), OUT і IN OUT. Вони використовуються для позначення відповідності вхідних, вихідних і модифікованих параметрів. Бажано не використовувати режими OUT і IN OUT при написанні функцій, щоб уникнути побічних ефектів.

Фактичний аргумент, що указується на місці IN-параметра, повинен бути константою, літералом, проініціалізованою змінної або виразом, і на відміну від OUT- і IN OUT-параметрів IN-параметр може мати значення за замовчуванням. Якщо параметр передається з варіантом IN, то в підпрограмі йому не можна присвоювати значення.

На місці OUT- або IN OUT-параметра може бути вказана тільки змінна. Як і змінні, OUT-параметри ініціалізуються NULL-значенням, і тип OUT-параметра не може бути підтипом, визначеним як NOT NULL. В іншому випадку генерується виключення VALUE\_ERROR.

Якщо при виконанні процедури або функції виникають виняткові ситуації, то управління передається до викликаючого блоку. Коли здійснюється нормальний вихід з підпрограми, то фактичним OUT- і IN OUT-аргументам присвоюються значення, а якщо виникають необроблені виключення, то значення не присвоюються.

Параметр тип визначає припустимий тип даних для параметра. В якості типу параметра можуть використовуватися практично всі основні типи даних мови. Однак якщо використовуються типи CHAR, VARCHAR2 або NUMBER, то не можна вказувати розмірність для цих типів даних, а для типу NUMBER – точність і масштаб..

**Порядок завдання параметрів.**

При виклику програм можна записати список фактичних аргументів, використовуючи або позиційну, іменовану, змішану нотації, або передачу параметрів за замовченням:

позиційна нотація– це передача списку параметрів простим перерахуванням, причому типи, кількість і порядок проходження параметрів повинні відповідати оголошеним раніше;

в іменованої нотації стрілка => використовується як оператор асоціації, який пов'язує формальний параметр зліва від стрілки з фактичним аргументом праворуч від неї. При іменованій нотації параметри можуть вказуватися в будь-якому порядку;

нотації можуть змішуватися (змішана нотація), але в цьому випадку позиційна вказівка параметрів має передувати іменованій;

існує можливість передачі параметрів за замовчуванням. При цьому формальним параметрам повинні бути привласнені значення або оператором присвоювання, або через ключове слово DEFAULT, і вони в списку фактичних параметрів повинні бути записані останніми.

## Програми і залежності. Перекомпіляція програм

Збережені функції і процедури зберігаються в скомпільованому вигляді в БД. При цьому, як правило, їх виконання зачіпає деякі об'єкти БД. Для забезпечення достовірності роботи таких процедур або функцій система постійно відстежує для кожної процедури або функції стан об'єктів, з якими вона пов'язана. Якщо якийсь із пов'язаних з нею об'єктів піддається модифікації за допомогою оператора DDL, то процедура або функція оголошується системою недійсною або недостовірною (INVALID). У цьому випадку процедуру або функцію, оголошену недостовірною, треба обов'язково перекомпілювати.

Для того щоб перекомпілювати збережену процедуру, використовується команда:

ALTER PROCEDURE ім’я\_процедури COMPILE;

Команда ALTER FUNCTION перекомпілює збережену функцію:

ALTER FUNCTION ім’я\_функції COMPILE;

Видалення процедур та функцій з БД здійснюється так:

DROP {PROCEDURE │ FUNCTION} ім’я\_підпрограми;

## Отримання інформації про процедури і функції

Інформацію про процедури та функції можна отримати з подання словника даних USER\_OBJECTS, наприклад, наступною командою:

SELECT \* FROM USER\_OBJECTS;

**Приклади.**

1. Створити збережену процедуру для підвищення вартості вкзаної книги в таблиці BOOKS на 10 %. Параметр: код книги.

Варіант 1.

CREATE OR REPLACE PROCEDURE INCREASE

(CODE\_BOOK BOOKS.CODE\_BOOK%TYPE) AS

Q NUMBER(1) := 0;

BEGIN

SELECT COUNT(\*) INTO Q FROM BOOKS

WHERE BOOKS.CODE\_BOOK = INCREASE.CODE\_BOOK;

IF Q <> 0 THEN

UPDATE BOOKS SET PRICE = PRICE + PRICE\*0.1

WHERE BOOKS.CODE\_BOOK = INCREASE.CODE\_BOOK;

ELSE

RAISE\_APPLICATION\_ERROR(-20105, ‘В таблиці BOOKS

відсутня книга з зазначеним кодом’);

END IF;

END INCREASE;

Виконання процедури INCREASE (варіант 1) реалізується оператором EXECUTE INCREASE (1);

Варіант 2.

CREATE OR REPLACE PROCEDURE INCREASE

(CODE\_BOOK BOOKS.CODE\_BOOK%TYPE) AS

BEGIN

UPDATE BOOKS SET PRICE = PRICE\*1.1

WHERE BOOKS.CODE\_BOOK = INCREASE.CODE\_BOOK;

IF SQL%NOTFOUND THEN

RAISE\_APPLICATION\_ERROR(-20105, ‘В таблиці BOOKS

відсутня книга з зазначеним кодом’);

END IF;

END INCREASE;

Виконання процедури INCREASE (варіант 2) реалізується оператором EXECUTE INCREASE (1);

2. Створити збережену процедуру, що при надходженні книжок продавцю або додає вказану кількість до кількості вже існуючих у продавця таких же книжок, поновлюючи відповідний запис в таблиці BOOKS\_DELIVERY, або додає новий запис до таблиці BOOKS\_DELIVERY, якщо у продавця книг такого найменування немає.

Параметри: код книги, кількість одиниць, прізвище продавця.

Створення процедури:

CREATE OR REPLACE PROCEDURE ADD\_BOOKS

(CODE\_BOOK NUMBER, QUANTITY NUMBER, SALESMAN VARCHAR2) IS

Q NUMBER(5) := 0;

BEGIN

SELECT COUNT(\*) INTO Q FROM BOOKS\_DELIVERY

WHERE BOOKS\_DELIVERY.CODE\_BOOK = ADD\_BOOKS.CODE\_BOOK AND BOOKS\_DELIVERY.SALESMAN = ADD\_BOOKS.SALESMAN;

IF Q <> 0 THEN

UPDATE BOOKS\_DELIVERY SET

QUANTITY = QUANTITY + ADD\_BOOKS.QUANTITY

WHERE BOOKS\_DELIVERY.CODE\_BOOK = ADD\_BOOKS.CODE\_BOOK AND BOOKS\_DELIVERY.SALESMAN = ADD\_BOOKS.SALESMAN;

ELSE

INSERT INTO BOOKS\_DELIVERY VALUES

(CODE\_OP.NEXTVAL,

ADD\_BOOKS.CODE\_BOOK, ADD\_BOOKS.SALESMAN,

ADD\_BOOKS.QUANTITY, SYSDATE);

END IF;

END ADD\_BOOKS;

Виконання процедури ADD\_BOOKS реалізується такими операторами:

EXECUTE ADD\_BOOKS(1, 10, 'Іванов І. І.');

EXECUTE ADD\_BOOKS(5, 10, 'Іванов І. І.');

При першому виклику процедури відбувається додавання 10 книжок до тих, що вже є у вказаного продавця. При другому виклику до таблиці BOOKS\_DELIVERY додається новий запис.

У MySQL функції це програми, що зберігаються на сервері, та можуть приймати параметри, а потім повертати якесь значення.

Створення функції

CREATE FUNCTION function\_name [ (parameter datatype [, parameter datatype]) ]

RETURNS return\_datatype

BEGIN

declaration\_section

executable\_section

END;

function\_name – це назва, що Ви надаєте функції

parameter – один або кілька параметрів, що надходять до функції. Параметри мають бути вхідними (або **IN parameters** (not OUT or INOUT parameters)) може бути посилання на параметр, але параметр не може бути зміненим функцією.

return\_datatype – тип даних, що повертаються

declaration\_section - місце функції де описуються локальні змінні.

executable\_section – місце функції, де пишеться її код.

Спочатку треба зазначити т.з. делімітер, а в кінці не забути написати :

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: WHILE income <= 3000 DO

SET income = income + starting\_value;

END WHILE label1;

RETURN income;

END; //

DELIMITER ;

### Виконання функції

Функція викликається так:

SELECT CalcIncome (1000);

Now that we have a database and a table to work with, we are ready to create a stored function. Let's create a function named **calcProfit**. This function takes two input parameters: the cost and the price of something. It calculates the profit by subtracting the cost from the price, and then returns the value to the calling expression.

To create this stored function, run the following MySQL statements:

DELIMITER $$

CREATE FUNCTION calcProfit(cost FLOAT, price FLOAT) RETURNS DECIMAL(9,2)

BEGIN

DECLARE profit DECIMAL(9,2);

SET profit = price-cost;

RETURN profit;

END$$

DELIMITER ;

The ***DELIMITER*** command at the beginning of these statements prevents MySQL from processing the function definition too soon. The ***DELIMITER*** command at the end of these statements returns processing to normal.

You can now use the stored function in a database query. The following SQL statement demonstrates how to do this:

SELECT \*, calcProfit(prod\_cost,prod\_price) AS profit FROM products;

This SQL statement returns the following results:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| prod\_id | prod\_name | prod\_cost | prod\_price | profit |
| 1 | Basic Widget | 5.95 | 8.35 | 2.40 |
| 2 | Micro Widget | 0.95 | 1.35 | 0.40 |
| 3 | Mega Widget | 99.95 | 140 | 40.05 |

As you can see, the **calcProfit** function automatically calculates the profit (price minus the cost) for each product in the table.

### Вилучення функції з БД

DROP FUNCTION [ IF EXISTS ] function\_name;

function\_name – назва функції, яку бажаєте вилучити

DROP FUNCTION CalcIncome;

## MySQL процедури користувача MySQL

Процедура це програма, що зберігається на сервері та може приймати параметри. Процедура не повертає значення як функція.

CREATE PROCEDURE procedure\_name [ (parameter datatype [, parameter datatype]) ]

BEGIN

declaration\_section

executable\_section

END;

procedure\_name – назва процедури

parameter – необов’язковий параметр (або кілька параметрів)

Параметри бувають таких типів:

**IN** – надається процедурі як посилання і не може бути зміненим.

**OUT** – значення його може бути змінене процедурою.

**IN OUT** – надається як посилання й може бути зміненим.

declaration\_section – місце де оголошуютьсяллокальні змінні

executable\_section – місце процедури де записано її код

Let's look at an example that shows how to create a procedure in MySQL:

DELIMITER //

CREATE procedure CalcIncome ( OUT ending\_value INT )

BEGIN

DECLARE income INT;

SET income = 50;

label1: WHILE income <= 3000 DO

SET income = income \* 2;

END WHILE label1;

SET ending\_value = income;

END; //

DELIMITER ;

Утворену процедуру викликають так::

CALL CalcIncome (@variable\_name);

SELECT @variable\_name;

### Виконання процедури EXEC

The EXEC command is used to execute a stored procedure.

The following SQL executes a stored procedure :

EXECUTE назва\_процедури (список\_фактичних\_параметрів);

### Вилученя процедури з серверу БД

DROP procedure [ IF EXISTS ] procedure\_name;

DROP procedure CalcIncome;

This example would drop the procedure called CalcIncome.

## Елементи мови PL/SQL

### Оператор IF-THEN-ELSE

Використовується у коді , коли є альтернатива вибору

IF condition1 THEN

{...statements to execute when condition1 is TRUE...}

[ ELSEIF condition2 THEN

{...statements to execute when condition1 is FALSE and condition2 is TRUE...} ]

[ ELSE

{...statements to execute when both condition1 and condition2 are FALSE...} ]

END IF;

ELSEIF – необовязково .

ELSE – необовязково

DELIMITER //

CREATE FUNCTION IncomeLevel ( monthly\_value INT )RETURNS varchar(20)

BEGIN

DECLARE income\_level varchar(20);

IF monthly\_value <= 4000 THEN

SET income\_level = 'Low Income';

ELSEIF monthly\_value > 4000 AND monthly\_value <= 7000 THEN

SET income\_level = 'Avg Income';

ELSE

SET income\_level = 'High Income';

END IF;

RETURN income\_level;

END; //

DELIMITER ;

Функція повертає varchar(20), що грутується на параметрі monthly\_value.

### Оператор WHILE

Використовується тоді, коли ви не знаєте кількості повторів циклу(можливо, що жодного разу).

[ label\_name: ] WHILE condition DO

{...statements...}

END WHILE [ label\_name ];

Приклад:

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: WHILE income <= 3000 DO

SET income = income + starting\_value;

END WHILE label1;

RETURN income;

END; //

DELIMITER ;

Цикл завершується коли длосягнуто 3000 виконання:

label1: WHILE income <= 3000 DO

### Оператор LEAVE

Використовується для завершення іменованого блоку MySQL: [LOOP](https://www.techonthenet.com/mysql/loops/gen_loop.php), WHILE, REPEAT операторів.

LEAVE label\_name;

label\_name – назва іменованого блоку

Приклад:

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: LOOP

SET income = income + starting\_value;

IF income < 3000 THEN

ITERATE label1;

END IF;

LEAVE label1;

END LOOP label1;

RETURN income;

END; //

DELIMITER ;

Оператор LEAVE завершує цикл label1: LOOP

Оператор [ITERATE](https://www.techonthenet.com/mysql/loops/iterate.php) w повторює цикл 3000 разів.

### Оператор ITERATE

ITERATE використовується для повтору тіла циклу

ITERATE label\_name;

label\_name назва іменованого блоку

Let's look at an example that shows how to use the ITERATE statement in MySQL:

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: LOOP

SET income = income + starting\_value;

IF income < 2500 THEN

ITERATE label1;

END IF;

LEAVE label1;

END LOOP label1;

RETURN income;

END; //

DELIMITER ;

### Оператор RETURN

RETURN використовують тоді, коли ви бажаєте вийцти з функції та повернути реультат її роботи. Також може використогвуватися для завершення й виходу з циклу.

RETURN result;

Кожна з функції повинна мати мінімум один або кілька RETURN.

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: LOOP

SET income = income + starting\_value;

IF income < 3000 THEN

ITERATE label1;

END IF;

LEAVE label1;

END LOOP label1;

RETURN income;

END; //

DELIMITER ;

For example:

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: LOOP

SET income = income + starting\_value;

IF income < 3000 THEN

ITERATE label1;

END IF;

RETURN income;

END LOOP label1;

RETURN starting\_value;

END; //

DELIMITER ;

### Оператор LOOP

LOOP використовують коли ви не знаєте загальної кількості ітерацій, але знаєте, що мінімум одна буде.

[ label\_name: ] LOOP

{...statements...}

END LOOP [ label\_name ];

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: LOOP

SET income = income + starting\_value;

IF income < 4000 THEN

ITERATE label1;

END IF;

LEAVE label1;

END LOOP label1;

RETURN income;

END; //

DELIMITER ;

### Оператор REPEAT

REPEAT використовують, якщо невідома кількість ітерацій.

[ label\_name: ] REPEAT

{...statements...}

UNTIL condition

END REPEAT [ label\_name ];

label\_name – назва циклу

statements – оператору коду , що реалізують цикл

condition умова припинення циклу.

DELIMITER //

CREATE FUNCTION CalcIncome ( starting\_value INT )

RETURNS INT

BEGIN

DECLARE income INT;

SET income = 0;

label1: REPEAT

SET income = income + starting\_value;

UNTIL income >= 4000

END REPEAT label1;

RETURN income;

END; //

DELIMITER ;

Оператор CASE

CASE функціонально такий же як IF-THEN-ELSE та має 2 синтаксису.

Варіанти сінтаксису

CASE expression

WHEN value\_1 THEN

{...statements to execute when expression equals value\_1...}

[ WHEN value\_2 THEN

{...statements to execute when expression equals value\_2...} ]

[ WHEN value\_n THEN result\_n

{...statements to execute when expression equals value\_n...} ]

[ ELSE

{...statements to execute when no values matched...} ]

END CASE;

або

CASE

WHEN condition\_1 THEN

{...statements to execute when condition\_1 is TRUE...}

[ WHEN condition\_2 THEN

{...statements to execute when condition\_2 is TRUE...} ]

[ WHEN condition\_n THEN

{...statements to execute when condition\_n is TRUE...} ]

[ ELSE

{...statements to execute when all conditions were FALSE...} ]

END CASE;

Expression – величина , що порівнюється зі списком величин ( value\_1, value\_2, ... value\_n).

value\_1, value\_2, ... value\_n – список величин. Якщо величина списку збігається з порівнюваною, то виконується відповідна гілка

condition\_1, condition2, ... condition\_n список умов.

DELIMITER //

CREATE FUNCTION IncomeLevel ( monthly\_value INT )

RETURNS varchar(20)

BEGIN

DECLARE income\_level varchar(20);

CASE monthly\_value

WHEN 4000 THEN

SET income\_level = 'Low Income';

WHEN 5000 THEN

SET income\_level = 'Avg Income';

ELSE

SET income\_level = 'High Income';

END CASE;

RETURN income\_level;

END; //

DELIMITER ;

**CASE з іншим синтаксисом.**

DELIMITER //

CREATE FUNCTION IncomeLevel ( monthly\_value INT )

RETURNS varchar(20)

BEGIN

DECLARE income\_level varchar(20);

CASE

WHEN monthly\_value <= 4000 THEN

SET income\_level = 'Low Income';

WHEN monthly\_value > 4000 AND monthly\_value <= 7000 THEN

SET income\_level = 'Avg Income';

ELSE

SET income\_level = 'High Income';

END CASE;

RETURN income\_level;

END; //

DELIMITER ;

## Курсор MySQL: Декларування курсору

The syntax to declare a cursor in MySQL is:

DECLARE cursor\_name CURSOR FOR select\_statement;

cursor\_name – назва курсору

select\_statement – операторр запиту, асоційований з курсором.

DECLARE c1 CURSOR FOR

SELECT site\_id FROM sites WHERE site\_name = name\_in;

В результаті курсор устанавлює значення site\_id коли site\_name = name\_in

DELIMITER //

CREATE FUNCTION FindSiteID ( name\_in VARCHAR(50) )

RETURNS INT

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE siteID INT DEFAULT 0;

DECLARE c1 CURSOR FOR

SELECT site\_id

FROM sites

WHERE site\_name = name\_in;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN c1;

FETCH c1 INTO siteID;

CLOSE c1;

RETURN siteID;

END; //

DELIMITER ;

Виклик функції

SELECT FindSiteID ('TechOnTheNet.com');

### Відкриття курсору OPEN

Відразу після декларування курсору, ви маєте відкрити курсор оператором OPEN.

OPEN cursor\_name;

DELIMITER //

CREATE FUNCTION FindSiteID ( name\_in VARCHAR(50) )

RETURNS INT

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE siteID INT DEFAULT 0;

DECLARE c1 CURSOR FOR

SELECT site\_id

FROM sites

WHERE site\_name = name\_in;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN c1;

FETCH c1 INTO siteID;

CLOSE c1;

RETURN siteID;

END; //

DELIMITER ;

### Оператор FETCH – отримання даних

Цей оператор читає рядки з БД, бо головна мета курсору – полегшення отримання даних.

FETCH [ NEXT [ FROM ] ] cursor\_name INTO variable\_list;

cursor\_name – назва курсору

variable\_list – список змінних, де зберігається інформація, прочитана курсором

DELIMITER //

CREATE FUNCTION FindSiteID ( name\_in VARCHAR(50) )

RETURNS INT

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE siteID INT DEFAULT 0;

DECLARE c1 CURSOR FOR

SELECT site\_id

FROM sites

WHERE site\_name = name\_in;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN c1;

FETCH c1 INTO siteID;

CLOSE c1;

RETURN siteID;

END; //

DELIMITER ;

### Закриття курсору CLOSE

**CLOSE має виконуватися обов’язково, та закривати курсор в кінці процедури або функції.**

CLOSE cursor\_name;

Below is a function that demonstrates how to use the CLOSE statement:

DELIMITER //

CREATE FUNCTION FindSiteID ( name\_in VARCHAR(50) )

RETURNS INT

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE siteID INT DEFAULT 0;

DECLARE c1 CURSOR FOR

SELECT site\_id

FROM sites

WHERE site\_name = name\_in;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN c1;

FETCH c1 INTO siteID;

CLOSE c1;

RETURN siteID;

END; //

DELIMITER ;

### Маніпулятор курсора Handler for Cursor's NOT FOUND condition

Якщо дані не знайдені курсором, то формується промилка NO DATA. Існує можливість налагодити маніпулятор курсора так, що умова NOT FOUND не завершить аварійно процедуру на сервері.

DECLARE CONTINUE HANDLER FOR NOT FOUND [ set\_condition ];

Спочатку треба декларувати програмну змінну.

DECLARE done INT DEFAULT FALSE;

Далі декларувати курсор.

DECLARE c1 CURSOR FOR

SELECT site\_id

FROM sites

WHERE site\_name = name\_in;

Далі декларувати маніпулятор.

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

.

DELIMITER //

CREATE FUNCTION FindSiteID ( name\_in VARCHAR(50) )

RETURNS INT

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE siteID INT DEFAULT 0;

DECLARE c1 CURSOR FOR

SELECT site\_id

FROM sites

WHERE site\_name = name\_in;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN c1;

FETCH c1 INTO siteID;

CLOSE c1;

RETURN siteID;

END; //

DELIMITER ;

## Тригери MySQL:

Тригер бази даних – це оформлений спеціальним чином іменований блок PL / SQL, що зберігається в БД. Кожен тригер пов’язаний з визначеною таблицею і автоматично запускається при виконанні одного з DML-операторів (INSERT, DELETE, UPDATE) або їх сукупності над цією таблицею.

Тригери можуть бути використані:

* для реалізації складних обмежень цілісності даних, які не можуть бути здійснені стандартним чином при створенні таблиці;
* запобігання невірних транзакцій;
* виконання процедур комплексної перевірки прав доступу та секретності даних;
* генерації деяких виразів на основі значень, наявних у стовпцях таблиць;
* при реалізації складних бізнес-правил для обробки даних (можливість відстежити оновлення даних у зв’язаних таблицях при зміні в одній з них).

### Створення і включення тригерів

Для створення і автоматичного включення тригера застосовується такий загальний синтаксис:

CREATE [OR REPLACE] TRIGGER *ім’я\_тригера*

{BEFORE | AFTER}

{INSERT | DELETE | UPDATE [OF *список\_стовпців*]}

ON *ім’я\_таблиці* [FOR EACH ROW] [WHEN *умова*]

<PL / SQL\_блок>

При наявності ключових слів OR REPLACE тригер створюється заново, якщо він вже існує.

Конструкція BEFORE | AFTER вказує на момент запуску тригера. Варіант BEFORE означає, що тригер буде запускатися перед виконання активізуючого DML-оператора; варіант AFTER означає, що тригер буде запускатися після виконання активізуючого DML-оператора.

Конструкція INSERT | DELETE | UPDATE [OF *список\_стовпців*] вказує тип активізуючого тригер DML-оператора. Дозволяється, використовуючи логічну операцію OR, задати сукупність активізуючи операторів, наприклад INSERT OR DELETE. Якщо при використанні варіанта UPDATE вказаний список стовпців, то тригер буде запускатися при модифікації одного із зазначених стовпців; якщо список стовпців відсутній, то тригер буде запускатися при зміні будь-якого із стовпців пов’язаної з тригером таблиці.

Конструкція FOR EACH ROW вказує на характер впливу тригера: строковий або операторний. Якщо конструкція FOR EACH ROW присутня, то тригер є строковим; при відсутності її тригер є операторних. Операторний тригер запускається один раз до або після виконання активізуючого тригер DML-оператора незалежно від того, скільки рядків у пов’язаної з тригером таблиці піддалось модифікації. Строковий тригер запускається один раз для кожного з рядків, які піддаються модифікації DML-оператором, активізуючого тригер.

За допомогою ключового слова WHEN можна задати додаткове обмеження на рядки пов’язаної з тригером таблиці, при модифікації яких може бути запущений тригер.

Конструкція PL / SQL\_блок представляє блок PL / SQL, який запускається при активізації тригера.

### Класифікація тригерів.

В основному розрізняють дванадцять типів тригерів. Тип тригера визначається поєднанням наступних трьох параметрів:

* характером впливу тригера на рядки пов’язаної з ним таблиці (строковий або операторний);
* моментом запуску тригера: до (BEFORE) або після (AFTER) виконання активізуючого тригер DML-оператора;
* типом активізуючого тригер DML-оператора (INSERT, DELETE, UPDATE).

### Порядок активації тригерів.

Якщо у таблиці є декілька типів тригерів, то вони активуються за такою схемою:

1. виконується операторний тригер BEFORE (якщо їх декілька, то нічого про порядок їх виконання сказати не можна);
2. виконується строковий тригер BEFORE;
3. виконується активізуючий тригер DML-оператор з наступною перевіркою всіх обмежень цілісності даних;
4. виконується строковий тригер AFTER з наступною перевіркою всіх обмежень цілісності даних;
5. виконується операторний тригер AFTER.

### Тригерні предикати.

Якщо в тригері вказується сукупність активізуючих тригер DML-операторів (наприклад, INSERT OR DELETE), то для розпізнавання того, який конкретно з DML-операторів виконується над пов’язаною з тригером таблицею, використовуються тригерні предикати: INSERTING, DELETING, UPDATING. Вони є логічними функціями, які повертають TRUE, якщо тип активізуючого оператора збігається з типом предиката, і FALSE – в протилежному випадку. Для завдання одних і тих же дій у разі виконання різних DML-операторів в умовному операторі тригерні предикати об’єднуються за допомогою логічних операцій.

### Псевдозаписи OLD І NEW.

Для строкових тригерів існують спеціальні конструкції, які дозволяють при виконанні DML-операторів над рядком таблиці звертатися як до старих значенням, які знаходились в ній до модифікації, так і до нових, які з’являться в рядку після її модифікації. Ці конструкції називаються псевдозаписами і позначаються **OLD І NEW.** Структура цих псевдозаписів ідентична структурі рядка модифікуємої таблиці, але оперувати можна тільки окремими полями псевдозапису. Звернення до полів псевдозапису відбувається за такою схемою: перед old або new ставиться символ двокрапка (:), далі через точку вказується назва поля. Значення, які беруть поля псевдозапису при виконанні активізуючих DML-операторів, визначаються таким чином.

1. Оператор INSERT – псевдозапис: new еквівалентний рядку, що вставляється, а псевдозапис: old у всіх полях має значення NULL.
2. Оператор DELETE – псевдозапис: old еквівалентний рядку, що видаляється, а псевдозапис: new у всіх полях має значення NULL.
3. Оператор UPDATE – псевдозапис: new еквівалентний рядку, отриманому в результаті модифікації, а псевдозапис: old у всіх полях має початкове значення рядка.

### Вмикання, вимивання та видалення тригерів.

Зберігаємий в БД тригер можна тимчасово вімкнути, не видаляючи його з БД. Для цього використується команда:

ALTER TRIGGER *ім’я\_тригера* DISABLE;

Ввімкнути тригер через деякий проміжок часу можна так

ALTER TRIGGER *ім’я\_тригера* ENABLE;

Заборонити або дозволити запуск всіх тригерів, пов’язаних з деякою таблицею, можна так

ALTER TABLE *ім’я\_таблиці* {DISABLE | ENABLE} ALL TRIGGERS;

де варіант DISABLE використовується для вимкнення, а варіант ENABLE – для ввімкнення всіх тригерів даної таблиці.

Видалення тригера з БД здійснюється командj.:

DROP TRIGGER *ім’я\_тригера*;

Інформацію про тригери, які зберігаються в БД, можна отримати з подання словника даних USER\_TRIGGERS, наприклад, такою командою:

SELECT \* FROM USER\_TRIGGERS;

### Приклади створення тригерів

1. Створити тригер, який перед вставкою чергового рядка в таблицю BOOKS\_DELIVERY перевіряє наявність зазначеного коду книги в таблиці BOOKS. При відсутності зазначеного коду книги в таблиці BOOKS має генеруватися виняток з видачею відповідного повідомлення.

Додавання нових рядків таблицю BOOKS\_DELIVERY виконується оператором INSERT. Оскільки тригер повинен запускатися перед виконанням кожного оператора INSERT, отже, він повинен бути строковим BEFORE-тригером. Для збереження цілісності даних необхідно перевірити, чи є коди книг в таблиці BOOKS, які мають бути внесені. Для цього за допомогою однорядкового оператора SELECT виконується вибірка інформації з таблиці BOOKS, де в умові вибірки використовується поле CODE\_BOOK псевдозапису: new. Якщо кількість рядків з даними кодом книги в таблиці BOOKS виявиться рівним нулю, буде згенеровано виняток і видано відповідне повідомлення.

Створення тригера TR1:

CREATE OR REPLACE TRIGGER TR1

BEFORE INSERT ON BOOKS\_DELIVERY

FOR EACH ROW

DECLARE

QUANTITY NUMBER (4);

BEGIN

SELECT COUNT (\*) INTO QUANTITY FROM BOOKS

WHERE CODE\_BOOK =: NEW.CODE\_BOOK;

IF QUANTITY = 0 THEN RAISE\_APPLICATION\_ERROR

(-20212, ‘У таблиці BOOKS немає інформації про дану книгу’);

END IF;

END TR1;

При створенні тригера за допомогою SQL\*PLUS, необхідно вказувати в рядку, наступному за останнім оператором, косу риску (/), щоб оператор CREATE ... TRIGGER виконався.

Дія тригера TR1 може бути перевірено виконанням наступного оператора, що здійснює вставку рядка в таблицю BOOKS\_DELIVERY і тим самим викликає активізацію тригера TR2:

INSERT INTO BOOKS\_DELIVERY VALUES (21, 15, ‘Іванов І. І.’, 15, ‘20 -01-06 ‘);

Оскільки код книги 15 відсутній в таблиці BOOKS, то буде згенеровано виняток і видано відповідне повідомлення.

2. Створити тригер, який забороняє вносити в таблицю BOOKS рядки зі значенням поля PRICE більше, ніж 500 грн., а також збільшувати ціни книг, інформація про яких зберігається в таблиці BOOKS, більш ніж на 20%. При порушенні даної вимоги має генеруватися виняток з видачею відповідного повідомлення.

Так як внесення нових рядків таблицю BOOKS здійснюється в результаті виконання оператора INSERT, а значення поля PRICE в таблиці BOOKS, що містить ціну книги, може бути змінено в результаті виконання оператора UPDATE, то в тригері вказується сукупність активізуючих DML-операторів. Оскільки тригер повинен запускатися перед виконанням кожного із зазначених DML-операторів, він є строковим BEFORE-тригером. Так як дії, виконувані тригером, різні для кожного з активізуючих DML-операторів, що модифікують таблицю BOOKS, то для розпізнавання типу DML-оператора використовуються відповідні тригерні предикати INSERTING і UPDAITING. Внаслідок того що при вставці нових рядків перевірці має бути піддано нове значення поля PRICE, а при модифікації значення поля PRICE нове значення повинно порівнюватися зі старим значенням, необхідно використовувати псевдозаписи: new і old.

Створення тригера TR2:

CREATE OR REPLACE TRIGGER TR2

BEFORE INSERT OR UPDATE OF PRICE ON BOOKS

FOR EACH ROW

BEGIN

IF INSERTING THEN

IF: NEW.PRICE> 5000 THEN

RAISE\_APPLICATION\_ERROR

(-20102, ‘У таблицю BOOKS не можна вносити записи з ціною книги > 500’);

END IF;

END IF;

IF UPDATING THEN

IF: NEW.PRICE>: OLD.PRICE \* 1.2 THEN

RAISE\_APPLICATION\_ERROR

(-20103, ‘У таблиці BOOKS не можна змінювати ціну книги більш ніж на 20% ‘);

END IF;

END IF;

END TR2;

Дія тригера TR2 може бути перевірена виконанням таких операторів, які, здійснюючи вставку рядків таблицю BOOKS і оновлення рядків у таблиці BOOKS, тим самим викликають його активізацію.

Оператор вставки рядків у таблицю BOOKS, що активізує тригер TR2:

INSERT INTO BOOKS VALUES (21, ‘Дюна’, ‘Герберт Ф.’, 5268, ‘Аст’, ‘Фантастика’);

Оператор відновлення рядків в таблиці BOOKS, що активізує тригер TR2:

UPDATE BOOKS SET PRICE = 600;

Оскільки ці оператори порушують вимоги, пропоновані до значенням і модифікації ціни книг, то у всіх випадках буде згенеровано виняток і видано відповідне повідомлення.

### BEFORE INSERT Trigger

BEFORE INSERT тригер виконується завжди перед додаванням записів до таблиці.

Синтаксис створення тригеру BEFORE INSERT у MySQL такий:

CREATE TRIGGER trigger\_name

BEFORE INSERT

ON table\_name FOR EACH ROW

BEGIN

-- variable declarations

-- trigger code

END;

trigger\_name – назва тригеру

BEFORE INSERT – індикація типу тригеру

table\_name – назва таблиці

ОБМЕЖЕННЯ

* Неможливо створити BEFORE trigger для подання (view).
* Ви можете оновити NEW величини.
* Ви не можете оновити OLD величини.

Let's look at an example of how to create an BEFORE INSERT trigger using the CREATE TRIGGER statement in MySQL.

Для таблиці:

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

created\_date DATE,

created\_by VARCHAR(30),

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

Створимо BEFORE INSERT тригер:

DELIMITER //

CREATE TRIGGER contacts\_before\_insert

BEFORE INSERT

ON contacts FOR EACH ROW

BEGIN

DECLARE vUser varchar(50);

-- Find username of person performing INSERT into table

SELECT USER() INTO vUser;

-- Update create\_date field to current system date

SET NEW.created\_date = SYSDATE();

-- Update created\_by field to the username of the person performing the INSERT

SET NEW.created\_by = vUser;

END; //

DELIMITER ;

### MySQL тригер AFTER INSERT

AFTER INSERT Trigger означає, що цей тригер виконується після операції INSERT.

CREATE TRIGGER trigger\_name

AFTER INSERT

ON table\_name FOR EACH ROW

BEGIN

-- variable declarations

-- trigger code

END;

trigger\_name – назва тригера

AFTER INSERT – тип тригера

table\_name – назва таблиці

Обмеження

* Ви не можете створити AFTER тригер для подання (view).
* Ви не можете оновити NEW значення.
* Ви не можете оновити OLD значення.

Якщо є таблиця:

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL, first\_name VARCHAR(25),

birthday DATE, CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

То ви можете створити тригер AFTER INSERT :

DELIMITER //

CREATE TRIGGER contacts\_after\_insert

AFTER INSERT

ON contacts FOR EACH ROW

BEGIN

DECLARE vUser varchar(50);

-- Find username of person performing the INSERT into table

SELECT USER() INTO vUser;

-- Insert record into audit table

INSERT INTO contacts\_audit ( contact\_id,created\_date,created\_by)

VALUES ( NEW.contact\_id, SYSDATE(), vUser );

END; //

DELIMITER ;

### Тригер MySQL: BEFORE UPDATE

BEFORE UPDATE означає, що тригер виконується перед виконанням операції UPDATE у певній таблиці.

CREATE TRIGGER trigger\_name

BEFORE UPDATE

ON table\_name FOR EACH ROW

BEGIN

-- variable declarations

-- trigger code

END;

trigger\_name – назва тригера

BEFORE UPDATE – тип тригера - виконується перед оновленням даних у таблиці

table\_name – назва таблиці

Обмеження

Заборонено створювати тригер BEFORE для подання (view).

Заборонено оновлювати величини NEW.

Заборонено оновлювати величини OLD.

Є таблиця:

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

created\_date DATE,

created\_by VARCHAR(30),

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

Створимо для неї BEFORE UPDATE тригер:

DELIMITER //

CREATE TRIGGER contacts\_before\_update

BEFORE UPDATE

ON contacts FOR EACH ROW

BEGIN

DECLARE vUser varchar(50);

-- Find username of person performing the INSERT into table

SELECT USER() INTO vUser;

-- Insert record into audit table

INSERT INTO contacts\_audit

( contact\_id, updated\_date, updated\_by)

VALUES ( NEW.contact\_id, SYSDATE(), vUser );

END; //

DELIMITER ;

### MySQL тригер: AFTER UPDATE

AFTER UPDATE тригер виконується після оновленння таблиці

CREATE TRIGGER trigger\_name

AFTER UPDATE

ON table\_name FOR EACH ROW

BEGIN

-- variable declarations

-- trigger code

END;

trigger\_name – назва тригера

AFTER UPDATE – оголошує, що тригер буде виконуватися після оновлення

table\_name – назва таблиці

Обмеження

Заборонено створювати AFTER тригер для подання (view).

Заборонено оновлювати NEW значення.

Заборонено оновлювати OLD значення.

Для таблиці:

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

created\_date DATE,

created\_by VARCHAR(30),

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

Визначимо тригер AFTER UPDATE:

DELIMITER //

CREATE TRIGGER contacts\_after\_update

AFTER UPDATE

ON contacts FOR EACH ROW

BEGIN

DECLARE vUser varchar(50);

-- Find username of person performing the INSERT into table

SELECT USER() INTO vUser;

-- Insert record into audit table

INSERT INTO contacts\_audit

( contact\_id, updated\_date, updated\_by)

VALUES

( NEW.contact\_id, SYSDATE(), vUser );

END; //

DELIMITER ;

### MySQL тригер : BEFORE DELETE

BEFORE DELETE тригер виконується перед операцією DELETE над таблицею.

CREATE TRIGGER trigger\_name

BEFORE DELETE

ON table\_name FOR EACH ROW

BEGIN

-- variable declarations

-- trigger code

END;

trigger\_name - назва тригера

BEFORE DELETE – тип тригера

table\_name – назва таблиці

Обмеження

Заборонено створювати тригер BEFORE для подання (view).

Можливо оновити NEW значення.

Заборонено оновлювати OLD значення.

Для таблиці

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

created\_date DATE,

created\_by VARCHAR(30),

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

Створимо тригер BEFORE DELETE:

DELIMITER //

CREATE TRIGGER contacts\_before\_delete

BEFORE DELETE

ON contacts FOR EACH ROW

BEGIN

DECLARE vUser varchar(50);

-- Find username of person performing the DELETE into table

SELECT USER() INTO vUser;

-- Insert record into audit table

INSERT INTO contacts\_audit ( contact\_id, deleted\_date, deleted\_by)

VALUES ( OLD.contact\_id, SYSDATE(), vUser );

END; //

DELIMITER ;

### MySQL тригер : AFTER DELETE

AFTER DELETE тригер виконується після операції DELETE.

CREATE TRIGGER trigger\_name

AFTER DELETE

ON table\_name FOR EACH ROW

BEGIN

-- variable declarations

-- trigger code

END;

trigger\_name – назва тригера

AFTER DELETE – тип тригера

table\_name – назва таблиці

Обмеження

Заборонено створювати AFTER тригер для подання (view).

Заборонено оновлювати NEW величини.

Заборонено оновлювати OLD величини.

Для таболиці:

CREATE TABLE contacts

( contact\_id INT(11) NOT NULL AUTO\_INCREMENT,

last\_name VARCHAR(30) NOT NULL,

first\_name VARCHAR(25),

birthday DATE,

created\_date DATE,

created\_by VARCHAR(30),

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

Створимио AFTER DELETE тригер:

DELIMITER //

CREATE TRIGGER contacts\_after\_delete

AFTER DELETE

ON contacts FOR EACH ROW

BEGIN

DECLARE vUser varchar(50);

-- Find username of person performing the DELETE into table

SELECT USER() INTO vUser;

-- Insert record into audit table

INSERT INTO contacts\_audit

( contact\_id,

deleted\_date,

deleted\_by)

VALUES

( OLD.contact\_id,

SYSDATE(),

vUser );

END; //

DELIMITER ;

### Вилучення тригера з сервера: DROP TRIGGER

DROP TRIGGER trigger\_name;

### Висновки

Тригери схожі на процедури і функції тим, що також є іменованими блоками PL / SQL і мають розділ оголошень, виконуваний розділ і розділ обробки виняткових ситуацій. Подібно модулям, тригери зберігаються як автономні об’єкти в базі даних і не можуть зберігатися локально в блоці або модулі. Процедура викликається явним чином з іншого блоку, при виклику їй можуть передаватися різні аргументи. Тригер ж виконується неявно всякий раз, коли відбувається активізуюча його подія, і тригер не має аргументів. Процес виконання тригера називається його активізацією (firing). Подією, що запускає тригер, є операція DML (INSERT, UPDATE або DELETE), виконувана над таблицею або поданням бази даних. У OracleSi ці функції розширені: тригер може спрацьовувати на системну подія, наприклад на запуск або зупинку бази даних, а також на певні види операцій DDL.

Тригери можна використовувати:

для реалізації складних обмежень цілісності даних, які неможливо реалізувати через декларативні обмеження, що встановлюються при створенні таблиці.

для контролю за інформацією, що зберігається в таблиці, за допомогою реєстрації внесених змін і користувачів, які виконують ці зміни.

для автоматичного оповіщення інших програм про те, що необхідно робити в разі зміни інформації, що міститься в таблиці.

для публікації інформації про різні події в середовищі "публікація– підписка".

[TCL (MySQL Transaction Control Language) 1](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747159)

[Introduction on Transaction 1](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747160)

[Understand the concept of a transaction 2](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747161)

[MySQL and the ACID Model 2](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747162)

[MySQL Transaction 4](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747163)

[SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT 7](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747164)

[LOCK and UNLOCK Tables 7](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747165)

[SET TRANSACTION Syntax 8](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747166)

[Transactional and Locking Statements 8](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747167)

[START TRANSACTION, COMMIT, and ROLLBACK Statements 9](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747168)

[Statements That Cannot Be Rolled Back 11](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747169)

[Statements That Cause an Implicit Commit 11](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747170)

[SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements 12](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747171)

[LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE Statements 13](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747172)

[LOCK TABLES and UNLOCK TABLES Statements 13](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747173)

[Table Lock Acquisition 14](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747174)

[Table Lock Release 15](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747175)

[Interaction of Table Locking and Transactions 16](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747176)

[LOCK TABLES and Triggers 16](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747177)

[Table-Locking Restrictions and Conditions 17](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747178)

[SET TRANSACTION Statement 18](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747179)

[Transaction Isolation Levels 18](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747180)

[Transaction Access Mode 18](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747181)

[Transaction Characteristic Scope 19](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747182)

[XA Transactions 20](file:///D:\2020_2021\БД\DBMS%20_Transaction.docx#_Toc57747183)

# TCL (MySQL Transaction Control Language)

## Introduction on Transaction

A transaction is a logical unit of work that contains one or more SQL statements. Transactions are atomic units of work that can be committed or rolled back. When a transaction makes multiple changes to the database, either all the changes succeed when the transaction is committed, or all the changes are undone when the transaction is rolled back.

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly with a COMMIT or ROLLBACK statement or implicitly when a DDL (Data Definition Language (DDL) is used to manage table and index structure and CREATE, ALTER, RENAME, DROP and TRUNCATE statements are to name a few data definition elements) statement is issued.

### ****Understand the concept of a transaction****

To understand the concept of a transaction, consider a banking database. Suppose a bank customer transfers money from his savings account (SB a/c) to his current account (CA a/c), the statement will be divided into four blocks :

* Debit SB a/c.
* Credit CA a/c.
* Record in Transaction Journal
* End Transaction

The SQL statement to debit SB a/c is as follows:

UPDATE sb\_accounts SET balance = balance – 1000 WHERE account\_no = 932656 ;

The SQL statement to credit OD a/c is as follows:

UPDATE ca\_accounts SET balance = balance + 1000 WHERE account\_no = 933456 ;

The SQL statement for record in transaction journal is as follows:

INSERT INTO journal VALUES  
(100896, 'Tansaction on Benjamin Hampshair a/c', '26-AUG-08' 932656, 933456, 1000);

The SQL statement for End Transaction is as follows :

COMMIT WORK;

## ****MySQL and the ACID Model****

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. In MySQL, InnoDB storage engine supports ACID-compliant features. The following sections discuss how MySQL features, in particular, the InnoDB storage engine, interact with the categories of the ACID model:

**Atomicity:**The atomicity aspect of the ACID model mainly involves InnoDB transactions. Related MySQL features include :

* Autocommit setting.
* COMMIT statement.
* ROLLBACK statement.
* Operational data from the INFORMATION\_SCHEMA tables.

**Consistency:**The consistency aspect of the ACID model mainly involves internal InnoDB processing to protect data from crashes. Related MySQL features include :

* InnoDB doublewrite buffer.
* InnoDB crash recovery.

**Isolation:**The isolation aspect of the ACID model mainly involves InnoDB transactions, in particular, the isolation level that applies to each transaction. Related MySQL features include :

* Autocommit setting.
* SET ISOLATION LEVEL statement.
* The low-level details of InnoDB locking. During performance tuning, you see these details through INFORMATION\_SCHEMA tables.

**Durability:**The durability aspect of the ACID model involves MySQL software features interacting with your particular hardware configuration. Because of the many possibilities depending on the capabilities of your CPU, network, and storage devices, this aspect is the most complicated to provide concrete guidelines for. Related MySQL features include:

* InnoDB doublewrite buffer turned on and off by the innodb\_doublewrite configuration option.
* Configuration option innodb\_flush\_log\_at\_trx\_commit.
* Configuration option sync\_binlog.
* Configuration option innodb\_file\_per\_table.
* Write buffer in a storage device, such as a disk drive, SSD, or RAID array.
* Battery-backed cache in a storage device.
* The operating system used to run MySQL, in particular, its support for the fsync() system call.
* Uninterruptible power supply (UPS) protecting the electrical power to all computer servers and storage devices that run MySQL servers and store MySQL data.
* Your backup strategy, such as frequency and types of backups, and backup retention periods.
* For distributed or hosted data applications, the particular characteristics of the data centers where the hardware for the MySQL servers is located, and network connections between the data centers.

## ****MySQL Transaction****

MySQL (here we maintain version 5.6) supports local transactions (within a given client session) through statements such as SET autocommit, START TRANSACTION, COMMIT, and ROLLBACK. Here is the syntax of START TRANSACTION, COMMIT, and ROLLBACK:

START TRANSACTION

transaction\_characteristic [, transaction\_characteristic] ...]

transaction\_characteristic:

WITH CONSISTENT SNAPSHOT

| READ WRITE

| READ ONLY

BEGIN [WORK]

COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET autocommit = {0 | 1}

These statements provide control over use of transactions :

* The START TRANSACTION or BEGIN statement begins a new transaction.
* COMMIT commits the current transaction, making its changes permanent.
* ROLLBACK rolls back the current transaction, canceling its changes.
* The SETautocommit statement disables or enables the default autocommit mode for the current session.

By default, MySQL runs with autocommit mode enabled. This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk to make it permanent. The change cannot be rolled back.

Currently (by default), MySQL runs with autocommit mode enabled.

mysql> select \* from student\_mast;

+------------+------------------+----------+

| STUDENT\_ID | NAME | ST\_CLASS |

+------------+------------------+----------+

| 2 | Neena Kochhar | 9 |

| 3 | Lex De Haan | 9 |

| 4 | Alexander Hunold | 11 |

+------------+------------------+----------+

3 rows in set (0.09 sec)

Let execute an update command:

mysql>mysql> UPDATE STUDENT\_MAST SET ST\_CLASS=8 WHERE STUDENT\_ID=2;

Query OK, 1 row affected (0.08 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql>mysql> select \* from student\_mast;

+------------+------------------+----------+

| STUDENT\_ID | NAME | ST\_CLASS |

+------------+------------------+----------+

| 2 | Neena Kochhar | 8 |

| 3 | Lex De Haan | 9 |

| 4 | Alexander Hunold | 11 |

+------------+------------------+----------+

3 rows in set (0.00 sec)

Now execute the ROLLBACK command to return in the previous stage :

mysql>mysql> ROLLBACK;

Query OK, 0 rows affected (0.03 sec)

mysql>mysql> select \* from student\_mast;

+------------+------------------+----------+

| STUDENT\_ID | NAME | ST\_CLASS |

+------------+------------------+----------+

| 2 | Neena Kochhar | 8 |

| 3 | Lex De Haan | 9 |

| 4 | Alexander Hunold | 11 |

+------------+------------------+----------+

3 rows in set (0.00 sec)

There is no roll back as MySQL runs with autocommit mode enabled.

To disable autocommit mode, use the START TRANSACTION statement. See the following example :

mysql>mysql> START TRANSACTION;

Query OK, 0 rows affected (0.00 sec)

mysql> UPDATE STUDENT\_MAST SET ST\_CLASS=10 WHERE STUDENT\_ID=2;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql> select \* from student\_mast;

+------------+------------------+----------+

| STUDENT\_ID | NAME | ST\_CLASS |

+------------+------------------+----------+

| 2 | Neena Kochhar | 10 |

| 3 | Lex De Haan | 9 |

| 4 | Alexander Hunold | 11 |

+------------+------------------+----------+

3 rows in set (0.00 sec)

mysql> ROLLBACK;

Query OK, 0 rows affected (0.07 sec)

mysql> select \* from student\_mast;

+------------+------------------+----------+

| STUDENT\_ID | NAME | ST\_CLASS |

+------------+------------------+----------+

| 2 | Neena Kochhar | 8 |

| 3 | Lex De Haan | 9 |

| 4 | Alexander Hunold | 11 |

+------------+------------------+----------+

3 rows in set (0.00 sec)

**MySQL statements that cannot be Rolled Back and statements that cause an implicit Commit**

In MySQL, some statements cannot be rolled back. DDL statements such as CREATE or DROP databases, CREATE, ALTER or DROP tables or stored routines. You should design a transaction without these statements.

The statements listed in this section (and any synonyms for them) implicitly end any transaction active in the current session, as if you had done a COMMIT before executing the statement.

* Data definition language (DDL) statements that define or modify database objects. ALTER DATABASE ... UPGRADE DATA DIRECTORY NAME, ALTER EVENT, ALTER PROCEDURE, ALTER SERVER, ALTER TABLE, ALTER VIEW, CREATE DATABASE, CREATE EVENT, CREATE INDEX, CREATE PROCEDURE, CREATE SERVER, CREATE TABLE, CREATE TRIGGER, CREATE VIEW, DROP DATABASE, DROP EVENT, DROP INDEX, DROP PROCEDURE, DROP SERVER, DROP TABLE, DROP TRIGGER, DROP VIEW, RENAME TABLE, TRUNCATE TABLE.
* ALTER FUNCTION, CREATE FUNCTION, and DROP FUNCTION also cause an implicit commit when used with stored functions, but not with UDFs. (ALTER FUNCTION can only be used with stored functions.)
* ALTER TABLE, CREATE TABLE, and DROP TABLE do not commit a transaction if the TEMPORARY keyword is used.
* Statements that implicitly use or modify tables in the MySQL database. CREATE USER, DROP USER, GRANT, RENAME USER, REVOKE, SET PASSWORD.
* Transaction-control and locking statements. BEGIN, LOCK TABLES, SET autocommit = 1 (if the value is not already 1), START TRANSACTION, UNLOCK TABLES.
* Data loading statements. LOAD DATA INFILE. LOAD DATA INFILE causes an implicit commit only for tables using the NDB storage engine.
* Administrative statements. ANALYZE TABLE, CACHE INDEX, CHECK TABLE, LOAD INDEX INTO CACHE, OPTIMIZE TABLE, REPAIR TABLE.
* Replication control statements. Beginning with MySQL 5.6.7: START SLAVE, STOP SLAVE, RESET SLAVE, CHANGE MASTER TO.

## ****SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT****

InnoDB supports the SQL statements SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT and the optional WORK keyword for ROLLBACK.

The SAVEPOINT statement sets a named transaction savepoint with a name of the identifier. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named savepoint without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does not release the row locks that were stored in memory after the savepoint.

Here is the syntax:

SAVEPOINT identifier

ROLLBACK [WORK] TO [SAVEPOINT] identifier

RELEASE SAVEPOINT identifier

## ****LOCK and UNLOCK Tables****

MySQL enables client sessions to acquire table locks explicitly for the purpose of cooperating with other sessions for access to tables or to prevent other sessions from modifying tables during periods when a session requires exclusive access to them. A session can acquire or release locks only for itself. One session cannot acquire locks for another session or release locks held by another session.

LOCK TABLES explicitly acquires table locks for the current client session. Table locks can be acquired for base tables or views. You must have the LOCK TABLES privilege, and the SELECT privilege for each object to be locked.

UNLOCK TABLES explicitly releases any table locks held by the current session. LOCK TABLES implicitly releases any table locks held by the current session before acquiring new locks.

Here is the syntax:

LOCK TABLES

tbl\_name [[AS] alias] lock\_type

[, tbl\_name [[AS] alias] lock\_type] ...

lock\_type:

READ [LOCAL]

| [LOW\_PRIORITY] WRITE

UNLOCK TABLES

### ****SET TRANSACTION Syntax****

SET [GLOBAL | SESSION] TRANSACTION

transaction\_characteristic [, transaction\_characteristic] ...

transaction\_characteristic:

ISOLATION LEVEL level

| READ WRITE

| READ ONLY

level:

REPEATABLE READ

| READ COMMITTED

| READ UNCOMMITTED

| SERIALIZABLE

* With the GLOBAL keyword, the statement applies globally for all subsequent sessions. Existing sessions are unaffected.
* With the SESSION keyword, the statement applies to all subsequent transactions performed within the current session.
* Without any SESSION or GLOBAL keyword, the statement applies to the next (not started) transaction performed within the current session.

## Transactional and Locking Statements

MySQL supports local transactions (within a given client session) through statements such as [SET autocommit](https://dev.mysql.com/doc/refman/8.0/en/commit.html), [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html), [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html), and [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html). See [Section 13.3.1, “START TRANSACTION, COMMIT, and ROLLBACK Statements”](https://dev.mysql.com/doc/refman/8.0/en/commit.html). XA transaction support enables MySQL to participate in distributed transactions as well. See [Section 13.3.8, “XA Transactions”](https://dev.mysql.com/doc/refman/8.0/en/xa.html).

### START TRANSACTION, COMMIT, and ROLLBACK Statements

START TRANSACTION

[transaction\_characteristic [, transaction\_characteristic] ...]

transaction\_characteristic: {

WITH CONSISTENT SNAPSHOT

| READ WRITE

| READ ONLY

}

BEGIN [WORK]

COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET autocommit = {0 | 1}

These statements provide control over use of [transactions](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_transaction):

* START TRANSACTION or BEGIN start a new transaction.
* COMMIT commits the current transaction, making its changes permanent.
* ROLLBACK rolls back the current transaction, canceling its changes.
* SET autocommit disables or enables the default autocommit mode for the current session.

By default, MySQL runs with [autocommit](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_autocommit) mode enabled. This means that, when not otherwise inside a transaction, each statement is atomic, as if it were surrounded by START TRANSACTION and COMMIT. You cannot use ROLLBACK to undo the effect; however, if an error occurs during statement execution, the statement is rolled back.

To disable autocommit mode implicitly for a single series of statements, use the START TRANSACTION statement:

START TRANSACTION;

SELECT @A:=SUM(salary) FROM table1 WHERE type=1;

UPDATE table2 SET summary=@A WHERE type=1;

COMMIT;

With START TRANSACTION, autocommit remains disabled until you end the transaction with COMMIT or ROLLBACK. The autocommit mode then reverts to its previous state.

START TRANSACTION permits several modifiers that control transaction characteristics. To specify multiple modifiers, separate them by commas.

* The WITH CONSISTENT SNAPSHOT modifier starts a [consistent read](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_consistent_read) for storage engines that are capable of it. This applies only to InnoDB. The effect is the same as issuing a START TRANSACTION followed by a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) from any InnoDB table. See [Section 15.7.2.3, “Consistent Nonlocking Reads”](https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html). The WITH CONSISTENT SNAPSHOT modifier does not change the current transaction [isolation level](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_isolation_level), so it provides a consistent snapshot only if the current isolation level is one that permits a consistent read. The only isolation level that permits a consistent read is [REPEATABLE READ](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_repeatable-read). For all other isolation levels, the WITH CONSISTENT SNAPSHOT clause is ignored. A warning is generated when the WITH CONSISTENT SNAPSHOT clause is ignored.
* The READ WRITE and READ ONLY modifiers set the transaction access mode. They permit or prohibit changes to tables used in the transaction. The READ ONLY restriction prevents the transaction from modifying or locking both transactional and nontransactional tables that are visible to other transactions; the transaction can still modify or lock temporary tables.

MySQL enables extra optimizations for queries on InnoDB tables when the transaction is known to be read-only. Specifying READ ONLY ensures these optimizations are applied in cases where the read-only status cannot be determined automatically. See [Section 8.5.3, “Optimizing InnoDB Read-Only Transactions”](https://dev.mysql.com/doc/refman/8.0/en/innodb-performance-ro-txn.html) for more information.

If no access mode is specified, the default mode applies. Unless the default has been changed, it is read/write. It is not permitted to specify both READ WRITE and READ ONLY in the same statement.

In read-only mode, it remains possible to change tables created with the TEMPORARY keyword using DML statements. Changes made with DDL statements are not permitted, just as with permanent tables.

For additional information about transaction access mode, including ways to change the default mode, see [Section 13.3.7, “SET TRANSACTION Statement”](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html).

If the [read\_only](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_read_only) system variable is enabled, explicitly starting a transaction with START TRANSACTION READ WRITE requires the [CONNECTION\_ADMIN](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_connection-admin) privilege (or the deprecated [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege).

**Important**

Many APIs used for writing MySQL client applications (such as JDBC) provide their own methods for starting transactions that can (and sometimes should) be used instead of sending a START TRANSACTION statement from the client. See [Chapter 29, *Connectors and APIs*](https://dev.mysql.com/doc/refman/8.0/en/connectors-apis.html), or the documentation for your API, for more information.

To disable autocommit mode explicitly, use the following statement:

SET autocommit=0;

After disabling autocommit mode by setting the [autocommit](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_autocommit) variable to zero, changes to transaction-safe tables (such as those for [InnoDB](https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html) or [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html)) are not made permanent immediately. You must use [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html) to store your changes to disk or ROLLBACK to ignore the changes.

[autocommit](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_autocommit) is a session variable and must be set for each session. To disable autocommit mode for each new connection, see the description of the [autocommit](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_autocommit) system variable at [Section 5.1.8, “Server System Variables”](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html).

BEGIN and BEGIN WORK are supported as aliases of START TRANSACTION for initiating a transaction. START TRANSACTION is standard SQL syntax, is the recommended way to start an ad-hoc transaction, and permits modifiers that BEGIN does not.

The BEGIN statement differs from the use of the BEGIN keyword that starts a [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) compound statement. The latter does not begin a transaction. See [Section 13.6.1, “BEGIN ... END Compound Statement”](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html).

**Note**

Within all stored programs (stored procedures and functions, triggers, and events), the parser treats BEGIN [WORK] as the beginning of a [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) block. Begin a transaction in this context with [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html) instead.

The optional WORK keyword is supported for COMMIT and ROLLBACK, as are the CHAIN and RELEASE clauses. CHAIN and RELEASE can be used for additional control over transaction completion. The value of the [completion\_type](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_completion_type) system variable determines the default completion behavior. See [Section 5.1.8, “Server System Variables”](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html).

The AND CHAIN clause causes a new transaction to begin as soon as the current one ends, and the new transaction has the same isolation level as the just-terminated transaction. The new transaction also uses the same access mode (READ WRITE or READ ONLY) as the just-terminated transaction. The RELEASE clause causes the server to disconnect the current client session after terminating the current transaction. Including the NO keyword suppresses CHAIN or RELEASE completion, which can be useful if the [completion\_type](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_completion_type) system variable is set to cause chaining or release completion by default.

Beginning a transaction causes any pending transaction to be committed. See [Section 13.3.3, “Statements That Cause an Implicit Commit”](https://dev.mysql.com/doc/refman/8.0/en/implicit-commit.html), for more information.

Beginning a transaction also causes table locks acquired with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) to be released, as though you had executed [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html). Beginning a transaction does not release a global read lock acquired with [FLUSH TABLES WITH READ LOCK](https://dev.mysql.com/doc/refman/8.0/en/flush.html#flush-tables-with-read-lock).

For best results, transactions should be performed using only tables managed by a single transaction-safe storage engine. Otherwise, the following problems can occur:

* If you use tables from more than one transaction-safe storage engine (such as InnoDB), and the transaction isolation level is not [SERIALIZABLE](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_serializable), it is possible that when one transaction commits, another ongoing transaction that uses the same tables sees only some of the changes made by the first transaction. That is, the atomicity of transactions is not guaranteed with mixed engines and inconsistencies can result. (If mixed-engine transactions are infrequent, you can use [SET TRANSACTION ISOLATION LEVEL](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) to set the isolation level to [SERIALIZABLE](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_serializable) on a per-transaction basis as necessary.)
* If you use tables that are not transaction-safe within a transaction, changes to those tables are stored at once, regardless of the status of autocommit mode.
* If you issue a [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement after updating a nontransactional table within a transaction, an [ER\_WARNING\_NOT\_COMPLETE\_ROLLBACK](https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html#error_er_warning_not_complete_rollback) warning occurs. Changes to transaction-safe tables are rolled back, but not changes to nontransaction-safe tables.

Each transaction is stored in the binary log in one chunk, upon [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html). Transactions that are rolled back are not logged. (***Exception***: Modifications to nontransactional tables cannot be rolled back. If a transaction that is rolled back includes modifications to nontransactional tables, the entire transaction is logged with a [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement at the end to ensure that modifications to the nontransactional tables are replicated.) See [Section 5.4.4, “The Binary Log”](https://dev.mysql.com/doc/refman/8.0/en/binary-log.html).

You can change the isolation level or access mode for transactions with the [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) statement. See [Section 13.3.7, “SET TRANSACTION Statement”](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html).

Rolling back can be a slow operation that may occur implicitly without the user having explicitly asked for it (for example, when an error occurs). Because of this, [SHOW PROCESSLIST](https://dev.mysql.com/doc/refman/8.0/en/show-processlist.html) displays Rolling back in the State column for the session, not only for explicit rollbacks performed with the [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement but also for implicit rollbacks.

**Note**

In MySQL 8.0, BEGIN, COMMIT, and ROLLBACK are not affected by [--replicate-do-db](https://dev.mysql.com/doc/refman/8.0/en/replication-options-replica.html#option_mysqld_replicate-do-db) or [--replicate-ignore-db](https://dev.mysql.com/doc/refman/8.0/en/replication-options-replica.html#option_mysqld_replicate-ignore-db) rules.

When InnoDB performs a complete rollback of a transaction, all locks set by the transaction are released. If a single SQL statement within a transaction rolls back as a result of an error, such as a duplicate key error, locks set by the statement are preserved while the transaction remains active. This happens because InnoDB stores row locks in a format such that it cannot know afterward which lock was set by which statement.

If a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) statement within a transaction calls a stored function, and a statement within the stored function fails, that statement rolls back. If [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) is executed for the transaction subsequently, the entire transaction rolls back.

### Statements That Cannot Be Rolled Back

Some statements cannot be rolled back. In general, these include data definition language (DDL) statements, such as those that create or drop databases, those that create, drop, or alter tables or stored routines.

You should design your transactions not to include such statements. If you issue a statement early in a transaction that cannot be rolled back, and then another statement later fails, the full effect of the transaction cannot be rolled back in such cases by issuing a [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement.

### Statements That Cause an Implicit Commit

The statements listed in this section (and any synonyms for them) implicitly end any transaction active in the current session, as if you had done a [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html) before executing the statement.

Most of these statements also cause an implicit commit after executing. The intent is to handle each such statement in its own special transaction. Transaction-control and locking statements are exceptions: If an implicit commit occurs before execution, another does not occur after.

* ***Data definition language (DDL) statements that define or modify database objects.*** [ALTER EVENT](https://dev.mysql.com/doc/refman/8.0/en/alter-event.html), [ALTER FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/alter-function.html), [ALTER PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/alter-procedure.html), [ALTER SERVER](https://dev.mysql.com/doc/refman/8.0/en/alter-server.html), [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html), [ALTER VIEW](https://dev.mysql.com/doc/refman/8.0/en/alter-view.html), [CREATE DATABASE](https://dev.mysql.com/doc/refman/8.0/en/create-database.html), [CREATE EVENT](https://dev.mysql.com/doc/refman/8.0/en/create-event.html), [CREATE FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/create-function.html), [CREATE INDEX](https://dev.mysql.com/doc/refman/8.0/en/create-index.html), [CREATE PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html), [CREATE ROLE](https://dev.mysql.com/doc/refman/8.0/en/create-role.html), [CREATE SERVER](https://dev.mysql.com/doc/refman/8.0/en/create-server.html), [CREATE SPATIAL REFERENCE SYSTEM](https://dev.mysql.com/doc/refman/8.0/en/create-spatial-reference-system.html), [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html), [CREATE TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html), [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html), [DROP DATABASE](https://dev.mysql.com/doc/refman/8.0/en/drop-database.html), [DROP EVENT](https://dev.mysql.com/doc/refman/8.0/en/drop-event.html), [DROP FUNCTION](https://dev.mysql.com/doc/refman/8.0/en/drop-function.html), [DROP INDEX](https://dev.mysql.com/doc/refman/8.0/en/drop-index.html), [DROP PROCEDURE](https://dev.mysql.com/doc/refman/8.0/en/drop-procedure.html), [DROP ROLE](https://dev.mysql.com/doc/refman/8.0/en/drop-role.html), [DROP SERVER](https://dev.mysql.com/doc/refman/8.0/en/drop-server.html), [DROP SPATIAL REFERENCE SYSTEM](https://dev.mysql.com/doc/refman/8.0/en/drop-spatial-reference-system.html), [DROP TABLE](https://dev.mysql.com/doc/refman/8.0/en/drop-table.html), [DROP TRIGGER](https://dev.mysql.com/doc/refman/8.0/en/drop-trigger.html), [DROP VIEW](https://dev.mysql.com/doc/refman/8.0/en/drop-view.html), [INSTALL PLUGIN](https://dev.mysql.com/doc/refman/8.0/en/install-plugin.html), [RENAME TABLE](https://dev.mysql.com/doc/refman/8.0/en/rename-table.html), [TRUNCATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/truncate-table.html), [UNINSTALL PLUGIN](https://dev.mysql.com/doc/refman/8.0/en/uninstall-plugin.html).

[CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) and [DROP TABLE](https://dev.mysql.com/doc/refman/8.0/en/drop-table.html) statements do not commit a transaction if the TEMPORARY keyword is used. (This does not apply to other operations on temporary tables such as [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html) and [CREATE INDEX](https://dev.mysql.com/doc/refman/8.0/en/create-index.html), which do cause a commit.) However, although no implicit commit occurs, neither can the statement be rolled back, which means that the use of such statements causes transactional atomicity to be violated. For example, if you use [CREATE TEMPORARY TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) and then roll back the transaction, the table remains in existence.

The [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) statement in InnoDB is processed as a single transaction. This means that a [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) from the user does not undo [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) statements the user made during that transaction.

[CREATE TABLE ... SELECT](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) causes an implicit commit before and after the statement is executed when you are creating nontemporary tables. (No commit occurs for CREATE TEMPORARY TABLE ... SELECT.)

* ***Statements that implicitly use or modify tables in the****mysql****database.*** [ALTER USER](https://dev.mysql.com/doc/refman/8.0/en/alter-user.html), [CREATE USER](https://dev.mysql.com/doc/refman/8.0/en/create-user.html), [DROP USER](https://dev.mysql.com/doc/refman/8.0/en/drop-user.html), [GRANT](https://dev.mysql.com/doc/refman/8.0/en/grant.html), [RENAME USER](https://dev.mysql.com/doc/refman/8.0/en/rename-user.html), [REVOKE](https://dev.mysql.com/doc/refman/8.0/en/revoke.html), [SET PASSWORD](https://dev.mysql.com/doc/refman/8.0/en/set-password.html).
* ***Transaction-control and locking statements.*** [BEGIN](https://dev.mysql.com/doc/refman/8.0/en/commit.html), [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), SET autocommit = 1 (if the value is not already 1), [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html), [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html).

[UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) commits a transaction only if any tables currently have been locked with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) to acquire nontransactional table locks. A commit does not occur for [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) following [FLUSH TABLES WITH READ LOCK](https://dev.mysql.com/doc/refman/8.0/en/flush.html#flush-tables-with-read-lock) because the latter statement does not acquire table-level locks.

Transactions cannot be nested. This is a consequence of the implicit commit performed for any current transaction when you issue a [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement or one of its synonyms.

Statements that cause an implicit commit cannot be used in an XA transaction while the transaction is in an ACTIVE state.

The [BEGIN](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement differs from the use of the BEGIN keyword that starts a [BEGIN ... END](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html) compound statement. The latter does not cause an implicit commit. See [Section 13.6.1, “BEGIN ... END Compound Statement”](https://dev.mysql.com/doc/refman/8.0/en/begin-end.html).

* ***Data loading statements.*** [LOAD DATA](https://dev.mysql.com/doc/refman/8.0/en/load-data.html). [LOAD DATA](https://dev.mysql.com/doc/refman/8.0/en/load-data.html) causes an implicit commit only for tables using the [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html) storage engine.
* ***Administrative statements.*** [ANALYZE TABLE](https://dev.mysql.com/doc/refman/8.0/en/analyze-table.html), [CACHE INDEX](https://dev.mysql.com/doc/refman/8.0/en/cache-index.html), [CHECK TABLE](https://dev.mysql.com/doc/refman/8.0/en/check-table.html), [FLUSH](https://dev.mysql.com/doc/refman/8.0/en/flush.html), [LOAD INDEX INTO CACHE](https://dev.mysql.com/doc/refman/8.0/en/load-index.html), [OPTIMIZE TABLE](https://dev.mysql.com/doc/refman/8.0/en/optimize-table.html), [REPAIR TABLE](https://dev.mysql.com/doc/refman/8.0/en/repair-table.html), [RESET](https://dev.mysql.com/doc/refman/8.0/en/reset.html) (but not [RESET PERSIST](https://dev.mysql.com/doc/refman/8.0/en/reset-persist.html)).
* ***Replication control statements***. [START REPLICA | SLAVE](https://dev.mysql.com/doc/refman/8.0/en/start-replica.html), [STOP REPLICA | SLAVE](https://dev.mysql.com/doc/refman/8.0/en/stop-replica.html), [RESET REPLICA | SLAVE](https://dev.mysql.com/doc/refman/8.0/en/reset-replica.html), [CHANGE MASTER TO](https://dev.mysql.com/doc/refman/8.0/en/change-master-to.html).

### SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements

SAVEPOINT identifier

ROLLBACK [WORK] TO [SAVEPOINT] identifier

RELEASE SAVEPOINT identifier

InnoDB supports the SQL statements [SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html), [ROLLBACK TO SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html), [RELEASE SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html) and the optional WORK keyword for [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html).

The [SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html) statement sets a named transaction savepoint with a name of ***identifier***. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

The [ROLLBACK TO SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html) statement rolls back a transaction to the named savepoint without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does not release the row locks that were stored in memory after the savepoint. (For a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock is released in the undo.) Savepoints that were set at a later time than the named savepoint are deleted.

If the [ROLLBACK TO SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html) statement returns the following error, it means that no savepoint with the specified name exists:

ERROR 1305 (42000): SAVEPOINT identifier does not exist

The [RELEASE SAVEPOINT](https://dev.mysql.com/doc/refman/8.0/en/savepoint.html) statement removes the named savepoint from the set of savepoints of the current transaction. No commit or rollback occurs. It is an error if the savepoint does not exist.

All savepoints of the current transaction are deleted if you execute a [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html), or a [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) that does not name a savepoint.

A new savepoint level is created when a stored function is invoked or a trigger is activated. The savepoints on previous levels become unavailable and thus do not conflict with savepoints on the new level. When the function or trigger terminates, any savepoints it created are released and the previous savepoint level is restored.

### LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE Statements

LOCK INSTANCE FOR BACKUP

UNLOCK INSTANCE

LOCK INSTANCE FOR BACKUP acquires an instance-level backup lock that permits DML during an online backup while preventing operations that could result in an inconsistent snapshot.

Executing the LOCK INSTANCE FOR BACKUP statement requires the [BACKUP\_ADMIN](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_backup-admin) privilege. The [BACKUP\_ADMIN](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_backup-admin) privilege is automatically granted to users with the [RELOAD](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_reload) privilege when performing an in-place upgrade to MySQL 8.0 from an earlier version.

Multiple sessions can hold a backup lock simultaneously.

UNLOCK INSTANCE releases a backup lock held by the current session. A backup lock held by a session is also released if the session is terminated.

LOCK INSTANCE FOR BACKUP prevents files from being created, renamed, or removed. [REPAIR TABLE](https://dev.mysql.com/doc/refman/8.0/en/repair-table.html) [TRUNCATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/truncate-table.html), [OPTIMIZE TABLE](https://dev.mysql.com/doc/refman/8.0/en/optimize-table.html), and account management statements are blocked. See [Section 13.7.1, “Account Management Statements”](https://dev.mysql.com/doc/refman/8.0/en/account-management-statements.html). Operations that modify InnoDB files that are not recorded in the InnoDB redo log are also blocked.

LOCK INSTANCE FOR BACKUP permits DDL operations that only affect user-created temporary tables. In effect, files that belong to user-created temporary tables can be created, renamed, or removed while a backup lock is held. Creation of binary log files is also permitted.

A backup lock acquired by LOCK INSTANCE FOR BACKUP is independent of transactional locks and locks taken by [FLUSH TABLES ***tbl\_name*** [, ***tbl\_name***] ... WITH READ LOCK](https://dev.mysql.com/doc/refman/8.0/en/flush.html#flush-tables-with-read-lock-with-list), and the following sequences of statements are permitted:

LOCK INSTANCE FOR BACKUP;

FLUSH TABLES tbl\_name [, tbl\_name] ... WITH READ LOCK;

UNLOCK TABLES;

UNLOCK INSTANCE;

FLUSH TABLES tbl\_name [, tbl\_name] ... WITH READ LOCK;

LOCK INSTANCE FOR BACKUP;

UNLOCK INSTANCE;

UNLOCK TABLES;

The [lock\_wait\_timeout](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_lock_wait_timeout) setting defines the amount of time that a LOCK INSTANCE FOR BACKUP statement waits to acquire a lock before giving up.

### LOCK TABLES and UNLOCK TABLES Statements

LOCK TABLES

tbl\_name [[AS] alias] lock\_type

[, tbl\_name [[AS] alias] lock\_type] ...

lock\_type: {

READ [LOCAL]

| [LOW\_PRIORITY] WRITE

}

UNLOCK TABLES

MySQL enables client sessions to acquire table locks explicitly for the purpose of cooperating with other sessions for access to tables, or to prevent other sessions from modifying tables during periods when a session requires exclusive access to them. A session can acquire or release locks only for itself. One session cannot acquire locks for another session or release locks held by another session.

Locks may be used to emulate transactions or to get more speed when updating tables. This is explained in more detail in [Table-Locking Restrictions and Conditions](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html#lock-tables-restrictions).

[LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) explicitly acquires table locks for the current client session. Table locks can be acquired for base tables or views. You must have the [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_lock-tables) privilege, and the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) privilege for each object to be locked.

For view locking, [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) adds all base tables used in the view to the set of tables to be locked and locks them automatically. For tables underlying any view being locked, [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) checks that the view definer (for SQL SECURITY DEFINER views) or invoker (for all views) has the proper privileges on the tables.

If you lock a table explicitly with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), any tables used in triggers are also locked implicitly, as described in [LOCK TABLES and Triggers](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html#lock-tables-and-triggers).

If you lock a table explicitly with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), any tables related by a foreign key constraint are opened and locked implicitly. For foreign key checks, a shared read-only lock ([LOCK TABLES READ](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html)) is taken on related tables. For cascading updates, a shared-nothing write lock ([LOCK TABLES WRITE](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html)) is taken on related tables that are involved in the operation.

[UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) explicitly releases any table locks held by the current session. [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) implicitly releases any table locks held by the current session before acquiring new locks.

Another use for [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) is to release the global read lock acquired with the [FLUSH TABLES WITH READ LOCK](https://dev.mysql.com/doc/refman/8.0/en/flush.html#flush-tables-with-read-lock) statement, which enables you to lock all tables in all databases. See [Section 13.7.8.3, “FLUSH Statement”](https://dev.mysql.com/doc/refman/8.0/en/flush.html). (This is a very convenient way to get backups if you have a file system such as Veritas that can take snapshots in time.)

A table lock protects only against inappropriate reads or writes by other sessions. A session holding a WRITE lock can perform table-level operations such as [DROP TABLE](https://dev.mysql.com/doc/refman/8.0/en/drop-table.html) or [TRUNCATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/truncate-table.html). For sessions holding a READ lock, [DROP TABLE](https://dev.mysql.com/doc/refman/8.0/en/drop-table.html) and [TRUNCATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/truncate-table.html) operations are not permitted.

The following discussion applies only to non-TEMPORARY tables. [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) is permitted (but ignored) for a TEMPORARY table. The table can be accessed freely by the session within which it was created, regardless of what other locking may be in effect. No lock is necessary because no other session can see the table.

#### *Table Lock Acquisition*

To acquire table locks within the current session, use the [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement, which acquires metadata locks .

The following lock types are available:

READ [LOCAL] lock:

* The session that holds the lock can read the table (but not write it).
* Multiple sessions can acquire a READ lock for the table at the same time.
* Other sessions can read the table without explicitly acquiring a READ lock.
* The LOCAL modifier enables nonconflicting [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statements (concurrent inserts) by other sessions to execute while the lock is held. (See [Section 8.11.3, “Concurrent Inserts”](https://dev.mysql.com/doc/refman/8.0/en/concurrent-inserts.html).) However, READ LOCAL cannot be used if you are going to manipulate the database using processes external to the server while you hold the lock. For InnoDB tables, READ LOCAL is the same as READ.

[LOW\_PRIORITY] WRITE lock:

* The session that holds the lock can read and write the table.
* Only the session that holds the lock can access the table. No other session can access it until the lock is released.
* Lock requests for the table by other sessions block while the WRITE lock is held.
* The LOW\_PRIORITY modifier has no effect. In previous versions of MySQL, it affected locking behavior, but this is no longer true. It is now deprecated and its use produces a warning. Use WRITE without LOW\_PRIORITY instead.

WRITE locks normally have higher priority than READ locks to ensure that updates are processed as soon as possible. This means that if one session obtains a READ lock and then another session requests a WRITE lock, subsequent READ lock requests wait until the session that requested the WRITE lock has obtained the lock and released it. (An exception to this policy can occur for small values of the [max\_write\_lock\_count](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_max_write_lock_count) system variable; see [Section 8.11.4, “Metadata Locking”](https://dev.mysql.com/doc/refman/8.0/en/metadata-locking.html).)

If the [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement must wait due to locks held by other sessions on any of the tables, it blocks until all locks can be acquired.

A session that requires locks must acquire all the locks that it needs in a single [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement. While the locks thus obtained are held, the session can access only the locked tables. For example, in the following sequence of statements, an error occurs for the attempt to access t2 because it was not locked in the [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement:

mysql> LOCK TABLES t1 READ;

mysql> SELECT COUNT(\*) FROM t1;

+----------+

| COUNT(\*) |

+----------+

| 3 |

+----------+

mysql> SELECT COUNT(\*) FROM t2;

ERROR 1100 (HY000): Table 't2' was not locked with LOCK TABLES

Tables in the INFORMATION\_SCHEMA database are an exception. They can be accessed without being locked explicitly even while a session holds table locks obtained with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html).

You cannot refer to a locked table multiple times in a single query using the same name. Use aliases instead, and obtain a separate lock for the table and each alias:

mysql> LOCK TABLE t WRITE, t AS t1 READ;

mysql> INSERT INTO t SELECT \* FROM t;

ERROR 1100: Table 't' was not locked with LOCK TABLES

mysql> INSERT INTO t SELECT \* FROM t AS t1;

The error occurs for the first [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) because there are two references to the same name for a locked table. The second [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) succeeds because the references to the table use different names.

If your statements refer to a table by means of an alias, you must lock the table using that same alias. It does not work to lock the table without specifying the alias:

mysql> LOCK TABLE t READ;

mysql> SELECT \* FROM t AS myalias;

ERROR 1100: Table 'myalias' was not locked with LOCK TABLES

Conversely, if you lock a table using an alias, you must refer to it in your statements using that alias:

mysql> LOCK TABLE t AS myalias READ;

mysql> SELECT \* FROM t;

ERROR 1100: Table 't' was not locked with LOCK TABLES

mysql> SELECT \* FROM t AS myalias;

#### *Table Lock Release*

When the table locks held by a session are released, they are all released at the same time. A session can release its locks explicitly, or locks may be released implicitly under certain conditions.

* A session can release its locks explicitly with [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html).
* If a session issues a [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement to acquire a lock while already holding locks, its existing locks are released implicitly before the new locks are granted.
* If a session begins a transaction (for example, with [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html)), an implicit [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) is performed, which causes existing locks to be released. (For additional information about the interaction between table locking and transactions, see [Interaction of Table Locking and Transactions](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html#lock-tables-and-transactions).)

If the connection for a client session terminates, whether normally or abnormally, the server implicitly releases all table locks held by the session (transactional and nontransactional). If the client reconnects, the locks are no longer in effect. In addition, if the client had an active transaction, the server rolls back the transaction upon disconnect, and if reconnect occurs, the new session begins with autocommit enabled. For this reason, clients may wish to disable auto-reconnect. With auto-reconnect in effect, the client is not notified if reconnect occurs but any table locks or current transaction are lost. With auto-reconnect disabled, if the connection drops, an error occurs for the next statement issued. The client can detect the error and take appropriate action such as reacquiring the locks or redoing the transaction. See [C API Automatic Reconnection Control](https://dev.mysql.com/doc/c-api/8.0/en/c-api-auto-reconnect.html).

**Note**

If you use [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html) on a locked table, it may become unlocked. For example, if you attempt a second [ALTER TABLE](https://dev.mysql.com/doc/refman/8.0/en/alter-table.html) operation, the result may be an error Table '***tbl\_name***' was not locked with LOCK TABLES. To handle this, lock the table again prior to the second alteration. See also [Section B.3.6.1, “Problems with ALTER TABLE”](https://dev.mysql.com/doc/refman/8.0/en/alter-table-problems.html).

#### *Interaction of Table Locking and Transactions*

[LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html" \o "13.3.6 LOCK TABLES and UNLOCK TABLES Statements) and [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) interact with the use of transactions as follows:

* [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) is not transaction-safe and implicitly commits any active transaction before attempting to lock the tables.
* [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) implicitly commits any active transaction, but only if [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) has been used to acquire table locks. For example, in the following set of statements, [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) releases the global read lock but does not commit the transaction because no table locks are in effect:
* FLUSH TABLES WITH READ LOCK;
* START TRANSACTION;
* SELECT ... ;
* UNLOCK TABLES
* Beginning a transaction (for example, with [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html)) implicitly commits any current transaction and releases existing table locks.
* [FLUSH TABLES WITH READ LOCK](https://dev.mysql.com/doc/refman/8.0/en/flush.html#flush-tables-with-read-lock) acquires a global read lock and not table locks, so it is not subject to the same behavior as [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) and [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) with respect to table locking and implicit commits. For example, [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html) does not release the global read lock. See [Section 13.7.8.3, “FLUSH Statement”](https://dev.mysql.com/doc/refman/8.0/en/flush.html).
* Other statements that implicitly cause transactions to be committed do not release existing table locks..
* The correct way to use [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) and [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) with transactional tables, such as InnoDB tables, is to begin a transaction with SET autocommit = 0 (not [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html)) followed by [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), and to not call [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) until you commit the transaction explicitly. For example, if you need to write to table t1 and read from table t2, you can do this:
* SET autocommit=0;
* LOCK TABLES t1 WRITE, t2 READ, ...;
* *...* *do* something *with* *tables* t1 *and* t2 here *...*
* COMMIT;

UNLOCK TABLES;

When you call [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), InnoDB internally takes its own table lock, and MySQL takes its own table lock. InnoDB releases its internal table lock at the next commit, but for MySQL to release its table lock, you have to call [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html). You should not have [autocommit = 1](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_autocommit), because then InnoDB releases its internal table lock immediately after the call of [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), and deadlocks can very easily happen. InnoDB does not acquire the internal table lock at all if [autocommit = 1](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_autocommit), to help old applications avoid unnecessary deadlocks.

* [ROLLBACK](https://dev.mysql.com/doc/refman/8.0/en/commit.html) does not release table locks.

#### *LOCK TABLES and Triggers*

If you lock a table explicitly with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), any tables used in triggers are also locked implicitly:

* The locks are taken as the same time as those acquired explicitly with the [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement.
* The lock on a table used in a trigger depends on whether the table is used only for reading. If so, a read lock suffices. Otherwise, a write lock is used.
* If a table is locked explicitly for reading with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), but needs to be locked for writing because it might be modified within a trigger, a write lock is taken rather than a read lock. (That is, an implicit write lock needed due to the table's appearance within a trigger causes an explicit read lock request for the table to be converted to a write lock request.)

Suppose that you lock two tables, t1 and t2, using this statement:

LOCK TABLES t1 WRITE, t2 READ;

If t1 or t2 have any triggers, tables used within the triggers are also locked. Suppose that t1 has a trigger defined like this:

CREATE TRIGGER t1\_a\_ins AFTER INSERT ON t1 FOR EACH ROW

BEGIN

UPDATE t4 SET count = count+1

WHERE id = NEW.id AND EXISTS (SELECT a FROM t3);

INSERT INTO t2 VALUES(1, 2);

END;

The result of the [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement is that t1 and t2 are locked because they appear in the statement, and t3 and t4 are locked because they are used within the trigger:

* t1 is locked for writing per the WRITE lock request.
* t2 is locked for writing, even though the request is for a READ lock. This occurs because t2 is inserted into within the trigger, so the READ request is converted to a WRITE request.
* t3 is locked for reading because it is only read from within the trigger.
* t4 is locked for writing because it might be updated within the trigger.

#### *Table-Locking Restrictions and Conditions*

You can safely use [KILL](https://dev.mysql.com/doc/refman/8.0/en/kill.html) to terminate a session that is waiting for a table lock. See [Section 13.7.8.4, “KILL Statement”](https://dev.mysql.com/doc/refman/8.0/en/kill.html).

[LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) and [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) cannot be used within stored programs.

Tables in the performance\_schema database cannot be locked with [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), except the setup\_***xxx*** tables.

The following statements are prohibited while a [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement is in effect: [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html), [CREATE TABLE ... LIKE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html), [CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/create-view.html), [DROP VIEW](https://dev.mysql.com/doc/refman/8.0/en/drop-view.html), and DDL statements on stored functions and procedures and events.

For some operations, system tables in the mysql database must be accessed. For example, the [HELP](https://dev.mysql.com/doc/refman/8.0/en/help.html) statement requires the contents of the server-side help tables, and [CONVERT\_TZ()](https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_convert-tz) might need to read the time zone tables. The server implicitly locks the system tables for reading as necessary so that you need not lock them explicitly. These tables are treated as just described:

mysql.help\_category

mysql.help\_keyword

mysql.help\_relation

mysql.help\_topic

mysql.time\_zone

mysql.time\_zone\_leap\_second

mysql.time\_zone\_name

mysql.time\_zone\_transition

mysql.time\_zone\_transition\_type

If you want to explicitly place a WRITE lock on any of those tables with a [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) statement, the table must be the only one locked; no other table can be locked with the same statement.

Normally, you do not need to lock tables, because all single [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) statements are atomic; no other session can interfere with any other currently executing SQL statement. However, there are a few cases when locking tables may provide an advantage:

* If you are going to run many operations on a set of MyISAM tables, it is much faster to lock the tables you are going to use. Locking MyISAM tables speeds up inserting, updating, or deleting on them because MySQL does not flush the key cache for the locked tables until [UNLOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) is called. Normally, the key cache is flushed after each SQL statement.

The downside to locking the tables is that no session can update a READ-locked table (including the one holding the lock) and no session can access a WRITE-locked table other than the one holding the lock.

* If you are using tables for a nontransactional storage engine, you must use [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) if you want to ensure that no other session modifies the tables between a [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) and an [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html). The example shown here requires [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) to execute safely:
* LOCK TABLES trans READ, customer WRITE;
* SELECT SUM(value) FROM trans WHERE customer\_id=some\_id;
* UPDATE customer
* SET total\_value=sum\_from\_previous\_statement
* WHERE customer\_id=some\_id;

UNLOCK TABLES;

Without [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html), it is possible that another session might insert a new row in the trans table between execution of the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/select.html) and [UPDATE](https://dev.mysql.com/doc/refman/8.0/en/update.html) statements.

You can avoid using [LOCK TABLES](https://dev.mysql.com/doc/refman/8.0/en/lock-tables.html) in many cases by using relative updates (UPDATE customer SET ***value***=***value***+***new\_value***) or the [LAST\_INSERT\_ID()](https://dev.mysql.com/doc/refman/8.0/en/information-functions.html#function_last-insert-id) function.

You can also avoid locking tables in some cases by using the user-level advisory lock functions [GET\_LOCK()](https://dev.mysql.com/doc/refman/8.0/en/locking-functions.html#function_get-lock) and [RELEASE\_LOCK()](https://dev.mysql.com/doc/refman/8.0/en/locking-functions.html#function_release-lock). These locks are saved in a hash table in the server and implemented with pthread\_mutex\_lock() and pthread\_mutex\_unlock() for high speed.

### SET TRANSACTION Statement

SET [GLOBAL | SESSION] TRANSACTION

transaction\_characteristic [, transaction\_characteristic] ...

transaction\_characteristic: {

ISOLATION LEVEL level

| access\_mode

}

level: {

REPEATABLE READ

| READ COMMITTED

| READ UNCOMMITTED

| SERIALIZABLE

}

access\_mode: {

READ WRITE

| READ ONLY

}

This statement specifies [transaction](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_transaction) characteristics. It takes a list of one or more characteristic values separated by commas. Each characteristic value sets the transaction [isolation level](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_isolation_level) or access mode. The isolation level is used for operations on [InnoDB](https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html) tables. The access mode specifies whether transactions operate in read/write or read-only mode.

In addition, [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) can include an optional GLOBAL or SESSION keyword to indicate the scope of the statement.

* [Transaction Isolation Levels](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html#set-transaction-isolation-level)
* [Transaction Access Mode](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html#set-transaction-access-mode)
* [Transaction Characteristic Scope](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html#set-transaction-scope)

#### *Transaction Isolation Levels*

To set the transaction isolation level, use an ISOLATION LEVEL ***level*** clause. It is not permitted to specify multiple ISOLATION LEVEL clauses in the same [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) statement.

The default isolation level is [REPEATABLE READ](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_repeatable-read). Other permitted values are [READ COMMITTED](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_read-committed), [READ UNCOMMITTED](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_read-uncommitted), and [SERIALIZABLE](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_serializable)..

#### *Transaction Access Mode*

To set the transaction access mode, use a READ WRITE or READ ONLY clause. It is not permitted to specify multiple access-mode clauses in the same [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) statement.

By default, a transaction takes place in read/write mode, with both reads and writes permitted to tables used in the transaction. This mode may be specified explicitly using [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) with an access mode of READ WRITE.

If the transaction access mode is set to READ ONLY, changes to tables are prohibited. This may enable storage engines to make performance improvements that are possible when writes are not permitted.

In read-only mode, it remains possible to change tables created with the TEMPORARY keyword using DML statements. Changes made with DDL statements are not permitted, just as with permanent tables.

The READ WRITE and READ ONLY access modes also may be specified for an individual transaction using the [START TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/commit.html) statement.

#### *Transaction Characteristic Scope*

You can set transaction characteristics globally, for the current session, or for the next transaction only:

* With the GLOBAL keyword:
  + The statement applies globally for all subsequent sessions.
  + Existing sessions are unaffected.
* With the SESSION keyword:
  + The statement applies to all subsequent transactions performed within the current session.
  + The statement is permitted within transactions, but does not affect the current ongoing transaction.
  + If executed between transactions, the statement overrides any preceding statement that sets the next-transaction value of the named characteristics.
* Without any SESSION or GLOBAL keyword:
  + The statement applies only to the next single transaction performed within the session.
  + Subsequent transactions revert to using the session value of the named characteristics.
  + The statement is not permitted within transactions:
  + mysql> START TRANSACTION;
  + Query OK, 0 rows affected (0.02 sec)
  + mysql> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  + ERROR 1568 (25001): Transaction characteristics can't be changed

while a transaction is in progress

A change to global transaction characteristics requires the [CONNECTION\_ADMIN](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_connection-admin) privilege (or the deprecated [SUPER](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_super) privilege). Any session is free to change its session characteristics (even in the middle of a transaction), or the characteristics for its next transaction (prior to the start of that transaction).

To set the global isolation level at server startup, use the [--transaction-isolation=***level***](https://dev.mysql.com/doc/refman/8.0/en/server-options.html#option_mysqld_transaction-isolation) option on the command line or in an option file. Values of ***level*** for this option use dashes rather than spaces, so the permissible values are [READ-UNCOMMITTED](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_read-uncommitted), [READ-COMMITTED](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_read-committed), [REPEATABLE-READ](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_repeatable-read), or [SERIALIZABLE](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_serializable).

Similarly, to set the global transaction access mode at server startup, use the [--transaction-read-only](https://dev.mysql.com/doc/refman/8.0/en/server-options.html#option_mysqld_transaction-read-only) option. The default is OFF (read/write mode) but the value can be set to ON for a mode of read only.

For example, to set the isolation level to [REPEATABLE READ](https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html#isolevel_repeatable-read) and the access mode to READ WRITE, use these lines in the [mysqld] section of an option file:

[mysqld]

transaction-isolation = REPEATABLE-READ

transaction-read-only = OFF

At runtime, characteristics at the global, session, and next-transaction scope levels can be set indirectly using the [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) statement, as described previously. They can also be set directly using the [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html) statement to assign values to the [transaction\_isolation](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_transaction_isolation) and [transaction\_read\_only](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_transaction_read_only) system variables:

* [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) permits optional GLOBAL and SESSION keywords for setting transaction characteristics at different scope levels.
* The [SET](https://dev.mysql.com/doc/refman/8.0/en/set-variable.html) statement for assigning values to the [transaction\_isolation](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_transaction_isolation) and [transaction\_read\_only](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_transaction_read_only) system variables has syntaxes for setting these variables at different scope levels.

The following tables show the characteristic scope level set by each [SET TRANSACTION](https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html) and variable-assignment syntax.

**Table 13.9 SET TRANSACTION Syntax for Transaction Characteristics**

| **Syntax** | **Affected Characteristic Scope** |
| --- | --- |
| SET GLOBAL TRANSACTION ***transaction\_characteristic*** | Global |
| SET SESSION TRANSACTION ***transaction\_characteristic*** | Session |
| SET TRANSACTION ***transaction\_characteristic*** | Next transaction only |

**Table 13.10 SET Syntax for Transaction Characteristics**

| **Syntax** | **Affected Characteristic Scope** |
| --- | --- |
| SET GLOBAL ***var\_name*** = ***value*** | Global |
| SET @@GLOBAL.***var\_name*** = ***value*** | Global |
| SET PERSIST ***var\_name*** = ***value*** | Global |
| SET @@PERSIST.***var\_name*** = ***value*** | Global |
| SET PERSIST\_ONLY ***var\_name*** = ***value*** | No runtime effect |
| SET @@PERSIST\_ONLY.***var\_name*** = ***value*** | No runtime effect |
| SET SESSION ***var\_name*** = ***value*** | Session |
| SET @@SESSION.***var\_name*** = ***value*** | Session |
| SET ***var\_name*** = ***value*** | Session |
| SET @@***var\_name*** = ***value*** | Next transaction only |
| **Syntax** | **Affected Characteristic Scope** |

It is possible to check the global and session values of transaction characteristics at runtime:

SELECT @@GLOBAL.transaction\_isolation, @@GLOBAL.transaction\_read\_only;

SELECT @@SESSION.transaction\_isolation, @@SESSION.transaction\_read\_only;

# DCL data control language

A **data control language** (**DCL**) is a syntax similar to a computer [programming language](https://en.wikipedia.org/wiki/Programming_language) used to control access to data stored in a database ([Authorization](https://en.wikipedia.org/wiki/Authorization_(computer_access_control))). In particular, it is a component of [Structured Query Language](https://en.wikipedia.org/wiki/Structured_Query_Language) (SQL). Data Control Language is one of the logical group in SQL Commands. [SQL](https://blog.ansi.org/2018/10/sql-standard-iso-iec-9075-2016-ansi-x3-135/)[[1]](https://en.wikipedia.org/wiki/Data_control_language#cite_note-1) is the standard language for relational database management systems. SQL statements are used to perform tasks such as insert data to a database, delete or update data in a database, or retrieve data from a database.

Though database systems use SQL, they also have their own additional proprietary extensions that are usually only used on their system.  For Example Microsoft SQL server uses Transact-SQL (T-SQL) which is an extension of SQL. Similarly Oracle uses PL-SQL which is their proprietary extension for them only. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

Examples of DCL commands include:

* [GRANT](https://en.wikipedia.org/wiki/Grant_(SQL)) to allow specified users to perform specified tasks.
* [REVOKE](https://en.wikipedia.org/wiki/Revoke_(SQL)) to remove the user accessibility to database object.

The operations for which privileges may be granted to or revoked from a user or role apply to both the [Data definition language](https://en.wikipedia.org/wiki/Data_definition_language) (DDL) and the [Data manipulation language](https://en.wikipedia.org/wiki/Data_manipulation_language) (DML), and may include [CONNECT](https://en.wikipedia.org/w/index.php?title=Connect_(SQL)&action=edit&redlink=1), [SELECT](https://en.wikipedia.org/wiki/Select_(SQL)), [INSERT](https://en.wikipedia.org/wiki/Insert_(SQL)), [UPDATE](https://en.wikipedia.org/wiki/Update_(SQL)), [DELETE](https://en.wikipedia.org/wiki/Delete_(SQL)), [EXECUTE](https://en.wikipedia.org/w/index.php?title=Execute_(SQL)&action=edit&redlink=1), and [USAGE](https://en.wikipedia.org/w/index.php?title=Usage_(SQL)&action=edit&redlink=1).

**Microsoft SQL Server:**

As per Microsoft SQL server there are four groups of SQL Commands. ( MSSQL)

* Data Manipulation Language (DML)
* Data Definition Language (DDL)
* Data Control Language (DCL[[2]](https://en.wikipedia.org/wiki/Data_control_language#cite_note-2))
* Transaction Control Language (TCL)

DCL commands are used for access control and permission management for users in the database. With them we can easily allow or deny some actions for users on the tables or records (row level security).

DCL commands are:

GRANT - We can give certain permissions on the table (and other objects) for certain users of database,

DENY - bans certain permissions from users.

REVOKE - with this command we can take back permission from users.

For Example GRANT can be used to give privileges to user to do SELECT, INSERT, UPDATE and DELETE on a specific table or more than one tables.

With REVOKE command we can take back privilege to default or revoking specific command like update or delete based on requirements.

**Example:**

Grant SELECT,INSERT,UPDATE,DELETE on Employee To User1

Revoke INSERT On Employee To user1

Deny Update On Employee to user1

GRANT in first case we gave privileges to user User1 to do SELECT, INSERT, UPDATE and DELETE on the table called employees.

REVOKE with this command we can take back privilege to default one, in this case, we take back command INSERT on the table employees for user User1.

DENY is a specific command. We can conclude that every user has a list of privilege which is denied or granted so command DENY is there to explicitly ban you some privileges on the database objects.:

[**ORACLE Database**](https://en.wikipedia.org/wiki/Oracle_Database) :

[Oracle](https://www.oracle.com/in/database/) divide SQL commands to different types. They are.

* Data Definition Language (DDL) Statements
* Data Manipulation Language (DML) Statements
* Transaction Control Statements
* Session Control Statements
* System Control Statement
* Embedded SQL Statements

Data definition language (DDL) statements let you to perform these tasks:

* Create, alter, and drop schema objects
* Grant and revoke privileges and roles
* Analyze information on a table, index, or cluster
* Establish auditing options
* Add comments to the data dictionary

So Oracle database DDL commands include the **Grant** and **revoke** privileges which is actually part of Data control Language in Microsoft SQL server.

Syntax for grant and revoke in Oracle:

**Example**:

GRANT SELECT, INSERT, UPDATE, DELETE ON db1.Employee TO user1;

REVOKE SELECT, INSERT, UPDATE, DELETE ON db1.Employee FROM user1;

**Transaction Control Statements in Oracle:**

Transaction control statements manage changes made by DML statements. The transaction control statements are:

* COMMIT
* ROLLBACK
* SAVEPOINT
* SET TRANSACTION
* SET CONSTRAINT

[**MySQL server**](https://en.wikipedia.org/wiki/MySQL_server)**:**

MySQL server they divide SQL statements into different type of statement

* Data Definition Statements
* Data Manipulation Statements
* Transactional and Locking Statements
* Replication Statements
* Prepared Statements
* Compound Statement Syntax
* Database Administration Statements
* Utility Statements

For details refer [MySQL Transactional statements](https://dev.mysql.com/doc/refman/8.0/en/account-management-statements.html)[[4]](https://en.wikipedia.org/wiki/Data_control_language#cite_note-4))

The grant, revoke syntax are as part of Database administration statementsàAccount Management System.

The GRANT statement enables system administrators to grant privileges and roles, which can be granted to user accounts and roles. These syntax restrictions apply:

* GRANT cannot mix granting both privileges and roles in the same statement. A given GRANT statement must grant either privileges or roles.
* The ON clause distinguishes whether the statement grants privileges or roles:
* With ON, the statement grants privileges
* Without ON, the statement grants roles.
* It is permitted to assign both privileges and roles to an account, but you must use separate GRANT statements, each with syntax appropriate to what is to be granted.

The REVOKE statement enables system administrators to revoke privileges and roles, which can be revoked from user accounts and roles.

**Examples**:

REVOKE INSERT ON \*.\* FROM 'jeffrey'@'localhost';

REVOKE 'role1', 'role2' FROM 'user1'@'localhost', 'user2'@'localhost';

REVOKE SELECT ON world.\* FROM 'role3';

GRANT ALL ON db1.\* TO 'jeffrey'@'localhost';

GRANT 'role1', 'role2' TO 'user1'@'localhost', 'user2'@'localhost';

GRANT SELECT ON world.\* TO 'role3';

In [PostgreSQL](https://en.wikipedia.org/wiki/PostgreSQL), executing DCL is [transactional](https://en.wikipedia.org/wiki/Database_transaction), and can be rolled back.

**Grant** and **Revoke** are the SQL commands are used to control the privileges given to the users in a Databases

[SQLite](https://en.wikipedia.org/wiki/SQLite) does not have any DCL commands as it does not have usernames or logins. Instead, [SQLite](https://en.wikipedia.org/wiki/SQLite) depends on [file system permissions](https://en.wikipedia.org/wiki/File_system_permissions) to define who can open and access a database.[[5]](https://en.wikipedia.org/wiki/Data_control_language#cite_note-5)

## MySQL DCL

### SQL GRANT Command

SQL GRANT is a command used to provide access or privileges on the database objects to the users.

GRANT privilege\_name ON object\_name

TO {user\_name |PUBLIC |role\_name} [WITH GRANT OPTION];

* ***privilege\_name*** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
* ***object\_name*** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
* ***user\_name*** is the name of the user to whom an access right is being granted.
* ***user\_name*** is the name of the user to whom an access right is being granted.
* ***PUBLIC*** is used to grant access rights to all users.
* ***ROLES*** are a set of privileges grouped together.
* ***WITH GRANT OPTION*** - allows a user to grant access rights to other users.

**For Example:** GRANT SELECT ON employee TO user1; This command grants a SELECT permission on employee table to user1.You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

### SQL REVOKE Command:

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are GRANT and REVOKE. Only Database Administrator's or owner's of the database object can provide/remove privileges on a database object.

The REVOKE command removes user access rights or privileges to the database objects.

The Syntax for the REVOKE command is:

REVOKE privilege\_name

ON object\_name

FROM {user\_name |PUBLIC |role\_name}

**For Example:** REVOKE SELECT ON employee FROM user1;This command will REVOKE a SELECT privilege on employee table from user1.When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

## Privileges and Roles:

Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

**1) System privileges** - This allows the user to CREATE, ALTER, or DROP database objects.  
**2) Object privileges** - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

|  |  |
| --- | --- |
| **System Privileges** | **Description** |
| CREATE object | allows users to create the specified object in their own schema. |
| CREATE ANY object | allows users to create the specified object in any schema. |

**The above rules also apply for ALTER and DROP system privileges.**

Few of the object privileges are listed below:

|  |  |
| --- | --- |
| **Object Privileges** | **Description** |
| INSERT | allows users to insert rows into a table. |
| SELECT | allows users to select data from a database object. |
| UPDATE | allows user to update data in a table. |
| EXECUTE | allows user to execute a stored procedure or a function. |

**Roles:**Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.

Some of the privileges granted to the system roles are as given below:

|  |  |
| --- | --- |
| **System Role** | **Privileges Granted to the Role** |
| CONNECT | CREATE TABLE,  CREATE VIEW,  CREATE SYNONYM,  CREATE SEQUENCE,  CREATE SESSION etc. |
| RESOURCE | CREATE PROCEDURE,  CREATE SEQUENCE,  CREATE TABLE,  CREATE TRIGGER etc.  The primary usage of the RESOURCE role  is to restrict access to database objects. |
| DBA | ALL SYSTEM PRIVILEGES |

### Creating Roles:

**The Syntax to create a role is:**

CREATE ROLE role\_name [IDENTIFIED BY password];

**For Example:** To create a role called "developer" with password as "pwd",the code will be as follows

CREATE ROLE testing [IDENTIFIED BY pwd];

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.

We can GRANT or REVOKE privilege to a role as below.

**For example:** To grant CREATE TABLE privilege to a user by creating a testing role:

First, create a testing Role

CREATE ROLE testing

Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE.

GRANT CREATE TABLE TO testing;

Third, grant the role to a user.

GRANT testing TO user1;

To revoke a CREATE TABLE privilege from testing ROLE, you can write:

REVOKE CREATE TABLE FROM testing;

### ****Drop role****

**The Syntax to drop a role from the database is as below:**

DROP ROLE role\_name;

**For example:** To drop a role called developer, you can write:

DROP ROLE testing;

## SQL Server: Grant/Revoke Privileges

Learn how to **grant and revoke privileges** in SQL Server (Transact-SQL) with syntax and examples.

**Description**

You can GRANT and REVOKE privileges on various database objects in SQL Server. We'll look at how to grant and revoke privileges on tables in SQL Server.

### Grant Privileges on Table

You can grant users various privileges to tables. These permissions can be any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, or ALL.

The syntax for granting privileges on a table in SQL Server is:

GRANT privileges ON object TO user;

**Object -** The name of the database object that you are granting permissions for. In the case of granting privileges on a table, this would be the table name.

**User -** The name of the user that will be granted these privileges.

**privileges**

The privileges to assign. It can be any of the following values:

| Privilege | Description |
| --- | --- |
| SELECT | Ability to perform SELECT statements on the table. |
| INSERT | Ability to perform INSERT statements on the table. |
| UPDATE | Ability to perform UPDATE statements on the table. |
| DELETE | Ability to perform DELETE statements on the table. |
| REFERENCES | Ability to create a constraint that refers to the table. |
| ALTER | Ability to perform ALTER TABLE statements to change the table definition. |
| ALL | ALL does not grant all permissions for the table. Rather, it grants the ANSI-92 permissions which are SELECT, INSERT, UPDATE, DELETE, and REFERENCES. |

### Example

Let's look at some examples of how to grant privileges on tables in SQL Server.

For example, if you wanted to grant SELECT, INSERT, UPDATE, and DELETE privileges on a table called employees to a user name smithj, you would run the following GRANT statement:

GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO smithj;

You can also use the ALL keyword to indicate that you wish to grant the ANSI-92 permissions (ie: SELECT, INSERT, UPDATE, DELETE, and REFERENCES) to a user named smithj. For example:

GRANT ALL ON employees TO smithj;

If you wanted to grant only SELECT access on the employees table to all users, you could grant the privileges to the public role. For example:

GRANT SELECT ON employees TO public;

### Revoke Privileges on Table

Once you have granted privileges, you may need to revoke some or all of these privileges. To do this, you can run a revoke command. You can revoke any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, or ALL.

### Syntax

The syntax for revoking privileges on a table in SQL Server is:

REVOKE privileges ON object FROM user;

**Object -** The name of the database object that you are revoking privileges for. In the case of revoking privileges on a table, this would be the table name.

**User -** The name of the user that will have these privileges revoked.

**privileges**

It is the privileges to assign. It can be any of the following values:

| Privilege | Description |
| --- | --- |
| SELECT | Ability to perform SELECT statements on the table. |
| INSERT | Ability to perform INSERT statements on the table. |
| UPDATE | Ability to perform UPDATE statements on the table. |
| DELETE | Ability to perform DELETE statements on the table. |
| REFERENCES | Ability to create a constraint that refers to the table. |
| ALTER | Ability to perform ALTER TABLE statements to change the table definition. |
| ALL | ALL does not revoke all permissions for the table. Rather, it revokes the ANSI-92 permissions which are SELECT, INSERT, UPDATE, DELETE, and REFERENCES. |

### Example

Let's look at some examples of how to revoke privileges on tables in SQL Server.

For example, if you wanted to revoke DELETE privileges on a table called employees from a user named anderson, you would run the following REVOKE statement:

REVOKE DELETE ON employees FROM anderson;

If you wanted to revoke ALL ANSI-92 permissions (ie: SELECT, INSERT, UPDATE, DELETE, and REFERENCES) on a table for a user named anderson, you could use the ALL keyword as follows:

REVOKE ALL ON employees FROM anderson;

If you had granted SELECT privileges to the public role (ie: all users) on the employees table and you wanted to revoke these privileges, you could run the following REVOKE statement:

REVOKE SELECT ON employees FROM public;

It seems, that this is a question that regularly shows up in forums or stackoverflow.

To start, let’s highlight the fact that in MySQL 8.0 it’s **not** any more possible to create a user directly from the [GRANT](https://dev.mysql.com/doc/refman/8.0/en/grant.html) command (ERROR 1410 (42000): You are not allowed to create a user with GRANT).

This means that to grant some privileges, the user must be [created](https://dev.mysql.com/doc/refman/8.0/en/create-user.html) first.

Let’s create a user ‘user1‘ with ‘ChangeMe‘ as password that the user will have to change:

mysql> create user 'user1' identified by 'ChangeMe' password expire;

Query OK, 0 rows affected (1.35 sec)

Let’s try to connect to MySQL using that new created user:

$ mysql -u user1 -pChangeMe -h localhost

mysql: [Warning] Using a password on the command line interface can be insecure.

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 347

Server version: 8.0.13

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

Nothing special, we are connected as expected… but didn’t I explicitly expired the password ?

Yes I did, let’s try any statement:

mysql> select now();

ERROR 1820 (HY000): You must reset your password using ALTER USER

statement before executing this statement.

We must change the password as expected.

Let’s change it to ‘MySQL8isGreat‘:

mysql> set password='MySQL8isGreat';

Query OK, 0 rows affected (0.34 sec)

And now we can use MySQL and run any statement we are allowed to do (that we have the privileges for).

mysql> select now();

|  |
| --- |
| now() |
| 2020-01-10 14:36:05 |

mysql> show databases;

|  |
| --- |
| database |
| information\_schema |

It seems I don’t have access to many databases…

The default privilege is very limited:

mysql> show grants;

Grants for user1@%

GRANT USAGE ON . TO user1@%

row in set (0.00 sec)

It’s now time to grant more privileges to our user… but which privileges are available ?

there are currently **46 privileges**!

To list them all, just run:

mysql> show privileges;

|  |  |  |
| --- | --- | --- |
| Privilege | Context | Comment |
| Alter | Tables | To alter the table |
| Alter routine | Functions,Procedures | To alter or drop stored functions/procedures |
| Create | Databases,Tables,  Indexes | To create new databases and tables |
| Create routine | Databases | To use CREATE FUNCTION/PROCEDURE |
| Create role | Server Admin | To create new roles |
| Create  temporary  tables | Databases | To use CREATE TEMPORARY TABLE |
| Create view | Tables | To create new views |
| Create user | Server Admin | To create new users |
| Delete | Tables | To delete existing rows |
| Drop | Databases,Tables | To drop databases, tables, and views |
| Drop role | Server Admin | To drop roles |
| Event | Server Admin | To create, alter, drop and execute events |
| Execute | Functions,Procedures | To execute stored routines |
| File | File access on server | To read and write files on the server |
| Grant option | Databases,Tables,Functions,Procedures | To give to other users those privileges you possess |
| Index | Tables | To create or drop indexes |
| Insert | Tables | To insert data into tables |
| Lock tables | Databases | To use LOCK TABLES (together with SELECT privilege) |
| Process | Server Admin | To view the plain text of currently executing queries |
| Proxy | Server Admin | To make proxy user possible |
| References | Databases,Tables | To have references on tables |
| Reload | Server Admin | To reload or refresh tables, logs and privileges |
| Replication client | Server Admin | To ask where the slave or master servers are |
| Replication slave | Server Admin | To read binary log events from the master |
| Select | Tables | To retrieve rows from table |
| Show databases | Server Admin | To see all databases with SHOW DATABASES |
| Show view | Tables | To see views with SHOW CREATE VIEW |
| Shutdown | Server Admin | To shut down the server |
| Super | Server Admin | To use KILL thread, SET GLOBAL, CHANGE MASTER etc |
| Trigger | Tables | To use triggers |
| Create tablespace | Server Admin | To create/alter/drop tablespaces |
| Update | Tables | To update existing rows |
| Usage | Server Admin | No privileges - allow connect only |
| XA\_RECOVER\_ADMIN | Server Admin |  |
| SET\_USER\_ID | Server Admin |  |
| ROLE\_ADMIN | Server Admin |  |
| RESOURCE\_GROUP\_USER | Server Admin |  |
| RESOURCE\_GROUP\_ADMIN | Server Admin |  |
| BINLOG\_ADMIN | Server Admin |  |
| SYSTEM\_VARIABLES\_ADMIN | Server Admin |  |
| GROUP\_REPLICATION\_ADMIN | Server Admin |  |
| CONNECTION\_ADMIN | Server Admin |  |
| REPLICATION\_SLAVE\_ADMIN | Server Admin |  |
| ENCRYPTION\_KEY\_ADMIN | Server Admin |  |
| BACKUP\_ADMIN | Server Admin |  |
| PERSIST\_RO\_ VARIABLES\_ ADMIN | Server Admin |  |

Let’s allow our user to create tables in the database users1 that we created for him and also allow him to perform the following actions:

Alter

Create

Delete

Drop

Index

Insert

Select

Update

Trigger

Alter routine

Create routine

Execute

Create temporary tables

mysql> grant alter,create,delete,drop,index,insert,select,update,trigger,

alter routine, create routine, execute, create temporary tables on user1.\* to 'user1';

Query OK, 0 rows affected (0.23 sec)

**NO NEED TO RUN FLUSH PRIVILEGES !**

And in the open session for user1, we can check the granted privileges:

mysql> show grants\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user1@%: GRANT USAGE ON . TO user1@%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user1@%: GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP,

INDEX, ALTER, CREATE TEMPORARY TABLES, EXECUTE, CREATE ROUTINE, ALTER

ROUTINE, TRIGGER ON user1.\* TO user1@%

2 rows in set (0.00 sec)

Now let’s imagine we want to have multiple users that will have access to the same database (mydatabase), instead of specifying all the grants for each users, let’s use a common [role](https://dev.mysql.com/doc/refman/8.0/en/roles.html) for all of them. We will call it ‘developer\_user‘:

mysql> create ROLE developer\_users;

mysql> grant alter,create,delete,drop, index,insert, select, update, trigger, alter routine,create routine, execute,

create temporary tables on mydatabase.\* to 'developer\_user';

Query OK, 0 rows affected (0.12 sec)

Let’s grant the role to user1:

mysql> grant 'developer\_user' to 'user1';

Query OK, 0 rows affected (0.16 sec)

Now back again in user1‘s session and let’s verify:

mysql> SELECT CURRENT\_ROLE();

+----------------+  
 | CURRENT\_ROLE() |

+----------------+  
 | NONE |

+----------------+  
 1 row in set (0.00 sec)

mysql> set role 'developer\_user';

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT CURRENT\_ROLE();

+----------------------+  
 | CURRENT\_ROLE() |

+----------------------+  
 | developer\_user@% |

+----------------------+  
1 row in set (0.00 sec)

mysql> show grants\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user1@%: GRANT USAGE ON . TO user1@%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user1@%: GRANT SELECT, INSERT, UPDATE, DELETE, CREATE,

DROP, INDEX, ALTER, CREATE TEMPORARY TABLES, EXECUTE,

CREATE ROUTINE, ALTER ROUTINE, TRIGGER ON mydatabase.\* TO user1@%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user1@%: GRANT SELECT, INSERT, UPDATE, DELETE, CREATE,

DROP, INDEX, ALTER, CREATE TEMPORARY TABLES, EXECUTE,

CREATE ROUTINE, ALTER ROUTINE, TRIGGER ON user1.\* TO user1@%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user1@%: GRANT developer\_user@% TO user1@%

4 rows in set (0.00 sec)

Now we would like that every time user1 logs into MySQL, his new role will be set:

mysql> set default role 'developer\_user' to 'user1';

Query OK, 0 rows affected (0.22 sec)

Let’s also create a user2 having the default role:

mysql> create user 'user2' identified by 'DontChangeMe' default role 'developer\_user';  
 Query OK, 0 rows affected (0.18 sec)

And we can immediately test it:

$ mysql -u user2 -pDontChangeMe -h localhost

mysql: [Warning] Using a password on the command line interface can be insecure.

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 352

Server version: 8.0.13 MySQL Community Server – GPL

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its

affiliates. Other names may be trademarks of their respective

owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show grants\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user2@%: GRANT USAGE ON . TO user2@%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user2@%: GRANT SELECT, INSERT, UPDATE, DELETE, CREATE,

DROP, INDEX, ALTER, CREATE TEMPORARY TABLES, EXECUTE,

CREATE ROUTINE, ALTER ROUTINE, TRIGGER ON mydatabase.\* TO user2@%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Grants for user2@%: GRANT developer\_user@% TO user2@%

3 rows in set (0.18 sec)

# SQL ін’єкція

SQL ін’єкція це метод ін’єкції коду, що може знищити вашу БД, є одним з найбільш використовуваним методом веб- хакінгу та засновується на розміщенні зловмисного коду у SQL операторах, використовуючи поле вводу даних веб-сторінки.

## SQL у веб сторінках

SQL ін’єкція зустрічається коли ви просите користувача внести будь-які дані, наприклад прізвище або userid, і замість імені або id, користувач надсилає вам SQL оператор, ціль якого невідома.

Розглянемо приклад,коли SELECT додає програмну змінну (txtUserId) до обраного рядку. Значення ця змінна отримує через поле діалогової форми за допомогою функції getRequestString:

txtUserId = getRequestString("UserId");

txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

Надалі покажемо потенційні загрози використання даних, отриманих від користувача і використаних у SQL операторах.

## SQL ін’єкція , що використовує завжди вірну умову 1=1

Розглянемо цей приклад знову. Спочатку, під час створення коду, вважалося, що його мета полягає у створенні SQL оператора для вибірки користувача, що має певний номер (user id).

Якщо ніщо не заважає дозволити користувачу внести некоректні дані, то він може внести щось подібне:

UserId: 

Тоді сформований програмою SQL оператор буде виглядати приблизно так:

SELECT \* FROM Users WHERE UserId = 105 OR 1=1;

Ця інструкція є валідною з точки зору синтаксису й буде після виконання повертати всі записи з таблиці "Users" тому, що **OR 1=1** завжди вірно.

Чи є наведений приклад загрозливим? А що як таблиця "Users" містить імена та паролі користувачів системи?

У цьому випадку хакер отримає доступ до імен та паролів всіх користувачів бази даних. Для цього треба просто внести у поле діалогової форми форми рядок: 105 OR 1=1.

## Ін’єкція, що базується на завжди вірній умові ""=""

Розглянемо приклад реєстрації користувача на веб- сайті, що має такі поля:

Username:  


Password:  


Дані, внесені користувачем надходять у програму таким чином:

uName = getRequestString("username");

uPass = getRequestString("userpassword");

і дозволяють створити такий оператор

sql = 'SELECT \* FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'

Наприклад, після внесення даних, ми матимемо такий оператор

SELECT \* FROM Users WHERE Name ="John Doe" AND Pass ="myPass"

Хакер зможе отримати доступ до БД завдяки доступу до імен та паролів користувачів. Для цього він має внести рядок " OR ""=" в поля даних форми:

User Name: 

Password: 

У цьому випадку сервер створить зовні коректний оператор SQL:SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

Цей оператор буде повертати всі рядки з таблиці "Users", тому , що який створюється сервером завжди вірний: **""=""**.

Цей оператор утворюється так:

SELECT \* FROM Users WHERE Name =" " AND Pass =" " – у подвійні лапки мають додаватися рядки, які вніс користувач

А ці рядки ми додаємо " OR ""=" " OR ""="

як ім’я та пароль

результат:

SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

## SQL ін’єкція, що базується на кількох операторах SQL

Більшість сучасних СУБД дозволяє виконувати послідовність кількох операторів, тобто групу SQL операторів, що складається з двох або більше SQL операторів, розділених крапкою з комою (; semicolons).

Наприклад, цей оператор поверне всі рядки за таблиці "Users", а потім знищить таблицю "Suppliers":

SELECT \* FROM Users; DROP TABLE Suppliers

Розглянемо приклад:

txtUserId = getRequestString("UserId");

txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

І форму, що дозволяє користувачу внести дані:

User id: 

У цьому випадку створиться такий коректний SQL оператор:

SELECT \* FROM Users WHERE UserId = 105; DROP TABLE Suppliers;

## Використання SQL параметрів для захисту БД

SQL параметри – це значення, що контрольовано додаються до SQL запиту під час виконання.

### Приклади захисту у середовищі ASP.NET

txtUserId = getRequestString("UserId");

txtSQL = "SELECT \* FROM Users WHERE UserId = @0";

db.Execute(txtSQL,txtUserId);

Параметр позначено у запиті маркером @.

SQL інтерпретатор перевіряє кожен параметрів для гарантування його коректності відповідно до обмежень стовпчика таблиці та перевірки того, що цей параметр не є часткою SQL, яка буде виконуватися.

txtNam = getRequestString("CustomerName");

txtAdd = getRequestString("Address");

txtCit = getRequestString("City");

txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";

db.Execute(txtSQL,txtNam,txtAdd,txtCit);

Ці приклади показують як побудувати параметризований запит узагальною web- мовою програмування.

SELECT оператор у ASP.NET:

txtUserId = getRequestString("UserId");

sql = "SELECT \* FROM Customers WHERE CustomerId = @0";

command = new SqlCommand(sql);

command.Parameters.AddWithValue("@0",txtUserId);

command.ExecuteReader();

INSERT INTO STATEMENT IN ASP.NET:

txtNam = getRequestString("CustomerName");

txtAdd = getRequestString("Address");

txtCit = getRequestString("City");

txtSQL = "INSERT INTO Customers (CustomerName,Address,City) Values(@0,@1,@2)";

command = new SqlCommand(txtSQL);

command.Parameters.AddWithValue("@0",txtNam);  
command.Parameters.AddWithValue("@1",txtAdd);  
command.Parameters.AddWithValue("@2",txtCit);  
command.ExecuteNonQuery();

INSERT INTO STATEMENT IN PHP:

$stmt = $dbh->prepare("INSERT INTO Customers (CustomerName,Address,City)

VALUES (:nam, :add, :cit)");

$stmt->bindParam(':nam', $txtNam);

$stmt->bindParam(':add', $txtAdd);

$stmt->bindParam(':cit', $txtCit);

$stmt->execute();

# Ключові слова SQL

|  |  |
| --- | --- |
| Keyword | Description |
| [ADD](https://www.w3schools.com/sql/sql_ref_add.asp) | Adds a column in an existing table |
| [ADD CONSTRAINT](https://www.w3schools.com/sql/sql_ref_add_constraint.asp) | Adds a constraint after a table is already created |
| [ALTER](https://www.w3schools.com/sql/sql_ref_alter.asp) | Adds, deletes, or modifies columns in a table, or changes the data type of a column in a table |
| [ALTER COLUMN](https://www.w3schools.com/sql/sql_ref_alter_column.asp) | Changes the data type of a column in a table |
| [ALTER TABLE](https://www.w3schools.com/sql/sql_ref_alter_table.asp) | Adds, deletes, or modifies columns in a table |
| [ALL](https://www.w3schools.com/sql/sql_ref_all.asp) | Returns true if all of the subquery values meet the condition |
| [AND](https://www.w3schools.com/sql/sql_ref_and.asp) | Only includes rows where both conditions is true |
| [ANY](https://www.w3schools.com/sql/sql_ref_any.asp) | Returns true if any of the subquery values meet the condition |
| [AS](https://www.w3schools.com/sql/sql_ref_as.asp) | Renames a column or table with an alias |
| [ASC](https://www.w3schools.com/sql/sql_ref_asc.asp) | Sorts the result set in ascending order |
| [BACKUP DATABASE](https://www.w3schools.com/sql/sql_ref_backup_database.asp) | Creates a back up of an existing database |
| [BETWEEN](https://www.w3schools.com/sql/sql_ref_between.asp) | Selects values within a given range |
| [CASE](https://www.w3schools.com/sql/sql_ref_case.asp) | Creates different outputs based on conditions |
| [CHECK](https://www.w3schools.com/sql/sql_ref_check.asp) | A constraint that limits the value that can be placed in a column |
| [COLUMN](https://www.w3schools.com/sql/sql_ref_column.asp) | Changes the data type of a column or deletes a column in a table |
| [CONSTRAINT](https://www.w3schools.com/sql/sql_ref_constraint.asp) | Adds or deletes a constraint |
| [CREATE](https://www.w3schools.com/sql/sql_ref_create.asp) | Creates a database, index, view, table, or procedure |
| [CREATE DATABASE](https://www.w3schools.com/sql/sql_ref_create_database.asp) | Creates a new SQL database |
| [CREATE INDEX](https://www.w3schools.com/sql/sql_ref_create_index.asp) | Creates an index on a table (allows duplicate values) |
| [CREATE OR REPLACE VIEW](https://www.w3schools.com/sql/sql_ref_create_or_replace_view.asp) | Updates a view |
| [CREATE TABLE](https://www.w3schools.com/sql/sql_ref_create_table.asp) | Creates a new table in the database |
| [CREATE PROCEDURE](https://www.w3schools.com/sql/sql_ref_create_procedure.asp) | Creates a stored procedure |
| [CREATE UNIQUE INDEX](https://www.w3schools.com/sql/sql_ref_create_unique_index.asp) | Creates a unique index on a table (no duplicate values) |
| [CREATE VIEW](https://www.w3schools.com/sql/sql_ref_create_view.asp) | Creates a view based on the result set of a SELECT statement |
| [DATABASE](https://www.w3schools.com/sql/sql_ref_database.asp) | Creates or deletes an SQL database |
| [DEFAULT](https://www.w3schools.com/sql/sql_ref_default.asp) | A constraint that provides a default value for a column |
| [DELETE](https://www.w3schools.com/sql/sql_ref_delete.asp) | Deletes rows from a table |
| [DESC](https://www.w3schools.com/sql/sql_ref_desc.asp) | Sorts the result set in descending order |
| [DISTINCT](https://www.w3schools.com/sql/sql_ref_select_distinct.asp) | Selects only distinct (different) values |
| [DROP](https://www.w3schools.com/sql/sql_ref_drop.asp) | Deletes a column, constraint, database, index, table, or view |
| [DROP COLUMN](https://www.w3schools.com/sql/sql_ref_drop_column.asp) | Deletes a column in a table |
| [DROP CONSTRAINT](https://www.w3schools.com/sql/sql_ref_drop_constraint.asp) | Deletes a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK constraint |
| [DROP DATABASE](https://www.w3schools.com/sql/sql_ref_drop_database.asp) | Deletes an existing SQL database |
| [DROP DEFAULT](https://www.w3schools.com/sql/sql_ref_drop_default.asp) | Deletes a DEFAULT constraint |
| [DROP INDEX](https://www.w3schools.com/sql/sql_ref_drop_index.asp) | Deletes an index in a table |
| [DROP TABLE](https://www.w3schools.com/sql/sql_ref_drop_table.asp) | Deletes an existing table in the database |
| [DROP VIEW](https://www.w3schools.com/sql/sql_ref_drop_view.asp) | Deletes a view |
| [EXEC](https://www.w3schools.com/sql/sql_ref_exec.asp) | Executes a stored procedure |
| [EXISTS](https://www.w3schools.com/sql/sql_ref_exists.asp) | Tests for the existence of any record in a subquery |
| [FOREIGN KEY](https://www.w3schools.com/sql/sql_ref_foreign_key.asp) | A constraint that is a key used to link two tables together |
| [FROM](https://www.w3schools.com/sql/sql_ref_from.asp) | Specifies which table to select or delete data from |
| [FULL OUTER JOIN](https://www.w3schools.com/sql/sql_ref_full_outer_join.asp) | Returns all rows when there is a match in either left table or right table |
| [GROUP BY](https://www.w3schools.com/sql/sql_ref_group_by.asp) | Groups the result set (used with aggregate functions: COUNT, MAX, MIN, SUM, AVG) |
| [HAVING](https://www.w3schools.com/sql/sql_ref_having.asp) | Used instead of WHERE with aggregate functions |
| [IN](https://www.w3schools.com/sql/sql_ref_in.asp) | Allows you to specify multiple values in a WHERE clause |
| [INDEX](https://www.w3schools.com/sql/sql_ref_index.asp) | Creates or deletes an index in a table |
| [INNER JOIN](https://www.w3schools.com/sql/sql_ref_inner_join.asp) | Returns rows that have matching values in both tables |
| [INSERT INTO](https://www.w3schools.com/sql/sql_ref_insert_into.asp) | Inserts new rows in a table |
| [INSERT INTO SELECT](https://www.w3schools.com/sql/sql_ref_insert_into_select.asp) | Copies data from one table into another table |
| [IS NULL](https://www.w3schools.com/sql/sql_ref_is_null.asp) | Tests for empty values |
| [IS NOT NULL](https://www.w3schools.com/sql/sql_ref_is_not_null.asp) | Tests for non-empty values |
| [JOIN](https://www.w3schools.com/sql/sql_ref_join.asp) | Joins tables |
| [LEFT JOIN](https://www.w3schools.com/sql/sql_ref_left_join.asp) | Returns all rows from the left table, and the matching rows from the right table |
| [LIKE](https://www.w3schools.com/sql/sql_ref_like.asp) | Searches for a specified pattern in a column |
| [LIMIT](https://www.w3schools.com/sql/sql_ref_top.asp) | Specifies the number of records to return in the result set |
| [NOT](https://www.w3schools.com/sql/sql_ref_not.asp) | Only includes rows where a condition is not true |
| [NOT NULL](https://www.w3schools.com/sql/sql_ref_not_null.asp) | A constraint that enforces a column to not accept NULL values |
| [OR](https://www.w3schools.com/sql/sql_ref_or.asp) | Includes rows where either condition is true |
| [ORDER BY](https://www.w3schools.com/sql/sql_ref_order_by.asp) | Sorts the result set in ascending or descending order |
| [OUTER JOIN](https://www.w3schools.com/sql/sql_ref_full_outer_join.asp) | Returns all rows when there is a match in either left table or right table |
| [PRIMARY KEY](https://www.w3schools.com/sql/sql_ref_primary_key.asp) | A constraint that uniquely identifies each record in a database table |
| [PROCEDURE](https://www.w3schools.com/sql/sql_ref_create_procedure.asp) | A stored procedure |
| [RIGHT JOIN](https://www.w3schools.com/sql/sql_ref_right_join.asp) | Returns all rows from the right table, and the matching rows from the left table |
| [ROWNUM](https://www.w3schools.com/sql/sql_ref_top.asp) | Specifies the number of records to return in the result set |
| [SELECT](https://www.w3schools.com/sql/sql_ref_select.asp) | Selects data from a database |
| [SELECT DISTINCT](https://www.w3schools.com/sql/sql_ref_select_distinct.asp) | Selects only distinct (different) values |
| [SELECT INTO](https://www.w3schools.com/sql/sql_ref_select_into.asp) | Copies data from one table into a new table |
| [SELECT TOP](https://www.w3schools.com/sql/sql_ref_top.asp) | Specifies the number of records to return in the result set |
| [SET](https://www.w3schools.com/sql/sql_ref_set.asp) | Specifies which columns and values that should be updated in a table |
| [TABLE](https://www.w3schools.com/sql/sql_ref_table.asp) | Creates a table, or adds, deletes, or modifies columns in a table, or deletes a table or data inside a table |
| [TOP](https://www.w3schools.com/sql/sql_ref_top.asp) | Specifies the number of records to return in the result set |
| [TRUNCATE TABLE](https://www.w3schools.com/sql/sql_ref_drop_table.asp) | Deletes the data inside a table, but not the table itself |
| [UNION](https://www.w3schools.com/sql/sql_ref_union.asp) | Combines the result set of two or more SELECT statements (only distinct values) |
| [UNION ALL](https://www.w3schools.com/sql/sql_ref_union.asp) | Combines the result set of two or more SELECT statements (allows duplicate values) |
| [UNIQUE](https://www.w3schools.com/sql/sql_ref_unique.asp) | A constraint that ensures that all values in a column are unique |
| [UPDATE](https://www.w3schools.com/sql/sql_ref_update.asp) | Updates existing rows in a table |
| [VALUES](https://www.w3schools.com/sql/sql_ref_values.asp) | Specifies the values of an INSERT INTO statement |
| [VIEW](https://www.w3schools.com/sql/sql_ref_view.asp) | Creates, updates, or deletes a view |
| [WHERE](https://www.w3schools.com/sql/sql_ref_where.asp) | Filters a result set to include only records that fulfill a specified condition |

# Синтаксис операторів SQL

|  |  |
| --- | --- |
| SQL Statement | Syntax |
| AND / OR | SELECT column\_name(s) FROM table\_name WHERE condition AND|OR condition |
| ALTER TABLE | ALTER TABLE table\_name ADD column\_name datatype  or  ALTER TABLE table\_name DROP COLUMN column\_name |
| AS (alias) | SELECT column\_name AS column\_alias FROM table\_name  or  SELECT column\_name FROM table\_name  AS table\_alias |
| BETWEEN | SELECT column\_name(s) FROM table\_name WHERE column\_name BETWEEN value1 AND value2 |
| CREATE DATABASE | CREATE DATABASE database\_name |
| CREATE TABLE | CREATE TABLE table\_name ( column\_name1 data\_type, column\_name2 data\_type, column\_name3 data\_type, ... ) |
| CREATE INDEX | CREATE INDEX index\_name ON table\_name (column\_name)  or  CREATE UNIQUE INDEX index\_name ON table\_name (column\_name) |
| CREATE VIEW | CREATE VIEW view\_name AS SELECT column\_name(s) FROM table\_name WHERE condition |
| DELETE | DELETE FROM table\_name WHERE some\_column=some\_value  or  DELETE FROM table\_name (**Note:**Deletes the entire table!!)  DELETE \* FROM table\_name (**Note:**Deletes the entire table!!) |
| DROP DATABASE | DROP DATABASE database\_name |
| DROP INDEX | DROP INDEX table\_name.index\_name (SQL Server) DROP INDEX index\_name ON table\_name (MS Access) DROP INDEX index\_name (DB2/Oracle) ALTER TABLE table\_name DROP INDEX index\_name (MySQL) |
| DROP TABLE | DROP TABLE table\_name |
| EXISTS | IF EXISTS (SELECT \* FROM table\_name WHERE id = ?) BEGIN --do what needs to be done if exists END ELSE BEGIN --do what needs to be done if not END |
| GROUP BY | SELECT column\_name, aggregate\_function(column\_name) FROM table\_name WHERE column\_name operator value GROUP BY column\_name |
| HAVING | SELECT column\_name, aggregate\_function(column\_name) FROM table\_name WHERE column\_name operator value GROUP BY column\_name HAVING aggregate\_function(column\_name) operator value |
| IN | SELECT column\_name(s) FROM table\_name WHERE column\_name IN (value1,value2,..) |
| INSERT INTO | INSERT INTO table\_name VALUES (value1, value2, value3,....)  *or*  INSERT INTO table\_name (column1, column2, column3,...) VALUES (value1, value2, value3,....) |
| INNER JOIN | SELECT column\_name(s) FROM table\_name1 INNER JOIN table\_name2 ON table\_name1.column\_name=table\_name2.column\_name |
| LEFT JOIN | SELECT column\_name(s) FROM table\_name1 LEFT JOIN table\_name2 ON table\_name1.column\_name=table\_name2.column\_name |
| RIGHT JOIN | SELECT column\_name(s) FROM table\_name1 RIGHT JOIN table\_name2 ON table\_name1.column\_name=table\_name2.column\_name |
| FULL JOIN | SELECT column\_name(s) FROM table\_name1 FULL JOIN table\_name2 ON table\_name1.column\_name=table\_name2.column\_name |
| LIKE | SELECT column\_name(s) FROM table\_name WHERE column\_name LIKE pattern |
| ORDER BY | SELECT column\_name(s) FROM table\_name ORDER BY column\_name [ASC|DESC] |
| SELECT | SELECT column\_name(s) FROM table\_name |
| SELECT \* | SELECT \* FROM table\_name |
| SELECT DISTINCT | SELECT DISTINCT column\_name(s) FROM table\_name |
| SELECT INTO | SELECT \* INTO new\_table\_name [IN externaldatabase] FROM old\_table\_name  *or*  SELECT column\_name(s) INTO new\_table\_name [IN externaldatabase] FROM old\_table\_name |
| SELECT TOP | SELECT TOP number|percent column\_name(s) FROM table\_name |
| TRUNCATE TABLE | TRUNCATE TABLE table\_name |
| UNION | SELECT column\_name(s) FROM table\_name1 UNION SELECT column\_name(s) FROM table\_name2 |
| UNION ALL | SELECT column\_name(s) FROM table\_name1 UNION ALL SELECT column\_name(s) FROM table\_name2 |
| UPDATE | UPDATE table\_name SET column1=value, column2=value,... WHERE some\_column=some\_value |
| WHERE | SELECT column\_name(s) FROM table\_name WHERE column\_name operator value |

## 

# Інші типи баз даних - DBMS Tutorials

Зараз Ви знаєте MySQL, Для вивчення інших DB спробуйте обрати

[SQL Tutorial](https://www.techonthenet.com/sql/index.php)

[Oracle Tutorial](https://www.techonthenet.com/oracle/index.php)

[SQL Server Tutorial](https://www.techonthenet.com/sql_server/index.php)

[MariaDB Tutorial](https://www.techonthenet.com/mariadb/index.php)

[PostgreSQL Tutorial](https://www.techonthenet.com/postgresql/index.php)

[SQLite Tutorial](https://www.techonthenet.com/sqlite/index.php)

<https://metanit.com/sql/mysql/>

https://www.w3resource.com/mysql/mysql-transaction.php

https://www.w3resource.com/mysql/mysql-spatial-data-types.php

https://www.sqlshack.com/learn-mysql-create-and-drop-temp-tables/

Бажаю успіхів!