ЗАКАРПАТСЬКИЙ УГОРСЬКИЙ ІНСТИТУТ ІМЕНІ ФЕРЕНЦА РАКОЦІ ІІ
II. RÁKÓCZI FERENC KÁRPÁTALJAI MAGYAR FŐISKOLA

Кафедра математики та інформатики
Matematika és Informatika Tanszék

ОСНОВИ МОВИ SQL/ SQL NYELV ALAPJAI

МЕТОДИЧНІ ВКАЗІВКИ ДО ПРАКТИЧНИХ ЗАНЯТЬ/ MÓDSZERTANI ÚTMUTATÓ GYAKORLATI FOGLALKOZÁSOKHOZ

Системи керування базами даних/ Adatbázis-kezelő rendszerek
(назва навчальної дисципліни/a tantárgy neve)
Перший (бакалаврський) / Alapképzés (BSc)
(рівень вищої освіти / felsőoktatás szintje)
01 Освіта/Педагогіка / 01 Oktatás/Pedagógia
(галузь знань / képzési ág)
Середня освіта (Інформатика) / Középiskolai oktatás (Informatika)
(освітня програма / képzési program)

Берегове / Beregszász 2024 р.

Методичні вказівки «Основи мови SQL» до практичних занять з дисципліни «Системи керування базами даних » розроблені на основі Освітньої програми підготовки бакалаврів з галузі знань «01 Освіта/Педагогіка» за напрямом «014 Середня освіта (Інформатика)» як для студентів денної, так і заочної форми навчання. Метою методичного посібника є поглиблення знань студентів з дисципліни « Системи керування базами даних». У роботі наведені основні команди мови SQL, а також їх структури, які виникають при використанні різних опціональних параметрів. Можливості мови SQL ілюструються простими прикладами. В методичці наводяться приклади використання команд SQL для роботи з таблицями баз даних. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу, а також студентами-математиками.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 1 від «13 » серпня 2024 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 24 від 26 серпня 2024 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від серпня 2024 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Олександр МІЦА – завідувач кафедри інформаційних управляючих систем та технологій УжНУ, доктор техніічих наук, професор (м. Ужгород)

Мирослав СТОЙКА – доцент кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, кандидат фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несе розробник.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua)

A „SQL nyelv alapjai” módszertani útmutató az „Adatbázis-kezelő rendszerek” tárgy gyakorlati foglalkozásaihoz a BSc „Középiskolai oktatás (Informatika)” képzési programja alapján lett kidolgozva nappali és levelező tagozatos hallgatók részére. A módszertani kézikönyv célja, hogy elmélyítse az „Adatbázis-kezelő rendszerek” tárgy t hallgatóinak ismereteit. Az útmutató bemutatja az SQL nyelv főbb parancsait, valamint azok struktúráit, amelyek különböző opcionális paraméterek használatakor keletkeznek. Az SQL nyelv lehetőségeit egyszerű példák illusztrálják. A kézikönyv példákat ad az SQL-parancsok használatára adatbázistáblákkal való munkavégzéshez. A módszertani utasítások a gyakorlati órákon és a tantárgy önálló munkája során használhatók, valamint a matematikus hallgatók is használhatják.

Az oktatási folyamatban történő felhasználását jóváhagyta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszéke
(2024. augusztus13, 1. számú jegyzőkönyv).

Megjelentetésre javasolta a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Oktatási és Módszertani Tanácsa
(2024. augusztus 26., 22. számú jegyzőkönyv).

Elektronikus formában (PDF fájlformátumban) történő kiadásra javasolta
a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Tudományos Tanácsa
(2024. augusztus 27., 1. számú jegyzőkönyv).

Kiadásra előkészítette a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola
Matematika és Informatika Tanszéke, valamint Kiadói Részlege.

A módszertani útmutató kidolgozója:

HOLOVÁCS József – professzor, műszaki tudományok doktora, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének professzora

Szakmai lektorok:

MITSA Oleksandr – Az UNE Információs vezérlési Rendszerek és Technológiák Tanszékének vezetője, műszaki tudományok doktora, professzor

SZTOJKA Miroszláv – a fizikai és matematikai tudományok kandidátusa, docens, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének docense

A kiadásért felelnek:

KUCSINKA Katalin – PhD, a fizikai és matematikai tudományok kandidátusa, a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Matematika és Informatika Tanszékének tanszékvezetője és docense

DOBOS Sándor – a II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola Kiadói Részlegének vezetője

A segédlet tartalmáért kizárólag a módszertani útmutató kidolgozója felel.

Kiadó: II. Rákóczi Ferenc Kárpátaljai Magyar Főiskola (cím: 90 202, Beregszász, Kossuth tér 6. E-mail: foiskola@kmf.uz.ua)

ЗМІСТ

Передмова

Важливі напрямки сучасних інформаційних технологій базуються на концепції баз даних. Дані в базах даних повинні бути структуровані та організовані так, щоб вони адекватно відображали предметні області реального світу. Для ефективного функціювання баз даних створені різні системи управління базами даних (СУБД). На даний час найбільш поширені реляційні СУБД, в яких використовується мова SQL.
Основними завданнями вивчення дисципліни „Системи управління базами даних", яка вивчається студентами спеціальності «Інформатика», є навчання студентів основам СУБД і застосуванню їх при проектуванні інформаційних систем різного призначення. У дисципліні «Системи управління базами даних» важливе місце займає вивчення мови SQL.

У пропонованому методичному посібнику описані основні команди мови SQL, наводяться приклади їх використання, та можливості розробки на їх основі компонент інформаційних систем. Посібник слід розглядати, як вступ до розробки інформаційних систем, які використовують мову SQL. Для більш поглибленого вивчення даної проблематики, рекомендується ознайомитись з наведеною в кінці посібника літературою. Наведені в посібнику приклади рекомендується виконати самостійно на комп'ютері з варіюванням вхідних даних та опціональних параметрів команд.

1. Реляційні бази даних та SQL

База даних (***БД***) — це організована колекція даних, до якої можна легко отримати доступ. Для управління базами даних використовують **системи управління базами даних** (***СУБД***).

Мова структурованих запитів (***SQL*** –. *"**S**tructured **Q**uery **L**anguage"*) — це стандартна мова запитів, яка використовується для роботи з реляційними базами даних. SQL використовується для створення та видалення баз даних, створення та видалення таблиць у базі даних, читання, вставки, оновлення та видалення даних з таблиць, а також у багатьох інших операцій з базами даних.

SQL використовується для "комунікації" між реляційними базами даних, а також в таких СУБД, як MySQL, PostgreSQL, Oracle, SQL Server та багато інших.

Є два поширені типи баз даних:

Нереляційні.

Реляційні.

Нереляційні СУБД

У нереляційних СУБД дані зберігаються в парах *ключ-значення.*

Приклад нереляційної БД

custumers ({"id": 1, "name": "Mark", "age" : "30"},

{ "id": 2, "name": "Ferenc", "age" : "25"})

Дані *customers* зберігаються в парах ключ-значення.

*Популярні нереляційні СУБД**:*** MongoDB, Amazon DynamoDB, Redis та інші.

Реляційні СУБД

У реляційних СУБД дані зберігаються у **табличному форматі**. Наприклад:

Таблиця customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

customers (покупці)— це таблиця бази даних. Перший рядок — це *атрибути* таблиці. Кожен наступний рядок містить фактичні *дані* про покупців. У реляційних СУБД дві чи більше таблиць можуть бути пов'язані між собою. Звідси і термін "реляційні" (*"relation" = "зв'язок"*). Наприклад:

Таблиця orders

order_id	product	total	customer_id
1	Paper	500	5
2	Pen	10	2
3	Marker	120	3
4	Books	1000	1
5	Erasers	20	4

Таблиця customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Doe	806-749-2958	UAE

Приклад зв'язків в реляційних СУБД

Таблиця *orders* (замовлення, покупки). Таблиці *customers* та *orders* пов'язані через `customer_id`.

Для доступу до даних реляційних баз даних використовується мова SQL.

Популярні реляційні СУБД: MySQL, PostgreSQL, MSSQL, Oracle та інші.

Мова структурованих запитів SQL

SQL використовується для:

- створення баз даних;
- створення таблиць в базі даних;
- читання даних з таблиць;
- вставки даних в таблиці;
- оновлення даних в таблиці;
- видалення даних з таблиці;
- видалення таблиць бази даних;
- видалення баз даних;

та інших операцій із базами даних.

SQL використовується у всіх реляційних базах даних, таких як MySQL, Oracle, MSSQL, PostgreSQL та інші.

Примітка: Основні SQL-команди однакові у всіх реляційних базах даних. Однак для деяких команд можуть бути відмінності.

на Зміст

2. Робота з базами даних в SQL

Команда CREATE DATABASE

Перш ніж ми зможемо працювати з таблицями бази даних, спочатку потрібно створити базу даних.

Для створення бази даних використовується команда **CREATE DATABASE**. Наприклад:

```
CREATE DATABASE my_db;
```

Ми створюємо базу даних з ім'ям *my_db*.

Команда CREATE DATABASE IF NOT EXISTS

CREATE DATABASE IF NOT EXISTS використовується для створення бази даних, якщо такої (з вказаним ім'ям) не існує на момент виконання команди. Наприклад:

```
CREATE DATABASE IF NOT EXISTS my_db;
```

Тут ми створюємо базу даних з ім'ям *my_db*, якщо такої немає на момент виконання команди.

Вибір активної бази даних (команда USE)

Іноді доводиться працювати з кількома базами даних. Щоб переключатися між доступними базами даних, ми можемо використати наступну команду:

```
USE my_db;
```

Тут ми вибираємо базу даних *my_db* і всі наступні SQL-операції будуть виконуватися всередині цієї бази даних.

Команда DROP DATABASE

DROP DATABASE використовується для видалення бази даних у системі управління базами даних (СУБД). Наприклад:

```
DROP DATABASE  my_db;
```

Тут ми видаляємо базу даних з ім'ям *my_db*. При видаленні бази даних всі таблиці та записи у базі даних також видаляються.

Ця команда виконується лише тоді, якщо у вас є **права адміністратора** або права **DROP**.

Отримання списку всіх баз даних

Щоб перевірити, чи видалена база даних, ми можемо запустити наступну команду, щоб вивести список доступних баз даних:

SHOW DATABASES;
Виводить список усіх доступних баз даних у СУБД.

Резервне копіювання

Команда BACKUP DATABASE

Важливо регулярно створювати резервні копії бази даних, щоб дані не були втрачені у разі пошкодження бази даних (БД). В SQL ми можемо створювати резервні копії БД за допомогою **оператора BACKUP DATABASE**. Наприклад:

```
BACKUP DATABASE orders TO DISK = 'C:\orders_backup.bak';
```

Тут ми створюємо файл резервної копії бази даних *orders* на диску *C* з ім'ям *orders_backup.bak*.

Примітка: Поширено використання розширення ***.bak*** для файлів резервних копій БД, однак це не є обов'язковим.

Резервне копіювання тільки нових змін

У SQL ми можемо зробити резервну копію лише нових змін у порівнянні з попередньою резервною копією, використовуючи **команду WITH DIFFERENTIAL**. Наприклад:

```
BACKUP DATABASE orders TO DISK = 'C:\orders_backup.bak'
          WITH DIFFERENTIAL;
```

Тут ми додаємо лише нові зміни до попереднього файлу резервної копії. Ця команда працює швидше, ніж створення резервної копії БД з нуля.

Відновлення бази даних із резервної копії

Для відновлення файлу резервної копії у системі управління базою даних (СУБД) використовується команда **RESTORE DATABASE**. Наприклад:

```
RESTORE DATABASE orders FROM DISK = 'C:\orders_backup.bak';
```

Тут ми відновлюємо файл резервної копії *orders_backup.bak* у базі даних *orders*.

На Зміст

3. Таблиці бази даних

3.1. Створення таблиці

Таблиця використовується для зберігання записів (даних). Для створення таблиці у базі даних використовується команда **CREATE TABLE**. Наприклад:

```
CREATE TABLE Companies (
  id int,
  name varchar(50),
  address text,
  email varchar(50),
  phone varchar(10)
);
```

Тут ми створюємо таблицю під назвою *Companies*. Таблиця містить стовпці (поля) *id, name, address, email* та *phone.*

Примітка: Ми повинні явно вказувати типи даних кожного стовпця під час створення таблиці.

CREATE TABLE IF NOT EXISTS

Команда **CREATE TABLE IF NOT EXISTS** використовується для створення таблиці, якщо такої (із вказаним ім'ям) немає на момент виконання команди. Наприклад:

```
CREATE TABLE IF NOT EXISTS Companies (
  id int,
  name varchar(50),
  address text,
  email varchar(50),
  phone varchar(10)
);
```

Тут ми створюємо таблицю з ім'ям *Companies*, якщо такої немає на момент виконання команди.

Типи даних

У SQL кожен стовпець (в таблиці) має власний **тип даних**, який обмежує те, що може зберігатися у стовпці. Наприклад, якщо тип даних стовпця — `INT`, то в цьому стовпці можуть зберігатися лише цілі числа, такі як `0`, `1`, `-1` тощо.

Різні СУБД підтримують різні типи даних.

Цілочисельні типи даних

Тип даних	Опис
BIT(n)	Може містити ***n***-бітові значення (***n*** може варіюватися від 1 до 64).
TINYINT	Може містити числа в діапазоні від -128 до 127.
SMALLINT	Може містити числа в діапазоні від -32,768 до 32,767.
MEDIUMINT	Може містити числа в діапазоні від -8,388,608 до 8,388,607.
INT	Може містити числа в діапазоні від -2,147,483,648 до 2,147,483,647.
BIGINT	Може містити числа в діапазоні від -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807.

Рядкові типи даних

Тип даних	Опис
CHAR(n)	Може містити символи фіксованої довжини (***n*** – це фіксована довжина, максимум 8000 символів).
VARCHAR(n)	Може містити символи **n** довжини (***n*** – це змінна довжини, максимум 8000 символів).
BINARY(n)	Може містити двійкові рядки фіксованої довжини (***n*** – фіксована довжина).
VARBINARY(n)	Може містити двійкові рядки заданої довжини (***n*** – це змінна довжина).
TINYTEXT	Може містити до 255 символів.
TEXT(n)	Може містити символи розміром до 65,535 байт.
MEDIUMTEXT	Може містити до 16,777,215 символів.
LONGTEXT	Може містити до 4,294,967,295 символів.
BLOB(n)	Може містити великі двійкові об’єкти розміром до 65,535 байт.
MEDIUMBLOB	Може містити великі двійкові об’єкти розміром до 16,777,215 байт.
LONGBLOB	Може містити великі двійкові об’єкти розміром до 4,294,967,295 байт.

Типи даних Дати та Часу

Тип даних	Опис
DATE	Може містити дату в форматі РРРР-ММ-ДД і в діапазоні від 1000-01-01 до 9999-12-31.
DATETIME	Може містити дату та час в форматі РРРР-ММ-ДД гг:хх:сс.
TIME	Може містити лише час в форматі гг:хх:сс та в діапазоні від -838:59:59 до 838:59:59.
YEAR	Може містити рік в 4-значному форматі та в діапазоні від 1901 до 2155.
TIMESTAMP	Може містити позначку часу від поточного часового поясу до UTC.

Примітка: MySQL підтримує багато інших типів даних.

Створення таблиці для зберігання дати та часу

При створенні таблиці слід вказувати тип даних для стовпця з датою. Наприклад:

```
-- Створюємо таблицю з безліччю стовпців з датою
CREATE TABLE Users (
  id INT,
  username VARCHAR(50),
  full_name VARCHAR(50),
  date_of_birth DATE,
  last_login DATETIME,
  registered_at TIMESTAMP
);
-- Вставляємо значення в таблицю Users
INSERT INTO Users
VALUES
 (1, 'janos', Nagy János', '1999-04-14', '2022-04-22
10:34:53.44', '2020-03-15 07:31:42.23');
```

CREATE TABLE AS

Ми також можемо створити таблицю, використовуючи записи з будь-якої іншої таблиці, використовуючи команду **CREATE TABLE AS**. Наприклад:

```
CREATE TABLE JapanCustomers
AS (
  SELECT *
  FROM Customers
  WHERE country = 'Japan'
```

```
);
```

Тут ми створюємо таблицю з ім'ям *JapanCustomers* і копіюємо в неї записи з вкладеного запиту.

Створення таблиці з первинним ключем (Primary Key)

Для створення таблиці з первинним ключем використовується наступна команда:

```
CREATE TABLE Companies (
  id int,
  name varchar(50),
  address text,
  email varchar(50),
  phone varchar(10),
  PRIMARY KEY (id)
);
```

Як вказати обмеження під час створення таблиці?

Ми також можемо додавати різні типи обмежень під час створення таблиці. Наприклад:

```
CREATE TABLE Companies (
  id int NOT NULL,
  name varchar(50) NOT NULL,
  address text,
  email varchar(50) NOT NULL,
  phone varchar(10)
);
```

Тут обмеження `NOT NULL` додається до стовпців *id*, *name* та *email*. Це означає, що ці стовпці не можуть бути порожніми (`NULL`).

Примітка: Інколи обмеження залежать від системи управління базами даних (СУБД), тобто ключові слова можуть відрізнятися у різних СУБД.

Команда DROP TABLE

DROP TABLE використовується для видалення таблиць у базі даних. Наприклад:

```
DROP TABLE my_table;
```

Тут ми видаляємо таблицю з ім'ям *my_table*.

Також переконайтеся, що у вас є **права адміністратора** або (права) **DROP** для запуску цієї команди.

Примітка: При видаленні таблиці всі записи таблиці також видаляються.

Команда DROP TABLE IF EXISTS

При спробі видалити неіснуючу таблицю SQL видасть помилку. Щоб вирішити цю проблему, можна додати частину **IF EXISTS** при видаленні таблиці. Наприклад:

```
DROP TABLE IF EXISTS my_table;
```

Тут ми видалимо таблицю лише в тому випадку, якщо вона існує з таким самим ім'ям.

3.2. Зміна структури таблиці

Можна змінити структуру таблиці за допомогою **команди ALTER TABLE**. Ми можемо:

Додати стовпець.

Перейменувати стовпець.

Змінити стовпець.

Видалити стовпець.

Перейменувати таблицю.

Додавання стовпця до таблиці

Ми можемо додати стовпці до таблиці за допомогою команди **ALTER TABLE з оператором ADD**. Наприклад:

```
ALTER TABLE Customers ADD phone varchar(10);
```

Тут ми додали стовпець з ім'ям *phone* до таблиці *Customers*.

Додавання декількох стовпців до таблиці

Ми також можемо додати відразу декілька стовпців до таблиці. Наприклад:

```
ALTER TABLE Customers ADD phone varchar(10), age int;
```

Тут ми додали стовпці *phone* та *age* у таблицю *Customers*.

Перейменування стовпця таблиці

Ми можемо перейменувати стовпці у таблиці за допомогою команди **ALTER TABLE з оператором RENAME COLUMN**. Наприклад:

```
ALTER TABLE Customers  RENAME COLUMN customer_id TO c_id;
```

Тут ми змінили ім’я стовпця *customer_id* на *c_id* у таблиці *Customers*.

Зміна стовпця у таблиці

Ми також можемо змінити тип даних стовпця за допомогою команди **ALTER TABLE з оператором MODIFY** або **ALTER COLUMN**. Наприклад:

```
ALTER TABLE Customers  MODIFY COLUMN age VARCHAR(2);
```

Тут ми змінили тип даних стовпця *age* на тип `VARCHAR(2)` в таблиці *Customers*.

Видалення стовпця таблиці

Ми також можемо видалити стовпці у таблиці за допомогою команди **ALTER TABLE з оператором DROP**. Наприклад:

```
ALTER TABLE Customers  DROP COLUMN age;
```

Тут ми видалили стовпець *age* з таблиці *Customers*.

Перейменування таблиці

Ми можемо змінити назву таблиці за допомогою команди **ALTER TABLE з оператором RENAME**. Наприклад:

```
ALTER TABLE Customers RENAME TO newCustomers;
```

Тут ми перейменували таблицю *Customers* на *newCustomers*.

3.3. Обмеження в SQL

У SQL ми можемо застосувати правила до стовпця, відомі як **обмеження**. Ці правила стосуються даних, які можуть зберігатися в стовпці. Наприклад, якщо стовпець має обмеження `NOT NULL`, то він не зможе зберігати **значення NULL**.

У SQL використовуються наступні обмеження:

Обмеження	**Опис**
NOT NULL	Значення не можуть бути NULL.
UNIQUE	Значення повинні бути унікальними і не можуть співпадати з уже існуючими.
PRIMARY KEY	Використовується для унікальної ідентифікації рядка.
FOREIGN KEY	Посилається на рядок в іншій таблиці.

CHECK	Перевіряє умову для вставки нового значення.
DEFAULT	Встановлює значення за замовчуванням, якщо воно не передано.
CREATE INDEX	Використовується для прискорення пошуку/читання даних.

Примітка: Ці обмеження називаються **обмеженнями цілісності даних**.

Обмеження NOT NULL

Обмеження NOT NULL означає, що стовпець не може зберігати значення `NULL`. Наприклад:

```
CREATE TABLE Colleges (
  college_id INT NOT NULL,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50)
);
```

Тут стовпці *college_id* та *college_code* таблиці *Colleges* не можуть містити значення `NULL`.

Примітка: Ключові слова `NOT NULL` використовуються для додавання обмежень, тоді як **умови IS NULL та IS NOT NULL** використовуються з оператором `WHERE` для вибору рядків з таблиці.

Видалення обмеження NOT NULL

Ми також можемо видалити обмеження `NOT NULL` для стовпця, якщо воно більше не потрібне. Наприклад:

```
ALTER TABLE Colleges  MODIFY college_id INT;
```

Обмеження NOT NULL з оператором ALTER TABLE

Ми також можемо додати обмеження `NOT NULL` до стовпця у існуючій таблиці за допомогою команди **ALTER TABLE**. Наприклад:

```
ALTER TABLE Colleges  MODIFY COLUMN college_id INT NOT NULL;
```

Тут ми додаємо обмеження `NOT NULL` до стовпця *college_id* в існуючій таблиці *Colleges*.

Тепер, коли ми спробуємо додати дані до таблиці *Colleges* без вказування значень для поля *college_id*, SQL видасть помилку. Наприклад:

```
INSERT INTO Colleges(college_code, college_name)
          VALUES ('NYC', "Waikiki");
```

Тут SQL видасть помилку, тому що нам не можна пропускати поле *college_id* у таблиці *Colleges* через діюче обмеження `NOT NULL`.

Обмеження UNIQUE

Обмеження UNIQUE означає, що стовпець повинен мати унікальне значення. Наприклад:

```
CREATE TABLE Colleges (
  college_id INT NOT NULL UNIQUE,
  college_code VARCHAR(20) UNIQUE,
  college_name VARCHAR(50)
);
```

Тут значення стовпця *college_code* повинно бути унікальним. Так само значення *college_id* повинно бути унікальним, до того ж ще й без значень `NULL`.

Порівняння UNIQUE з DISTINCT

Обмеження `UNIQUE` використовується, щоб зробити значення стовпця унікальним. Однак, щоб вибрати унікальні рядки із таблиці, слід використовувати **SELECT DISTINCT**. Наприклад:

```
SELECT DISTINCT country  FROM Customers;
```

Тут ми вибираємо унікальні значення стовпця *country* з таблиці *Customers*.

Обмеження PRIMARY KEY

Обмеження PRIMARY KEY — це просто комбінація обмежень `NOT NULL` та `UNIQUE`. Це означає, що значення стовпця використовується для унікальної ідентифікації рядка: стовпець не може містити значення, що повторюються, а також значення `NULL`.

. Наприклад:

```
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50)
);
```

Тут значення стовпця *college_id* є унікальним ідентифікатором рядка. До того ж він не може зберігати значення `NULL` і повинен бути унікальним (`UNIQUE`).
Примітка: У таблиці може бути лише один первинний ключ.

Синтаксис створення первинного ключа:

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50),
  CONSTRAINT CollegePK PRIMARY KEY (college_id)
);
```

Тут стовпець *college_id* має первинний ключ. Це означає, що значення цього стовпця повинні бути унікальними, а також не містити значення `NULL`.

Помилка первинного ключа

Якщо ми спробуємо вставити нульові або повторювані значення в стовпець з *первинним ключем*, то отримаємо помилку. Наприклад:

```
-- Перша операція вставки даних проходить успішно
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School");

-- Друга операція вставки даних - помилка у зв'язку з обмеженням
UNIQUE.
-- Значення college_id не є унікальним
INSERT INTO Colleges(college_id, college_code, college_name)
 VALUES (1, "ARD12", "Star Public School");
```

Тут SQL видасть помилку, тому що не можна вставити повторюване значення для поля *college_id* через обмеження `UNIQUE`.

Первинний ключ для декількох стовпців

Первинний ключ можна додати відразу декільком стовпцям. Наприклад:

```
CREATE TABLE Colleges (
  college_id INT,
  college_code VARCHAR(20),
  college_name VARCHAR(50),
  CONSTRAINT CollegePK PRIMARY KEY (college_id, college_code)
);
```

Тут обмеження `PRIMARY KEY` з ім’ям *CollegePK* складається із стовпців *college_id* та *college_code*. Це означає, що стовпці *college_id* і *college_code* повинні мати унікальні значення і не можуть містити значення NULL.

Тепер спробуємо вставити дані до таблиці *Colleges*:

```
-- Перша операція вставки даних проходить успішно
INSERT INTO Colleges(college_id, college_code, college_name)
```

```
VALUES (1, "ARD12", "Star Public School");
-- Друга операція вставки даних проходить успішно
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (2, "ARD13", "Star Public School");
-- Третя операція вставки даних - помилка у зв'язку з обмеженням
UNIQUE.
-- Стовпець college_id вже має значення 1, а стовпець
college_code вже має значення "ARD12"
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School");
```

PRIMARY KEY з командою ALTER TABLE

Ми також можемо додати обмеження `PRIMARY KEY` до стовпця в уже існуючій таблиці за допомогою оператора **ALTER TABLE**. Наприклад:

Для одного стовпця

```
ALTER TABLE Colleges ADD PRIMARY KEY (college_id);
```

Для декількох стовпців

```
ALTER TABLE Colleges
ADD CONSTRAINT CollegePK PRIMARY KEY (college_id, college_code);
```

Тут ми додаємо обмеження `PRIMARY KEY` до вказаних стовпців у існуючій таблиці.

Автоінкремент первинного ключа AUTO_INCREMENT

Звичайною практикою є автоматичне збільшення значення первинного ключа при вставці нового рядка. Наприклад:

```
-- Ключове слово AUTO_INCREMENT автоматично збільшує значення
CREATE TABLE Colleges (
  college_id INT AUTO_INCREMENT,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50),
  CONSTRAINT CollegePK PRIMARY KEY (college_id)
);

-- Вставляємо рядок без вказування значення для college_id
(первинний ключ)
INSERT INTO Colleges(college_code, college_name)
VALUES ("ARD13", "Star Public School");
```

Видалення первинного ключа

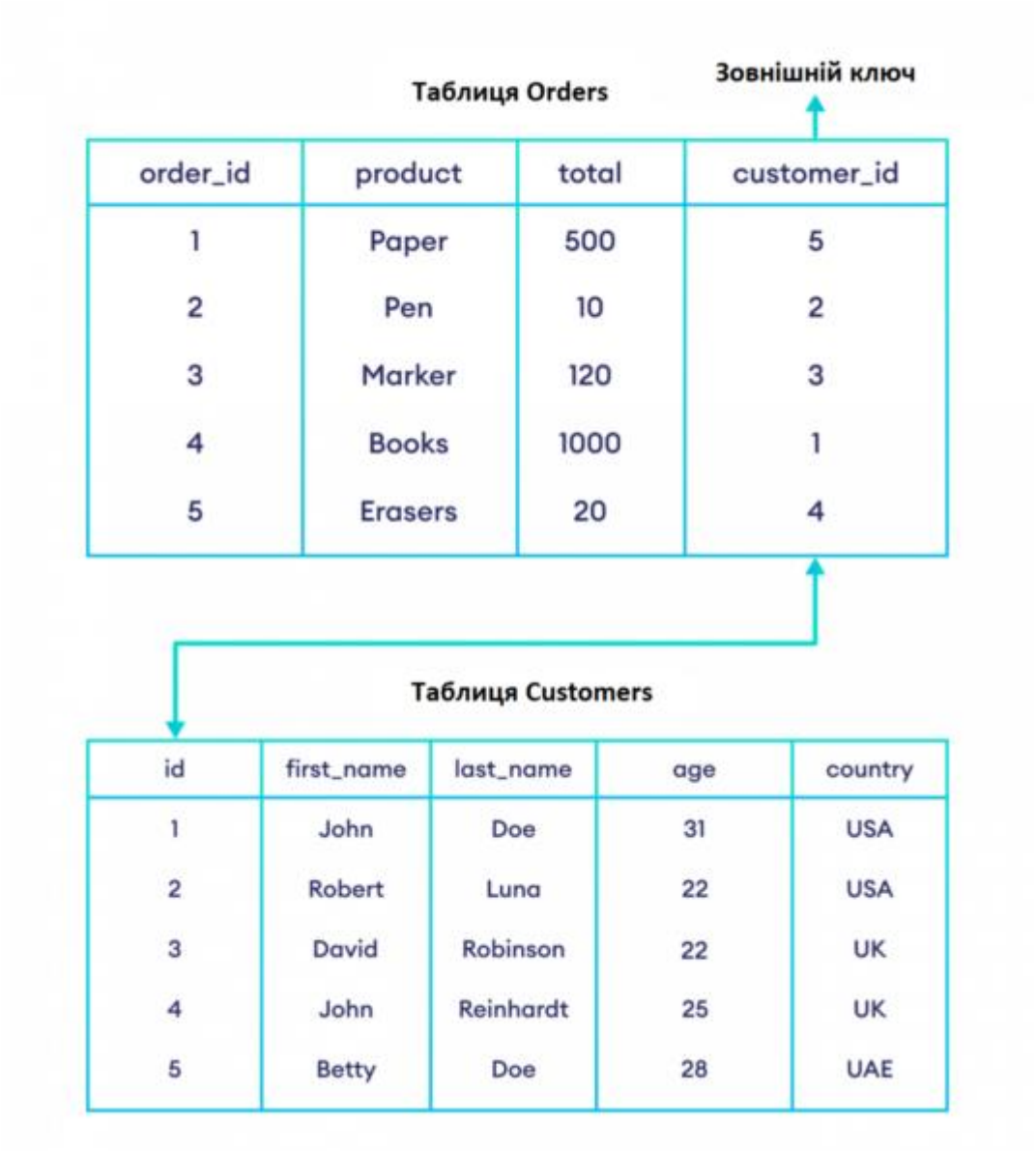
Ми можемо видалити обмеження `PRIMARY KEY` у таблиці за допомогою оператора `DROP`. Наприклад:

```
ALTER TABLE Colleges DROP PRIMARY KEY;
```

Тут ми видаляємо обмеження `PRIMARY KEY` з таблиці *Colleges*.

Зовнішній ключ (FOREIGN KEY)

Зовнішній ключ потрібен для того, щоб зв'язати дві різні таблиці між собою. Зовнішній ключ може посилатися на будь-який стовпець у батьківській таблиці. Проте загальноприйнятою практикою є посилання зовнішнього ключа на **первинний ключ (primary key)** батьківської таблиці. Наприклад:



Тут поле *customer_id* в таблиці *Orders* є `FOREIGN KEY`, який посилається на поле *id* в таблиці *Customers*. Це означає, що значенням *customer_id* (таблиці *Orders*) повинно бути значення зі стовпця *id* (таблиці *Customers*).

Створення зовнішнього ключа

Тепер подивимось, як ми можемо додати обмеження `FOREIGN KEY`:

```
-- Ця таблиця не має зовнішнього ключа
CREATE TABLE Customers (
  id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT CustomersPK PRIMARY KEY (id)
);
-- Додаємо зовнішній ключ до поля customer_id.
-- Зовнішній ключ посилається на поле id таблиці Customers
CREATE TABLE Orders (
  order_id INT,
  item VARCHAR(40),
  amount INT,
  customer_id INT REFERENCES Customers(id),
  CONSTRAINT OrdersPK PRIMARY KEY (order_id)
);
```

Тут стовпець *customer_id* таблиці *Orders* посилається на стовпець *id* таблиці *Customers*.

Обмеження REFERENCES

В деяких СУБД обмеження **REFERENCES** у стовпці використовується для посилання на дані з іншої таблиці. Наприклад:

```
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  customer_id int REFERENCES Customers(id)
);
```

Тут значенням ***customer_id*** в таблиці *Orders* повинно бути значення зі стовпця ***id*** таблиці *Customers*.

Вставка даних у таблицю із зовнішнім ключем

Спробуємо вставити дані у таблицю із зовнішнім ключем.

```
-- Спочатку вставляємо дані до таблиці без зовнішнього ключа
INSERT INTO Customers
VALUES
(1, 'John', 'Doe', 31, 'USA'),
(2, 'Robert', 'Luna', 22, 'USA');
-- Перша операція вставки даних проходить успішно
INSERT INTO Orders
VALUES
(1, 'Keyboard', 400, 2),
(2, 'Mouse', 300, 2),
```

```
(3, 'Monitor', 12000, 1);
-- Друга операція вставки даних призводить до помилки, оскільки
customer_id зі значенням 7 немає
INSERT INTO Orders
VALUES (4, 'Keyboard', 400, 7);
```

Використання зовнішнього ключа

Використовують по двом головним причинам:

Нормалізація даних. `FOREIGN KEY` допомагає нормалізувати дані у декількох таблицях та зменшити надмірність. Це означає, що у базі даних може бути кілька таблиць, пов'язаних одна з одною.

Запобігання вставці некоректних даних. Якщо дві таблиці в базі даних пов'язані через поле (атрибут), використання `FOREIGN KEY` гарантує, що в це поле не будуть вставлені неправильні дані. Це допомагає усунути помилки на рівні бази даних.

FOREIGN KEY з оператором ALTER TABLE

Можна додати обмеження `FOREIGN KEY` до існуючої таблиці за допомогою оператора **ALTER TABLE**. Наприклад:

```
CREATE TABLE Customers (
  id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT CustomersPK PRIMARY KEY (id)
);

CREATE TABLE Orders (
  order_id INT,
  item VARCHAR(40),
  amount INT,
  customer_id INT,
  CONSTRAINT OrdersPK PRIMARY KEY (order_id)
);

-- Додаємо зовнішній ключ до поля customer_id.
-- Зовнішній ключ посилається на поле id таблиці Customers
ALTER TABLE Orders
 ADD FOREIGN KEY (customer_id) REFERENCES Customers(id);
```

Декілька зовнішніх ключів у таблиці

Таблиця може мати декілька зовнішніх ключів. Припустимо, нам потрібно записати всі транзакції, де кожен користувач одночасно є покупцем та продавцем.

```
-- Ця таблиця не має зовнішнього ключа
CREATE TABLE Users (
  id INT,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  age INT,
  country VARCHAR(10),
  CONSTRAINT CustomersPK PRIMARY KEY (id)
);

-- Додаємо зовнішній ключ до полів buyer та seller.
-- Зовнішній ключ посилається на поле id таблиці Users
CREATE TABLE Transactions (
  transaction_id INT,
  amount INT,
  seller INT REFERENCES Users(id),
  buyer INT REFERENCES Users(id),
  CONSTRAINT TransactionsPK PRIMARY KEY (transaction_id)
);
```

Тут ми створюємо два зовнішні ключі (*buyer* та *seller*) у таблиці *Transactions*.

Обмеження DEFAULT

Обмеження **DEFAULT** використовується для встановлення значень за замовчуванням при спробі вставити порожнє (`NULL`) значення в стовпець. Наприклад:

```
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR(20),
  college_country VARCHAR(20) DEFAULT 'Japan'
);
```

Тут значенням за замовчуванням для стовпця *college_country* є `Japan`.
Якщо ми спробуємо зберегти значення `NULL` в стовпці *college_country*, то значенням стане `Japan`. Наприклад:

```
-- Вставляємо значення 'Japan' в стовпець college_country
INSERT INTO Colleges (college_id, college_code)
VALUES (1, 'ARP76');
-- Вставляємо значення 'UAE' в стовпець college_country
INSERT INTO Colleges (college_id, college_code, college_country)
VALUES (2, 'JWS89', 'UAE');
```

Обмеження DEFAULT з ALTER TABLE

Ми також можемо додати обмеження `DEFAULT` до існуючого стовпця за допомогою оператора **ALTER TABLE**. Наприклад:

```
ALTER TABLE Colleges ALTER college_country SET DEFAULT 'Japan';
```

Тут значенням за замовчуванням для стовпця *college_country* є `Japan`, якщо хтось спробує вставити `NULL`.

Видалення обмеження DEFAULT

Ми можемо видалити обмеження `DEFAULT`, використовуючи оператор `DROP`. Наприклад:

```
ALTER TABLE Colleges ALTER college_country DROP DEFAULT;
```

Тут ми видаляємо обмеження `DEFAULT` зі стовпця *college_country*.

Обмеження CHECK

Обмеження **CHECK** використовується для вказування умови, яка має бути виконана для вставки значення в таблицю. Наприклад:

```
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  amount INT CHECK (amount > 0)
);
```

Тут стовпець *amount* приймає значення лише більше `0`, це є умовою для перевірки. Тепер давайте спробуємо вставити дані в таблицю *Orders*.

Приклад 1

```
-- Вставляємо значення 100.
-- Дані додані
INSERT INTO Orders(amount) VALUES(100);
```

Приклад 2

```
-- Вставляємо значення -5.
-- Помилка при додаванні даних
INSERT INTO Orders(amount) VALUES(-5);
```

Примітка: Обмеження `CHECK` використовується для перевірки даних лише при вставці. Щоб перевірити, чи існує рядок, слід використовувати **оператор EXISTS**.

Приклад. Обмеження CHECK перевіряє умову, перш ніж дозволити помістити значення в стовпець. Наприклад:

```
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  amount int CHECK (amount >= 100)
);
```

Тут значення стовпця *amount* повинно бути більше або рівним 100. В протилежному випадку SQL видасть помилку.

Створення іменованого обмеження CHECK

Популярною практикою є створення **іменованих обмежень**, щоб їх було легше змінювати та видаляти. Наприклад:

```
-- Створюємо іменоване обмеження amountCK для стовпця amount.
-- Обмеження перевіряє, щоб значення для вставки було більшим 0
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  amount INT,
  CONSTRAINT amountCK CHECK (amount > 0)
);
```

Створення обмеження CHECK у існуючій таблиці

Ми можемо додати обмеження `CHECK` до існуючої таблиці за допомогою оператора **ALTER TABLE**. Наприклад:

```
-- Додаємо неіменоване обмеження CHECK
ALTER TABLE Orders
 ADD CHECK (amount > 0);
```

Також можна додати іменоване обмеження `CHECK`:

```
-- Додаємо іменоване обмеження CHECK - amountCK
ALTER TABLE Orders
ADD CONSTRAINT amountCK CHECK (amount > 0);
```

Примітка: Якщо ми спробуємо додати обмеження `amount > 0` до стовпця, значення якого вже менше `0`, ми отримаємо помилку.

Видалення обмеження CHECK

Ми можемо видалити обмеження `CHECK`, використовуючи оператор `DROP`. Наприклад:

```
-- Видаляємо обмеження CHECK – amountCK
```

```
ALTER TABLE Orders
 DROP CHECK amountCK;
```

Оператори в SQL

Оператори — це символи (і ключові слова), які використовуються для виконання операцій зі значеннями.

Арифметичні оператори

Арифметичні оператори виконують прості арифметичні операції, такі як додавання, віднімання, множення тощо.

Оператор	Опис
+	Додавання
–	Віднімання
*	Множення
/	Ділення
%	Ділення з остачею

- *Оператор додавання*

```
-- Повертає новий стовпець з ім'ям total_amount,
-- який має значення (поле amount + 100)
SELECT item, amount, amount+100 AS total_amount
 FROM Orders;
```

- *Оператор віднімання*

```
-- Повертає новий стовпець з ім'ям offer_price,
-- який має значення (поле amount - 20)
SELECT item, amount, amount-20 AS offer_price
 FROM Orders;
```

- *Оператор множення*

```
-- Повертає новий стовпець з ім'ям total_amount,
-- який має значення (поле amount * 4)
SELECT item, amount, amount*4 AS total_amount
 FROM Orders;
```

- *Оператор ділення*

```
-- Повертає новий стовпець з ім'ям half_amount,
-- який має значення (поле amount / 2)
SELECT item, amount, amount/2 AS half_amount
 FROM Orders;
```

- *Ділення з остачею*

```
-- Повертаємо 1, як остачу від ділення
SELECT 10 % 3 AS result;
```

Оператори порівняння в SQL

Ми можемо порівняти два значення, використовуючи оператори порівняння в SQL. Ці оператори повертають або `1` (означає `true`) або `0` (означає `false`).

Оператор	Опис
=	Дорівнює
<	Менше
>	Більше
<=	Менше/Дорівнює
>=	Більше/Дорівнює
<>, !=	Не дорівнює

- *Оператор Дорівнює*

```
-- Повертає рядки, де customer_id має значення 4
SELECT order_id, item, amount FROM Orders WHERE customer_id = 4;
```

- *Оператор Менше*

```
-- Повертаємо рядки, де сума замовлення (поле amount) менше 400
(не включаючи)
SELECT order_id, item, amount FROM Orders WHERE amount < 400;
```

- *Оператор Більше*

```
-- Повертає рядки, де сума замовлення (поле amount) більше 400
(не включаючи)
SELECT order_id, item, amount FROM Orders
WHERE amount > 400;
```

- *Оператор Менше/Дорівнює*

```
-- Повертає рядки, де сума замовлення (поле amount) менше або
дорівнює 400
SELECT order_id, item, amount FROM Orders
WHERE amount <= 400;
```

- *Оператор Більше/Дорівнює*

```
-- Повертає рядки, де сума замовлення (поле amount) більше або
дорівнює 400
SELECT order_id, item, amount FROM Orders
```

```
WHERE amount >= 400;
```

- *Оператор Не дорівнює*

```
-- Повертає рядки, де сума замовлення (поле amount) не дорівнює
400
SELECT order_id, item, amount
FROM Orders
WHERE amount != 400;
```

Замість `!=` ми також можемо використовувати `<>` для операцій “не дорівнює”.

На Зміст

3.4. Індексування таблиць

CREATE INDEX

Якщо стовпець має **обмеження CREATE INDEX**, дані витягуватимуться швидше, якщо ми використаємо саме цей стовпець для отримання даних.

Індекс — це об’єкт бази даних, створений для підвищення продуктивності пошуку даних. Таблиці в базі даних можуть мати велику кількість рядків, які зберігаються в довільному порядку, і їх пошук за вказаними критеріями шляхом послідовного перегляду таблиці рядок за рядком може тривати багато часу. Як закладка в книзі, індекс допомагає швидко отримати доступ до необхідних даних в таблиці, згідно SQL-запиту. Таким чином, використання індексів дозволяє прискорити отримання даних.

Наприклад:

```
-- Створюємо таблицю
CREATE TABLE Colleges (
  college_id INT PRIMARY KEY,
  college_code VARCHAR(20) NOT NULL,
  college_name VARCHAR(50)
);
-- Створюємо індекс
CREATE INDEX college_index
ON Colleges(college_code);
```

Тут ми створюємо індекс з ім’ям *college_index* у таблиці *Colleges*, використовуючи стовпець *college_code*.

Примітка: Побачити різницю у швидкості при малій кількості даних у таблиці проблематично. Однак при великій кількості даних можна легко помітити різницю в швидкості між використанням індексів і без них.

CREATE UNIQUE INDEX для унікальних значень

Якщо потрібно створити індекси для унікальних значень у стовпці, слід використовувати обмеження `CREATE UNIQUE INDEX`. Наприклад:

```
-- Створюємо унікальний індекс
CREATE UNIQUE INDEX college_index ON Colleges(college_code);
```

Тут ми створюємо унікальний індекс з ім'ям *college_index* у таблиці *Colleges*, використовуючи стовпець *college_code*.

Видалення індексу з таблиці

Для видалення індексу з таблиці використовується оператор `DROP INDEX`. Наприклад:

```
ALTER TABLE Colleges DROP INDEX college_index;
```

Тут ми видаляємо обмеження *college_index* із таблиці *Colleges*.

Примітка: Видалення індексу означає, що видаляється лише індекс. Дані у вихідній таблиці залишаються незмінними.

Обмеження UNIQUE з оператором ALTER TABLE

Ми також можемо додати обмеження `UNIQUE` до існуючого стовпця за допомогою команди **ALTER TABLE**. Наприклад:

Для одного стовпця

```
ALTER TABLE Colleges ADD UNIQUE (college_id);
```

Для декількох стовпців

```
ALTER TABLE Colleges ADD UNIQUE UniqueCollege (college_id,
college_code);
```

Тут ми додаємо обмеження `UNIQUE` до вказаних стовпців у існуючій таблиці.

Помилка при вставці повторюваних значень

Якщо ми спробуємо вставити повторювані значення в стовпець з обмеженням `UNIQUE`, то отримаємо помилку.

```
CREATE TABLE Colleges (
  college_id INT NOT NULL UNIQUE,
  college_code VARCHAR(20) UNIQUE,
  college_name VARCHAR(50)
);

-- Вставляємо значення в таблицю Colleges
INSERT INTO Colleges(college_id, college_code, college_name)
VALUES (1, "ARD12", "Star Public School"), (2, "ARD12", "Galaxy
School");
```

Тут ми намагаємося вставити значення `ARD12` у стовпець *college_code* у двох різних рядках. Оскільки це призведе до дублювання значень, а стовпець *college_code* має обмеження `UNIQUE`, ми отримаємо помилку.

Підрахунок унікальних рядків

Якщо потрібно підрахувати кількість унікальних рядків, слід використовувати **функцію COUNT()** з `SELECT DISTINCT`. Наприклад:

```
SELECT COUNT(DISTINCT country) FROM Customers;
```

Тут ми повертаємо кількість унікальних країн (*country*).

на Зміст

4. Команди DML (Data Manipulation Language)

4.1. Вставка нових рядків у таблицю

Команда INSERT INTO використовується для вставки нових рядків у таблицю. Наприклад:

```
INSERT INTO Customers(customer_id, first_name, last_name, age,
country)
     VALUES (5, 'János', 'Nagy', 31, 'USA');
```

Тут ми вставляємо новий рядок з вказаними значеннями у таблицю *Customers*.

Примітка: Якщо потрібно вставити дані з іншої існуючої таблиці, слід використовувати команду INSERT INTO SELECT.

У таблиці **поле з ідентифікатором** зазвичай унікальне і автоматично збільшується на одиницю. У таких випадках ми можемо не вказувати значення для цього стовпця під час вставки рядка. Наприклад:

```
INSERT INTO Customers(first_name, last_name, age, country)
     VALUES ('Imre', 'Kiss', 48, 'USA');
```

Тут SQL автоматично встановлює новий *customer_id* для нового рядка та вставляє його до таблиці.

Вставка відразу декілька рядків

Можна вставити відразу декілька рядків у таблицю. Наприклад:

```
INSERT INTO Customers(first_name, last_name, age, country)
VALUES
('Ádám', 'Novák', 31, 'HU'),
('József', 'Szabó', 43, 'HU'),
('Iván', 'Pejter', 26, 'UA');
```

Тут ми вставляємо три рядки у таблицю *Customers*.

Вставка рядків без вказування імен стовпців

Можна вставляти значення без вказування імен стовпців. Наприклад:

```
INSERT INTO Customers
VALUES (5, 'Chris', 'Evans', 42, 'USA');
```

Тут ми вставляємо новий рядок у стовпець, вказавши при цьому коректний порядок значень.

Примітка: Якщо явно не вказувати імена стовпців, то порядок значень в SQL-запиті повинен відповідати порядку стовпців у таблиці.

Пропуск значень при вставці рядка

Якщо пропустити імена стовпців під час вставки рядка, то значеннями цих стовпців буде **NULL**.

```
INSERT INTO Customers(first_name, last_name, age)
VALUES ('Brad', 'Pitt', 58);
```

Тут значенням стовпця *country* (останній стовпець, який ми пропустили) буде `NULL`. Однак стовпець з ідентифікатором (*customer_id*), як і раніше, буде збільшено на одиницю через принцип автоматичного інкременту, про який ми говорили вище.

Примітка: Якщо нульові значення недопустимі для стовпця, то SQL-запит призведе до помилки.

4.2. Редагування рядків таблиці

Команда UPDATE

UPDATE використовується для редагування існуючих рядків у таблиці. Наприклад:

```
UPDATE Customers SET first_name = 'Johnny'
 WHERE customer_id = 1;
```

Тут ми змінюємо значення стовпця *first_name* на `Johnny`, де *customer_id* дорівнює `1`.

Примітка: Якщо потрібно вставити новий рядок замість оновлення існуючого, слід використовувати команду **INSERT INTO**.

Оновлення відразу декілька значень у рядку

Можна оновити відразу декілька значень у рядку. Наприклад:

```
UPDATE Customers SET first_name = 'Johnny', last_name = 'Depp'
 WHERE customer_id = 1;
```

Тут ми змінюємо значення стовпця *first_name* на `Johnny`, а *last_name* на `Depp`, де *customer_id* дорівнює `1`.

Оновлення декількох рядків

Можна оновити відразу декілька рядків. Наприклад:

```
UPDATE Customers SET country = 'NP' WHERE age = 22;
```

Тут ми змінюємо значення стовпця *country* на `NP`, якщо значенням стовпця *age* є `22`. Якщо є більше одного рядка з віком `22`, всі відповідні рядки будуть відредаговані.

Оновлення всіх рядків

Ми можемо оновити відразу всі рядки в таблиці. Для цього потрібно просто не вказувати оператор `WHERE`. Наприклад:

```
UPDATE Customers SET country = 'USA';
```

Тут ми змінюємо значення стовпця *country* на `NP` для всіх рядків.

Примітка: Будьте обережні під час використання оператора `UPDATE`. Якщо пропустити/забути оператор `WHERE`, всі рядки будуть змінені, і ці зміни будуть незворотними.

4.3. Видалення рядків таблиці

DELETE FROM TABLE

Команда **DELETE FROM** використовується для видалення рядків у таблиці. Наприклад:

```
DELETE FROM Customers WHERE customer_id = 5;
```

Тут ми видаляємо рядок з таблиці *Customers*, де *customer_id* дорівнює `5`.

Видалення всіх рядків таблиці

За допомогою оператора `WHERE` ми можемо вказати, які рядки потрібно видалити. Однак якщо потрібно видалити відразу всі рядки, достатньо буде просто не вказувати оператор `WHERE`. Наприклад:

```
DELETE FROM Customers;
```

Тут ми видаляємо усі рядки з таблиці *Customers*.

Примітка: Будьте обережні при використанні оператора `DELETE FROM`. Записи можуть бути безповоротно втрачені, якщо у вас не виявиться резервних копій.

Команда TRUNCATE TABLE

TRUNCATE TABLE — це ще один спосіб одночасного видалення одразу всіх рядків у таблиці. Наприклад:

```
TRUNCATE TABLE Customers;
```

Тут ми робимо те саме, що й вище — видаляємо всі рядки з таблиці *Customers*.

Примітка: Оператор `TRUNCATE TABLE` заборонено використовувати у зв'язці з оператором `WHERE`.

Порівняння DELETE FROM з TRUNCATE TABLE

Основна відмінність між обома операторами полягає в тому, що оператор `DELETE FROM` може використовуватися у зв'язці з оператором `WHERE`, а `TRUNCATE TABLE` — ні. Це означає, що ми можемо видалити один або кілька рядків, використовуючи оператор `DELETE FROM`, тоді як оператор `TRUNCATE TABLE` видаляє відразу весь вміст таблиці.

Ми можемо виконати роботу оператора `TRUNCATE TABLE` оператором `DELETE FROM`, просто не вказуючи оператор `WHERE`. Наприклад, наступний код:

```
DELETE FROM Customers;
```

ідентичний

```
TRUNCATE TABLE Customers;
```

на Зміст

5. Вибірка даних з таблиць

Команда **SELECT в SQL** використовується для отримання (вибору) даних з таблиці бази даних. Наприклад:

```
SELECT first_name, last_name FROM Customers;
```

За допомогою цієї команди ми вибираємо (SELECT) ім'я (*first_name*) та прізвище (*last_name*) кожного клієнта з таблиці *Customers*.

SELECT *

Щоб отримати *всі стовпці* з таблиці, використовується символ *. Наприклад:

```
SELECT * FROM Customers;
```

Тут ми відбираємо весь вміст таблиці *Customers*.

Оператор WHERE

Оператор **WHERE** при виконанні різних команд відбирає рядки таблиці, які задовільняють задану умову. Команда SELECT теж може використовуватись з WHERE. Наприклад:

```
SELECT * FROM Customers WHERE last_name = 'Doe';
```

Тут відбираються всі клієнти з таблиці *Customers* з прізвищем Doe.
Розглянемо інший приклад:

```
SELECT age, country FROM Customers WHERE country = 'USA';
```

Тут ми відбираються поля віку (*age*) та країни (*country*) всіх клієнтів, країною яких є США (*USA*).

Примітка: У SQL текстові дані поміщають або в одинарні, або в подвійні лапки (як у прикладі з 'USA').

Оператори для побудови умов

Оператор WHERE може використовувати інші оператори для побудови умов. Розглянемо деякі з найпопулярніших операторів:

1. Оператор Дорівнює (=)

```
SELECT * FROM Customers WHERE first_name = 'David';
```

Тут ми відбираємо всіх клієнтів з іменем David з таблиці *Customers*.

2. Оператор Більше (>)

```
SELECT * FROM Customers WHERE age > 25;
```

Тут ми відбираємо всіх клієнтів з таблиці *Customers*, вік (*age*) яких перевищує 25 років.

Примітка: Якщо умова WHERE не відповідає жодному рядку, то повертається пустий результат.

Приклад. Запросити дані з використанням дат

Можна використовувати запити для отримання даних із фільтрацією за датами. Наприклад:

```
SELECT * FROM Teams WHERE registered = "2022-10-12";
```

Тут ми повертаємо команди (*Teams*), які були зареєстровані 2022-10-12.
Розглянемо інший приклад:

```
SELECT * FROM Teams  WHERE registered > "2022-11-15";
```

Тут ми обираємо лише ті команди, які були зареєстровані після 2022-11-15.

Оператори AND, OR та NOT
Логічні оператори AND, OR та NOT в SQL використовуються з операторами WHERE або HAVING.

Оператор AND

Оператор **AND** витягує дані тільки, якщо всі умови істинні (true). Наприклад:

```
SELECT first_name, last_name
     FROM Customers
          WHERE country = 'USA' AND last_name = 'Doe';
```

Тут ми відбираємо ім'я (*first_name*) та прізвище (*last_name*) тих клієнтів з таблиці *Customers*, країна яких — USA,а прізвище — Doe.

Оператор OR

Оператор OR відбирає дані, якщо хоча б одна з умов істинна (true). Наприклад:

```
SELECT first_name, last_name FROM Customers
          WHERE country = 'USA' OR last_name = 'Doe';
```

Тут ми відбираємо ім'я (*first_name*) та прізвище (*last_name*) тих клієнтів з таблиці *Customers*, чия країна — USA **АБО** прізвище — Doe.

Оператор NOT

Оператор **NOT** витягує дані тільки, якщо умова **хибна** (false). Наприклад:

```
SELECT first_name, last_name FROM Customers
          WHERE NOT country = 'USA';
```

Тут ми відбираємо ім'я (*first_name*) та прізвище (*last_name*) тих клієнтів із таблиці *Customers*, країною яких **НЕ** є USA.

Поєднання декількох операторів

Можна також комбінувати декілька операторів AND, OR та NOT в одній SQL-команді. Наприклад, якщо ми хочемо вибрати всіх клієнтів з таблиці *Customers*, країна яких — USA або UK і вік менше 26 років:

```
SELECT * FROM Customers
     WHERE (country = 'USA' OR country = 'UK') AND age < 26;
```

Розглянемо інший приклад:

```
SELECT * FROM Customers
     WHERE NOT country = 'USA' AND NOT last_name = 'Doe';
```

Тут ми відбираємо всіх клієнтів із таблиці *Customers*, країною яких НЕ є USA, а прізвище — НЕ Doe.

SELECT DISTINCT

Команда **SELECT DISTINCT** витягує унікальні дані з таблиці. Всередині таблиці в рядках або стовпцях дані можуть повторюватися, тому коли потрібно отримати унікальні дані без дублів, використовується SELECT DISTINCT. Наприклад:

```
SELECT DISTINCT country  FROM Customers;
```

Тут ми відбираємо список країн із таблиці *Customers*.
Розглянемо інший приклад:

```
SELECT DISTINCT country, first_name  FROM Customers;
```

Тут ми відбираємо рядки, якщо комбінація країни (*country*) та імені (*first_name*) унікальна.

Оператор DISTINCT з функцією COUNT()

Якщо потрібно підрахувати кількість унікальних рядків, можна використати **функцію COUNT()** з оператором DISTINCT. Наприклад:

```
SELECT COUNT(DISTINCT country)     FROM Customers;
```

Тут ми рахуємо кількість унікальних країн (без повторів).

Псевдоніми (оператор AS)

Ключове слово **AS** використовується для присвоювання стовпцям або таблицям тимчасового імені (*псевдоніма*), яке можна використовувати для ідентифікації цього стовпця або таблиці пізніше. Псевдонім існує лише на час виконання запиту. Наприклад:

```
SELECT first_name AS name FROM Customers;
```

Тут ми відбираємо імена (*first_name*) клієнтів із таблиці *Customers*. Однак у результаті ми отримаємо стовпець з ім'ям ***name*** замість *first_name*.

Декілька псевдонімів AS

Можна використовувати псевдоніми для кількох стовпців відразу. Наприклад:

```
SELECT customer_id AS cid, first_name AS name
     FROM Customers;
```

Тут ми відбираємо дані зі стовпцями *cid* (замість *customer_id*) та *name* (замість *first_name*).

Використання AS у виразах

Можна комбінувати дані з декількох стовпців і представити їх в одному за допомогою функції CONCAT(). Наприклад:

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name
          FROM Customers;
```

Тут ми відбираємо ім'я (*first_name*) та прізвище (*last_name*) клієнтів з таблиці *Customers*, поєднуючи їх в одному стовпці з іменем *full_name*.

SELECT TOP

Команда **SELECT TOP** використовується для вибору фіксованої кількості рядків із бази даних. Наприклад:

```
SELECT TOP 2 * FROM Customers;
```

Тут ми відбираємо перші 2 рядки з таблиці *Customers*.

Розглянемо ще один приклад:

```
SELECT TOP 2 first_name, last_name FROM Customers;
```

Тут ми відбираємо поля *first_name* та *last_name* з перших 2 рядків таблиці *Customers*.

LIMIT

Ключове слово **LIMIT** використовується для вибору фіксованої кількості рядків у MySQL, PostGresQL та SQLITE. Наприклад:

```
SELECT first_name, age FROM Customers LIMIT 2;
```

Тут ми відбираємо перші 2 рядки з таблиці *Customers*.

LIMIT з OFFSET

Ключове слово **OFFSET** використовується для позначення місця, звідки слід витягувати рядки. Наприклад:

```
SELECT first_name, last_name FROM Customers
          LIMIT 2 OFFSET 3;
```

Тут ми відбираємо два рядки, починаючи з четвертого рядка. OFFSET 3 означає, що перші 3 рядки виключені.

Примітка: Ключове слово TOP підтримується не у всіх системах управління базами даних (СУБД). Різні СУБД використовують різні ключові слова для вибору фіксованої кількості рядків. Наприклад:

ключове слово **TOP** використовується в SQL Server, MS Access;
ключове слово **LIMIT** використовується в MySQL, PostgreSQL, SQLite;

Оператор IN

Оператор **IN** використовується з оператором WHERE для зіставлення значень у списку. Наприклад:

```
SELECT first_name, country FROM Customers WHERE country IN
('USA', 'UK');
```

Тут ми відбираємо рядки, якщо країна (*country*) — USA або UK.
Оператор IN також можна використовувати для вибору рядків, які містять певне значення. Наприклад:

```
SELECT first_name, country FROM Customers WHERE 'USA' IN
(country);
```

Тут ми відбираємо рядки, в яких в полі *country* знаходиться значення USA.

Оператор NOT IN

Оператор **NOT IN** використовується для виключення рядків, що відповідають вказаним значенням у списку. Він повертає всі рядки, крім виключених. Наприклад:

```
SELECT first_name, country FROM Customers WHERE country NOT IN
('UK', 'UAE');
```

Тут ми відбираємо всі рядки, в яких в полі *country* немає значення UK та UAE.

Оператор IN з повторюваними значеннями

Оператор IN ігнорує повторювані значення (дублі) у списку. Наприклад, наступний код:

```
SELECT first_name, country FROM Customers WHERE country IN
('USA', 'UK', 'USA');
```

рівнозначний

```
SELECT first_name, country FROM Customers WHERE country IN
('USA', 'UK');
```

Оператор IN з підзапитом

Припустимо, нам потрібна інформація про клієнтів, які розмістили замовлення. Ми можемо це зробити за допомогою наступного підзапиту:

```
SELECT customer_id, first_name FROM Customers
     WHERE customer_id IN (   SELECT customer_id   FROM Orders);
```

У нас є 2 таблиці — *Customers (Клієнти)* та *Orders (Замовлення)*. Ми вибираємо ідентифікатор клієнта (*customer_id*) та його ім'я (*first_name*) з таблиці *Customers*, лише якщо той самий ідентифікатор також є в таблиці *Orders*.

Оператор BETWEEN

Оператор **BETWEEN** використовується з оператором WHERE для зіставлення значень у вказаному діапазоні. Наприклад:

```
SELECT item, amount FROM Orders WHERE amount BETWEEN 300 AND
500;
```

Тут ми відбираємо всі замовлення з сумами від 300 до 500, включно з 300 та 500.

Оператор NOT BETWEEN

Оператор **NOT BETWEEN** використовується для виключення рядків, що відповідають значенням у вказаному діапазоні. Він повертає всі рядки, крім виключених. Наприклад:

```
SELECT item, amount FROM Orders
          WHERE amount NOT BETWEEN 300 AND 500;
```

Тут ми відбираємо всі замовлення, за винятком рядків з сумами від 300 до 500.

Оператор BETWEEN та робота з текстом

Оператор BETWEEN також може працювати і з текстовими даними. Наприклад:

```
SELECT item, amount FROM Orders
     WHERE item BETWEEN 'I' AND 'L';
```

Тут ми відбираємо всі замовлення, в яких назва позиції (товару) знаходиться між буквами I та L.

Якщо потрібно вибрати всі слова, які починаються з літери L, тоді потрібно додати знак %:

```
SELECT item, amount FROM Orders
                WHERE item BETWEEN 'I' AND 'L%';
```

NULL-значеннями

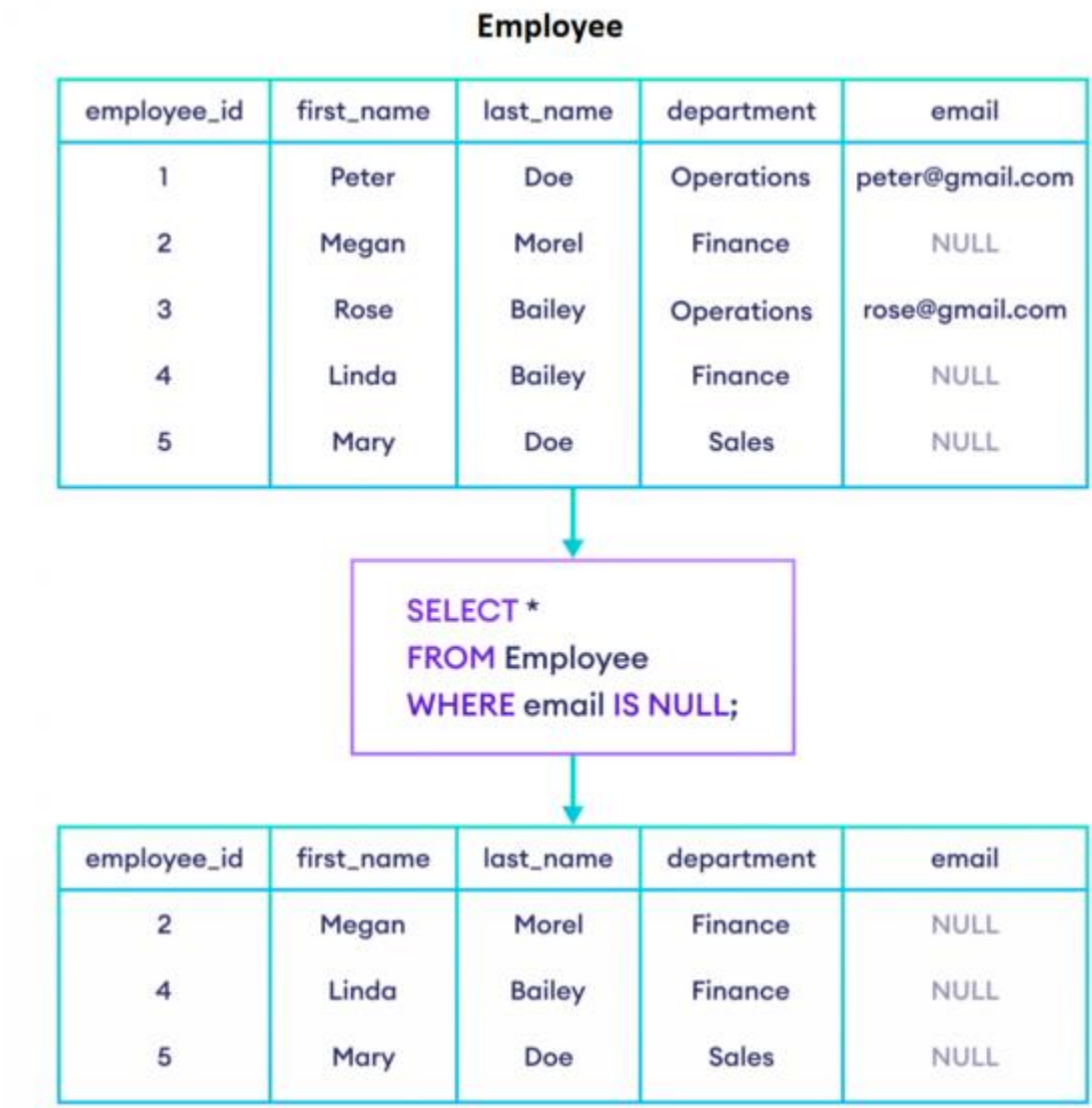
Умова **IS NULL** використовується для вибору рядків, в яких вказане поле має значення NULL. **Умова IS NOT NULL** використовується для вибору рядків, в яких вказане поле НЕ є NULL. Можна використовувати функцію COUNT() з умовою IS NULL для підрахунку кількості порожніх рядків.

Умова IS NULL

Умова **IS NULL** використовується для вибору рядків, в яких вказане поле має значення NULL. Наприклад:

```
SELECT * FROM Employee WHERE email IS NULL;
```

Тут ми вибираємо працівників, які не мають електронної пошти.



Примітка: Порожні поля вважаються як NULL в SQL. Проте 0, ' ' та пробіли не є NULL.

Умова IS NOT NULL

Умова **IS NOT NULL** використовується для вибору рядків, в якиз вказане поле не є `NULL`. Наприклад:

```
SELECT * FROM Employee WHERE email IS NOT NULL;
```

Тут ми вибираємо працівників, які мають електронну пошту.

Умова IS NULL з функцією COUNT()

Можна використовувати **функцію COUNT()** з умовою `IS NULL` для підрахунку кількості порожніх рядків. Наприклад:

```
SELECT COUNT(*)  FROM Employee WHERE email IS NULL;
```

Тут ми отримуємо загальну кількість працівників, які не мають електронної пошти. Таким же чином ми можемо використовувати функцію COUNT() з умовою IS NOT NULL для підрахунку кількості непустих полів.

Функція MAX()

Функція **MAX()** повертає найбільше значення стовпця. Наприклад:

```
SELECT MAX(age) FROM Customers;
```

Тут ми отримуємо найбільше значення зі стовпця *age*.

Функція MIN()

Функція **MIN()** повертає найменше значення стовпця. Наприклад:

```
SELECT MIN(age) FROM Customers;
```

Тут ми отримуємо найменше значення зі стовпця *age*.

Використання псевдонімів з MAX() та MIN()

У вище наведених прикладах ми отримували по 1 стовпцю на виході — *MAX (age)* та *MIN (age)*. Цим полям також можна присвоїти псевдоніми (тимчасові імена) за допомогою **оператора AS**. Наприклад:

```
SELECT MAX(age) AS max_age FROM Customers;
```

Тепер ми отримаємо стовпець з іменем *max_age* замість *MAX(age)*.

Функції MAX() та MIN() з рядками

Функції MAX() та MIN() також працюють з іншими типами даних, такими як рядки (текст), а не тільки з числами. Наприклад:

```
SELECT MIN(first_name) AS min_first_name FROM Customers;
```

Тут ми вибираємо найменше значення стовпця *first_name* з урахуванням алфавітного порядку.

Вибір рядка з максимальним (мінімальним) значенням

Якщо нам потрібно повністю витягнути рядок(и), що містить максимальне/мінімальне значення, ми можемо використати вкладений **оператор SELECT** наступним чином:

```
-- Відбираємо повністю рядки, які містять найменший вік (age)
SELECT * FROM Customers
     WHERE age = ( SELECT MIN(age)  FROM Customers );
```

Тут ми повністю відбираємо всі рядки з найменшим значенням віку (*age*).

Функція COUNT()

Функція **COUNT()** підраховує кількість рядків у таблиці. Наприклад:

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

Тут ми підраховуємо та повертаємо кількість рядків у таблиці *Customers*.

Використання псевдонімів з функцією COUNT()

У вищевказаному прикладі на виході ми отримали стовпець з іменем COUNT(*). Ми можемо змінити його назву за допомогою **оператора AS**. Наприклад:

```
SELECT COUNT(*) AS total_customers FROM Customers;
```

Тепер у нас є стовпець total_customers замість COUNT(*).

Функція COUNT() з оператором WHERE

Розглянемо приклад використання функції COUNT() з **оператором WHERE**:

```
SELECT COUNT(country) AS customers_in_UK FROM Customers
          WHERE country = 'UK';
```

Тут ми виводимо кількість клієнтів із Великобританії (*UK*).

Функція COUNT() з оператором DISTINCT

Якщо нам потрібно підрахувати кількість унікальних рядків (без дублів), ми можемо використати функцію COUNT() з **оператором DISTINCT**. Наприклад:

```
SELECT COUNT(DISTINCT country) FROM Customers;
```

Тут ми виводимо кількість унікальних країн.

Функція COUNT() з оператором GROUP BY

Функція COUNT() може використовуватися з **оператором GROUP BY** для підрахунку рядків зі схожими значеннями. Наприклад:

```
SELECT country, COUNT(*) AS customers FROM Customers
          GROUP BY country;
```

Тут ми підраховуємо та виводимо кількість клієнтів у кожній країні.

Функція COUNT() з оператором HAVING

Розглянемо приклад використання функції COUNT() з **оператором HAVING**:

```
SELECT COUNT(customer_id), country FROM Customers
     GROUP BY country  HAVING COUNT(customer_id) > 1;
```

Тут ми підраховуємо кількість клієнтів, групуючи їх за країнами (*country*), а потім виводимо, якщо в кожній країні їх налічується більше 1.

Функція COUNT() зі значеннями NULL

SELECT COUNT(*) повертає кількість всіх рядків, незалежно від **значень NULL** (тобто разом з ними).

SELECT COUNT(attribute) повертає кількість рядків, що містять ненульові значення у вказаному стовпці.

Функція SUM()

Функція **SUM()** використовується для обчислення суми чисел у стовпці. Наприклад:

```
SELECT SUM(amount) AS total_sales FROM Orders;
```

Тут ми отримуємо суму всіх замовлень із таблиці *Orders*.

Розглянемо інший приклад:

```
SELECT SUM(amount) AS total_of_cus4 FROM Orders
          WHERE customer_id = 4;
```

Тут ми отримуємо загальну суму, яку має заплатити клієнт з ідентифікатором 4 за всі свої замовлення.

Функція AVG()

Функція **AVG()** використовується для обчислення середнього арифметичного чисел у стовпці. Наприклад:

```
SELECT AVG(age) AS average_age FROM Customers;
```

Тут ми отримуємо середній вік усіх клієнтів.

Розглянемо інший приклад:

```
SELECT DISTINCT customer_id, AVG(amount) AS average_spends
     FROM Orders GROUP BY customer_id;
```

Тут ми отримуємо середні витрати кожного клієнта.

Оператор ORDER BY

Оператор **ORDER BY** використовується для сортування даних у порядку зростання або спадання. Наприклад:

```
SELECT * FROM Customers ORDER BY first_name;
```

Тут ми вибираємо всіх клієнтів, а потім сортуємо їх у порядку зростання за іменем (*first_name*).

ORDER BY ASC

Ми можемо явно використовувати ***ключове слово*** **ASC** для сортування вибраних записів у порядку зростання. Наприклад:

```
SELECT * FROM Customers ORDER BY age ASC;
```

Тут ми вибираємо всіх клієнтів, а потім сортуємо їх у порядку зростання віку (*age*).

Примітка: Оператор `ORDER BY` за замовчуванням виконує сортування даних у порядку зростання; немає потреби явно вказувати `ASC`.

ORDER BY DESC

Ключове слово **DESC** використовується для сортування вибраних записів у порядку спадання. Наприклад:

```
SELECT * FROM Customers ORDER BY age DESC;
```

Тут ми вибираємо всіх клієнтів, а потім сортуємо їх у порядку спадання віку.

ORDER BY з декількома стовпцями

Ми також можемо використовувати оператор `ORDER BY` з декількома стовпцями. Наприклад:

```
SELECT * FROM Customers ORDER BY first_name, age;
```

Тут ми вибираємо всі записи, а потім сортуємо їх за іменем (*first_name*). Якщо імена повторюються, то записи сортуємо вже за віком (*age*).

ORDER BY з оператором WHERE

Ми також можемо використовувати оператор `ORDER BY` з командою **SELECT WHERE**. Наприклад:

```
SELECT last_name, age FROM Customers
     WHERE NOT country = 'UK' ORDER BY last_name DESC;
```

Тут ми вибираємо поля *last_name* та *age* з таблиці *Customers*, у яких у полі *country* вказано НЕ Великобританія (*UK*). Потім вибрані записи сортуються в порядку спадання за прізвищем (*last_name*).

Примітка: Оператор `WHERE` повинен знаходитися перед оператором `ORDER BY` у випадку їх спільного використання.

Групування рядків. Оператор GROUP BY

Оператор **GROUP BY** використовується для групування рядків по одному або декільком стовпцям. Наприклад:

```
SELECT country, COUNT(*) AS number FROM Customers GROUP BY
country;
```

Тут ми групуємо рядки по стовпцю *country* та підраховуємо кількість записів для кожної країни (використовуючи **функцію COUNT()**).

Примітка: Оператор `GROUP BY` використовується у поєднанні з агрегатними функціями, такими як **MIN(), MAX()**, **SUM(), AVG()**, COUNT() та ін.

Ім’я стовпця виконання функції `COUNT(*)` — *number*, через використання псевдоніма (**оператор AS**).

Розглянемо інший приклад. Спробуємо знайти загальну суму замовлень для кожного клієнта:

```
SELECT customer_id, SUM(amount) AS total FROM Orders
          GROUP BY customer_id;
```

Тут ми сумуємо значення стовпця *amount* після групування рядків за ідентифікатором *customer_id*.

Оператор GROUP BY з JOIN

Ми також можемо використовувати оператор `GROUP BY` з **операторами JOIN**. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Count(Orders.order_id) AS order_count
```

```
FROM Customers LEFT JOIN Orders
ON Customers.customer_id = Orders.customer_id
GROUP BY Customers.customer_id;
```

Тут ми об'єднуємо таблиці *Customers* та *Orders*, а потім групуємо результат по полю *customer_id*. Це дозволяє нам дізнатися кількість замовлень, розміщених кожним клієнтом.

Оператор GROUP BY з декількома стовпцями

Оператор `GROUP BY` також можна використовувати для групування рядків на основі декількох стовпців. Наприклад:

```
SELECT country, state, MIN(age) as min_age FROM Persons
          GROUP BY country, state;
```

Тут ми групуємо всіх людей з однаковою країною (*country*) та штатом (*state*) та вказуємо найменший вік (*age*) людини у кожній групі.

Оператор GROUP BY з оператором HAVING

Ми можемо використовувати оператор `GROUP BY` з **оператором HAVING. Оператор HAVING** в SQL використовується, коли потрібно відфільтрувати дані від використання агрегатних функцій, таких як **MIN() і MAX()**, **SUM() і AVG()** та **COUNT()**.

```
SELECT COUNT(customer_id), country FROM Customers
GROUP BY country HAVING COUNT(customer_id) > 1;
```

Тут ми рахуємо кількість клієнтів (*customer_id*), групуючи їх за країнами (*country*), а потім повертаємо результат, якщо на країну є більше 1 клієнта.

Примітка: Оператор `HAVING` додали через те, що оператор `WHERE` не підтримує агрегатні функції. Крім того, перед оператором HAVING необхідно використовувати **GROUP BY**.

Розглянемо приклад. Якщо ми хочемо вибрати всі замовлення, сума (*amount*) яких менше 500, ми можемо зробити:

```
SELECT customer_id, amount FROM Orders WHERE amount < 500;
```

Тепер, якщо нам потрібно порахувати суму всіх замовлень кожного клієнта (*customer_id*):

```
SELECT customer_id, SUM(amount) AS total FROM Orders
          GROUP BY customer_id;
```

Ми отримуємо згруповані дані.

Оператор LIKE. Символи підстановки

Символ підстановки % використовується для представлення нуля чи більше символів. Наприклад:

```
SELECT * FROM Customers WHERE last_name LIKE 'R%';
```

Тут ми вибираємо клієнтів, чиє прізвище (*last_name*) починається з літери **R**, за якою слідує нуль або більше символів. Підстановочний знак _ в SQL

Символ підстановки _ використовується для представлення одного символу в рядку. Наприклад:

```
SELECT * FROM Customers WHERE country LIKE 'U_';
```

Тут ми вибираємо клієнтів, назва країни (*country*) яких починається з літери **U** і супроводжується лише одним символом.

Символ підстановки [] використовується для представлення одного з зазначених у квадратних дужках символу. У дужках можна вказати відразу кілька символів, але вибиратиметься лише один із них. Наприклад:

```
SELECT * FROM Customers
          WHERE country LIKE 'U[KA]%';
```

Тут ми вибираємо клієнтів, назва країни (*country*) яких починається з літери **U**, а друга літера **K** або **A** — щось одне. Після другої літери із заданого шаблону допускається будь-яка кількість символів.

Символи підстановки в PostgreSQL та MySQL:

`%` — нуль або більше символів;

`_` — одиночний символ.

Оператор **LIKE** використовується з оператором **WHERE** для отримання даних, які відповідають заданому рядковому шаблону. Наприклад:

```
SELECT * FROM Customers WHERE country LIKE 'UK';
```

Тут ми вибираємо клієнтів, чия країна — `UK`.

Примітка: Хоча в цьому прикладі оператор `LIKE` поводиться так само, як оператор `=`, це не одне й те саме. Оператор `=` використовується для перевірки рівності, тоді як оператор `LIKE` використовується лише для зіставлення рядкових шаблонів.

Оператор NOT LIKE

Ми також можемо інвертувати роботу оператора `LIKE` та ігнорувати дані, що відповідають заданому рядковому шаблону, за допомогою **оператора NOT**. Наприклад:

```
SELECT * FROM Customers WHERE country NOT LIKE 'USA';
```

Тут ми вибираємо всіх клієнтів, крім тих, чия країна — США (*USA*).

Оператор LIKE з декількома значеннями

Ми можемо використовувати оператор `LIKE` з кількома рядковими шаблонами для зіставлення за допомогою оператора `OR`. Наприклад:

```
SELECT * FROM Customers
          WHERE last_name LIKE 'R%t' OR last_name LIKE '%e';
```

Тут ми вибираємо клієнтів, чиє прізвище (*last_name*) починається з букви **R** і закінчується буквою **t**, або клієнтів, чиє прізвище закінчується буквою **e**.

Вибірка з двох або більше таблиць оператором UNION

Оператор **UNION** в SQL вибирає рядки з двох або більше таблиць. Якщо рядки в таблицях повторюються, вони виводяться лише один раз (тобто без дублів). Наприклад:

```
SELECT age
FROM Teachers
UNION
SELECT age
FROM Students;
```

Тут ми відбираємо стовпець *age* з таблиць *Teachers* і *Students*, ігноруючи рядки, що повторюються.
Щоб використовувати оператор `UNION` в SQL, слід пам’ятати, що:

- **Кількість стовпців у всіх таблицях має бути однаковою**. Наприклад, у вищезазначеному прикладі в таблицях *Teachers* та *Students* по три стовпці.
- **Типи даних стовпців мають бути однаковими.** Наприклад, у вищезазначеному прикладі стовпець *age* у таблиці *Teachers* є цілочисленного типу даних, як і стовпець *age* у таблиці *Students*.
- **Стовпці повинні знаходитися в однаковому порядку в кожній таблиці.** Наприклад, у вищезазначеному прикладі порядок стовпців у таблиці *Teachers — id-name-age*, як і в таблиці *Students*.

Оператор UNION ALL

Оператор **UNION ALL** вибирає рядки з двох або більше таблиць, подібно до роботи оператора `UNION`. Однак, на відміну від `UNION`, оператор `UNION ALL` не ігнорує рядки, що повторюються (тобто враховує дублі).

Спробуємо виконати попередню команду SQL, використовуючи `UNION ALL` замість `UNION`:

```
SELECT age
FROM Teachers
UNION ALL
SELECT age
FROM Students;
```

Тут ми вибираємо рядки з обох таблиць, включно з дублями.

Рекомендується використовувати оператор `UNION ALL` у випадках, коли ви впевнені, що дані є унікальними. Таким чином, можна покращити продуктивність.

Підзапити

У SQL ми можемо помістити один запит всередині іншого запиту, зробивши **підзапит**. У підзапиті результат зовнішнього запиту залежить від результатів внутрішнього запиту. Ось чому підзапити також називають **вкладеними запитами**. Наприклад:

```
SELECT * FROM Customers
WHERE age = (   SELECT MIN(age)   FROM Customers );
```

Тут спочатку виконується підзапит (внутрішній запит) — ми вибираємо найменше значення *age* з таблиці *Customers*. Потім виконується зовнішній запит — ми вибираємо рядки, в яких *age* дорівнює результату підзапиту (найменшому значенню *age*).

Розглянемо інший приклад. Припустимо, нам потрібні відомості про клієнтів, які розмістили замовлення. Ось як ми можемо це зробити за допомогою підзапиту:

```
SELECT customer_id, first_name FROM Customers
WHERE customer_id IN (  SELECT customer_id   FROM Orders );
```

Спочатку ми вибираємо *customer_id* з таблиці *Orders*, потім вибираємо рядки з таблиці *Customers*, у яких *customer_id* збігається з результатами підзапиту.

Підзапит та оператор JOIN

У деяких сценаріях ми можемо отримати однакові результати, використовуючи як підзапит, так і оператор JOIN. Наприклад, результат виконання наступної SQL-команди:

```
SELECT DISTINCT Customers.customer_id, Customers.first_name
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer_id
 ORDER BY Customers.customer_id;
```

рівнозначний результату виконання

```
SELECT customer_id, first_name FROM Customers
WHERE customer_id IN (   SELECT customer_id   FROM Orders);
```

Примітка: Рекомендується використовувати оператор `JOIN` замість підзапитів (наскільки це можливо), оскільки швидкість виконання операцій з `JOIN` вища і процес краще оптимізований, ніж під час використання підзапитів.

Оператори ANY та ALL

Оператор ANY порівнює значення першої таблиці з усіма значеннями другої таблиці та повертає рядок, якщо є збіг з будь-яким значенням. Наприклад, якщо ми хочемо знайти вчителів, вік яких аналогічний віку вказаного учня, ми можемо використати:

```
SELECT * FROM Teachers
WHERE age = ANY (   SELECT age   FROM Students );
```

Підзапит:

```
SELECT age FROM Students
```

повертає всі дані віку з таблиці *Students*. А умова:

```
    WHERE age = ANY (...)
```

порівнює вік учнів (повертається від підзапиту) із віком вчителя. За наявності збігу вибирається відповідний рядок таблиці *Teachers*.

Оператор ALL

Оператор **ALL** порівнює значення першої таблиці з усіма значеннями другої таблиці та повертає рядок, якщо є збіг з усіма значеннями. Наприклад, якщо ми хочемо знайти вчителів, вік яких більше, ніж у всіх учнів, ми можемо використати:

```
SELECT * FROM Teachers
WHERE age > ALL (  SELECT age  FROM Students);
```

Підзапит:

```
SELECT age FROM Students
```

повертає всі дані віку з таблиці *Students*. А умова:

```
    WHERE age > ALL (...)
```

порівнює вік учнів (повертається від підзапиту) із віком вчителя. Якщо вік вчителя більший, ніж вік усіх учнів, вибирається відповідний рядок таблиці *Teachers*.

Оператори ANY та ALL з операторами порівняння

Ми можемо використовувати будь-які оператори порівняння, такі як `=`, `>`, `<` та інші, з операторами `ANY` та `ALL`. Розглянемо приклад, коли нам потрібні вчителі, вік яких менше, ніж у будь-якого учня:

```
SELECT *  FROM Teachers
    WHERE age < ANY (  SELECT age   FROM Students );
```

Тут ми вибираємо рядки з таблиці *Teachers*, якщо вік (*age*) у **зовнішньому запиті** менший за будь-який вік у **підзапиті**.

Перевірка умов з CASE

Оператор **CASE** у SQL використовується для перевірки умов та виконання операцій над даними. Синтаксис **виразу CASE** завжди починається з ключового слова `CASE` і закінчується ключовим словом `END`, за яким слідує псевдонім (**оператор AS**) імені стовпця. Наприклад:

```
SELECT customer_id, first_name,
CASE
       WHEN age >= 18 THEN 'Allowed'
END AS can_vote
FROM Customers;
```

Тут ми перевіряємо кожен рядок на відповідність заданому виразу `CASE` та виконуємо додаткову дію у разі відповідності. Якщо вік (*age*) більше або дорівнює 18, то вибираються відповідні рядки *customer_id* та *first_name* і додається додатковий стовпець *can_vote* зі значенням `Allowed`.

Розглянемо інший приклад. Припустимо, ми хочемо надати клієнтам знижку 10% на кожне замовлення на різдвяному розпродажі, якщо сума перевищує 400.

```
SELECT order_id, customer_id,
CASE
    WHEN amount >= 400 THEN (amount - amount * 10/100)
END AS offer_price
 FROM Orders;
```

Тут ми перевіряємо, чи сума замовлення (*amount*) більша або дорівнює 400. Якщо ця умова виконана, додається новий стовпець *offer_price*, що містить суми замовлень з урахуванням 10% знижки (формула `amount - amount*10/100`).

Також можна використовувати відразу декілька умов усередині одного виразу `CASE`.

```
SELECT customer_id, first_name,
CASE
    WHEN country = 'USA' THEN 'United States of America'
    WHEN country = 'UK' THEN 'United Kingdom'
END AS country_name
FROM Customers;
```

Тут ми вибираємо стовпці *customer_id*, *first_name* та додаємо стовпець з ім'ям *country_name*. Значенням *country_name* буде:

United States of America, якщо країна (*поле country*) — **USA**.
United Kingdom, якщо країна (*поле country*) — **UK**.

CASE з ELSE

Оператор **CASE** може мати необов'язкову частину ELSE.
Частина **ELSE** виконується, якщо жодна з умов в операторі `CASE` не виконується.
Наприклад:

```
SELECT customer_id, first_name,
CASE
    WHEN country = 'USA' THEN 'United States of America'
    WHEN country = 'UK' THEN 'United Kingdom'
    ELSE 'Unknown Country'
END AS country_name
FROM Customers;
```

Тут ми вибираємо стовпці *customer_id*, *first_name* та додаємо стовпець з ім'ям *country_name*. Значенням *country_name* буде:

United States of America, якщо країна (*поле country*) — **USA**;
United Kingdom, якщо країна (*поле country*) — **UK**;
Unknown Country, якщо країною (*поле country*) не є ні USA, ні UK (спрацьовує частина `ELSE`).

Оператор EXISTS

Оператор **EXISTS** виконує зовнішній запит SQL, якщо внутрішній запит (**підзапит**) не повертає `NULL`. Принцип роботи оператора `EXISTS`:

1. Запускається зовнішній запит
2. Запускається підзапит
3. Якщо підзапит повертає NULL, то результат виконання зовнішнього запиту ігнорується
4. Якщо підзапит повертає дані, то результат виконання зовнішнього запиту відображається

Цей процес повторюється для кожного рядка зовнішнього запиту.

Розглянемо приклад. Припустимо, нам потрібно показати всіх клієнтів, які здійснили замовлення. У підзапиті ми перевіряємо наявність зробленого замовлення клієнта (по полю *customer_id*) і якщо це підтверджується, то в результаті виводимо ідентифікатор та ім'я клієнта.

```
SELECT customer_id, first_name FROM Customers
WHERE EXISTS (   SELECT order_id   FROM Orders
  WHERE Orders.customer_id = Customers.customer_id );
```

Оператор NOT EXISTS

Ми також можемо використовувати **оператор NOT** для інвертування роботи оператора `EXISTS`. Команда SQL виконується, якщо підзапит повертає порожній результат (тобто **NULL-значення**). Наприклад, виконаємо попередню SQL-команду, але вже з оператором `NOT EXISTS`:

```
SELECT customer_id, first_name  FROM Customers
WHERE NOT EXISTS (   SELECT order_id   FROM Orders
  WHERE Orders.customer_id = Customers.customer_id );
```

Тут ми виводимо всіх клієнтів, які НЕ зробили замовлення.

Приклади використання оператора EXISTS

Видалення (DROP) таблиці, якщо вона існує (EXISTS)

Ми можемо видалити таблицю (команда `DROP`), якщо вона вже існує (команда `IF EXISTS`). Наприклад:

```
DROP TABLE IF EXISTS my_table;
```

Створити (CREATE) таблицю, якщо вона не існує (NOT EXISTS)

Ми можемо створити таблицю (команда `CREATE`), якщо вона не існує (команда `IF NOT EXISTS`). Наприклад:

```
CREATE TABLE IF NOT EXISTS Companies (
  id int,
  name varchar(50),
  address text,
  email varchar(50),
  phone varchar(10)
);
```

Ще один приклад.
Наступна SQL-команда вибирає всі замовлення з таблиці *Orders* клієнтів віком від 23 років.

```
SELECT * FROM Orders
```

```
WHERE EXISTS (
  SELECT custom er_id
  FROM Customers
  WHERE Orders.customer_id = Customers.customer_id
  AND Customers.age > 23
);
```

на Зміст

6. Об’єднання двох таблиць JOIN

Оператор **JOIN** об’єднує дві таблиці на основі спільного стовпця і вибирає записи зі співпадаючими значеннями в цих стовпцях. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
JOIN Orders
 ON Customers.customer_id = Orders.customer;
```

Тут ми вибираємо стовпці ***customer_id*** та ***first_name*** (з таблиці *Customers*) та стовпець ***amount*** (з таблиці *Orders*). В результаті отримуємо ті рядки, в яких є збіг між ***customer_id*** (таблиці *Customers*) та ***customer*** (таблиці *Orders*).

Типи JOIN

Оператор `JOIN`, яку ми виконали вище, називається `INNER JOIN`. Існує 4 типи оператора `JOIN`:

INNER JOIN (те ж саме, що і `JOIN`)
LEFT JOIN
RIGHT JOIN
FULL OUTER JOIN

Оператор JOIN та псевдоніми

Ми можемо використовувати **псевдоніми (оператор AS)** з іменами таблиць, щоб зробити код коротшим і чистішим. Наприклад:

```
SELECT C.customer_id, C.first_name, O.amount
FROM Customers AS C
JOIN Orders AS O ON C.customer_id = O.customer;
```

Крім того, ми також можемо тимчасово змінити імена стовпців, використовуючи псевдоніми. Наприклад:

```
SELECT C.customer_id AS cid, C.first_name AS name, O.amount FROM
Customers AS C
JOIN Orders AS O
 ON C.customer_id = O.customer;
```

Цей фрагмент коду працює так само, як і попередній.

Оператор INNER JOIN

Оператор **INNER JOIN** об’єднує дві таблиці на основі спільного стовпця і вибирає записи зі співпадаючими значеннями в цих стовпцях. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Ось як працює цей код:

Ми вибираємо стовпці ***customer_id*** та ***first_name*** (з таблиці *Customers*) та стовпець ***amount*** (з таблиці *Orders*). В результате отримуємо ті рядки, в яких є збіг між ***customer_id*** (таблиці *Customers*) та ***customer*** (таблиці *Orders*).

Синтаксис оператора INNER JOIN

Синтаксис оператора `INNER JOIN` наступний:

```
SELECT стовпці
FROM таблиця1
INNER JOIN таблиця2
ON таблиця1.ім'я_стовпця = таблиця2.ім'я_стовпця;
```

Оператор INNER JOIN з оператором WHERE

Ось приклад використання оператора `INNER JOIN` з оператором **WHERE**:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Тут ми об’єднуємо дві таблиці та вибираємо рядки, у яких сума (*amount*) більше або дорівнює 500.

Оператор INNER JOIN з псевдонімами

Ми можемо використовувати **псевдоніми (оператор AS)** з оператором `INNER JOIN`, щоб зробити код коротшим та чистішим. Наприклад:

```
SELECT C.cat_name, P.prod_title FROM Categories AS C
INNER JOIN Products AS P
ON C.cat_id = P.cat_id;
```

Тут ми вибираємо спільні рядки між таблицями *Categories* та *Products*.

Оператор INNER JOIN з трьома таблицями

Ми також можемо об’єднати більше двох таблиць, використовуючи оператор `INNER JOIN`. Наприклад:

```
SELECT C.customer_id, C.first_name, O.amount, S.status
FROM Customers AS C
INNER JOIN Orders AS O
ON C.customer_id = O.customer
INNER JOIN Shippings AS S
ON C.customer_id = S.customer;
```

Тут ми:
об’єднуємо таблиці *Customers* та *Orders* на основі **customer_id**;
об’єднуємо таблиці *Customers* та *Status* на основі **customer_id**.
Команда повертає ті рядки, у яких є збіг між значеннями стовпців в обох умовах об’єднання.

Примітка: Для запуску цієї команди у кожній таблиці має бути спільний стовпець *customer_id* (або *customer*).

Оператор LEFT JOIN

Оператор **LEFT JOIN** об’єднує дві таблиці на основі спільного стовпця і вибирає не тільки записи зі співпадаючими значеннями в цих стовпцях, але і всі інші рядки з лівої таблиці. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
LEFT JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Ось як працює цей код:
Тут ми вибираємо стовпці ***customer_id*** та ***first_name*** (з таблиці *Customers*) та стовпець ***amount*** (з таблиці *Orders*). В результате отримуємо ті рядки, в яких є збіг між ***customer_id*** (таблиці *Customers*) та ***customer*** (таблиці *Orders*) разом зі **всіма іншими рядками з (лівої) таблиці** *Customers*.

Синтаксис оператора LEFT JOIN

Синтаксис оператора `LEFT JOIN` наступний:

```
SELECT стовпці
FROM таблиця1
LEFT JOIN таблиця2
 ON таблиця1.назва_стовпця = таблиця2.назва_стовпця;
```

Оператор LEFT JOIN з оператором WHERE

Ось приклад використання оператора `LEFT JOIN` з оператором **WHERE**:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
LEFT JOIN Orders
ON Customers.customer_id = Orders.customer
 WHERE Orders.amount >= 500;
```

Тут ми об'єднуємо дві таблиці та вибираємо рядки, у яких сума (*amount*) більше або дорівнює 500.

Оператор LEFT JOIN з псевдонімами

Ми можемо використовувати **псевдоніми (оператор AS)** з оператором `LEFT JOIN`, щоб зробити код коротшим та чистішим. Наприклад:

```
SELECT C.cat_name, P.prod_title
FROM Categories AS C
LEFT JOIN Products AS P
ON C.cat_id = P.cat_id;
```

Тут ми вибираємо спільні рядки між таблицями *Categories* і *Products*.

Оператор RIGHT JOIN

Оператор **RIGHT JOIN** об'єднує дві таблиці на основі спільного стовпця і вибирає не тільки записи зі співпадаючими значеннями в цих стовпцях, але і всі інші рядки **з правої таблиці**. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
RIGHT JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Ось як працює цей код:
Тут ми вибираємо стовпці ***customer_id*** та ***first_name*** (з таблиці *Customers*) та стовпець ***amount*** (з таблиці *Orders*). В результате отримуємо ті рядки, в яких є збіг між ***customer_id*** (таблиці *Customers*) та ***customer*** (таблиці *Orders*) разом **зі всіма іншими рядками з (*правої*) таблиці** *Orders*.

Синтаксис оператора RIGHT JOIN

Синтаксис оператора `RIGHT JOIN` наступний:

```
SELECT стовпці
FROM таблиця1
RIGHT JOIN таблиця2
 ON таблиця1.назва_стовпця = таблиця2.назва_стовпця;
```

Оператор RIGHT JOIN з оператором WHERE

Приклад використання оператора `RIGHT JOIN` з оператором **WHERE**:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
RIGHT JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Тут ми об'єднуємо дві таблиці та вибираємо рядки, у яких сума (*amount*) більше або дорівнює 500.

Оператор RIGHT JOIN з псевдонімами AS

Ми можемо використовувати **псевдоніми (оператор AS)** з оператором `RIGHT JOIN`, щоб зробити код коротшим та чистішим. Наприклад:

```
SELECT C.cat_name, P.prod_title
FROM Categories AS C
RIGHT JOIN Products AS P
ON C.cat_id = P.cat_id;
```

Тут ми вибираємо спільні рядки між таблицями *Categories* та *Products*.

Оператор FULL OUTER JOIN

Оператор **FULL OUTER JOIN** об'єднує дві таблиці на основі спільного стовпця і вибирає **всі рядки з обох таблиць**. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
FULL OUTER JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Ось як працює цей код:
Тут ми вибираємо стовпці ***customer_id*** та ***first_name*** (з таблиці *Customers*) та стовпець ***amount*** (з таблиці *Orders*). В результате отримуємо ті рядки, в яких є збіг між ***customer_id*** (таблиці *Customers*) та ***customer*** (таблиці *Orders*) разом **зі всіма іншими рядками з обох таблиць**.

Синтаксис оператора FULL OUTER JOIN

Синтаксис оператора `FULL OUTER JOIN` наступний:

```
SELECT стовпці
```

```
FROM таблиця1
FULL OUTER JOIN таблиця2
 ON таблиця1.назва_стовпця = таблиця2.назва_стовпця;
```

Оператор FULL OUTER JOIN з оператором WHERE

Ось приклад використання оператора `FULL OUTER JOIN` з **оператором WHERE**:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
FULL OUTER JOIN Orders
ON Customers.customer_id = Orders.customer
WHERE Orders.amount >= 500;
```

Тут ми об'єднуємо дві таблиці та вибираємо рядки, у яких сума (*amount*) більше або дорівнює 500.

Оператор FULL OUTER JOIN з псевдонімами

Ми можемо використовувати псевдоніми (оператор AS) з оператором FULL OUTER `JOIN`, щоб зробити код коротшим та чистішим. Наприклад:

```
SELECT C.cat_name, P.prod_title
FROM Categories AS C
FULL OUTER JOIN Products AS P
 ON C.cat_id = P.cat_id;
```

Тут ми вибираємо спільні рядки між таблицями *Categories* та *Products*.

на Зміст

7. Копіювання записів таблиці

Команда SELECT INTO

Команда **SELECT INTO** використовується для копіювання записів з однієї таблиці до ***нової*** таблиці. Ця команда створює саме нову таблицю. Якщо в базі даних вже є таблиця з такою назвою, то SQL видасть помилку. Наприклад:

```
SELECT * INTO CustomersCopy FROM Customers;
```

Тут ми копіюємо всі дані з таблиці *Customers* до нової таблиці *CustomersCopy*.

Примітка: Якщо потрібно скопіювати дані до існуючої таблиці (а не створювати нову таблицю), слід використовувати **команду INSERT INTO SELECT**.

Копіювання вибраних стовпців

Можна скопіювати лише вибрані стовпці зі старої таблиці до нової. Наприклад:

```
SELECT customer_id, country INTO CustomersCopy FROM Customers;
```

Тут ми копіюємо лише стовпці *customer_id* та *country* у таблицю *CustomersCopy*.

Копіювання дані, що відповідають умові

Ми можемо використовувати **оператор WHERE** з `SELECT INTO` для копіювання тих рядків, які відповідають вказаній умові. Наприклад:

```
SELECT customer_id, age INTO JapanCustomersAge
FROM Customers WHERE country = 'Japan';
```

Тут ми створюємо таблицю *JapanCustomersAge* зі стовпцями *customer_id* та *age*, а потім копіюємо рядки в нову таблицю, якщо значенням стовпця *country* є `Japan`.

Копіювання дані до іншої бази даних

За замовчуванням `SELECT INTO` створює нову таблицю в поточній базі даних. Якщо потрібно скопіювати дані до таблиці в іншій базі даних, слід використовувати **оператор IN**. Наприклад:

```
SELECT * INTO CustomersCopy IN another_db.mdb FROM Customers;
```

Тут ми копіюємо таблицю *Customers* в таблицю *CustomersCopy* в базі даних *another_db.mdb*.

Примітка: Користувач повинен мати **права** для копіювання даних до таблиці з іншої бази даних.

Копіювання дані з двох таблиць в одну

Можна скопіювати дані з двох різних таблиць у нову таблицю, використовуючи **оператор JOIN** з `SELECT INTO`. Наприклад:

```
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
INTO CustomerOrders FROM Customers
JOIN Orders ON Customers.customer_id = Orders.customer_id;
```

Тут ми копіюємо **customer_id** та **first_name** з таблиці *Customers* та **amount** з таблиці *Orders* в нову таблицю *CustomerOrders*.

Копіювання лише структури таблиці

Ми також можемо використовувати оператор `SELECT INTO` для створення нової таблиці із заданою структурою (без копіювання даних). Для цього слід використовувати оператор `WHERE` із значенням `false`.

```
SELECT * INTO NewCustomers FROM Customers WHERE false;
```

Тут ми створюємо порожню таблицю з ім'ям *NewCustomers* із тією самою структурою, яку має таблиця *Customers*.

INSERT INTO SELECT

Команда **INSERT INTO SELECT** використовується для копіювання записів з однієї таблиці до іншої ***існуючої*** таблиці. Наприклад:

```
INSERT INTO OldCustomers SELECT * FROM Customers;
```

Тут ми копіюємо всі записи з таблиці *Customers* до таблиці *OldCustomers*.

Примітка: Щоб запустити вищезазначений SQL-запит, у базі даних вже повинна бути таблиця з ім'ям *OldCustomers*, а імена стовпців таблиці *OldCustomers* і таблиці *Customers* повинні збігатися. Якщо потрібно скопіювати дані до ***нової*** таблиці (не до існуючої), слід використовувати **команду SELECT INTO**.

Копіювання лише вибраних стовпців

Також можна скопіювати лише вибрані стовпці з однієї таблиці до іншої. Наприклад:

```
INSERT INTO OldCustomers(customer_id, age) SELECT customer_id,
age
FROM Customers;
```

Тут ми копіюємо дані тільки зі стовпців *customer_id* та *age* до таблиці *OldCustomers*.

Примітка: Якщо в таблиці *OldCustomers*, крім *customer_id* та *age*, є інші стовпці, то значеннями цих стовпців буде **NULL**.

Копіювання даних, що відповідають умові

Ми можемо використати оператор `WHERE` з `INSERT INTO` для копіювання рядків, які відповідають вказаній умові. Наприклад:

```
INSERT INTO OldCustomers SELECT * FROM Customers WHERE country =
'USA';
```

Тут ми копіюємо всі рядки з таблиці *Customers*, де значенням стовпця *country* є `USA`, в таблицю *OldCustomers*.

Копіювання даних з двох таблиць в одну

Можна скопіювати дані з двох різних таблиць в одну, використовуючи **оператор JOIN** з `INSERT INTO SELECT`. Наприклад:

```
INSERT INTO OldCustomerOrders
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
JOIN Orders
ON Customers.customer_id = Orders.customer_id;
```

Тут ми копіюємо **customer_id** та **first_name** з таблиці *Customers* та **amount** з таблиці *Orders* в існуючу таблицю *OldCustomerOrders*.

Примітка: Якщо в існуючій таблиці вже є дані, то будуть додані нові рядки. Стовпці в існуючій таблиці можуть видавати такі помилки, як `NOT NULL Constraint Fail`, `UNIQUE Constraint Failed` під час копіювання даних.

Уникайте дублювання даних в INSERT INTO SELECT

Якщо вже є рядок зі схожим значенням, SQL може видати помилку при використанні команди `INSERT INTO SELECT`. Проте ми можемо пропустити копіювання повторюваних рядків, використовуючи **оператор NOT EXISTS**. Наприклад:

```
INSERT INTO OldCustomers(customer_id, age)
SELECT customer_id, age
FROM Customers
WHERE NOT EXISTS(
  SELECT customer_id
  FROM OldCustomers
  WHERE OldCustomers.customer_id = Customers.customer_id
);
```

Тут ми скопіюємо рядок у таблицю лише в тому випадку, якщо *customer_id* у таблиці не має такого ж значення, яке ми намагаємося скопіювати.

на Зміст

8. Коментарі в SQL

Коментарі — це рядки тексту в коді, які допомагають усім, хто його читає, краще зрозуміти, що й навіщо робить код. Коментарі повністю ігноруються (тобто не виконуються) системами управління базами даних (СУБД).

Однорядкові коментарі

У SQL подвійне тире `--` використовується для написання однорядкового коментаря. Наприклад:

```
-- Отримання всіх даних з таблиці Students
SELECT *
FROM Students;
```

СУБД повністю ігнорують цей рядок під час виконання коду.

Коментарі на одному рядку з кодом

Також можна писати коментарі на одному рядку із SQL-командами. Наприклад:

```
SELECT * -- вибираємо всі дані
FROM Students; -- з таблиці Students
```

Тут коментарями є:
`-- вибираємо всі дані`
`-- з таблиці Students`

Багаторядкові коментарі

В SQL багаторядкові коментарі починаються з `/*` і закінчуються `*/`. Наприклад:

```
/* Вибираємо всі дані
з таблиці Students */
SELECT *
FROM Students;
```

Тут все, що знаходиться між `/*` та `*/`, є коментарем та ігнорується СУБД.

Коментарі всередині рядків коду

Подібно до однорядкових коментарів, багаторядкові коментарі можна поміщати безпосередньо в рядки SQL-коду. Наприклад:

```
SELECT *
FROM /* тут ім'я таблиці */ Students;
SQL-код вище рівнозначний:
SELECT *
FROM Students;
```

Використання коментарів у налагодженні коду

Припустимо, ми хочемо не виконувати певні рядки SQL-коду. У таких випадках замість видалення цих рядків ми можемо їх просто закоментувати. Це допомагає в **тестуванні** та **налагодженні**. Наприклад:

```
/* SELECT *
FROM Customers; */
-- Ці 2 рядки коду вище ігноруються СУБД
SELECT *
 FROM Students;
```

Тут ми отримаємо дані лише з таблиці *Students*.

на Зміст

9. Представлення (VIEW)

В SQL **представлення (view)** містить рядки і стовпці, аналогічні таблиці, проте без фактичних даних. Представлення можна розглядати як **віртуальну таблицю**, створену з однієї або кількох таблиць, щоб спростити роботу з даними.

Створення представлення

Представлення в SQL можна створювати за допомогою **команди CREATE VIEW**. Наприклад:

```
CREATE VIEW us_customers AS
SELECT customer_id, first_name
FROM Customers
WHERE Country = 'USA';
```

Тут представлення з ім'ям *us_customers* створюється з таблиці *Customers*. Тепер, щоб вибрати клієнтів, які живуть у `USA`, ми можемо просто написати:

```
SELECT * FROM us_customers;
```

Оновлення представлення

Можна змінити або оновити існуюче представлення за допомогою команди **CREATE OR REPLACE VIEW**. Наприклад:

```
CREATE OR REPLACE VIEW us_customers AS
SELECT *
FROM Customers
WHERE Country = 'USA';
```

Тут ми оновили представлення *us_customers*, щоб отримати всі поля таблиці *Customers*.

Видалення представлення

Ми можемо видалити представлення за допомогою **команди DROP VIEW**. Наприклад:

```
DROP VIEW us_customers;
```

Тут ми видаляємо представлення з ім'ям *us_customers*.

Примітка: Якщо представлення на момент виконання команди видалення не існує, то SQL видасть помилку.

Представлення для складних запитів

Припустимо, що **A** та **B** — це дві таблиці, з яких ми хочемо отримати дані. Для цього ми можемо використати **оператори JOIN**. Однак використання `JOIN` щоразу може втомлювати. В якості альтернативи можна створити представлення для простого вилучення даних.

Створимо представлення:

```
CREATE VIEW order_details AS
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
JOIN Orders
ON Customers.customer_id = Orders.customer_id;
```

Тепер, щоб отримати дані, ми можемо написати:

```
SELECT * FROM order_details;
```

Тут ми отримуємо дані з представлення *order_details*.

на Зміст

10. Збережені процедури

Збережена процедура в SQL є набором команд, які виконують певні дії. Збережені процедури схожі на **функції в програмуванні** і створюються для того, щоб можна було повторно використовувати набір команд. Вони можуть приймати параметри та виконувати операції, коли ми їх викликаємо.

Створення збереженої процедури

Збережена процедура створюється за допомогою команди **CREATE PROCEDURE**, за якою слідує необхідний набір SQL-команд. Наприклад:

```
DELIMITER //
CREATE PROCEDURE us_customers ()
BEGIN
SELECT customer_id, first_name
FROM Customers
WHERE Country = 'USA';
END //
 DELIMITER ;
```

Виконання збереженої процедури - CALL

Тепер, якщо нам потрібно отримати всіх клієнтів, які живуть у США, ми можемо просто викликати збережену процедуру, яку написали раніше. Наприклад:

```
CALL us_customers();
```

Параметризована збережена процедура

Ми можемо передавати власні дані в збережені процедури так, щоб один і той же набір SQL-команд працював по-різному для різних даних. Припустимо, ми хочемо отримати рядки, в яких стовпці *country* є значення `USA`. Наш запит виглядатиме так:

```
SELECT *
FROM Customers
WHERE country = 'USA';
```

І знову, якщо ми хочемо отримати рядки, в яких у стовпці *country* є значення `UK`, ми виконаємо наступне:

```
SELECT *
FROM Customers
 WHERE country = 'UK';
```

Зверніть увагу, що в цих двох прикладах все те саме, за винятком значення для пошуку в стовпці *country*.

Таким чином, замість повторного написання одного і того ж коду, ми можемо створити збережену процедуру і просто викликати її з різними значеннями. Наприклад:

```
DELIMITER //
CREATE PROCEDURE ctr_customers (ctr VARCHAR(50))
BEGIN
SELECT customer_id, first_name
FROM Customers
WHERE Country = ctr;
END //
DELIMITER ;
```

Тут `ctr` — параметр, який нам потрібно вказати при виклику збереженої процедури. Наприклад:

```
-- Викликаємо збережену процедуру, вказавши аргумент 'USA'
CALL ctr_customers ('USA');
 -- Викликаємо ту ж збережену процедуру, але тепер з аргументом
'UK'
CALL ctr_customers ('UK');
```

Параметризовані процедури

Збережена процедура також може приймати декілька параметрів. Наприклад:

```
-- Створюємо збережену процедуру, яка приймає 2 параметри:
cus_id та max_amount
DELIMITER //
CREATE PROCEDURE order_details (cus_id INT, max_amount INT)
BEGIN
SELECT Customers.customer_id, Customers.first_name,
Orders.amount
FROM Customers
JOIN Orders
ON Customers.customer_id = Orders.customer_id
where Customers.customer_id = cus_id AND Orders.amount <
max_amount;
END //
DELIMITER ;
```

Тепер, щоб викликати збережену процедуру, нам потрібно всього лише написати:

```
CALL order_details (4, 400);
```

Тут ми передаємо два аргументи в процедуру.

Видалення збереженої процедури

Ми можемо видалити збережену процедуру за допомогою команди **DROP PROCEDURE**. Наприклад:

```
DROP PROCEDURE order_details;
```

Тут ми видаляємо збережену процедуру, яку створили раніше.

на Зміст

11. SQL-ін'єкції

SQL-ін'єкція — це метод отримання несанкціонованого доступу (різновид злому) до бази даних, при якому шкідливий код виконується прямо з поля вводу звичайної форми.

Якщо SQL-ін'єкція пройшла успішно, неавторизовані особи зможуть читати, створювати, оновлювати або навіть видаляти записи з таблиць бази даних. Цей метод використовується хакерами, пентестерами, тестувальниками тощо.

Приклад 1. SQL-ін'єкція з використанням декількох стейтментів

Припустимо, у нас на сайті є форма для пошуку товарів по ID. Фрагмент PHP-коду пошуку товару виглядатиме приблизно так:

```
$prod_id = $_GET["prod_id"];
$sql = "SELECT * FROM Products WHERE product_id = " . $prod_id;
```

Якщо користувач введе значення `20`, то SQL інтерпретує це як:

```
SELECT * FROM Products WHERE product_id = 20
```

Нічого підозрілого, правда? Але якщо користувач введе `20; DROP TABLE Products;`? Давайте подивимося, як SQL інтерпретує це:

```
SELECT * FROM Products WHERE product_id = 20; DROP TABLE
Products;
```

Ця команда видалить таблицю *Products* з бази даних. Це стало можливим, оскільки більшість систем управління базами даних (СУБД) можуть виконувати декілька команд одночасно.

Приклад 2. SQL-ін'єкція з використанням завжди істинної умови

Інший спосіб виконати SQL-ін’єкцію — це передати умову, результат якої завжди true, щоб дані завжди витягувалися, незважаючи ні на що.

Розглянемо інший фрагмент PHP-коду, де у нас є форма входу на наш веб-сайт, і користувачам потрібно вказати свій логін та пароль.

```
$username = $_POST["username"];
$password = $_POST["password"];
 $sql = "SELECT * FROM Users WHERE username = \"" . $username .
"\" AND password = \"" . $password . "\"";
```

Якщо користувач введе логін як `root` і пароль як `pass`, то SQL інтерпретує це як:

```
SELECT * FROM Users WHERE username = "root" AND password =
"pass"
```

Цей фрагмент коду виглядає нормально, коли користувач вводить коректні дані.

Але що робити, якщо користувач введе ім’я користувача як `invalid_user" OR "1"="1` і пароль як `invalid_pass" OR "1"="1`? Подивимось, як SQL відреагує на це:

```
SELECT * FROM Users WHERE username = "invalid_user" OR "1"="1"
AND password = "invalid_pass" OR "1"="1"
```

Оскільки `"1"="1"` завжди `true`, незалежно від того, який логін та пароль введе користувач, SQL витягне всіх користувачів з бази даних.

Захист від SQL-ін'єкцій

Спосіб 1. Валідація користувацького вводу

Завжди слід перевіряти дані, які вводить користувач, перед їх фактичною відправкою до бази даних. Хорошими практиками є видалення пробілів, парсинг спеціальних символів, обмеження розміру вводу тощо.
Наприклад:

```
$data = $_POST["name"];
$data = trim($data);
$data = stripslashes($data);
$data = htmlspecialchars($data);
```

Цей фрагмент PHP-коду певною мірою “перевіряє” вхідні дані.

Спосіб 2. Використання ORM

ORM (скор. від *“**O**bject **R**elational **M**apping”*) — це інструмент, який конвертує SQL-команди в код мови програмування та навпаки.

При використанні ORM здебільшого не потрібно писати “чистий” SQL-код. Оскільки ORM розроблені з урахуванням хороших практик та протоколів безпеки, вони надають захист, залишаючись при цьому простими у використанні.
Наприклад, наступний SQL-код:

```
SELECT * FROM Users WHERE id = 5;
```

виглядатиме як

```
select(Users).where(Users.id == 5)
```

в SQLAlchemy ORM для Python-кода.

Спосіб 3. Використання підготовлених запитів

Підготовлені запити — це SQL-код із плейсхолдерами. Передані аргументи просто розміщуються на місці плейсхолдерів.
Наприклад:

```
$sql = "INSERT INTO Users (first_name, last_name, email) VALUES
(?, ?, ?)";
mysqli_stmt_bind_param($sql, "sss", $first_name, $last_name,
$email);
$first_name = "Nagy";
$last_name = "János";
$email = "nagyjanos@mail.com";
mysqli_stmt_execute($stmt);
```

Тут передані значення поміщаються на місце `?` та структура SQL-коду зберігається.

SQL-ін'єкція — це дуже поширений спосіб злому. При використанні “сирого” SQL-коду, його слід ретельно **протестувати та перевірити**.

Ще одним альтернативним варіантом захисту при розробці програм/сайтів є використання фреймворків (таких як *Django*, *Laravel*, *ASP.NET* та ін.) замість того, щоб писати код з нуля. Фреймворки за замовчуванням обробляють SQL-ін'єкції, а також вирішують багато інших проблем.

на Зміст

Список рекомендованої літератури

1. **https://www.programiz.com/**
2. **https://ravesli.com/sql-tutorial/**
3. Michael J. Hernandez, John L. Viescas. SQL-lekérdezések földi halandóknak + CD – Gyakorlati útmutató az SQL nyelvű adatkezeléshez. – Budapest. – Kiskapu kiadó, 2009. – 568 old.
4. Lynn Beighley, Michael Morrison. Agyhullám: PHP & MySQL – Agytornásztató tanfolyam. – Budapest. – Kiskapu kiadó, 2011. – 811 old.
5. Гайдаржи В.І., Ізварін І.В. Бази даних в інформаційних системах: Підручник – Київ: Університет «Україна», 2018. – 417 с.
6. Павлиш В. А., Гліненко Л. К., Шаховська Н. Б. Основи інформаційних технологій і систем: Підручник; Національний університет «Львівська політехніка». – Львів: Вид-во Львівська політехніка, 2018. – 619 с.

на Зміст

ПРЕДМЕТНИЙ ПОКАЖЧИК
(КОМАНДИ, ФУНКЦІЇ, ОПЕРАТОРИ, ОБМЕЖЕННЯ)

на Зміст

Основи мови SQL/ SQL nyelv alapjai методичні вказівки до практичних занять з дисципліни «Системи керування базами даних » для здобувачів першого (бакалаврського) рівня вищої освіти денної та заочної форм навчання, освітня програма: «Середня освіта (Інформатика)», галузь знань: «01 Освіта/Педагогіка», спеціальність (спеціалізація): «014 Середня освіта (014.04 Інформатика)» / Розробник: Йожеф Головач. – Берегове: ЗУІ ім. Ф.Ракоці ІІ, 2024, 80 с. (українською мовою).

Метою методичного посібника є поглиблення знань студентів з дисципліни «Системи керування базами даних». У роботі наведені основні команди мови SQL, а також їх структури, які виникають при використанні різних опціональних параметрів. Можливості мови SQL ілюструються простими прикладами. В методичці наводяться приклади використання команд SQL для роботи з таблицями баз даних. Методичні вказівки можуть бути використані на практичних заняттях та при самостійній роботі по даному курсу, а також студентами-математиками.

Виробничо-практичне видання

Основи мови SQL/ SQL nyelv alapjai
Методичні вказівки до практичних занять з дисципліни
«Системи керування базами даних »
2024 р.

Затверджено до використання у навчальному процесі
на засіданні кафедри математики та інформатики ЗУІ ім. Ф.Ракоці ІІ
(протокол № 1 від 13 серпня 2024 року)

Розглянуто та рекомендовано Навчально-методичною радою
Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 22 від 26 серпня 2024 року)

Рекомендовано до видання в електронній формі (PDF)
рішенням Вченої ради Закарпатського угорського інституту імені Ференца Ракоці ІІ
(протокол № 7 від 27 серпня 2024 року)

Підготовлено до видання в електронній формі (PDF) кафедрою математики та інформатики
спільно з Видавничим відділом Закарпатського угорського інституту імені Ференца Ракоці ІІ

Розробник методичних вказівок:

Йожеф ГОЛОВАЧ – професор кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доктор технічних наук

Рецензенти:

Олександр МІЦА – завідувач кафедри інформаційних управляючих систем та технологій УжНУ, доктор техніічних наук, професор (м. Ужгород)

Мирослав СТОЙКА – доцент кафедри математики та інформатики Зака рпатського угорського інституту імені Ференца Ракоці ІІ, кандидат фізико-математичних наук, доцент

Відповідальні за випуск:

Каталін КУЧІНКА – завідувач кафедри математики та інформатики Закарпатського угорського інституту імені Ференца Ракоці ІІ, доцент, кандидат фізико-математичних наук

Олександр ДОБОШ – начальник Видавничого відділу ЗУІ ім. Ф.Ракоці ІІ

За зміст методичних вказівок відповідальність несе розробник.

Видавництво: Закарпатський угорський інститут імені Ференца Ракоці ІІ (адреса: пл. Кошута 6, м. Берегове, 90202. Електронна пошта: foiskola@kmf.uz.ua) *Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції Серія ДК7637 від 19 липня 2022 року*
Шрифт «Times New Roman». Розмір сторінок методичних вказівок: А4 (210х297мм).