

Міністерство освіти і науки України  
Національний університет «Запорізька політехніка»

**МЕТОДИЧНІ ВКАЗІВКИ  
до лабораторних робіт  
з дисципліни «Бази даних»  
«Робота з СКБД PostgreSQL»**

для студентів спеціальності  
121 «Інженерія програмного забезпечення»  
усіх форм навчання

PostgreSQL



Методичні вказівки до лабораторних робіт з дисципліни Бази даних. Робота з СКБД PostgreSQL для студентів спеціальності 121 «Інженерія програмного забезпечення» усіх форм навчання /Уклад.: Є. О. Гофман –Запоріжжя: НУ «Запорізька політехніка», 2024. – 31 с.

Укладач:	Є. О. Гофман, доцент, к.т.н.
Рецензент:	А. О. Олійник, професор, д.т.н.
Відповідальний за випуск:	С.О. Субботін, професор, д.т.н.

Затверджено  
на засіданні кафедри  
«Програмні засоби»  
Протокол № 5 від 06.12.2023р.

## ЗМІСТ

ЛАБОРАТОРНА РОБОТА № 1 «СТВОРЕННЯ СХЕМИ БАЗИ ДАНИХ» .....	4
ЛАБОРАТОРНА РОБОТА № 2 «ПРОЕКТУВАННЯ ЗАПИТІВ НА ВИБІРКУ ДАНИХ» .....	12
ЛАБОРАТОРНА РОБОТА № 3 «ПРОЕКТУВАННЯ СКЛАДНИХ ЗАПИТІВ НА ВИБІРКУ ТА МОДИФІКАЦІЮ ДАНИХ» .....	16
ЛАБОРАТОРНА РОБОТА № 4 «РОБОТА З ПОДАННЯМИ» .....	20
ЛАБОРАТОРНА РОБОТА № 5 «РОБОТА З ФУНКЦІЯМИ І ТРИГЕРАМИ» .....	23
ЛІТЕРАТУРА.....	31

## ЛАБОРАТОРНА РОБОТА № 1 «СТВОРЕННЯ СХЕМИ БАЗИ ДАНИХ»

### 1.1 Мета роботи

Метою роботи є ознайомлення з **системою керування базами даних PostgreSQL**, придбання початкових навичок роботи з нею і створення схеми бази даних.

### 1.2 Завдання до лабораторної роботи

1.2.1 Ознайомитися зі змістом пункту 1.3 даних методичних указівок.

1.2.2 Відповідно до індивідуального завдання розробити схему бази даних за допомогою інструментів **PgAdmin** та мови **SQL**.

1.2.3 Ввести дані.

### 1.3 Основні теоретичні відомості

При проектуванні бази даних треба визначити, яка саме інформація повинна входити до бази даних і чи повинна вся збережена в базі даних інформація розташовуватися в одній таблиці або її краще розділити на декілька таблиць.

#### 1.3.1 Відомості про PostgreSQL

**PostgreSQL** — широко розповсюджена **система керування базами даних (СКБД)** з відкритим початковим кодом. Прототип був розроблений в Каліфорнійському університеті Берклі в 1987 році під назвою POSTGRES, після чого активно розвивався і доповнювався.

**PostgreSQL** дозволяє виконувати програмний код безпосередньо сервером бази даних. Цей код може бути написаний на **SQL** — мові, що орієнтована на оброблення даних. Але гнучкішим буде код, написаний з використанням однієї із мов програмування, з якими PostgreSQL може працювати. До таких мов належать:

- Вбудована мова, яка зветься **PL/pgSQL**, подібна до процедурної мови **PL/SQL** компанії **Oracle**.
- Мови розробки сценаріїв: **PL/Perl**, **PL/Python**, **PL/Tcl**, **PL/Ruby**, **PL/sh**.
- Класичні мови програмування **C**, **C++**, **Java** (за допомогою PL/Java).

Зазначений програмний код може виконуватись із привілеями користувача.

#### 1.3.2 Інсталяція PostgreSQL

Посилання на інсталятор - <https://www.postgresql.org/download/windows/>

Інсталятор автоматично встановлює і налаштовує сервер **PostgreSQL**, а також GUI клієнт **PgAdmin**. В процесі інсталяції буде запропоновано визначити пароль суперкористувача для підключення до серверу баз даних.

#### 1.3.3 Інтерфейс PgAdmin

Інтерфейс **PgAdmin** включає 3 основні групи елементів:

1. Рядок меню – надає доступ до інструментів **PgAdmin**;
2. Дерево об'єктів (Object Explorer) – використовується для навігації між об'єктами серверу баз даних;
3. Робоча зона – область, що відображає перелік відкритих інструментів та дозволяє з ними працювати.

### 1.3.4 Створення баз даних

Запуск процедури створення бази даних (БД) починається з виклику контекстного меню натисканням ПКМ на відповідному об'єкті (рис. 1.3.1).

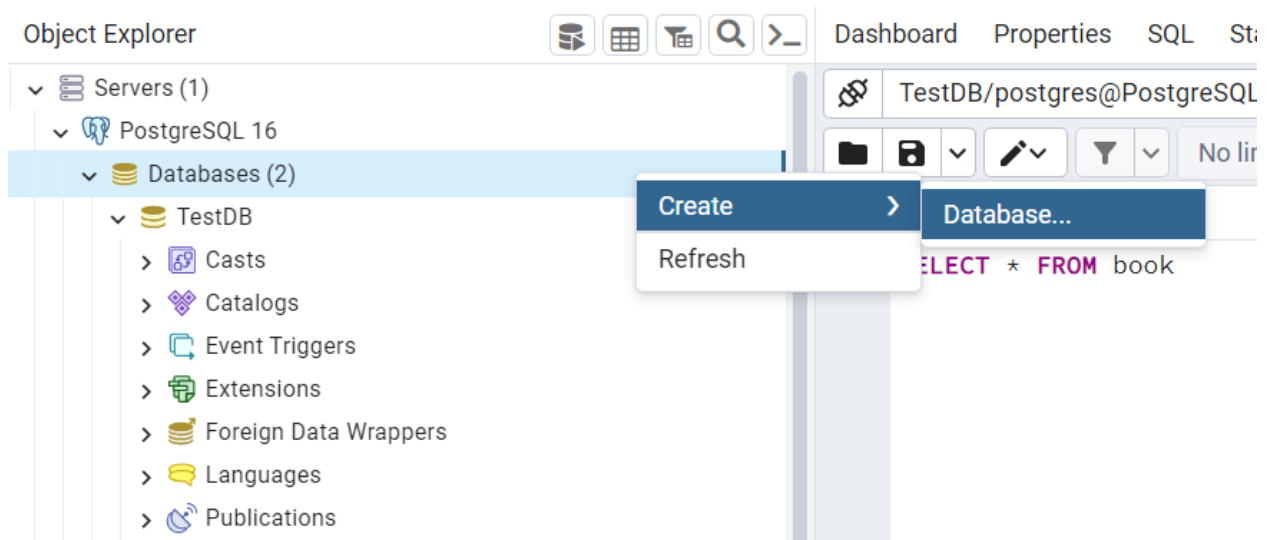


Рис. 1.3.1 – виклик вікна створення БД

*\* Аналогічно створюються та видаляються інші об'єкти БД: таблиці, колонки і т.д.*

Для створення 1-ї бази даних достатньо вказати лише ім'я, інші параметри мають прийнятні значення за замовченням.

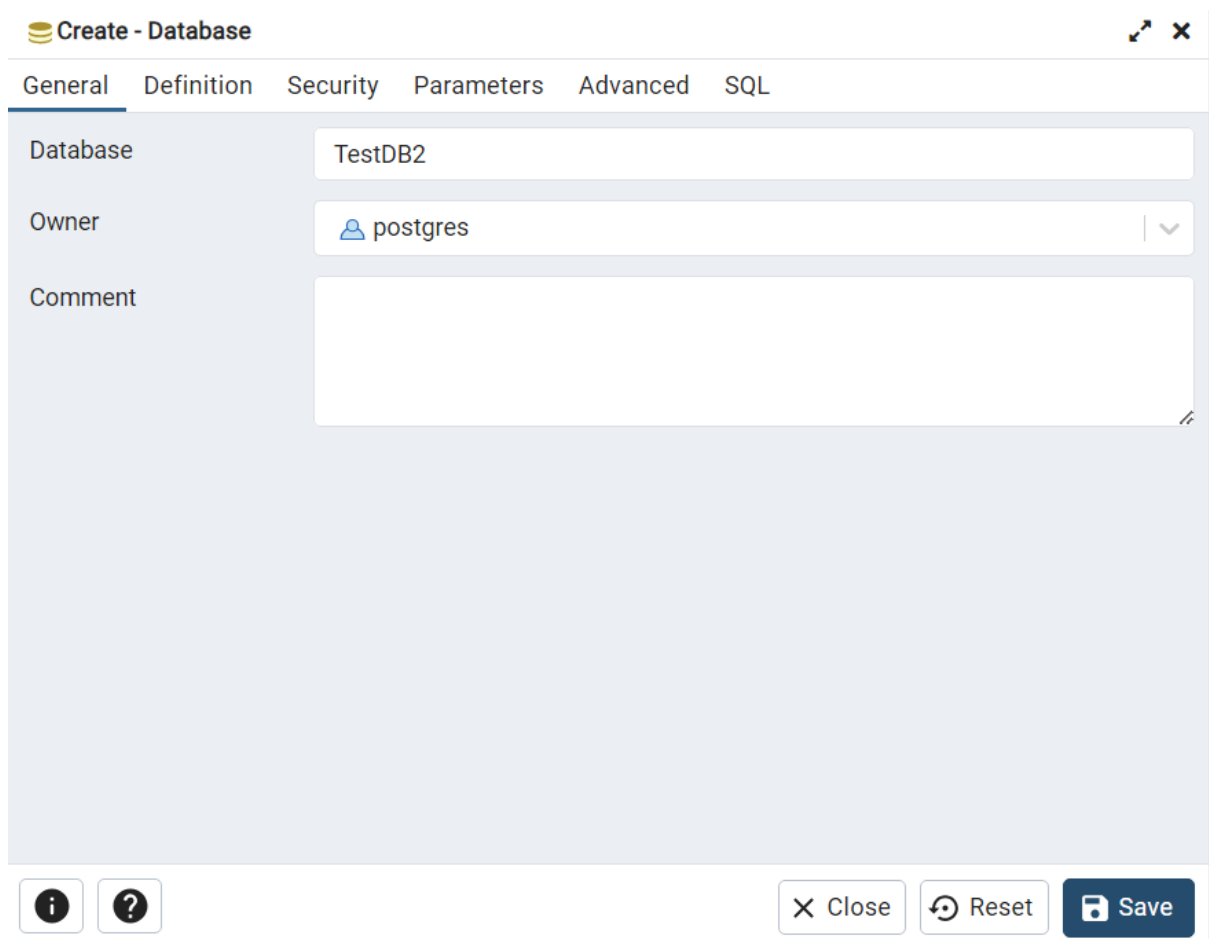


Рис. 1.3.2 – вікно створення БД

Переглянути текст SQL запиту на створення бази даних можна на останній вкладці діалогового вікна **Create – Database** (рис.1.3.3).

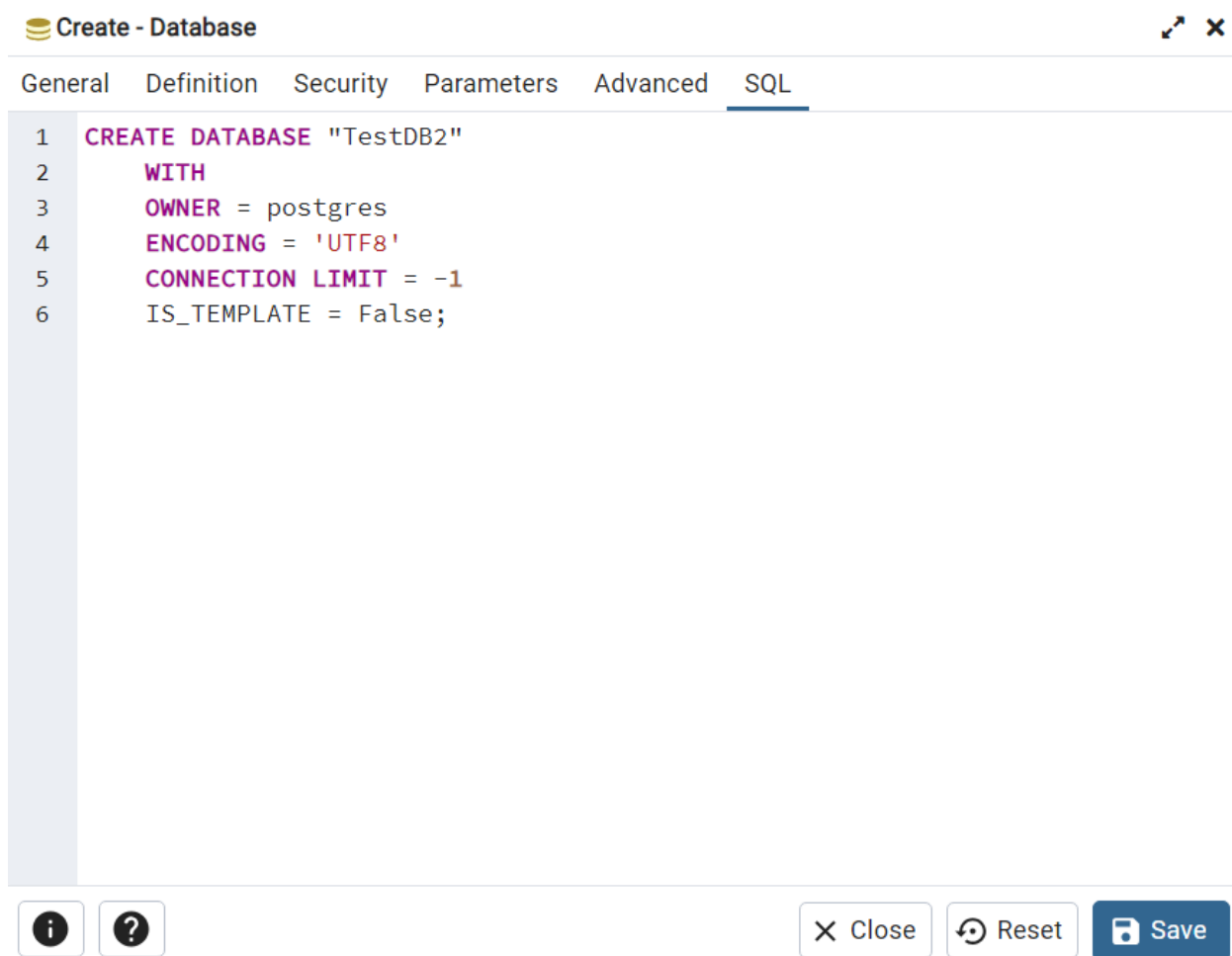


Рис. 1.3.3 – генератор SQL коду

*\* Цей код можна ввести безпосередньо у **Query Tool** (інструмент - інтерпретатор SQL команд) і так само буде створена база даних.*

*\*\* **Query Tool** можна використовувати також для створення та редагування інших об'єктів БД.*

### 1.3.5 Вимоги до даних у PostgreSQL

**PostgreSQL** підтримує реляційну модель даних, тому дані в ній зберігаються у вигляді 2-мірних зв'язаних таблиць.

Якщо казати спрощено, кожна таблиця зберігає дані про деяку сутність (наприклад: сутність клієнт).

В стовпчиках зберігаються різні дані про клієнтів (наприклад: ПІБ, телефон, індивідуальний податковий номер, дата додавання). Всі ці дані мають різні типи: текст, число, дату, час.

Кожен рядок зберігає дані про екземпляр сутності, тобто про конкретного клієнта (наприклад: Іваненко Іван Іванович, +380965544332, 3309776614, 10.10.2023). Цей набір даних називається записом або кортежем.

Типи даних, які підтримуються PostgreSQL, за властивостями можна об'єднати у 3 групи (табл. 1.3.1 – 1.3.3): числові, текстові, дата/час.

Табл. 1.3.1 - числові типи даних (цілі та дробові)

	Name	Bytes	Description	Range
Integral Numbers	smallint	2	small-range integer	$2^{16} \rightarrow -32.768$ to $32.767$
	integer	4	typical choice for integers	$2^{32} \rightarrow -2.147.483.648$ to $2.147.483.647$
	bigint	8	large-range integer	$2^{64} \rightarrow -9.223.372.036.854.775.808$ to $9.223.372.036.854.775.807$
Real Numbers	decimal / numeric	variable	User-specified precision, exact	$\pm 3.4 * 10^{38}$ to $+3.4 * 10^{38}$
	real / float4	4	User-specified precision, inexact	6 decimal digits precision
	double precision / float8 / float	8	User-specified precision, inexact	15 decimal digits precision

Табл. 1.3.2 - числові з автоінкрементом та текстові

	Name	Bytes	Description	Range
Integral Numbers	smallserial	2	autoincrementing small-range integer	1 to 32.767
	serial	4	autoincrementing mid-range integer	1 to 2.147.483.647
	bigserial	8	Autoincrementing large-range integer	1 to 9.223.372.036.854.775.807
Characters	char	variable	fixed-length character string	based on encoding
	varchar	variable	fixed-length character string	based on encoding
	text	variable	fixed-length character string	based on encoding
Logical	Boolean / bool	1	used in logic	True / False

Табл. 1.3.3 - дані, що відображають дату, час та часовий пояс

	Name	Bytes	Description	Range
Temporal	date	4	stores only date	4713 B.C. $\rightarrow$ 294.276 AD
	time	8	stores only time	00:00:00 $\rightarrow$ 24:00:00
	timestamp	8	stores date & time	4713 B.C. $\rightarrow$ 294.276 AD
	interval	16	stores difference between timestamps	-178.000.000 $\rightarrow$ +178.000.000
	timestampz	8	stores a timestamp + timezone	4713 B.C. $\rightarrow$ 294.276 AD + tz

### 1.3.6 Створення таблиць та зв'язків

Для встановлення зв'язків між таблицями необхідно встановити зв'язок між тими полями, у яких міститься загальна (за сенсом) інформація. Ці поля можуть мати різні імена, але:

- тип даних;
- довжина полів;
- інформація в обох полях відповідних записів (це особливо важливо);

повинні бути однаковими в обох таблицях.

Як правило, зв'язок устанавлюється з'єднанням ключових полів таблиць:

- первинного ключа однієї таблиці та зовнішнього ключа в іншій – зв'язок 1 до багатьох;
- 2-х первинних ключів з різних таблиць – зв'язок 1 до 1.

Кожна таблиця повинна містити первинний ключ — одне або декілька полів, вміст яких унікальний для кожного запису.

Коли поле однієї таблиці посилається на інше поле (в іншій таблиці), воно зветься зовнішнім ключем (foreign key). Поле, на яке він посилається, зветься його батьківським ключем (parent key). Імена зовнішнього та батьківського ключів можуть не співпадати, але тип і розмір даних повинні бути однаковими.

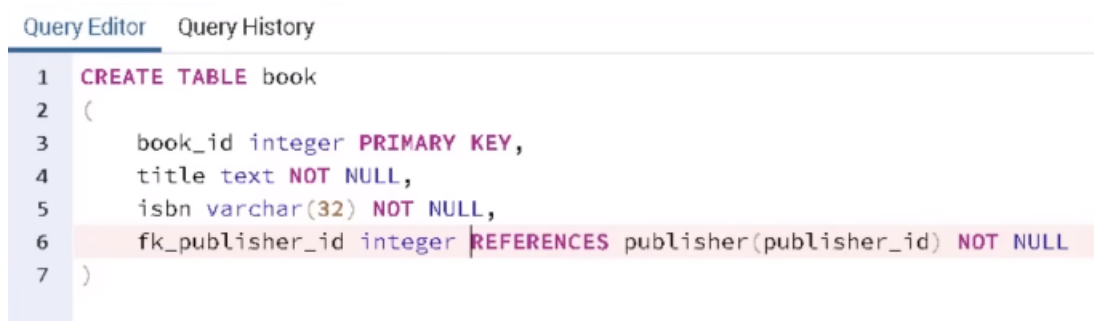
*\* Якщо батьківський ключ має тип з автоінкрементом, то відповідний зовнішній ключ повинний бути цілим.*

Таким чином, основна ідея посилальної цілісності полягає в тому, що при введенні даних до зовнішнього ключа батьківський ключ перевіряється на наявність в ньому такого значення. При відсутності команда відхиляється.

Таблиці можуть створюватись у **Query Tool** за допомоги команди **CREATE TABLE** (рис. 1.3.4). Безпосередньо після команди вказується назва таблиці. В дужках через кому записуються назви колонок із вказанням типів даних і обмежувальних умов.

Ключове слово **NOT NULL** означає, що при додаванні нового кортежу, відповідне поле не може залишитись порожнім.

Зв'язок з іншою таблицею вказується за допомогою ключового слова **REFERENCES**. Колонка, що є зовнішнім ключем в поточній таблиці, помічається словом **REFERENCES**, далі вказується таблиця (та в дужках колонка), з якою необхідно утворити зв'язок.



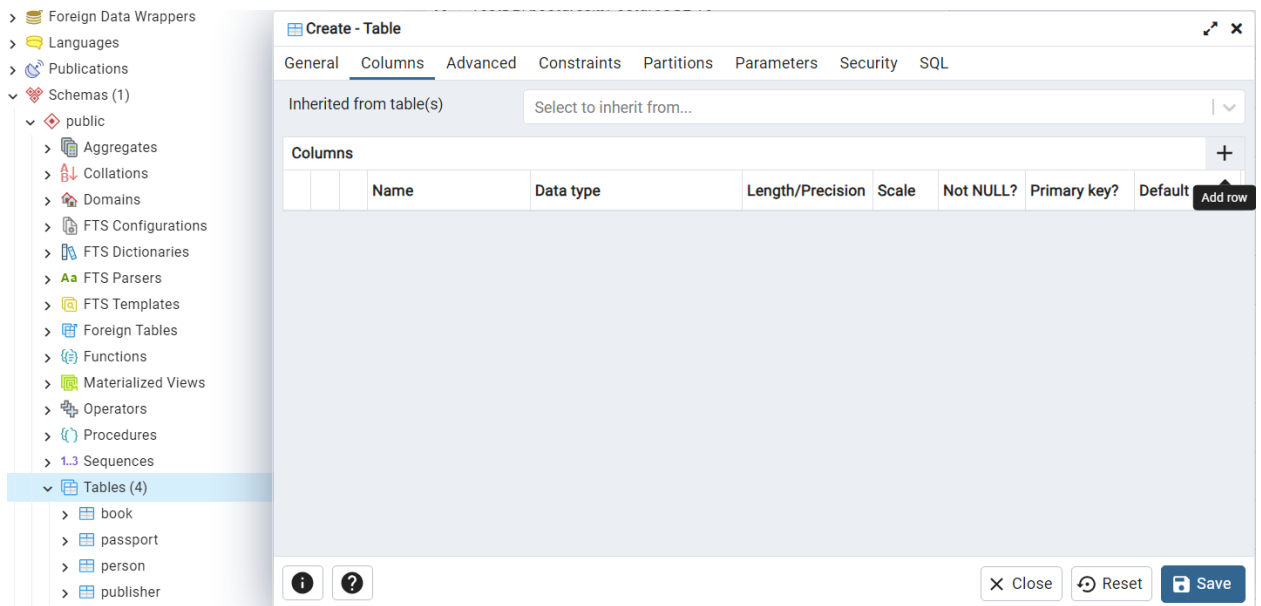
```

1 CREATE TABLE book
2 (
3     book_id integer PRIMARY KEY,
4     title text NOT NULL,
5     isbn varchar(32) NOT NULL,
6     fk_publisher_id integer REFERENCES publisher(publisher_id) NOT NULL
7 )
  
```

Рис. 1.3.4 – приклад синтаксису команди **CREATE TABLE**

Також можна створювати таблиці з використанням діалогового вікна **CREATE - TABLE** (рис. 1.3.5), що викликається у контекстному меню об'єктів групи **Tables**.



Рис. 1.3.5 - діалогове вікно **CREATE - TABLE**

Зв'язки між таблицями також можна встановлювати та редагувати з використанням інструменту **ERD Tool** (рис. 1.3.6), що викликається у рядку меню на вкладці Tools.

Крім того, можна визвати **ERD Tool** для відображення схеми даних на основі вже створених таблиць. Для цього необхідно визвати контекстне меню вже створеної таблиці і вибрати **ERD for table**.

У **ERD Tool** можна зберігати схему даних, також можна генерувати відповідний (схемі даних) SQL код.

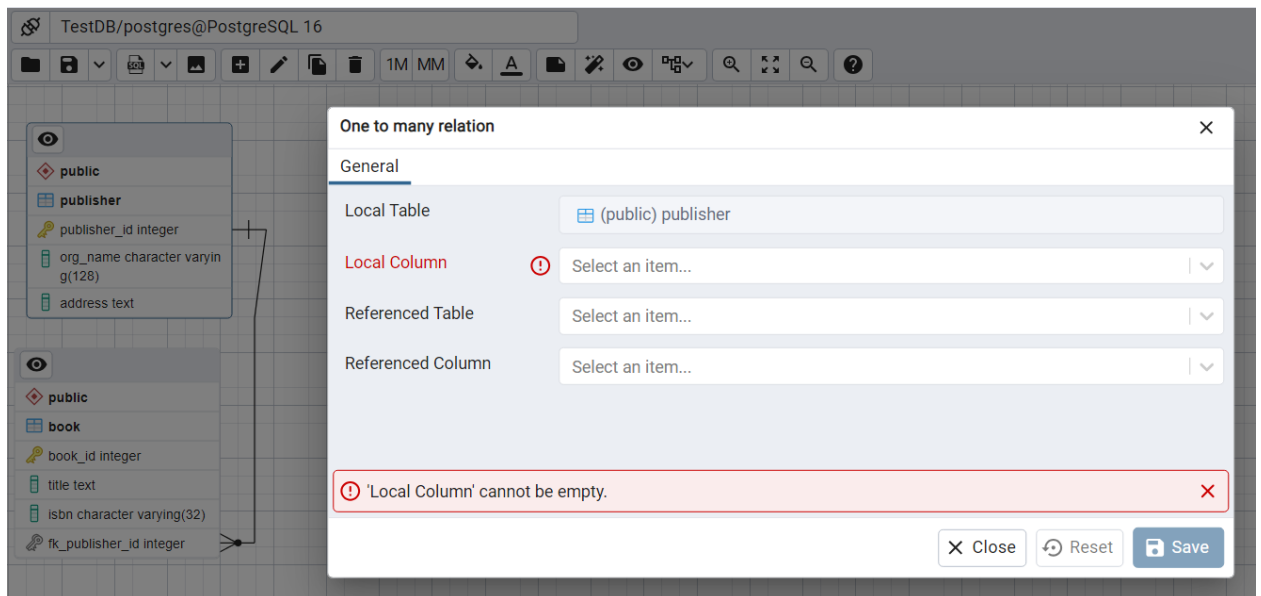


Рис. 1.3.6 – вікно ERD Tool

### 1.3.7 Додавання даних у таблиці

Дані до таблиць можна додавати за допомогою команди **INSERT INTO** (рис. 1.3.7), безпосередньо після команди вказується назва таблиці. Наступним вказується ключове слово **VALUES**, після якого у дужках через кому вказуються значення строго у порядку розташування колонок в таблиці.



Рис. 1.3.7 – синтаксис команди INSERT INTO

Дані можна також редагувати, використовуючи діалогове вікно **View/Edit Data** (рис. 1.3.8), що викликається з контекстного меню кожної таблиці.

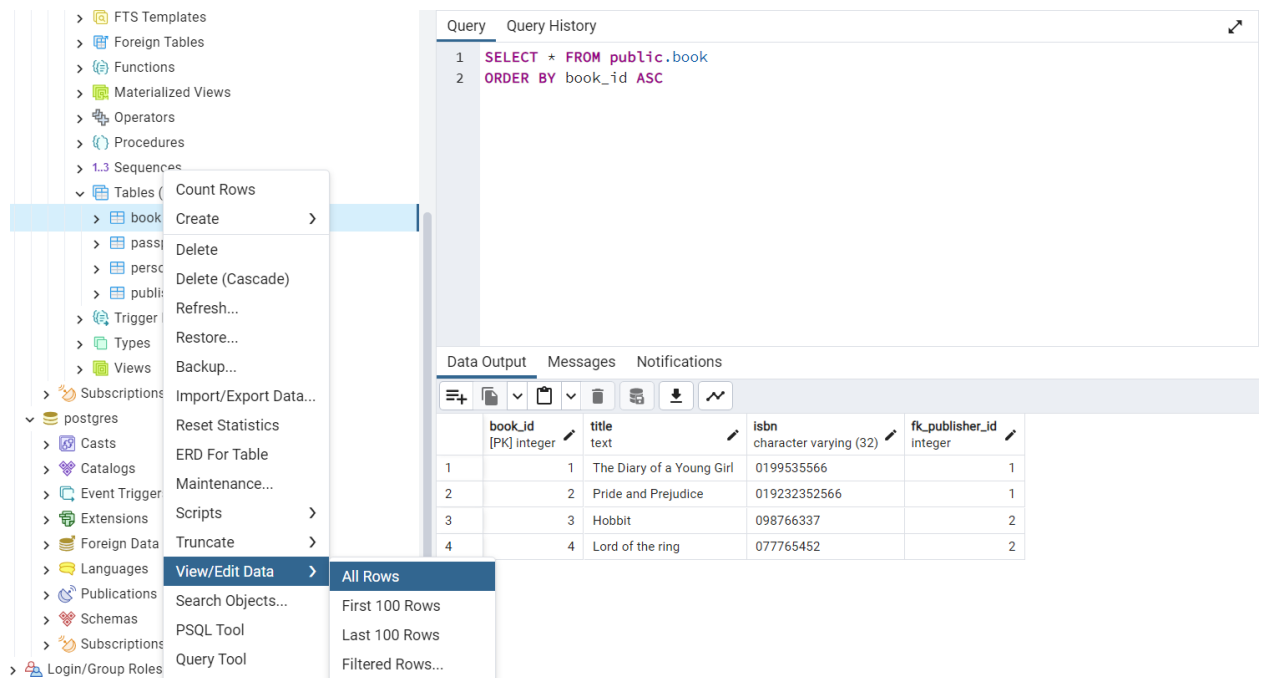


Рис. 1.3.8 – діалогове вікно View/Edit Data

### 1.3.8 Редагування структури таблиць з використанням Query tool

Для редагування структури таблиць використовується команда **ALTER TABLE** (рис. 1.3.9), безпосередньо після команди вказується назва таблиці. Для додавання та вилучення компонентів таблиці використовуються відповідно ключові слова **ADD** та **DROP**.

Приклад виконання послідовності SQL команд:

1. Видалення колонки - зовнішнього ключа;
2. Створення нової колонки;
3. Призначення їй статусу зовнішнього ключа;
4. Визначення оновленого зв'язку;

наведено на рис. 1.3.9.

Query	Query History
1	<b>ALTER TABLE</b> book
2	<b>DROP COLUMN</b> fk_publisher_id;
3	
4	<b>ALTER TABLE</b> book
5	<b>ADD COLUMN</b> fk_publ_id integer;
6	
7	<b>ALTER TABLE</b> book
8	<b>ADD CONSTRAINT</b> fk_publ_id
9	<b>FOREIGN KEY</b> (fk_publ_id) <b>REFERENCES</b> publisher(publisher_id)
10	

Рис. 1.3.9 – синтаксис команди ALTER TABLE

\* *Різні SQL команди обов'язково розділяються крапкою з комою.*

\*\* *SQL команди також називаються запитами (до СКБД).*

### Контрольні питання

- 1.4.1 Які типи даних використовуються в PostgreSQL?
- 1.4.2 Як задається режим автоматичної нумерації записів?
- 1.4.3 Якою командою створюється база даних?
- 1.4.4 Що таке тип даних, як він задається?
- 1.4.5 Як задається ключове поле?
- 1.4.6 Як задається зв'язок між двома таблицями?
- 1.4.7 Що таке первинні та зовнішні ключі?
- 1.4.8 Як вилучається зв'язок між таблицями?
- 1.4.9 Що таке поняття цілісності даних?
- 1.4.10 Як виконується редагування структури таблиці?

## ЛАБОРАТОРНА РОБОТА № 2 «ПРОЕКТУВАННЯ ЗАПИТІВ НА ВИБІРКУ ДАНИХ»

### 2.1 Мета роботи

Метою роботи є придбання навичок розробки запитів до СКБД PostgreSQL.

### 2.2 Завдання до лабораторної роботи

2.2.1 Ознайомитися зі змістом пункту 2.3 методичних указівок.

2.2.2 Отримати від викладача завдання, згідно з яким розробити відповідні запити до створеної в попередній роботі БД, використовуючи можливості СКБД PostgreSQL.

2.2.3 Оформити звіт, який повинен містити концептуальну модель предметної області, схему БД, тексти запитів та результати їхнього виконання.

### 2.3 Основні теоретичні відомості

#### 2.3.1 Розробка запитів до бази даних

Загальний формат команди SELECT такий:

**SELECT** [ALL|DISTINCT] список полів, що вибираються  
**FROM** список таблиць  
 [WHERE умова вибірки]  
 [GROUP BY умова угруповання]  
 [HAVING умова вибірки групи]  
 [ORDER BY умова впорядкування];

DISTINCT – аргумент, який усуває дублювання значень із результату виконання речення SELECT. Альтернативою DISTINCT є ключове слово ALL, яке використовується по умовчанням.

#### Вибір рядків: речення WHERE

Речення WHERE задає предикат, умову, що може бути вірною або помилковою для кожного рядка таблиці. У результаті вибираються тільки ті рядки з таблиці, для яких предикат має значення "істина".

У табл. 2.3.1 наведені оператори та ключові слова для завдання умов вибору.

Табл. 2.3.1 - оператори завдання умов

Оператор, ключове слово	Приклад використання
оператори порівняння (=, <, >, >=, <=, <>, !=);	where ціна > 1000;
комбінації умовних і логічних операцій - (AND, OR, NOT)	where ціна < 5000 and ціна > 2000
діапазони (BETWEEN і NOT BETWEEN)	where ціна between 450 and 500
списки (IN, NOT IN)	where товар in ("монітор", "принтер")
невідомі значення (IS NULL і IS NOT NULL)	where телефон is null
відповідності символів (LIKE і NOT LIKE)	where телефон not like "050%"; where назва like "%БД%"

## Агрегатні функції

Агрегатні функції (табл. 2.3.2) використовуються для одержання узагальнюючих значень. Вони дають єдине значення для цілої групи рядків таблиці.

Табл. 2.3.2 – перелік агрегатних функцій

Функція	Результат
<b>COUNT</b>	Визначає кількість рядків або значень поля, обраних за допомогою запиту, що <b>не є NULL-значеннями</b>
<b>SUM</b>	Обчислює арифметичну суму всіх обраних значень даного поля
<b>AVG</b>	Обчислює середнє значення для всіх обраних значень даного поля
<b>MAX</b>	Обчислює найбільше з всіх обраних значень даного поля
<b>MIN</b>	Обчислює найменше з всіх обраних значень даного поля

## Групування даних

Речення GROUP BY розділяє таблицю на набори, а агрегатна функція обчислює для кожного з них підсумкове значення. Ці значення називаються агрегатним вектором.

Приклад (рис. 2.3.1): скільки різних книжок було закуплено у кожного видавництва і яка середня вартість кожної вибірки книг?

```

Query  Query History
1  SELECT publisher_id, COUNT(book_id), AVG(price)
2  FROM books
3  GROUP BY publisher_id
4  ORDER BY publisher_id

```

Рис. 2.3.1 – приклад запиту з використанням агрегатних функцій

До списку вибору включаються стовпчики, за якими проводиться угруповання, та агрегатні функції, підсумок за якими треба відобразити.

## Речення HAVING

Речення WHERE накладає обмеження на рядки, а HAVING на групи. Як правило, речення HAVING використовується разом із реченням GROUP BY.

Якщо в списку вибору є агрегатні функції, речення WHERE виконується перед ними, тоді як речення HAVING застосовується до всього запиту в цілому, після розбивки на групи та обчислення значень функцій (рис. 2.3.2).

В умові речення WHERE не можуть знаходитися агрегатні функції. Крім того, елементи речення HAVING повинні включатися до списку вибору. На речення WHERE це обмеження не поширюється. У реченні HAVING може міститися будь-яка кількість умов.

```

Query  Query History
1  SELECT publisher_id, COUNT(book_id), AVG(price)
2  FROM books
3  GROUP BY publisher_id
4  HAVING AVG(price) > 300
5  ORDER BY publisher_id

```

Рис. 2.3.2 – приклад застосування речення HAVING.

### Операція INNER JOIN

Вона використовується у реченні FROM і з'єднує записи двох таблиць, якщо поля, за якими проводиться з'єднання, містять однакові значення. Ця операція також називається внутрішнім з'єднанням.

В наступному прикладі необхідно не просто порахувати кількість книжок за видавництвами, а і вивести назву кожного видавництва, а також його контактний телефон (рис. 2.3.3). Дані про видавництва зберігаються в іншій таблиці pub\_houses.

Query	Query History
1	<code>SELECT pub_houses.pub_name, pub_houses.tel_number, COUNT(book_id), AVG(price)</code>
2	<code>FROM books INNER JOIN pub_houses</code>
3	<code>ON books.publisher_id = pub_houses.publisher_id</code>
4	<code>GROUP BY pub_houses.pub_name, pub_houses.tel_number</code>
5	<code>HAVING AVG(price) &gt; 300</code>
6	<code>ORDER BY pub_houses.pub_name</code>

Рис. 2.3.3 – приклад застосування операції INNER JOIN.

### Операція LEFT JOIN, RIGHT JOIN

Операція LEFT JOIN використовується для створення лівого зовнішнього з'єднання, до якого включаються всі записи з першої (лівої) таблиці, навіть якщо немає співпадаючих значень для записів із другої (правої) таблиці.

Операція RIGHT JOIN використовується для створення правого зовнішнього з'єднання, до якого включаються всі записи з другої (правої) таблиці, навіть якщо немає співпадаючих значень для записів із першої (лівої) таблиці.

За замовчуванням, якщо вказати JOIN, буде виконана операція INNER JOIN.

### Запити на об'єднання

Операція UNION може об'єднувати результати декількох запитів, таблиць або інструкцій SQL (рис. 2.3.4).

Query	Query History
1	<code>SELECT * FROM books</code>
2	<code>UNION</code>
3	<code>SELECT * FROM books_archive</code>

Рис. 2.3.4 – приклад застосування операції UNION

Для об'єднання двох і більш запитів необхідно, щоб їхні стовпчики, що входять у набір вихідних даних, були сумісні по об'єднанню.

**Контрольні питання**

- 2.4.1 Основний формат команди Select?
- 2.4.2 Які оператори використовуються для завдання умов вибірки даних?
- 2.4.3 Що таке агрегатні функції?
- 2.4.4 Для чого використовується групування даних у запитах?
- 2.4.5 Що таке з'єднання таблиць, які існують різновиди цієї операції?
- 2.4.6 Для чого використовуються запити на об'єднання?
- 2.4.7 Послідовність виконання логічних і умовних операторів у предикаті запиту?
- 2.4.8 В чому полягає різниця між застосуванням речень WHERE і HAVING?
- 2.4.9 Чим відрізняються операції з'єднання і об'єднання таблиць?
- 2.4.10 Для чого використовується речення ORDER BY?

## ЛАБОРАТОРНА РОБОТА № 3 «ПРОЕКТУВАННЯ СКЛАДНИХ ЗАПИТІВ НА ВИБІРКУ ТА МОДИФІКАЦІЮ ДАНИХ»

### 3.1 Мета роботи

Метою роботи є придбання навичок розробки запитів до СКБД PostgreSQL.

### 3.2 Завдання до лабораторної роботи

3.2.1 Ознайомитися зі змістом пункту 2.3 методичних указівок.

3.2.2 Отримати від викладача завдання, згідно з яким розробити відповідні запити до створеної в попередній роботі БД, використовуючи можливості СКБД PostgreSQL.

3.2.3 Оформити звіт, який повинен містити концептуальну модель предметної області, схему БД, тексти запитів та результати їхнього виконання.

### 3.3 Основні теоретичні відомості

Підзапит (subquery) – це додатковий метод маніпуляцій із декількома таблицями. Це – оператор SELECT, вкладений:

- у речення WHERE, HAVING або SELECT іншого оператора SELECT;
- в оператор INSERT, UPDATE або DELETE;
- в інший підзапит.

Підзапити повертають результати внутрішнього запиту в зовнішнє речення.

#### 3.3.1 Запити на модифікацію даних

##### Запит на додавання даних

Команда INSERT дозволяє додавати рядки за допомогою ключового слова VALUES або за допомогою оператора SELECT.

Як було зазначено в першій лабораторній роботі, при використанні ключового слова VALUES, для кожного рядку, що додається, використовується свій оператор INSERT.

Для додавання даних з однієї або декількох таблиць у команді INSERT можна використовувати вкладений оператор SELECT:

**INSERT INTO** ім'я\_таблиці (список стовпчиків, які вставляються)  
**(SELECT** список\_стовпчиків **FROM** список\_таблиць  
**WHERE** умови)

Якщо рядки, які додаються, містять значення окремих стовпчиків таблиці, то стовпчики, що залишилися, (що не модифікуються) повинні припускати нульовий статус. У протилежному випадку команда буде відхилена.

Оператор SELECT у команді INSERT дозволяє взяти дані з декількох або з усіх стовпчиків однієї таблиці та вставити їх в іншу таблицю. Якщо вставлені значення тільки для частини стовпчиків, визначити значення для інших стовпчиків можна буде пізніше за допомогою оператора UPDATE.

При вставці рядків з однієї таблиці в іншу ці таблиці повинні мати сумісну структуру. Якщо стовпчики в обох таблицях сумісні за типами та визначені в однаковому порядку у відповідних операторах CREATE TABLE, перераховувати їх у команді INSERT необов'язково.



Приклад копіювання всіх даних з таблиці books у таблицю books\_archive наведено на рис. 3.3.1.

Query	Query History
1	<b>INSERT INTO</b> books_archive
2	<b>(SELECT * FROM</b> books)

Рис. 3.3.1 – копіювання всіх даних із однієї таблиці в іншу.

Таким чином може бути виконано архівування даних. Коли виконується робота з даними поточного звітного періоду, всі дані, що стосуються минулих періодів доцільно скопіювати в архівну таблицю та прибрати з поточної.

### Запит на видалення даних

Видалення рядків даних виконується за допомоги команди DELETE FROM, яка має такий формат:

```
DELETE
FROM ім'я_таблиці
[WHERE умови_відбору]
```

Речення WHERE може мати вкладений оператор SELECT. Наприклад, щоб прибрати дані з таблиці books, які нещодавно були скопійовані в таблицю books\_archive можна виконати запит, зазначений на рис. 3.3.2.

Query	Query History
1	<b>DELETE FROM</b> books
2	<b>WHERE</b> book_id <b>IN</b> ( <b>SELECT</b> book_id <b>FROM</b> books_archive)

Рис. 3.3.2 – запит на видалення даних.

### Запит на оновлення даних

Зміна значень полів виконується за допомоги команди UPDATE, яка має такий формат:

```
UPDATE ім'я_таблиці
SET ім'я_стовпчика = нове_значення
[WHERE умова_відбору]
```

При відсутності речення WHERE зміни вносяться у всі рядки до стовпчиків, зазначених у реченні SET.

Якщо необхідно зберегти в базі даних проміжні дані, щоб не витратити машинний час кожен раз на їх розрахунок, можна також скористатися запитом на оновлення даних.

Наприклад, необхідно мати в постійному доступі інформацію щодо загальної вартості товарних залишків по кожній книзі, що є у каталозі. Одна і та ж сама книга може доставлятися різними партіями у різний час. Текст запиту, що вирішує зазначену задачу наведено на рис. 3.3.3.

Query	Query History
1	<b>UPDATE</b> books
2	<b>SET</b> sum_price = subquery.total
3	<b>FROM</b>
4	(
5	<b>SELECT</b> books.book_id,
6	sum(stock.quantity * books.price) <b>AS</b> total
7	<b>FROM</b> books
8	<b>JOIN</b> stock <b>ON</b> stock.book_id = books.book_id
9	<b>GROUP BY</b> books.book_id
10	<b>ORDER BY</b> books.book_id
11	) <b>AS</b> subquery
12	
13	<b>WHERE</b> books.book_id = subquery.book_id

Рис. 3.3.3 – запит на оновлення даних.

\* У рядку 6 визначається обчислювальна колонка *total*, в яку записується сума добутків кількості на вартість для кожної книги з каталогу (таблиці *books*). Саме для кожної книги тому, що рядок 9 визначає умову групування результату обчислення за первинним ключем таблиці *books*.

\*\* Коли в запиті на оновлення використовується речення *FROM* (тобто визначається таблиця – джерело даних), у реченні *WHERE* необхідно обов'язково вказати поле за яким буде перевірятись відповідність рядків між таблицею – джерелом та таблицею – приймачем даних.

### 3.3.2 Складні запити на вибірку даних

Спрощена форма синтаксису підзапиту показана на рис. 3.3.4.



Рис. 3.3.4 – спрощена форма синтаксису підзапиту.

Підзапити бувають трьох типів, у залежності від елементів у реченні WHERE зовнішнього запити:

- підзапити, що не повертають жодного або повертають декілька елементів (починаються з IN або з оператора порівняння, містять ключові слова ANY або ALL);
- підзапити, що повертають єдине значення (починаються з простого оператора порівняння);
- підзапити, що являють собою тест на існування (починаються з EXISTS).

Наприклад, якщо необхідно вибрати книги, вартість яких вище середньої вартості всіх книг, що є у каталозі, можна виконати наступний запит (рис. 3.3.5).

Query	Query History
1	SELECT * FROM books
2	WHERE price > (SELECT AVG(price) FROM books)
3	

Рис. 3.3.5 – запит на вибірку даних.

### Контрольні питання

- 3.4.1 Наведіть синтаксис запити на додавання рядків.
- 3.4.2 Для чого слугує ключове слово VALUES у запиті на додавання рядків?
- 3.4.3 Як додати рядки у таблицю з іншої таблиці?
- 3.4.4 Наведіть синтаксис запити на видалення рядків.
- 3.4.5 Чи можна використовувати підзапити у запитах на видалення рядків?
- 3.4.6 Наведіть синтаксис запити на оновлення даних.
- 3.4.7 Для чого слугує команда SET у запиті на оновлення?
- 3.4.8 Коли дані оновлюються з іншої таблиці, яким чином відбувається синхронізація рядків між таблицею-джерелом та таблицею-приймачем даних?
- 3.4.9 В яких командах запити SELECT може використовуватись підзапит?
- 3.4.10 За допомоги якого ключового слова проводиться за тест на існування при відборі рядків у запиті SELECT?

## ЛАБОРАТОРНА РОБОТА № 4 «РОБОТА З ПОДАННЯМИ»

### 4.1 Мета роботи

Метою роботи є придбання навичок створення та використання подань в середовищі PostgreSQL.

### 4.2 Завдання до лабораторної роботи

4.2.1 Ознайомитися зі змістом пункту 4.3 даних методичних указівок.

4.2.2 Згідно з отриманим завданням створити відповідні об'єкти БД. Навести приклади їхнього використання.

### 4.3 Основні теоретичні відомості

Бувають ситуації, коли користувачеві необхідно дозволити доступ не до всієї інформації, що зберігається в таблиці, а лише до її частини: наприклад, необхідно, щоб продавці могли переглядати таблицю Salespeople, але не могли бачити комісійні своїх колег. Або ситуація, коли щодня необхідно одержувати інформацію на основі різних таблиць за допомогою складних запитів. Для того, щоб не повторювати щораз складний запит, можна використовувати подання.

Подання є **віртуальним відношенням** (*virtual relation*), якого реально в базі даних не існує, але яке створюється на вимогу окремого користувача в момент надходження цієї вимоги.

З погляду користувача подання є відношенням, що постійно існує та з яким можна працювати, як із базовим відношенням.

Однак подання не зберігається в базі даних так, як базові відношення (хоча його визначення зберігається в системному каталозі). Вміст подання визначається як результат виконання запиту до одного або декількох базових відношень.

Подання - це вікно, через яке користувач може побачити інформацію з БД. При зверненні до подання виконується запит до базової таблиці або таблиць, на основі яких створено подання.

Наприклад, необхідно дозволити покупцям бачити в таблиці books тільки дані про назву книги, жанр та ціну продажу. У цьому випадку можна створити подання book\_catalog на основі таблиці books (рис. 4.3.1), а потім дозволити всім покупцям читати інформацію з цього подання.

Query	Query History
1	CREATE OR REPLACE VIEW book_catalog AS
2	SELECT title, genre, price
3	FROM books
4	ORDER BY title

Рис. 4.3.1 – створення подання на основі таблиці books.

Зверніть увагу на те, що в пропозиції CREATE VIEW після імені подання не зазначені імена стовпців, з яких воно буде складатися. Тому беруться імена стовпців базової таблиці або їхні псевдоніми, на основі яких будується подання.

Подання, засновані на запиті, що використовує речення GROUP BY, називаються груповими. При цьому допускається використання речення HAVING.

При створенні подання також можна використовувати з'єднання таблиць.

Для розрахунку даних щодо загальної вартості товарних залишків по кожній книзі можна також використовувати подання (рис. 4.3.2).

Query Query History

```

1 CREATE VIEW cost_of_inv_balances AS
2     SELECT books.book_id,
3           sum(stock.quantity::double precision * books.price) AS total
4     FROM books
5     JOIN stock ON stock.book_id = books.book_id
6     GROUP BY books.book_id
7     ORDER BY books.book_id;
```

Рис. 4.3.2 – створення подання з використанням з'єднання таблиць.

Це подання побудовано на основі запиту, що з'єднує дві таблиці. Звернутися до нього можна також через запит. Наприклад, необхідно визначити всі книги, де вартість товарних залишків перевищує 5 000 грн (рис. 4.3.3).

```

9 SELECT * FROM cost_of_inv_balances
10 WHERE total > 5000
```

Рис. 4.3.2 – створення запиту до подання.

Будь які операції над поданнями автоматично транслюються в операції над відношеннями, на підставі яких це подання було створено. Подання мають динамічну природу, тобто зміни в базових відносинах, які можуть вплинути на вміст подання, негайно відбиваються на вмісті цього подання.

Якщо користувачі вносять у подання деякі припустимі зміни, ці зміни негайно заносяться до відповідних базових таблиць.

Якщо не визначені інші правила безпеки, за замовченням всі дані які не є агрегатними значеннями і які відображаються у поданні можна змінити і зміни будуть відображені у базових таблицях. Відповідно, ті дані, що не відображаються у поданні змінити не можна.

Але є одне виключення, коли додається новий рядок у подання в якому зазначена умова відбору (WHERE) і цей рядок не відповідає умові, він не з'явиться у відношенні, але буде доданий до базової таблиці.

### Матеріалізовані подання

Команда CREATE MATERIALIZED VIEW створює матеріалізоване подання. Заданий запит виконується та наповнює подання в момент виклику команди. Оновити подання пізніше можна, виконавши команду REFRESH MATERIALIZED VIEW.

Матеріалізоване подання статичне, на відміну від звичайного. Змінювати дані у ньому неможливо.

Матеріалізоване подання варто використовувати, наприклад, коли необхідні постійно контакти клієнтів різним спеціалістам. Контакти змінюються не часто, а блокування базової таблиці client відбувається кожен раз коли спеціаліст до неї звертається, що сповільнює транзакції, які націлені на редагування таблиці client.

**Контрольні питання**

- 4.4.1 Яка різниця між базовим і віртуальним відношеннями?
- 4.4.2 Що таке подання, його призначення?
- 4.4.3 Як створити подання?
- 4.4.4 Як використовується операція з'єднання при створенні подання?
- 4.4.5 Чи можна створити подання на основі іншого подання?

## ЛАБОРАТОРНА РОБОТА № 5 «РОБОТА З ФУНКЦІЯМИ І ТРИГЕРАМИ»

### 5.1 Мета роботи

Метою роботи є придбання навичок створення та використання функцій і тригерів в середовищі PostgreSQL.

### 5.2 Завдання до лабораторної роботи

5.2.1 Ознайомитися зі змістом пункту 5.3 даних методичних указівок.

5.2.2 Згідно з отриманим завданням створити відповідні об'єкти БД. Навести приклади їхнього використання.

### 5.3 Основні теоретичні відомості

Функції у PostgreSQL, як правило, складаються з SQL-операторів та низки спеціальних керуючих структур. Збережена функція може бути корисна, коли одну і ту ж послідовність дій необхідно виконати з різних додатків або з різних платформ або як засіб інкапсуляції функціональних можливостей. Збережені функції в базах даних можна вважати аналогом об'єктно-орієнтованого підходу у програмуванні. Вони дозволяють керувати способом доступу до даних.

#### LANGUAGE SQL у функціях

Розглянемо приклад функції, яка відшукує у БД найдешевші книги. Кількість книг визначається параметром (рис. 5.3.1).

Query	Query History
1	<b>DROP FUNCTION IF EXISTS</b> fn_min_price
2	
3	<b>CREATE FUNCTION</b> fn_min_price(amount int)
4	<b>RETURNS SETOF</b> books <b>AS</b>
5	\$body\$
6	<b>SELECT</b> * <b>FROM</b> books
7	<b>ORDER BY</b> price <b>ASC</b>
8	<b>LIMIT</b> amount;
9	\$body\$
10	<b>LANGUAGE SQL</b>
11	
12	<b>SELECT</b> (fn_min_price(3)).*

Рис. 5.3.1 - приклад видалення, створення та виклику функції.

\* **DROP FUNCTION** – видалити функцію, **IF EXISTS** – якщо існує (запобігає помилці, якщо функція вже видалена), останнім позначається ім'я функції, яку необхідно видалити.

\*\* Після ключового слова **CREATE FUNCTION** визначається ім'я функції та перелік параметрів (через кому в дужках, в форматі: ім'я, тип\_даних); після ключового слова **RETURNS** визначається тип об'єкту, який повертає функція (**SETOF books** – набір кортежів типу books, якщо об'єкт буде 1 – **SETOF** опускається, функція не повертає нічого – **RETURNS void**); тіло функції починається з ключового слова **AS** і виділяється тегами **\$body\$**, також обов'язково визначається мова, якою написана функція, за допомогою ключового слова **LANGUAGE**.

**\*\*\* Виклик функції здійснюється за допомоги ключового слова *SELECT* (якщо ім'я функції не взяти в дужки та не додати комбінацію символів *"."*, відбудеться неформатований вивід даних).**

Варто звернути увагу, що відбір найдешевших книг у тілі функції (рис. 5.3.1) відбувається не за допомоги ключового слова *WHERE*, як це звичайно робилося, а використовуючи сортування та обмеження кількості виведених кортежів.

### **LANGUAGE plpgsql у функціях**

PL/pgSQL належить до сімейства процедурних мов, вона подібна мові PL/SQL системи Oracle. Мова PL/pgSQL є процедурним розширенням мови SQL, тому може використовувати всі її типи даних, оператори і функції. Це підвищує гнучкість використання і швидкодію команд SQL, оскільки в програмному блоці вони виконуються за одну операцію замість звичайної обробки кожної команди. Важливою особливістю PL/pgSQL є високий ступінь адаптованості програм до всіх платформ, на яких вони базуються.

Структура мови. Мова PL/pgSQL відносно проста, кожен її логічновідокремлений фрагмент коду існує у вигляді функції. До певної міри програми подібні до написаних мовою C: всі змінні обов'язково оголошуються перед використанням, функції отримують аргументи при виклику і повертають потрібні значення при закінченні роботи та ін. PL/pgSQL також надає можливість використовувати структури управління потоком виконання, такі як *IF*, *ELSE*, *WHILE* та *FOR*.

Регістр символів в іменах функцій PL/pgSQL не регламентований. У ключових словах і ідентифікаторах допускається використання довільних комбінацій символів верхнього та нижнього регістрів.

Програмний блок. Виготовити програмний блок (функцію) можна командою SQL *CREATE FUNCTION*, у її складі може міститися інструкція *OR REPLACE*, яка дозволяє редагувати функцію.

Функція являє собою блок, який містить секцію *DECLARE* – оголошення даних та *BEGIN* – команди, які виконуються. Закінчується програмний блок словом *END*.

Розглянемо приклад функції, яка шукає у таблиці *authors* випадкового автора для нагородження і виводить дані про нього (рис. 5.3.2).

Query    Query History

```

1  CREATE FUNCTION get_random_author()
2  RETURNS SETOF authors AS
3  $body$
4  DECLARE
5  rand int;
6  BEGIN
7      SELECT random()*(MAX(author_id)-MIN(author_id))+ MIN(author_id)
8      FROM authors INTO rand;
9      RETURN QUERY
10     SELECT * FROM public.authors WHERE author_id = rand;
11 END;
12 $body$
13 LANGUAGE plpgsql

```



Рис. 5.3.2 – функція пошуку випадкового автора.

\* Для збереження результату запиту у змінну може бути використана конструкція `SELECT ... INTO`.

\*\* Для того, щоб повернути результат запиту в якості значення функції використовується команда `RETURN QUERY`.

Окрім роботи з даними за допомоги `plpgsql` можна виконувати типові для програміста арифметичні і логічні задачі, звісно, якщо це доречно робити на стороні серверу.

Наприклад, необхідно порахувати суму непарних чисел від 1 до `max_num` і визначити чи буде парним результат. Дану задачу вирішує наведена нижче функція (рис. 5.3.3).

Query	Query History
1	<b>CREATE FUNCTION</b> plpgsql_syntax_test(max_num int)
2	<b>RETURNS</b> varchar <b>AS</b>
3	\$body\$
4	<b>DECLARE</b>
5	i int <b>DEFAULT</b> 1;
6	tot_sum int <b>DEFAULT</b> 0;
7▼	<b>BEGIN</b>
8	FOR i <b>IN</b> 1 .. max_num <b>BY</b> 2
9▼	<b>LOOP</b>
10	tot_sum := tot_sum + i;
11	<b>END LOOP</b> ;
12▼	<b>IF</b> tot_sum % 2 = 0 <b>THEN</b>
13	RETURN CONCAT(tot_sum, ' - even number');
14	<b>ELSE</b>
15	RETURN CONCAT(tot_sum, ' - odd number');
16	<b>END IF</b> ;
17	<b>END</b> ;
18	\$body\$
19	<b>LANGUAGE</b> plpgsql;
20	
21	<b>SELECT</b> plpgsql_syntax_test(5)

Data Output	Messages	Notifications
<div> <div>⊞</div> <div>📄</div> <div>▼</div> <div>📋</div> <div>▼</div> <div>🗑️</div> <div>🗄️</div> <div>⬇️</div> <div>📈</div> </div>		
	plpgsql_syntax_test character varying	
1	9 - odd number	

Рис. 5.3.3 – приклад вирішення арифметичних задач.

\* змінна типу `varchar` зберігає динамічний масив символів, таким чином зручно вертати з функції рядок, який може включити декілька значень змінних різних типів (рядок формується за допомоги вбудованої функції `CONCAT`).

**\*\*** крок циклу задається ключовим словом *BY*, обернений порядок проходження циклу можна задати за допомоги ключового слова *REVERSE* (*FOR i IN REVERSE max\_num .. 1*).

### Тригери у PostgreSQL

Тригер PostgreSQL – це функція, яка запускається автоматично в об'єкті бази даних у разі виникнення події.

Приклади подій бази даних, які можуть активувати тригер, включають INSERT, UPDATE, DELETE, TRUNCATE. Тригери можуть викликатись до, після та під час події. Крім того, тригер автоматично скидається під час видалення таблиці.

Тригер може бути позначений оператором *FOR EACH ROW* під час його створення. Такий тригер викликатиметься один раз для кожного рядка, зміненого операцією. Тригер може бути позначений оператором *FOR EACH STATEMENT* під час його створення. Цей тригер буде виконано лише один раз для конкретної операції.

**Приклад 1.** Необхідно заблокувати модифікацію даних у таблиці *books* у вихідні дні. Ця задача вирішується у два етапи. Спочатку створюється функція, що блокує модифікацію даних (рис. 5.3.4), потім створюється тригер, який визначає умови виникнення події та автоматично викликає функцію (рис. 5.3.5).

Query	Query History
1	<b>CREATE OR REPLACE FUNCTION</b> fn_block_weekend_changes()
2	<b>RETURNS TRIGGER</b>
3	<b>LANGUAGE</b> plpgsql
4	<b>AS</b>
5	\$body\$
6 ▼	<b>BEGIN</b>
7	<b>RAISE NOTICE</b> 'No changes allowed on the weekend';
8	<b>RETURN</b> null;
9	<b>END;</b>
10	\$body\$

Рис. 5.3.4 – приклад функції, яка блокує ввід даних до таблиці.

\* Функція повертає значення типу *TRIGGER*, це позначає, що повертається об'єкт (в даному випадку кортеж або рядок) який викликав подію.

**\*\*** Після ключового слова *RETURN* можна зазначити: *null* – нічого (не робити), *old* – варіант кортежу до модифікації, *new* – варіант кортежу після модифікації.

**\*\*\*** Для звернення до окремого значення кортежу можна використати оператор “.” (*OLD.price* або *NEW.title*), **але тільки якщо тригер створений для кожного рядка (FOR EACH ROW).**

Зверніть увагу на те, що функції, які повертають значення типу *TRIGGER*, зберігаються у окремому розділі дерева об'єктів БД – *Trigger Functions*.

Видалення та створення тригерів відбувається за допомоги команд *DROP TRIGGER* та *CREATE TRIGGER* відповідно (рис. 5.3.5).

Query	Query History
1	<b>DROP TRIGGER IF EXISTS</b> tr_block_weekend_changes <b>ON</b> books;
2	
3	<b>CREATE TRIGGER</b> tr_block_weekend_changes
4	<b>BEFORE UPDATE OR INSERT OR DELETE</b>
5	<b>ON</b> public.books
6	<b>FOR EACH ROW</b>
7	<b>WHEN</b> (
8	<b>EXTRACT ('DOW' FROM CURRENT_TIMESTAMP) BETWEEN 6 AND 7</b>
9	)
10	<b>EXECUTE PROCEDURE</b> fn_block_weekend_changes();
11	
12	<b>UPDATE</b> books
13	<b>SET</b> price = 330
14	<b>WHERE</b> book_id = 1;

Data Output	Messages	Notifications
ЗАМЕЧАНИЕ: No changes allowed on the weekend		
UPDATE 0		
Query returned successfully in 35 msec.		

Рис. 5.3.5 – створення тригера, який визначає умови блокування вводу даних до таблиці.

\* умови виникнення подій, які перевіряє тригер, можна розділити на три категорії:

1. Що відбувається з даними (**UPDATE** - змінення, **INSERT** - додавання або **DELETE** – видалення);

2. У якому саме об'єкті відбувається модифікація даних (**ON books**), і саме до нього буде підключений тригер;

3. Коли за глобальною шкалою часу відбувається модифікація даних (**WHEN Day Of Week** поточної дати знаходиться між 6-м та 7-м днем тижня).

\*\* **EXECUTE PROCEDURE** запускає розроблену раніше функцію (рис. 5.3.4).

\*\*\* Тригер буде зберігатися у підрозділі *Triggers* дерева об'єктів таблиці *books*.

**Приклад 2.1.** Розглянемо ще раз знайомий приклад: необхідно оновити вартість товарних залишків по кожній книзі з каталогу, але перерахунок необхідно проводити автоматично, кожен раз, коли модифікуються дані про кількість книг на складі.

По-перше розробимо тригер-функцію, **тому що не можливо створити тригер, який посилається на неіснуючу функцію.**

Функція, що оновлює вартість товарних залишків має вже знайомий вигляд (рис. 3.5.6), вона використовує складний запит на оновлення, який включає підзапит, що розраховує суму добутків кількості на вартість по кожній книзі.

Query Query History

```

1 CREATE OR REPLACE FUNCTION fn_tr_mod_sum_price()
2 RETURNS TRIGGER
3 LANGUAGE plpgsql
4 AS
5 $body$
6 BEGIN
7     UPDATE books
8     SET sum_price = subquery.total
9     FROM
10    (
11     SELECT books.book_id, SUM(stock.quantity * books.price) AS total
12     FROM books
13     JOIN stock ON stock.book_id = books.book_id
14     GROUP BY books.book_id
15    ) AS subquery
16 WHERE books.book_id = subquery.book_id;
17 RAISE NOTICE 'Quantity changed';
18 RETURN NEW;
19 END;
20 $body$

```

Рис. 5.3.6 – тригер-функція, яка оновлює вартість товарних залишків.

Далі створюємо тригер, який буде спрацьовувати при редагуванні таблиці stock (рис. 5.3.7) та викликати описану вище тригер-функцію.

```

22 CREATE TRIGGER tr_stock_mod
23 AFTER INSERT OR UPDATE OR DELETE
24 ON public.stock
25 FOR EACH STATEMENT
26 EXECUTE PROCEDURE fn_tr_mod_sum_price();

```

Рис. 5.3.7 – тригер – модифікатор вартості товарних залишків.

\* Тригер спрацьовує після оновлення, додавання або видалення даних (AFTER UPDATE, INSERT, DELETE).

\*\* Тригер спрацьовує тільки 1 раз, коли відбувся якийсь з зазначених видів модифікації (FOR EACH STATEMENT).

Зазначена вище тригер-функція відносно проста у реалізації, але вона має суттєвий недолік. Кожен раз, коли змінюється кількість однієї позиції на складі, виконується перерахунок вартості товарних залишків по всім позиціям у каталозі. Це доволі неефективно з точки зору використання обчислювальної потужності серверу.

**Приклад 2.2.** Необхідно оновити вартість товарних залишків по кожній книзі з каталогу, але окрім автоматизації перерахунку, необхідно оновлювати тільки ті позиції, дані про кількість яких зазнали змін.

Нижче визначена функція, яка вирішує поставлену задачу (рис. 5.3.8).

Query	Query History
1	CREATE OR REPLACE FUNCTION fn_tr_mod_sum_price()
2	RETURNS TRIGGER
3	LANGUAGE plpgsql
4	AS
5	\$body\$
6	BEGIN
7	IF OLD.book_id NOT IN (SELECT book_id FROM stock) THEN
8	UPDATE books
9	SET sum_price = 0
10	WHERE books.book_id = OLD.book_id;
11	RETURN NEW;
12	END IF;
13	
14	UPDATE books
15	SET sum_price = subquery.total
16	FROM
17	(
18	SELECT books.book_id, SUM(stock.quantity * books.price) AS total
19	FROM books
20	JOIN stock ON stock.book_id = books.book_id
21	WHERE books.book_id IN (OLD.book_id, NEW.book_id)
22	GROUP BY books.book_id
23	) AS subquery
24	WHERE books.book_id = subquery.book_id;
25	RAISE NOTICE 'Quantity changed';
26	RETURN NEW;
27	END;
28	\$body\$

Рис. 5.3.8 – остаточна версія тригер-функції, яка оновлює вартість товарних залишків.

\* Додатково накладаємо умову у реченні *WHERE* підзапиту (рядок 21), яка додає до розрахунку тільки ті книги, значення *book\_id* яких належать множині (*OLD.book\_id*, *NEW.book\_id*). У запиті на оновлення будуть виправлені тільки ті рядки, які розраховувались у підзапиті.

\*\* Є такий випадок, коли видаляється зі складу останній рядок з даними про деяку книгу, тоді *OLD.book\_id* буде існувати, но такого значення вже не буде у переліку складу. Для оброблення даного виключення можна записати нулі в усі значення *sum\_price* таблиці *books* для книг, даних про які немає у таблиці *stock* (тобто для книг, які відсутні на складі).

Тригер, що викликає зазначену функцію наведено на рис. 5.3.9.

Query	Query History
1	CREATE TRIGGER tr_stock_mod
2	AFTER INSERT OR UPDATE OR DELETE
3	ON public.stock
4	FOR EACH ROW
5	EXECUTE PROCEDURE fn_tr_mod_sum_price();

Рис. 5.3.9 – тригер – модифікатор вартості товарних залишків.

*\* Цей тригер, на відміну від попереднього, необхідно виконувати для кожного рядка, інакше не можливо звернутися до значень " OLD." та "NEW.". Крім того, існують запити, які обробляють по декілька позицій.*

*\*\* Цей тригер доречно визначити саме після модифікації, тому що інші тригери або правила (виконані пізніше) можуть заблокувати модифікацію даних в базовій таблиці, це спричинить втрату цілісності даних.*

### **Контрольні питання**

- 5.4.1 Як визначається функція, які команди при визначені є обов'язковими?
- 5.4.2 Як визначається мова програмування, якою написана функція?
- 5.4.3 Як визначаються аргументи функції?
- 5.4.4 Для чого необхідні функції у СКБД PostgreSQL?
- 5.4.5 Чи може функція повернути таблицю?
- 5.4.6 Для чого використовуються тригери у СКБД PostgreSQL?
- 5.4.7 Як визначається тригер?
- 5.4.8 Як у тригер-функції звернутися до об'єкту, який був модифікований?
- 5.4.9 Як при створенні тригеру визначити прив'язку до глобальної шкали часу?
- 5.4.10 Чим відрізняються тригери, що визначені FOR EACH STATEMENT та FOR EACH ROW?

**ЛІТЕРАТУРА**

1. Корнієнко С.К. Проектування інформаційного забезпечення автоматизованих систем: Навч. Посібник / С.К Корнієнко . – Запоріжжя: ЗНТУ, 2015. – 224 с.
2. Методичні вказівки до лабораторних робіт з дисципліни Бази даних. Робота з СКБД MySQL для студентів спеціальності 121 «Інженерія програмного забезпечення» усіх форм навчання /Уклад.: С. К. Корнієнко –Запоріжжя: НУ «Запорізька політехніка», 2021. –34с.