ПРИКЛАДНІ ПРОБЛЕМИ КОМП'ЮТЕРНИХ НАУК,
БЕЗПЕКИ ТА МАТЕМАТИКИ, 2024, № 3

УДК 004.9

МЕТОДИ МОДЕЛЮВАННЯ ІЄРАРХІЧНИХ СТРУКТУР В РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ

Булатецька Л.В. (ORCID: 0000-0002-7202-826X),
Булатецький В.В. (ORCID: 0000-0002-9883-4550).
Волинський національний університет імені Лесі Українки, Луцьк, Україна

METHODS OF MODELING HIERARCHICAL STRUCTURES IN RELATIONAL DATABASES

Bulatetska L.V., Bulatetskyi V.V.
Lesya Ukrainka Volyn National University, Lutsk, Ukraine

Abstract. The paper examines the main methods for representing hierarchical structures in relational databases and typical queries for these data structures. The methods reviewed include Adjacency List, Nested Sets, Closure Table, and Materialized Path. Each of these models has its own advantages and disadvantages depending on the types of operations being performed. Quantitative metrics for data retrieval times were obtained for data represented in the database using these methods. Based on these metrics, an analysis of the appropriateness of using these methods was conducted, depending on the characteristics of nesting and data volumes. Adjacency List is simple to implement and convenient for operations at the level of individual nodes, such as adding or deleting a leaf, but it is not optimal for complex queries involving retrieving subtrees or paths between nodes. Nested Sets demonstrate high efficiency in retrieving subtrees, but operations such as adding and deleting nodes are complex and resource-intensive. Closure Table offers high flexibility for queries involving ancestor and descendant relationships but requires significant resources to maintain data consistency during frequent updates. Materialized Path is efficient for queries involving retrieving paths between nodes and subtrees.

Keywords: Adjacency List, Nested Sets, Closure Table, Materialized Path, hierarchical structures, recursive query, relational data model.

Вступ. Реляційні бази даних здебільшого призначені для зберігання та обробки табличних даних, де взаємозв'язки між даними описуються через реляційні таблиці. Однак у багатьох реальних сценаріях дані мають ієрархічну (деревовидну) структуру. Таблиці, що містять батьківські та дочірні елементи, часто використовуються для представлення такої ієрархічної інформації. Типовий приклад — категорії з численними рівнями вкладеності. У бізнесі ця структура може відображати організаційну ієрархію компанії, де кожен відділ має підвідділи, а ті, своєю чергою, підрозділи. У вебсервісах ієрархічні структури часто використовуються для категоризації товарів, у бібліотеках — для організації книжкових колекцій, що дозволяє легко знаходити книги за тематикою та підтематикою. Існує кілька поширених методів подання ієрархічних структур у реляційних базах даних, таких як Adjacency List, Nested Sets, Closure Table і Materialized Path [1-2]. Для забезпечення ефективної роботи з ієрархічними даними важливо зберігати їх у форматі, що мінімізує час вибірки та ресурси, необхідні для цього процесу.

Метою роботи є проведення аналізу ефективності операцій управління даними для ієрархічних структур у реляційних базах даних на основі різних моделей подання, таких як Adjacency List, Nested Sets, Closure Table та Materialized Path, з метою виявлення оптимальних підходів для виконання різних типів запитів та операцій.

Методологія дослідження. Для проведення аналізу оцінки продуктивності кожного методу було створено тестові сценарії генерації даних для ієрархічних структур з різною глибиною та кількістю вузлів. Для цього було заповнено таблицю побудовану у вигляді списків суміжних вершин (Adjacency List) випадковими даними. Для того, щоб у всіх чотирьох моделях були однакові дані, було заповнено таблиці інших ієрархічних моделей даними з таблиці моделі списків суміжних вершин. Дослідження проведено з

деревовидними структурами, які мають 100, 1000 і 10000 вузлів. Враховуючи те, що дані генерувалися випадковим чином, кількість рівнів дерева теж генерується випадковим чином. Отримані дерева мали глибину 12, 13 та 18 рівнів для таблиць які складаються з 100, 1000 та 10000 записів відповідно.

В цій роботі для моделювання ієрархічних структур було обрано реляційну СУБД Oracle Database 18c XE (Express Edition). Сервер баз даних був розгорнутий на платформі віртуалізації Oracle VirtualBox 7.0.18 з гостьовою системою 2xCPU core, 4GB RAM, MS Windows 10 Pro. Базова система: Intel Core i3, 16GB RAM, SSD NVMe 512GB, MS Windows 11 Pro. Робоча станція була представлена ноутбуком Lenovo ThinkPad T450 з процесором Intel Core i5, 8GB RAM, SSD 180GB SATA-3, MS Windows 10 Pro. В якості комунікаційного середовища використано Mesh WiFi-5 на базі трьох тридіапазонних юнітів Linksys з швидкістю до 3Gbps.

Результати дослідження та їхнє обговорення

Списки суміжних вершин (Adjacency List). Метод моделювання ієрархічних структур даних у вигляді списків суміжності (Adjacency List) є одним із найпоширеніших способів побудови дерев у реляційних базах даних. Кожен запис у таблиці відповідає вузлу дерева, зберігаючи унікальний ідентифікатор (id) та посилання на батьківський вузол (parent_id). Якщо вузол є коренем, parent_id має значення NULL. Ця модель представляє базовий тип ієрархії, де кожний нащадок має одного предка, що в теорії графів відповідає орієнтованому ациклічному графу [1].

На рис. 1 подано SQL запит на створення таблиці CATEGORY Adjacency_List та заповнене дерево з елементами від A до L, які зберігаються в цій таблиці [2-5].

```
CREATE TABLE CATEGORY_Adjacency_List (
id INTEGER PRIMARY KEY,
title VARCHAR2(5) NOT NULL,
parent_id INTEGER REFERENCES CATEGORY ON DELETE CASCADE
);
```

ID	TITLE	PARENT_ID
1	A	
2	B	1
3	C	1
4	D	1
5	E	2
6	F	3
7	G	3
8	H	4
9	I	4
10	J	8
11	K	8
12	L	8

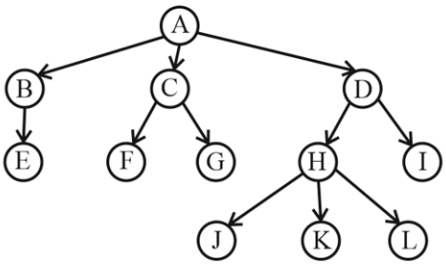


Рис. 1. SQL запит на створення таблиці, яка представляє деревовидну структуру списків суміжних вершин (Adjacency List) та дерево змодельоване у вигляді списків суміжності з елементами від A до L, які зберігаються в таблиці CATEGORY_ Adjacency_List.

На рис. 2 подано запити, які повертають всі вузли дерева, які є листками, тобто не мають нащадків [5]. Підзапит (рис. 2, *а*) вибирає всі записи з тієї ж таблиці CATEGORY_Adjacency_List, позначеної як t2, де parent_id дорівнює id з першої таблиці t1. Якщо такий запис існує, це означає, що у вузла є нащадки. Використання NOT EXISTS гарантує, що обираються тільки ті записи, які не мають нащадків. В запиті (рис. 2, *б*) використана умова фільтрації. Після LEFT JOIN для кожного рядка з таблиці t1, якщо не було знайдено відповідності в таблиці t2 (тобто, t2.id є NULL), то ці рядки будуть відображені в результатах. Це означає, що для рядків з таблиці t1, які не мають жодного дочірнього елемента в t2, t2.id буде NULL.

а)
```
SELECT id, title
FROM CATEGORY_Adjacency_List t1
WHERE NOT EXISTS
        (SELECT 1
        FROM CATEGORY_Adjacency_List t2
        WHERE t2.parent_id = t1.id );
```

б)
```
SELECT t1.id,t1.title
FROM CATEGORY_Adjacency_List t1 LEFT
JOIN CATEGORY_Adjacency_List t2
ON t1.id=t2.parent_id
WHERE t2.id IS NULL;
```

Рис. 2. Запити, які повертають всі вузли дерева, які не мають нащадків

На рис. 3 подано запит виведення дочірніх елементів та батьківського вузла для вузла title= 'H'.

Для відображення всього дерева ми повинні з'єднати таблицю саму з собою, стільки разів, скільки рівнів має наше дерево. Щоб знайти кількість рівнів ієрархії потрібно виконати рекурсивний запит поданий на рис 4 [6,7].

а)
```
SELECT id, title
FROM CATEGORY_Adjacency_List
WHERE parent_id=
        (SELECT id
        FROM CATEGORY_Adjacency_List
        WHERE title= 'H');
```

б)
```
SELECT t2.id, t2.title
FROM CATEGORY_Adjacency_List t1
JOIN CATEGORY_Adjacency_List t2
ON t1.parent_id = t2.id
WHERE t1.title= 'H';
```

Рис. 3. Запит на виведення дочірніх (*а*) та батьківського (*б*) елементів вузла з вершиною title= 'H'

```
SELECT MAX(level) AS max_level
FROM CATEGORY_Adjacency_List
START WITH parent_id IS NULL
CONNECT BY PRIOR id = parent_id;
```

Рис. 4. Запит для визначення кількості рівнів в ієрахії

Для згенерованого дерева (100 вузлів) кількість рівнів виявилось рівним 12, тому, щоб вивести все дерево, потрібно з'єднати таблицю саму з собою 12 разів (рис. 5)

```
SELECT  t1.title AS level_1, t2.title AS level_2,
        t3.title AS level_3, t4.title AS level_4
        ... t12.title AS level_12
FROM CATEGORY_Adjacency_List t1
LEFT JOIN CATEGORY_Adjacency_List t2 ON (t2.parent_id=t1.id)
LEFT JOIN CATEGORY_Adjacency_List t3 ON (t3.parent_id=t2.id)
LEFT JOIN CATEGORY_Adjacency_List t4 ON (t4.parent_id=t3.id)
...
LEFT JOIN CATEGORY_Adjacency_List t12 ON (t12.parent_id=t11.id)
WHERE t1.id IS NULL;
```

Рис. 5. Виведення всього дерева способом з'єднання таблиці саму з собою стільки разів скільки рівнів має дерево.

Щоб знайти шлях між вузлами дерева (рис. 6), потрібно побудувати запит, аналогічний до запиту для відображення всього дерева, вказавши елементи початку і кінця шляху і з'єднати таблицю саму з собою стільки разів, скільки рівнів між вузлами. Такі запити не є універсальними, оскільки при зміні кількості рівнів дерева їх потрібно переписувати. Якщо глибина дерева або рівень елемента невідомі, стандартний оператор SELECT не підходить, і необхідно створювати рекурсивний запит (рис. 7) [6,7]. Аналогічним способом можна вивести все піддерево (рис. 8).

Додавання нового вузла в дерево здійснюється за допомогою простої SQL-команди INSERT (рис. 9, *а*). При видаленні вузла, відповідно до правила ON DELETE CASCADE, автоматично видаляється і все піддерево (рис. 9, *б*).

```
SELECT  t1.title AS level_1, t2.title AS level_2,
        t3.title AS level_3, t4.title AS level_4
                        ... t12.title AS level_10
FROM CATEGORY_Adjacency_List t1
LEFT JOIN CATEGORY_Adjacency_List t2 ON (t2.parent_id=t1.id)
LEFT JOIN CATEGORY_Adjacency_List t3 ON (t3.parent_id=t2.id)
LEFT JOIN CATEGORY_Adjacency_List t4 ON (t4.parent_id=t3.id)
...
LEFT JOIN CATEGORY_Adjacency_List t10 ON (t10.parent_id=t9.id)
WHERE t1.title='D' AND t10.title='K';
```

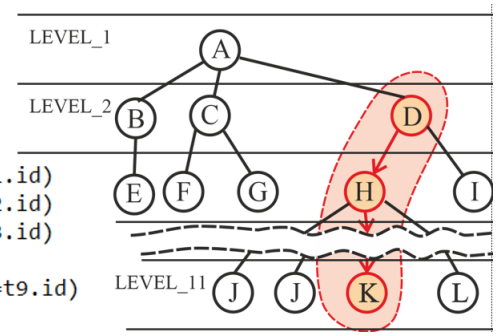


Рис. 6. Виведення від вузла title='D' на 2 рівні до вузла title='K', який знаходиться на 11 рівні.

```
SELECT SYS_CONNECT_BY_PATH(title,'/') as Path
FROM CATEGORY_Adjacency_List
WHERE title='K'
START WITH title='D'
CONNECT BY PRIOR id = parent_id;
```

Рис. 7. Рекурсивний запит на виведення шляху від вузла title='D' на 2 рівні до вузла title='K', який знаходиться на 11 рівні.

від вузла title='D' на 2 рівні до вузла title='K', який знаходиться на 11 рівні.

```
SELECT lpad(' ',3*level)||title as TREE
FROM CATEGORY_Adjacency_List
START WITH title= 'D'
CONNECT BY PRIOR id = parent_id
ORDER SIBLINGS BY title;
'
```

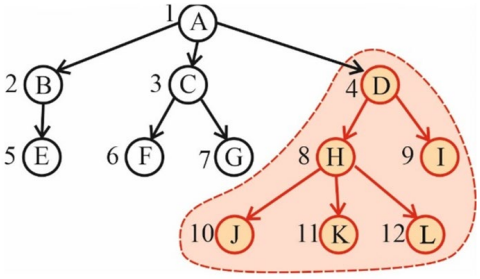


Рис. 8. Виведення всього піддерева з вершиною 'D'

а)
```
INSERT INTO CATEGORY_Adjacency_List (id, title, parent_id)
                                    VALUES  (101, 'M', 6);
```

б)
```
DELETE FROM CATEGORY WHERE id=101;
```

Рис. 9. Додавання (*а*) та видалення (*б*) вузла дерева з усіма дочірніми елементами.

Модель вкладених множин (Nested Sets). Інший спосіб представлення дерев полягає в їх відображенні у вигляді вкладених множин (рис 10) [2,3,8]. Кожен вузол дерева має 2 значення: Left (значення ліворуч від вузла) та Right (значення праворуч від вузла). Процедура визначення цих значень полягає у обході дерева ліворуч праворуч і нарощуванні лічильника на 1 під час проходження вузла. На рис. 10 пунктирними стрілками показано процес обходу дерева. При такі організації деревовидної структури дуже легко отримати всіх нащадків для деякого вузла [8].

```
CREATE TABLE CATEGORY_Nested_Sets
(id INTEGER PRIMARY KEY,
title VARCHAR2(5),
lft INTEGER NOT NULL,
rgt INTEGER NOT NULL
);
```

ID	TITLE	LFT	RGT
1	A	1	24
2	B	2	5
3	C	6	11
4	D	12	23
5	E	3	4
6	F	7	8
7	G	9	10
8	H	13	20
9	I	21	22
10	J	14	15
11	K	16	17
12	L	18	19

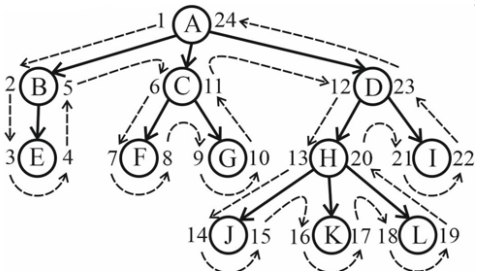


Рис. 10. Дерево змодельоване у вигляді вкладених множин з елементами від A до L, які зберігаються в таблиці CATEGORY_Nested_Sets

Перевагами цієї моделі є проста операція отримання нащадків без рівня вкладеності та проста операція отримання батьківських вузлів. До недоліків слід віднести відсутність посилкової цілісності, складні операції видалення, додавання та переміщення вузла, а також громістка операція підрахунку рівня вкладеності нащадків вузла. Головною причиною недоліків операцій видалення, додавання та переміщення вузла є необхідність оновлення значень LEFT та RIGHT для всіх порушених вузлів у дереві, що суттєво відбивається на продуктивності. Не проста операція підрахунку рівня вкладеності всім нащадкам вузла, також впливає на продуктивність особливо, якщо дерево досить велике. Як правило модель «Nested Sets» вибирають для швидких операцій вибірки всіх нащадків і предків, тому для даного способу швидше за все буде доречним перенести рівень вкладеності в іншу таблицю і оновити її за будь-яких змін у дереві.

На рис 11 подано запит, який повертає всі вузли дерева, які є листками, тобто не мають нащадків.

```
SELECT id, title
FROM CATEGORY_Nested_Sets
WHERE rgt = lft + 1;
```

Рис. 11. Запит який повертає всі вузли дерева моделі ієрархічних вкладених множин, які є не мають нащадків

Цей запит (рис. 11) обирає id та title з таблиці CATEGORY_Nested_Sets для всіх записів, де значення rgt дорівнює lft + 1. Це умова визначає листкові вузли у моделі вкладених множин, оскільки у таких вузлів немає нащадків.

Для отримання прямих дочірніх елементів певного вузла в моделі «Nested Sets» потрібно виконати один із запитів поданих на рис. 12. Запити знаходять всі прямі дочірні елементи вузла з назвою 'D' у таблиці CATEGORY_Nested_Sets. Спочатку обираються межі (lft і rgt) для вузла 'D', а потім використовуються ці межі для пошуку дочірніх елементів, що знаходяться всередині цих меж. Умова NOT EXISTS гарантує, що обираються лише прямі дочірні елементи, без підвузлів. Перший запит (рис. 12, *а*) виконує з'єднання таблиці саму з собою (JOIN), щоб знайти вузол 'D' і одразу використовує його межі (lft і rgt) для фільтрації дочірніх елементів. Другий запит (рис. 12, *б*) використовує зону визначення заголовків (WITH ParentNode AS), щоб спочатку вибрати межі (lft і rgt) вузла з назвою 'D' і потім використовує ці межі в основному запиті. Основна різниця полягає в тому, що другий запит спрощує розуміння

написаного запиту шляхом використання CTE (Common Table Expressions, загальний табличний вираз [6,7]).

а)
```
SELECT t1.id, t1.title, t1.lft, t1.rgt
FROM CATEGORY_Nested_Sets t1
JOIN CATEGORY_Nested_Sets t2
ON t1.lft > t2.lft AND t1.rgt < t2.rgt
WHERE t2.title= 'D'
  AND NOT EXISTS (
    SELECT 1
    FROM CATEGORY_Nested_Sets t3
    WHERE t3.lft > t2.lft AND t3.rgt < t2.rgt
    AND t1.lft > t3.lft AND t1.rgt < t3.rgt
  );
```

б)
```
WITH ParentNode AS (
    SELECT lft, rgt
    FROM CATEGORY_Nested_Sets
    WHERE title= 'D'
)
SELECT t1.id, t1.title, t1.lft, t1.rgt
FROM CATEGORY_Nested_Sets t1, ParentNode t2
WHERE t1.lft > t2.lft AND t1.rgt < t2.rgt
  AND NOT EXISTS (
    SELECT 1
    FROM CATEGORY_Nested_Sets t3
    WHERE t3.lft > t2.lft AND t3.rgt < t2.rgt
    AND t1.lft > t3.lft AND t1.rgt < t3.rgt
    );
```

Рис. 12. Запити, знаходять всі прямі дочірні елементи вузла з назвою 'H'

На рис. 13 подано запити які призначені для знаходження найближчого батьківського вузла для вузла 'D' у таблиці CATEGORY_Nested_Sets.

а)
```
SELECT t2.id, t2.title
FROM CATEGORY_Nested_Sets t1
JOIN CATEGORY_Nested_Sets t2
ON t1.lft BETWEEN t2.lft AND t2.rgt
WHERE t1.title= 'D'
AND t2.lft < t1.lft AND t2.rgt > t1.rgt
ORDER BY t2.lft DESC
FETCH FIRST 1 ROWS ONLY;
```

б)
```
SELECT id, title
FROM CATEGORY_Nested_Sets
WHERE lft <
        (SELECT lft
        FROM CATEGORY_Nested_Sets
        WHERE title= 'D')
AND rgt >
        (SELECT rgt
        FROM CATEGORY_Nested_Sets
        WHERE title= 'D')
ORDER BY lft DESC FETCH FIRST 1 ROWS ONLY;
```

Рис. 13. Запит, який знаходить батьківський вузол для вузла 'D'

Перший запит (рис. 13, *а*) використовує JOIN для з'єднання таблиці самої з собою, щоб знайти всі батьківські вузли для вузла з title='D', потім фільтрує ці вузли за умовами t2.lft < t1.lft і t2.rgt > t1.rgt, та вибирає верхній вузол. Другий запит виконує два підзапити, щоб знайти lft та rgt значення вузла з title='D', потім використовує ці значення для фільтрації всієї таблиці і вибирає вузол, де lft менше, а rgt більше цих значень, сортує результати і вибирає перший рядок. Основна різниця між запитами в тому, що перший використовує з'єднання таблиці самої з собою, тоді як другий використовує вкладені підзапити для знаходження меж значень.

На рис. 14 подано запит, який виводить шлях від одного вузла до іншого. Запит знаходить всі вузли, які є нащадками вузла з назвою 'D' і предками вузла з назвою 'K'. Він порівнює значення lft і rgt для обмеження результатів лише тими вузлами, які знаходяться між межами вузлів 'D' та 'K'. Результати впорядковуються за значенням lft у зростаючому порядку, щоб показати ієрархію з кореня до листя.

На рис. 15 подано запит, який знаходить всі вузли, які є нащадками вузла з назвою 'D'. Він використовує підзапити, щоб отримати значення lft і rgt для вузла 'D' і обмежує результати тільки тими вузлами, чиї lft і rgt значення знаходяться всередині цих меж. Результати впорядковуються за значенням lft у зростаючому порядку, щоб показати ієрархію з кореня до листя.

Для того, щоб додати новий вузол в дерево, потрібно знати значення правої межі parentRgt батьківського вузла. Далі потрібно оновити межі для всіх вузлів, що знаходяться праворуч від батьківського вузла, збільшивши їх значення на 2. І тільки після цього можна додати новий вузол. Враховуючи те, що для додавання вузла,

потрібно виконати не один запит а декілька, то варто ці запити об'єднати в одну збережену функцію (рис. 16).

```
SELECT id, title
FROM CATEGORY_Nested_Sets
WHERE
        lft <= (SELECT lft FROM CATEGORY_Nested_Sets WHERE title='K')
    AND rgt >= (SELECT rgt FROM CATEGORY_Nested_Sets WHERE title='K')
    AND lft >= (SELECT lft FROM CATEGORY_Nested_Sets WHERE title='D')
    AND rgt <= (SELECT rgt FROM CATEGORY_Nested_Sets WHERE title='D')
ORDER BY lft;
```

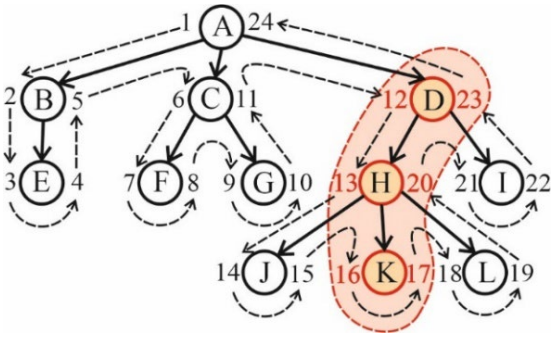


Рис. 14. Запит, який знаходить шлях від вузла 'D' до вузла 'K'

```
SELECT id, title FROM CATEGORY_Nested_Sets
WHERE lft>=
        (SELECT lft
        FROM CATEGORY_Nested_Sets
        WHERE title= 'D')
AND rgt <=
        (SELECT rgt
        FROM CATEGORY_Nested_Sets
        WHERE title= 'D')
ORDER BY lft;
```

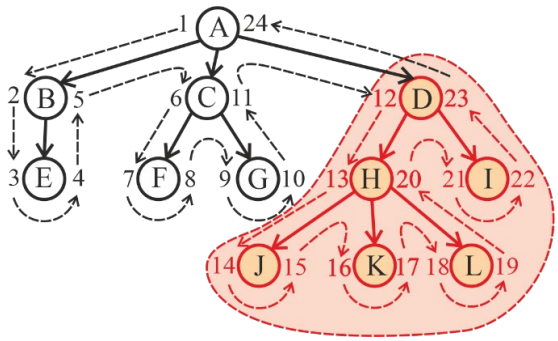


Рис. 15. Запит, який виводить піддерево вузла 'D'

Для видалення вузла з усіма його нащадками в моделі ієрархічних вкладених множин (Nested Sets) необхідно визначити межі (lft і rgt) вузла, який потрібно видалити, видалити всі вузли, які мають межі, що знаходяться в межах вузла, який видаляється, оновити межі всіх вузлів, які залишилися, щоб видалити пробіли, які утворилися після видалення вузла і його нащадків. Всі кроки, які виконані за допомогою SQL-запитів варто об'єднати в збережену процедуру (рис. 17).

Таблиці зв'язків (Closure Table). Подання дерева у вигляді таблиці зв'язків зображено на рис. 18. У цьому моделюванні вузли зберігаються в одній таблиці, а зв'язки – в іншій. Перша таблиця містить назви вузлів (title) та їхні ідентифікатори. Друга таблиця зберігає зв'язки між кожним вузлом і вузлами на шляху до вершини, включно з коренем. Зв'язок задається трьома числами: ідентифікатором початкового вузла, кінцевого вузла та відстанню – кількістю дуг між ними [5]. Як видно з рис. 18. одним з недоліків такого подання деревовидної структури – це надмірність зберігання даних у таблиці CATEGORY_link, які необхідні для опису всіх зв'язків.

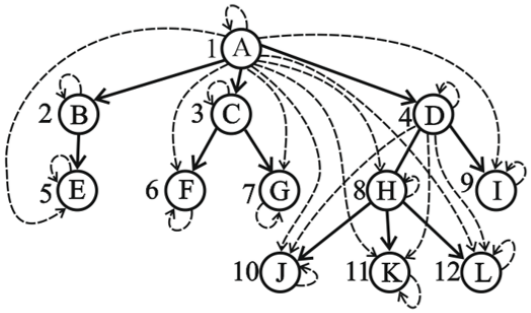
```
CREATE OR REPLACE PROCEDURE add_node (
    new_id        IN INTEGER,
    new_title     IN VARCHAR2,
    parent_id     IN INTEGER
) IS
    parent_rgt INTEGER;
BEGIN
    -- Отримати праву межу батьківського вузла
    SELECT rgt INTO parent_rgt FROM CATEGORY_Nested_Sets WHERE id = parent_id;

    -- Оновити праві межі всіх вузлів, що знаходяться праворуч від батьківського вузла
    UPDATE CATEGORY_Nested_Sets
    SET rgt = rgt + 2
    WHERE rgt >= parent_rgt;

    -- Оновити ліві межі всіх вузлів, що знаходяться праворуч від батьківського вузла
    UPDATE CATEGORY_Nested_Sets
    SET lft = lft + 2
    WHERE lft > parent_rgt;

    -- Додати новий вузол
    INSERT INTO CATEGORY_Nested_Sets (id, title, lft, rgt)
    VALUES (new_id, new_title, parent_rgt, parent_rgt + 1);
END;
/
```

Рис. 16. Процедура додавання нового вузла в дерево

```
CREATE OR REPLACE PROCEDURE delete_node_and_descendants(
    p_node_id IN INTEGER
) AS
    v_lft INTEGER;
    v_rgt INTEGER;
    v_width INTEGER;
BEGIN
    -- Отримуємо межі (lft та rgt) вузла, який потрібно видалити
    SELECT lft, rgt INTO v_lft, v_rgt FROM CATEGORY_Nested_Sets WHERE id = p_node_id;

    -- Визначаємо ширину вузла, який потрібно видалити
    v_width := v_rgt - v_lft + 1;

    -- Видаляємо всі вузли, що знаходяться в межах вузла, який видаляється
    DELETE FROM CATEGORY_Nested_Sets
    WHERE lft BETWEEN v_lft AND v_rgt;

    -- Оновлюємо межі всіх вузлів, що залишилися, зменшуючи значення меж,
    --які знаходяться праворуч від видалених вузлів
    UPDATE CATEGORY_Nested_Sets
    SET rgt = rgt - v_width
    WHERE rgt > v_rgt;

    UPDATE CATEGORY_Nested_Sets
    SET lft = lft - v_width
    WHERE lft > v_rgt;
END delete_node_and_descendants;
/
```

Рис. 17. Процедура видалення вузла з усіма його нащадками

```
CREATE TABLE CATEGORY_Closure_Table
(
id INTEGER PRIMARY KEY,
title VARCHAR2(5) NOT NULL
);
CREATE TABLE CATEGORY_link
(
Id_from INTEGER NOT NULL REFERENCES CATEGORY ON DELETE CASCADE,
Id_to INTEGER NOT NULL REFERENCES CATEGORY ON DELETE CASCADE,
distance INTEGER NOT NULL,
PRIMARY KEY (Id_from, Id_to)
);
```



ID	TITLE
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	H
9	I
10	J
11	K
12	L

ID_FROM	ID_TO	DISTANCE
1	1	0
1	2	1
1	3	1
1	4	1
1	5	2
1	6	2
1	7	2
1	8	2
1	9	2
1	10	3
1	11	3
1	12	3
2	2	0
2	5	1
3	3	0
3	6	1
3	7	1
4	4	0
4	8	1
4	9	1
4	10	2
4	11	2
4	12	2
5	5	0
6	6	0
7	7	0
8	8	0
8	10	1
8	11	1
8	12	1
9	9	0
10	10	0
11	11	0
12	12	0

Рис. 18. Подання дерева у вигляді таблиці зв'язків

Якщо за зразок прийняти модель списків суміжності, яка не містить жодної надмірності, то для методу таблиці зв'язків на кожен рівень потрібно стільки додаткових записів у таблиці CATEGORY_link, скільки елементів знаходиться на даному рівні дерева, помноженого на номер рівня. Надмірність зберігання даних можна оцінити як

$$\sum_{1}^{N} \text{Count(i)} \cdot \text{i}$$

де Count(i) – кількість вузлів на *i*-му рівні дерева, починаючи з кореня; *N* – число рівнів у дереві.

Однак переваги, отримані від надмірності зберігання, очевидні – запити більш короткі та швидкі. Чим глибше в дереві знаходиться вузол, тим більше потрібно записів для опису зв'язків. У той же час цей спосіб дає можливість отримати всіх батьків або нащадків одним простим запитом, не вдаючись до рекурсії.

До переваг цієї моделі можна віднести збереження посилкової цілісності даних, проста операція видалення вузла з усіма його нащадками завдяки цілісності посилання (ON DELETE CASCADE), проста операція отримання нащадків без рівня вкладеності, проста операція отримання батьків без рівня вкладеності, проста операція додавання вузла в дерево.

До недоліків – велика кількість записів у таблиці зв'язків через необхідність зберігати зв'язки кожного елемента дерева з усіма його предками. Варто також відзначити, що розмір таблиці зв'язків може змінюватися не тільки від операцій додавання або видалення вузла, а й операції переміщення. У випадку операції переміщення можливі обидва варіанти, як збільшення кількості записів у таблиці зв'язків, так і зменшення, все залежить від того, на яку глибину вкладеності буде переміщений вузол.

Запит поданий на рис. 19 поверне всі вузли дерева, які не мають нащадків. Підзапит перевіряє, чи існує запис у таблиці CATEGORY_link, де Id_from дорівнює id з таблиці CATEGORY_Closure_Table де немає самопокликань. Якщо такий запис існує, це означає, що вузол має нащадків, і він не буде обраний основним запитом.

```
SELECT t.id, t.title
FROM CATEGORY_Closure_Table t
WHERE NOT EXISTS (
                SELECT 1
                FROM CATEGORY_link l
                WHERE l.Id_from = t.id and l.distance<>0);
```

Рис. 19. Запит, який повертає всі вузли дерева, які не мають нащадків

Запити, що подано на рис. 20 знаходять всі прямі дочірні елементи вузла з title='H' та його батьківський вузол у таблиці CATEGORY_Closure_Table за допомогою таблиці зв'язків CATEGORY_LINK, для цього ці таблиці з'єднують за допомогою операторів LEFT JOIN. В результуючий набір відбираються тільки ті вузли, які знаходяться на відстані 1 від вузла з title='H', тобто прямі дочірні елементи.

а)
```
SELECT id, title
FROM CATEGORY_Closure_Table t1
LEFT JOIN CATEGORY_LINK t2
ON t1.id=t2.id_to
WHERE t2.id_from=
        (SELECT id
        FROM CATEGORY_Closure_Table
        WHERE title= 'H')
and distance=1;
```

б)
```
SELECT id, title
FROM CATEGORY_Closure_Table t1
LEFT JOIN CATEGORY_LINK t2
ON t1.id=t2.id_from
WHERE t2.title= 'H' and distance=1;
```

Рис. 20. запит на виведення дочірніх (*а*) та батьківського (*б*) елементів вузла з вершиною title= 'H'

Щоб знайти шлях від одного вузла до іншого в цій моделі, достатньо відобразити лише ті елементи, які пов'язані з кінцевим вузлом, виключивши при цьому вузли, що ведуть до початкового вузла (рис. 21).

```
SELECT id, title
FROM CATEGORY_Closure_Table t1
LEFT JOIN CATEGORY_LINK t2
ON t1.id=t2.id_from
WHERE t2.id_from>=
        (SELECT id
        FROM CATEGORY_Closure_Table
        WHERE title='D')
and t2.id_to =
        (SELECT id
        FROM CATEGORY_Closure_Table
        WHERE title='K'
);
```

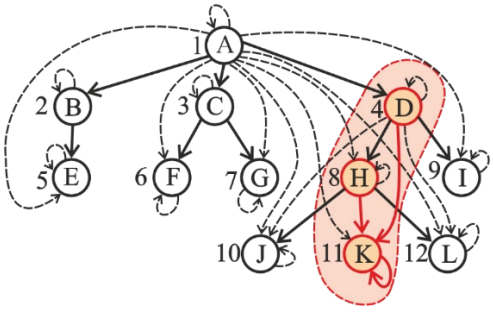


Рис. 21. Запит, який знаходить шлях від вузла 'D' до вузла 'K'

Подібним способом можна вивести всі вузли піддерева з певною вершиною, вибравши в результуючий набір всі вузли, які мають зв'язок з заданою вершиною, де поле id_from рівне id вершини піддерева (рис. 22). Для додавання нових даних потрібно буде добавляти дані в дві таблиці (рис. 23, *а*). Операція видалення всього піддерева виконується за допомогою операції DELETE з умовою, що видаляються всі вузли, які

мають зв'язок з вершиною піддерева (рис. 23, *б*). При видаленні, враховуючи використання правила ON DELETE CASCADE в таблиці CATEGORY_link будуть видалені всі зв'язки вузлів піддерева.

```
SELECT id, title
FROM CATEGORY_Closure_Table t1
LEFT JOIN CATEGORY_LINK t2
ON t1.id=t2.id_to
WHERE t2.id_from>=
        (SELECT id
        FROM CATEGORY_Closure_Table
        WHERE title='D');
```

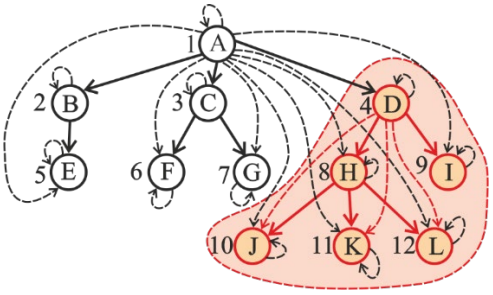


Рис. 22. Запит, який виведе піддерево вузла 'D'

а)
```
INSERT INTO CATEGORY_Closure_Table VALUES (101, 'M');
INSERT INTO CATEGORY_LINK (Id_from, Id_to, distance)
SELECT Id_from,101, distance+1 FROM CATEGORY_LINK
WHERE Id_to=6
UNION
SELECT 101,101, 0 FROM CATEGORY_LINK;
```

б)
```
DELETE FROM CATEGORY_Closure_Table
WHERE id IN (SELECT id_to FROM CATEGORY_link WHERE id_from=29);
```

Рис. 23. Запити додавання (*а*) та видалення (*б*) даних

Матеріалізований шлях (Materialized Path). На рис. 24 показано приклад моделі «Materialized Path». Ідея даної моделі полягає у зберіганні повного шляху для кожного вузла у дереві. У полі (path) зберігається ланцюжок всіх предків кожного вузла. За кількістю роздільників у шляху можна визначити глибину вкладеності вузла. Модель «Materialized Path» є наочним і найбільш інтуїтивно зрозумілим поданням дерева.

```
CREATE TABLE CATEGORY_Materialized_Path (
id INTEGER PRIMARY KEY,
title VARCHAR2(5) NOT NULL,
path VARCHAR(60) NOT NULL
);
```

ID	TITLE	PATH
1	A	A
2	B	A/B
3	C	A/C
4	D	A/D
5	E	A/B/E
6	F	A/C/F
7	G	A/C/G
8	H	A/D/H
9	I	A/D/I
10	J	A/D/H/J
11	K	A/D/H/K
12	L	A/D/H/L

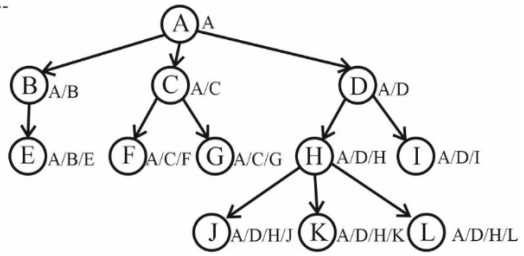


Рис. 24. Подання дерева у вигляді матеріалізованих шляхів.

Серед переваг даної моделі можна виділити простоту написання запиту на отримання нащадків та батьківських вузлів. Крім того, легко виконати простий запит на видалення вузла, а також на переміщення вузла. До недоліків можна віднести відсутність посилкової цілісності, а також складність операції підрахунку рівня вкладеності вузла, хоча в більшості баз даних є необхідні функції для ефективної роботи з рядками. Основним недоліком є неефективність запитів через пошук по підрядку.

На рис. 25 подано запит, який повертає всі вузли дерева, які не мають нащадків. В цьому запиті підзапит перевіряє, чи існує запис у таблиці CATEGORY_Materialized_Path, де значення path у t2 починається з path у t1, додаючи символ «/». Це означає, що такий запис є нащадком вузла t1. Якщо такий запис існує, то t1 не буде обраний основним запитом, оскільки він не є листком.

```
SELECT id, title
FROM CATEGORY_Materialized_Path t1
WHERE NOT EXISTS
        (SELECT 1
        FROM CATEGORY_Materialized_Path t2
        WHERE t2.path LIKE t1.path || '/%' );
```

Рис. 25. Запит, який повертає всі вузли дерева, які є листками

На рис. 26 подано запити на виведення дочірніх (*а, б*) та батьківського (*в, г*) елементів вузла з вершиною title= 'H'.

а)
```
SELECT t1.id, t1.title
FROM CATEGORY_Materialized_Path t1
JOIN CATEGORY_Materialized_Path t2
ON t1.path LIKE t2.path || '/%'
WHERE t2.title= 'H' AND
LENGTH(REPLACE(t1.path, t2.path || '/', '')) -
LENGTH(REPLACE(REPLACE(t1.path, t2.path || '/', ''), '/', '')) = 0;
```

б)
```
WITH ParentNode AS (
SELECT id, path
FROM CATEGORY_Materialized_Path
WHERE title = 'H'
)
SELECT t1.id, t1.title
FROM CATEGORY_Materialized_Path t1
JOIN ParentNode p ON t1.path LIKE t2.path || '/%'
WHERE REGEXP_LIKE(t1.path, '^' || t2.path || '/[^/]+$');
```

в)
```
SELECT t2.id, t2.title
FROM CATEGORY_Materialized_Path t1
JOIN CATEGORY_Materialized_Path t2
ON t1.path LIKE t2.path || '/%'
WHERE t1.title= 'H'
ORDER BY LENGTH(t2.path) DESC
FETCH FIRST 1 ROWS ONLY;
```

г)
```
SELECT t1.id, t.title
FROM CATEGORY_Materialized_Path t
WHERE
        (SELECT path
        FROM CATEGORY_Materialized_Path
        WHERE title= 'H') LIKE t.path || '/%'
ORDER BY LENGTH(t.path) DESC
FETCH FIRST 1 ROWS ONLY;
```

Рис. 26. Запити на виведення дочірніх (*а,б*) та батьківського (*в, г*) елементів вузла з вершиною title= 'H'

Перший запит (рис. 26, *а*) виведення дочірніх елементів знаходить прямі дочірні елементи вузла з title='H', використовуючи JOIN і перевіряє шлях з допомогою функції INSTR. Другий запит (рис. 26, *б*) досягає того ж результату, але спершу створює CTE (Common Table Expression) ParentNode, який містить шлях вузла з title= 'H', а потім використовує його для виконання основного запиту. Відмінність полягає в тому, що другий підхід із CTE може бути більш читабельним та ефективним при складніших умовах, забезпечуючи кращу структурованість коду.

Перший запит виведення батьківського елементу (рис. 26, *в*) знаходить найближчого предка вузла з title='H', об'єднуючи таблицю CATEGORY_Materialized_Path саму з собою на основі збігу шляхів і сортує результати за довжиною шляху у спадному порядку, повертаючи тільки один рядок. Другий запит (рис. 26, *г*) виконує аналогічну операцію, але використовує підзапит для отримання шляху вузла з title= 'H' і порівнює його з шляхами предків, сортує їх за довжиною шляху у спадному порядку і повертає тільки один рядок. Основна відмінність полягає у використанні підзапиту в другому запиті замість JOIN, що може вплинути на продуктивність у залежності від розміру даних.

Для того, щоб вивести шлях від одного вузла дерева до іншого достатньо витягнути підрядок зі значення стовпця path для рядка кінцевого вузла починаючи з позиції, де вперше зустрічається значення title з рядка початкового вузла (рис. 27). Використання функції INSTR визначає позицію title початкового вузла у рядку path, а SUBSTR витягує частину рядка path, починаючи з цієї позиції до кінця. Таким чином, результатом буде підрядок path, який починається зі значення title початкового вузла.

```
SELECT SUBSTR(path,
        INSTR(path, (Select title from CATEGORY_Materialized_Path where title='D')))
        AS trimmed_path
FROM CATEGORY_Materialized_Path
WHERE title='K';
```

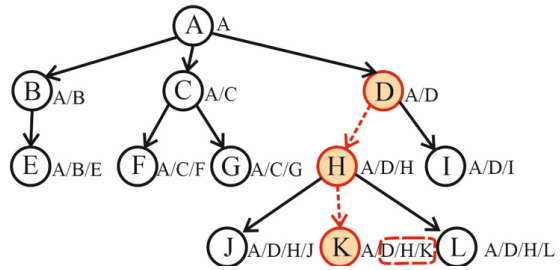
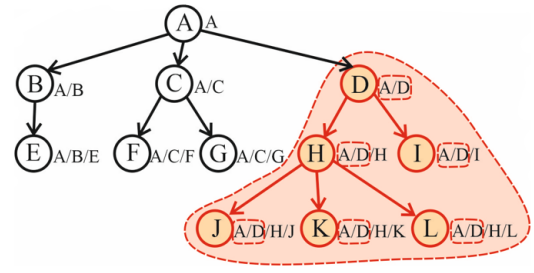


Рис. 27. Запит, який знаходить шлях від вузла 'D' до вузла 'K'

Для виведення піддерева потрібно перевірити для всіх рядків, де значення path починається з path батьківського вузла піддерева (рис. 28). Використання функції INSTR (рис. 28) перевіряє, чи path з вибраного рядка починається з того ж значення, що і path батьківського вузла (повертає 1 для початку рядка). Це означає, що запит витягує всі вузли, які є дочірніми або рівними батьківському вузлу.




```
SELECT id, title
FROM CATEGORY_Materialized_Path
WHERE INSTR(path,
        (Select path from CATEGORY_Materialized_Path where title='D'))=1;
```

Рис. 28. Запит, який виведе піддерево вузла 'D'

Для додавання нового вузла потрібно об'єднати шлях path з батьківського вузла з і нового title (рис. 29, *а*). Для видалення вузла дерева з усіма його нащадками потрібно видалити всі рядки з таблиці CATEGORY_Materialized_Path, де шлях містить рядок шляху з батьківського вузла піддерева (рис.29, *б*).

а)
```
INSERT INTO CATEGORY_Materialized_Path (id, title, path)
VALUES (101, 'M', CONCAT((
        SELECT path||'/'
        FROM CATEGORY_Materialized_Path
        WHERE id = 25), 'M')
        );
```

б)
```
DELETE FROM CATEGORY_Materialized_Path
WHERE INSTR(path, (Select path from CATEGORY_Materialized_Path where id=101))=1;
```

Рис. 29. Додавання (*а*) та видалення (*б*) вузла дерева з усіма дочірніми елементами.

Кількісні показники часу виконання запитів. В табл. 2 подано порівняльна характеристика складності схеми бази даних для різних моделей.

Таблиця 2. Порівняльна характеристика складності схеми бази даних

Критерії порівняння	Adjacency List	Nested Sets	Closure Table	Materialized Path
Таблиці	1	1	2	1
Посилання	1	0	2	0
Колонки	3	4	5	3
Кількість рядків в таблицях для 100 вузлів дерева і кількості рівнів 12	100	100	100 546	100
Кількість рядків в таблицях для 1000 вузлів дерева і кількості рівнів 13	1000	1000	1000 7129	1000
Кількість рядків в таблицях для 10000 вузлів дерева і кількості рівнів 18	10000	10000	10000 93110	10000

Для кожного запиту було проведено 10 вибірок, визначено час виконання запиту кожної вибірки та знайдено їх середнє значення. Проведемо оцінювання затрат часу на вибірку всіх листків дерева, які зберігаються в базі даних всіма розглянутими методами. Згенероване випадковим чином дерево, яке містить 100, 1000, 10000 вузлів мало 43, 504 та 4977 вузлів відповідно, які не мають нащадків. Як видно з рис. 30 вибірка всіх вузлів дерева, які не мають нащадків, для моделі Materialized Paths виконуються набагато довше порівняно з іншими моделями, що може бути пов'язано з тим, що матеріалізовані шляхи використовують рядкові операції для визначення ієрархічних зв'язків. Запити, які включають функції обробки рядків, такі як INSTR, LIKE та SUBSTR, можуть бути менш ефективними, особливо на великих наборах даних. Ці функції вимагають додаткових обчислень для кожного рядка, що може значно збільшити час виконання запиту. Шлях у форматі рядка може бути досить довгим, особливо для глибоко вкладених ієрархій. Обробка довгих рядків займає більше часу і ресурсів, що також впливає на загальну продуктивність запитів. Для порівняння, інші моделі, такі як Adjacency List, Nested Sets або Closure Table, показують кращий результат, при чому запит моделі Adjacency List з використанням підзапиту виконується швидше.

На рис. 31 подано час виконання запиту виведення прямих дочірніх вузлів, батьківського вузла, який знаходиться на 5 рівні. Для таблиць з кількістю записів 100, 1000, 10000 результатом вибірки було 3, 5 і 10 рядків відповідно. Як видно з рис. 31 час виконання вибірки для моделі Adjacency List та Closure Table майже не залежить від

кількості даних в таблиці. Тоді як для моделей Nested Sets та Materialized Path час виконання вибірки значно зростає при збільшенні кількості даних в таблиці.

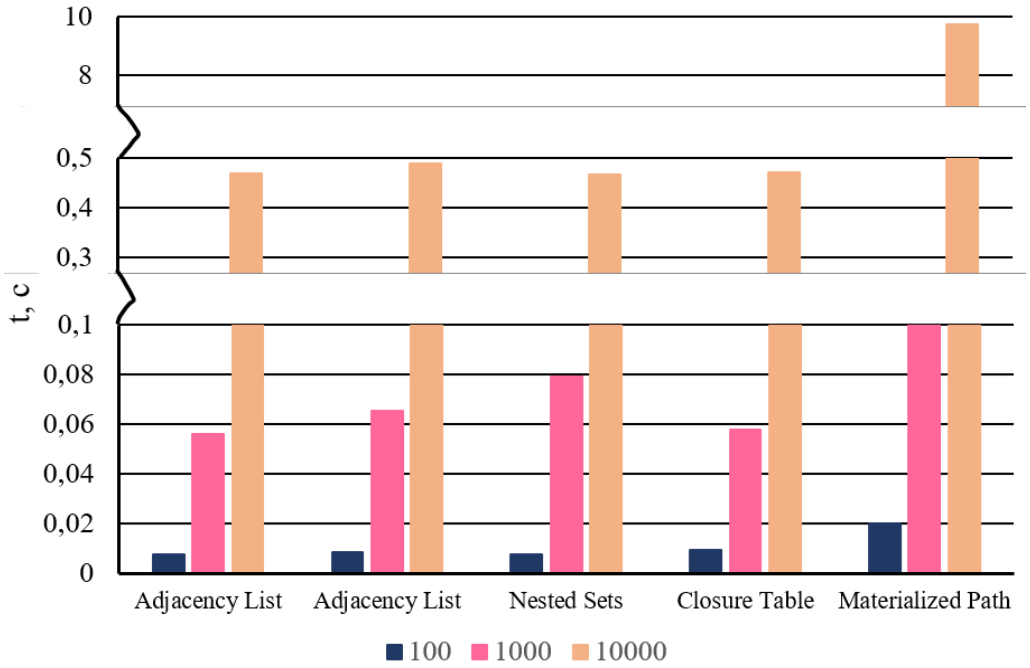


Рис. 30. Час виконання запиту виведення листків дерева (Adjacency List1: запит на рис. 2, *а*; Adjacency List2: запит на рис. 2, *б*; Nested Sets: запит на рис. 11; Closure Table: запит на рис. 19; Materialized Path: запит на рис. 25).

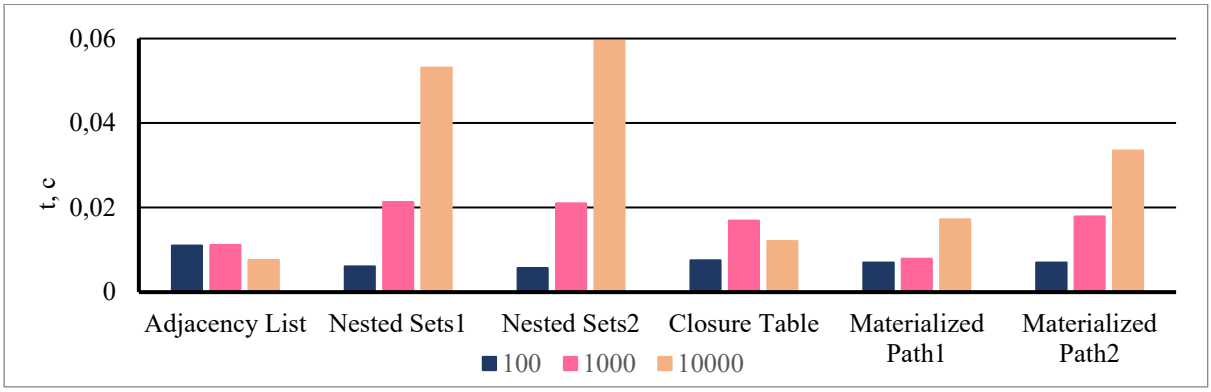


Рис. 31. Час виконання запиту виведення прямих дочірніх вузлів, батьківського вузла, який знаходиться на 5 рівні (Adjacency List1: запит на рис. 3, *а*; Nested Sets1: запит на рис. 12*а*; Nested Sets1: запит на рис. 12, *б*; Closure Table: запит на рис. 20, *а*; Materialized Path1: запит на рис. 26, *а*; Materialized Path2: запит на рис. 26, *б*)

Для малої кількості даних (100 записів) час виконання запиту на виведення прямих дочірніх вузлів найменший для моделі Nested Sets. Для моделі Materialized Path час виконання вибірки із збільшенням кількості даних в таблицях зростає повільніше ніж для моделі Nested Sets. Nested Sets є потужним підходом для швидкого отримання всіх нащадків. Але через необхідність здійснювати складні діапазонні пошуки і додаткові перевірки для отримання прямих дочірніх вузлів, цей підхід є менш ефективним для великих наборів даних у порівнянні з іншими підходами, такими як Adjacency List, Closure Table та Materialized Path. У Adjacency List кожен вузол просто містить посилання на батьківський елемент, що робить запит на отримання прямих дочірніх вузлів простим SELECT, де умова базується лише на порівнянні значення ідентифікатора

батька. Це дає швидші результати, особливо на великих наборах даних. Closure Table зберігає всі можливі шляхи між вузлами, що дозволяє теж швидко отримувати прямі дочірні елементи через простий SELECT з умовою, яка перевіряє відстань між вузлами. Обидва запити моделі Materialized Path (рис.31) обмежують діапазон пошуку за допомогою перевірки на префікс у шляху (LIKE t2.path || '/%'). Це дозволяє значно зменшити кількість рядків, які потрібно перевірити, зосереджуючи обчислення лише на тих шляхах, які потенційно є дочірніми. Час виконання запиту виведення батьківського вузла (рис. 32) для Adjacency List мало залежить від кількості записів в таблиці, для інших моделей час дещо зростає.

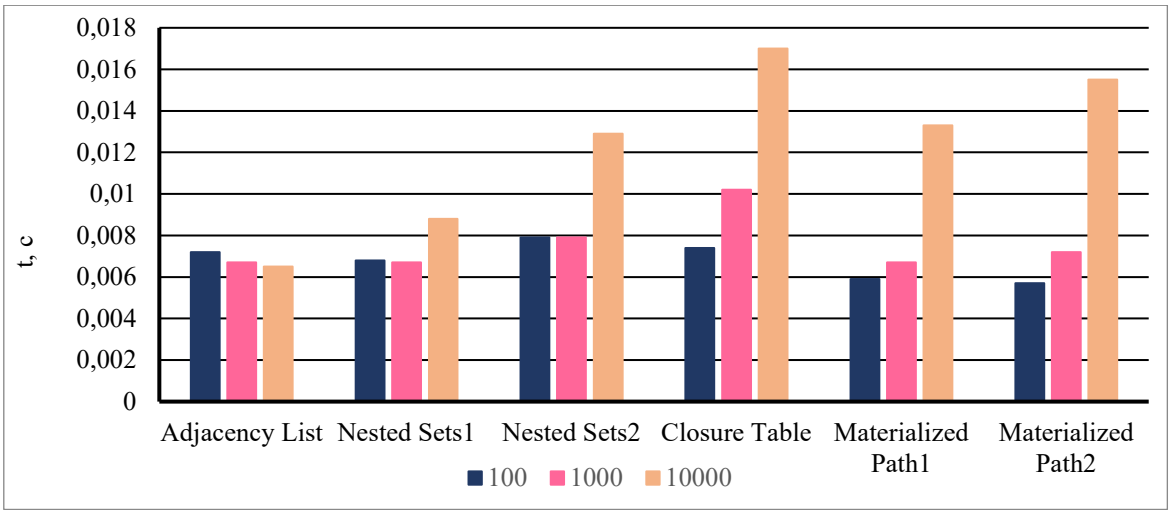


Рис. 32. Час виконання запиту виведення батьківського вузла, для вузла який знаходиться на 5 рівні (Adjacency List: запит на рис. 3 *б*; Nested Sets1: запит на рис. 13,*а*; Nested Sets1: запит на рис. 13, *б*; Closure Table: запит на рис. 20,*б*; Materialized Path1: запит на рис. 26, *в*; Materialized Path2: запит на рис. 26, *г*)

На рис. 33 подано час виконання запиту виведення шляху від вузла який знаходиться на 2 рівні до вузла, який знаходиться на 11 рівні. Для таблиць з кількістю записів 100, 1000, 10000 результатом вибірки було 10 рядків.

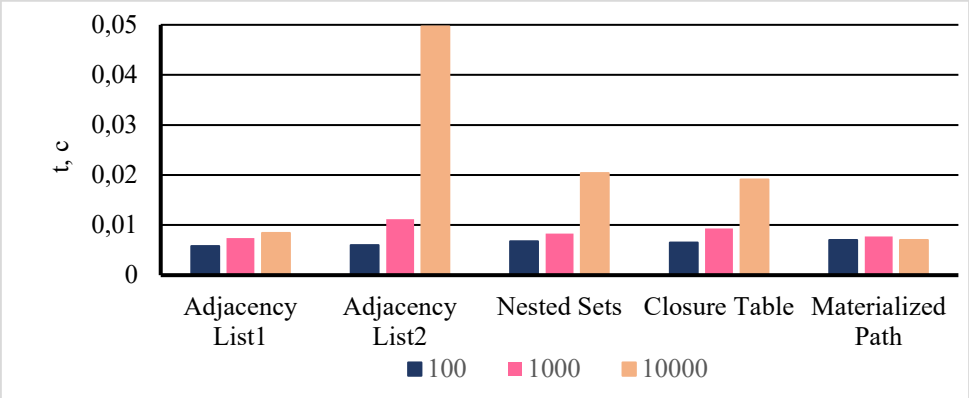


Рис. 33. Час виконання запиту виведення шляху від вузла який знаходиться на 2 рівні до вузла, який знаходиться на 11 рівні (Adjacency List1: запит на рис. 6; Adjacency List2: запит на рис. 7; Nested Sets: запит на рис. 14; Closure Table: запит на рис. 21; Materialized Path: запит на рис. 27)

Найкращі результати для малої кількості даних показує модель Adjacency List, але при кількості даних більше 1000 рекурсивний запит моделі Adjacency List стає

неефективним, а запит написаний шляхом з'єднання таблиці самої з собою громіздким. Крім того, для того щоб написати цей запит потрібно знати скільки рівнів міститься між кінцевими вершинами шляху. Найкращі результати для великої кількості даних показує модель Materialized Path. У Nested Sets потрібно виконувати складні діапазонні операції, щоб знайти всі нащадки вузла і відслідкувати шлях між вузлами, що є затратним на великих наборах даних. Closure Table для визначення точного шляху вимагає додаткових фільтрацій, поданих у вигляді підзапитів, що додає складності запиту і впливає на його час виконання.

Для виведення всього піддерева (рис. 34) найкращий результат показує модель Closure Table. Було виконано вибірку піддерева з вершиною яка знаходиться на 5 рівні де в результаті для таблиць з кількістю записів 100, 1000, 10000 отримали 13, 249 і 838 рядків відповідно. Непогані результати показує модель Materialized Path для малої кількості даних в базі.

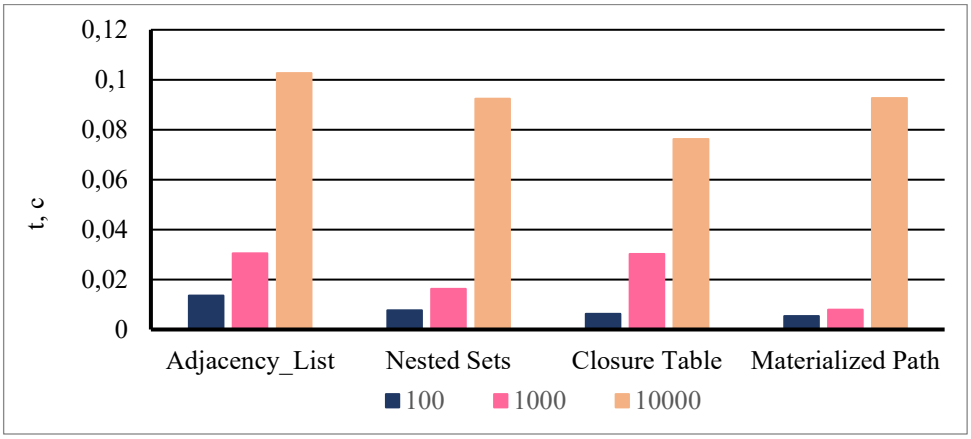


Рис. 34. Час виконання запиту виведення піддерева з вершиною, яка знаходиться на 5 рівні (Adjacency List: запит на рис. 8; Nested Sets: запит на рис. 15; Closure Table: запит на рис. 22; Materialized Path: запит на рис. 28)

Час виконання запитів на додавання або видалення листка в деревоподібній структурі даних майже не залежить від кількості рядків у таблиці (рис. 35), оскільки ці операції мають локальну природу і не вимагають повного перегляду всіх рядків у таблиці. Операції додавання або видалення листка дерева працюють тільки з невеликою підмножиною рядків у таблиці, зосереджуючись на вузлах, що безпосередньо залучені в операцію (тобто вузол, що додається або видаляється, та його батько чи предки).

Додавання нового листка в Closure Table вимагає вставки кількох рядків, що представляють зв'язки нового вузла з його предками. Однак кількість вставок лінійно залежить лише від кількості предків, а не від загальної кількості рядків у таблиці. Цей запит виконується довше ніж в інших моделях. Видалення листка також вимагає видалення кількох рядків з таблиці зв'язків моделі Closure Table.

В Nested Sets додавання нового листка може вимагати оновлення значень «ліва» і «права» межа для деяких вузлів, але це оновлення стосується лише підмножини вузлів і зазвичай не залежить від загальної кількості рядків. Видалення листка також передбачає оновлення значень «ліва» і «права» для підмножини вузлів, що є локальною операцією цього вузла. Знову ж таки, ця операція не вимагає перегляду всієї таблиці.

На рис. 36 подано час виконання запиту на видалення всього піддерева батьківського вузла, який знаходиться на 5 рівні. Для таблиць з кількістю записів 100, 1000, 10000 було видалено 63, 53 і 868 рядків відповідно. Час виконання запиту на видалення всього піддерева (усіх нащадків батьківського вузла) для моделі Closure Table

могло бути ефективним, але видалення великої кількості рядків займає деякий час, так як додатково видаляються всі дані про зв'язки видалених вузлів. В результаті з таблиці зв'язків було видалено 546, 7129 та 93110 рядків для таблиць з даними з 100, 1000 і 10000 рядків відповідно. У Nested Sets вузли піддерева мають значення «ліва» і «права» межа, що дозволяє швидко визначити всі вузли піддерева за допомогою діапазонного запиту. Видалення піддерева може бути виконано за один SQL-запит, але ще певний час займає оновлення значення «ліва» і «права» межа для решти вузлів, що може бути повільним для великих дерев. У Materialized Path можна швидко визначити всі нащадки вузла шляхом пошуку підрядка в рядку, тому видалення піддерева відбувається найшвидше, порівняно з іншими моделями.

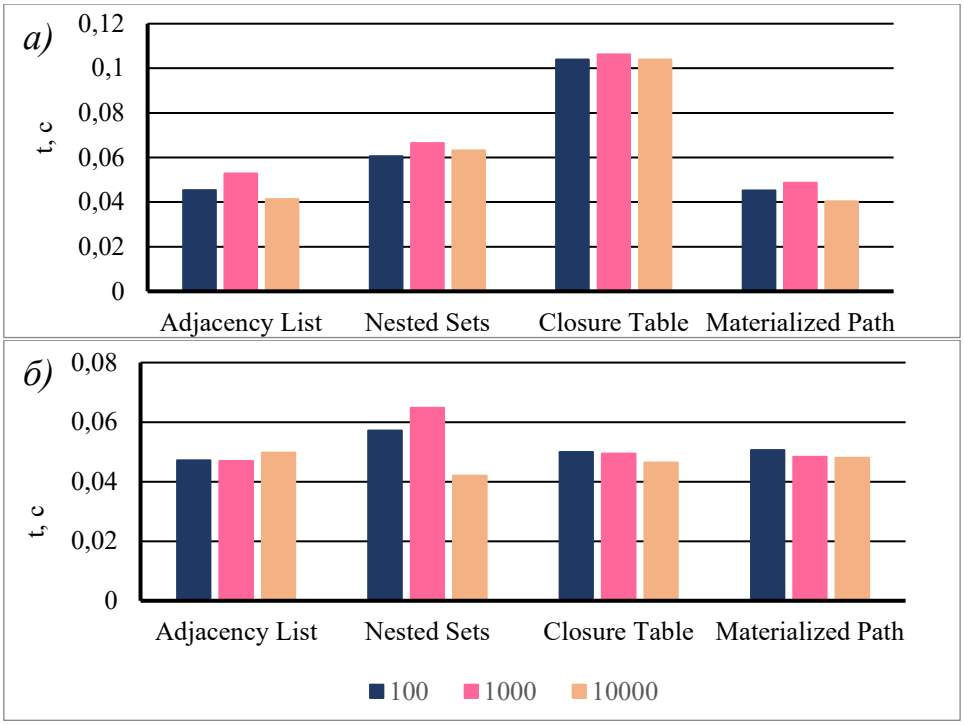


Рис. 35. Час виконання запиту на додавання нового листка дерева (*а*) та видалення вузла дерева, яке є листком (*б*)

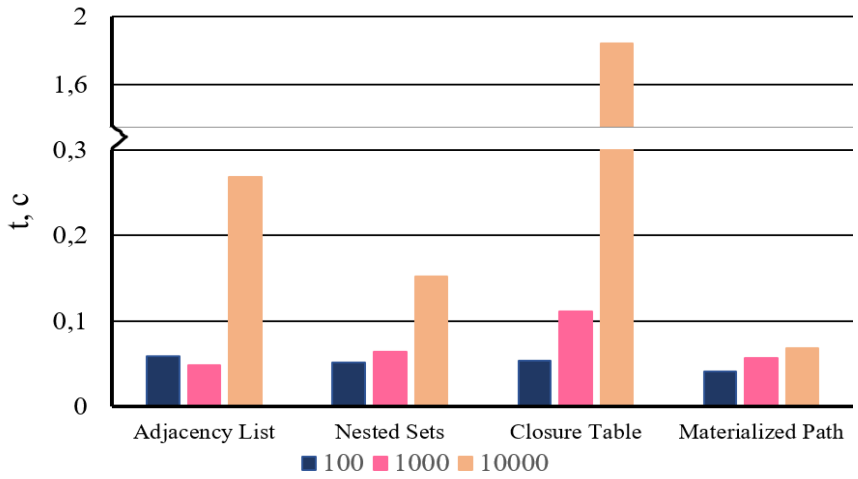


Рис. 36. Час виконання запиту на видалення піддерева з вершиною, яка знаходиться на 5 рівні.

Висновки. У роботі розглянуто основні способи представлення ієрархічних структур у реляційних базах даних та типові запити до цих структур даних. До розглянутих методів належить Adjacency List, Nested Sets, Closure Table та Materialized Path. Були отримані кількісні показники часу вибірки даних, які представлені в базі даних розглянутими методами. Час виконання базових операцій, таких як додавання, видалення та переміщення вузлів, суттєво залежить від обраної моделі. Найменший час на додавання та видалення листків демонструють Adjacency List та Materialized Path, тоді як Nested Sets та Closure Table потребують більше ресурсів для складних операцій, таких як видалення піддерева. Запити на вибірку піддерева найефективніше виконуються в моделі Materialized Path за допомогою простих операцій над рядками. Операції додавання і видалення окремих вузлів не демонструють значної залежності від загальної кількості рядків у таблиці, оскільки ці операції здебільшого локальні. Однак, при обробці великих наборів даних, ефективність моделей може суттєво змінюватися. Наприклад, видалення піддерева в Closure Table є значно менш ефективним порівняно з іншими моделями через необхідність видалення великої кількості вузлів з таблиці зв'язків.

Кожен з методів зберігання деревоподібних структур у реляційній базі даних має свої сильні та слабкі сторони. Вибір відповідного методу залежить від конкретних вимог завдання, яке необхідно вирішити.

Бібліографія

1. Креневич А.П. Алгоритми і структури даних. Підручник. – К.: ВПЦ "Київський Університет", 2021. – 200 с
2. Joe Celko's Trees and Hierarchies in SQL for Smarties. (2004). Elsevier. https://doi.org/10.1016/b978-1-55860-920-4.x5000-4
3. Celko J. Trees and Hierarchies in SQL for Smarties. – Morgan-Kaufmann, 2012. – 296 p.
4. Маркітан В. О., Возняк М. А., Булатецька Л. В. Деревовидні структури в SQL. *Математика. Інформаційні технології. Освіта.* : матеріали XI міжнар. науково-практ. конф. Луцьк, 3–5 червн. 2022 р. Луцьк, 2022. С. 109–111.
5. Зберігання ієрархічих структур в реляційних базах даних. / В. О. Маркітан, М. А. Возняк, Л. В. Булатецька, В. В. Булатецький. *Кібербезпека: освіта, наука, техніка.* 2022. Т. 4, №16. С. 85–97. DOI: https://doi.org/10.28925/2663-4023.2022.16.8597
6. Буй,Д., & Поляков,С. (2010). Рекурсивні запити в SQL-подібних мовах: приклади, змістова і формальна семантика. Проблеми програмування, (2-3).
7. Буй,Д., & Поляков,С. (2010). Композиційна семантика рекурсивних виразів та їхніх узагальнень в SQL-подібних мовах. Наукові записки НаУКМА. Комп'ютерні науки., 112, 21–26.
8. Дерево каталогів NESTED SETS (вкладені множини) і управління ним | open2web. *open2web | open2web*. URL: https://open2web.com.ua/blog/derevo-katalogov-nested-sets-vlozhennye-mnozhestva-i-upravlenie-im.html (дата звернення: 10.08.2024).

References

1. Krenevych A.P. Alhorytmy i struktury danykh. Pidruchnyk. – K.: VPTs "Kyivskyi Universytet", 2021. – 200 s
2. Joe Celko's Trees and Hierarchies in SQL for Smarties. (2004). Elsevier. https://doi.org/10.1016/b978-1-55860-920-4.x5000-4
3. Celko J. Trees and Hierarchies in SQL for Smarties. – Morgan-Kaufmann, 2012. – 296 p.
4. Markitan V. O., Vozniak M. A., Bulatetska L. V. Derevovydni struktury v SQL. Matematyka. Informatsiini tekhnolohii. Osvita. : materialy KhI mizhnar. naukovo-prakt. konf. Lutsk, 3–5 chervn. 2022 r. Lutsk, 2022. S. 109–111.
5. Zberihannia iierarkhichykh struktur v reliatsiinykh bazakh danykh. / V. O. Markitan, M. A. Vozniak, L. V. Bulatetska, V. V. Bulatetskyi. Kiberbezpeka: osvita, nauka, tekhnika. 2022. T. 4, №16. S. 85–97. DOI: https://doi.org/10.28925/2663-4023.2022.16.8597
6. Bui,D., & Poliakov,S. (2010). Rekursyvni zapyty v SQL-podibnykh movakh: pryklady, zmistova i formalna semantyka. Problemy prohramuvannia, (2-3).
7. Bui,D., & Poliakov,S. (2010). Kompozytsiina semantyka rekursyvnykh vyraziv ta yikhnikh uzahalnen v SQL-podibnykh movakh. Naukovi zapysky NaUKMA. Kompiuterni nauky., 112, 21–26.
8. Derevo katalohiv NESTED SETS (vkladeni mnozhyny) i upravlinnia nym | open2web. open2web | open2web. URL: https://open2web.com.ua/blog/derevo-katalogov-nested-sets-vlozhennye-mnozhestva-i-upravlenie-im.html (data zvernennia: 10.08.2024).