МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ВОЛИНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЛЕСІ УКРАЇНКИ
Кафедра комп'ютерних наук та кібербезпеки

На правах рукопису

БУЛАТЕЦЬКА ЛЕСЯ ВІТАЛІЇВНА

АНАЛІЗ МЕТОДІВ МОДЕЛЮВАННЯ ІЄРАРХІЧНИХ СТРУКТУР В РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ

Спеціальність: 122 Комп'ютерні науки
Освітньо-професійна програма: Комп'ютерні науки та інформаційні технології
Робота на здобуття освітнього ступеня «магістр»

Науковий керівник:
Собчук Андрій Валентинович,
доктора філософії, доцент

РЕКОМЕНДОВАНО ДО ЗАХИСТУ
Протокол №_____
засідання кафедри комп'ютерних наук
та кібербезпеки
від _____ 2024 р.
Завідувач кафедри

(_____)___Гришанович Т. О._____

ЛУЦЬК – 2024

ЗМІСТ

ВСТУП

Актуальність теми. Реляційні бази даних здебільшого призначені для зберігання та обробки табличних даних, де взаємозв'язки між даними описуються через реляційні таблиці. Кожен рядок реляційної таблиці містить дані про один окремий об'єкт, а кожен стовпець – характеристики цих об'єктів. При такій організації, рядки є незалежними один від одного. Проте, в багатьох реальних сценаріях дані мають ієрархічну структуру, наприклад, організаційні структури компаній, каталоги продуктів, сторінки вебсайтів тощо.

Таблиці, що містять батьківські та дочірні елементи, часто використовуються для представлення ієрархічної інформації. Типовий приклад таких даних – категорії, яких може бути досить багато з великим рівнем вкладеності одна в одну. В бізнесі така структура може відображати організаційну структуру компанії, де кожен відділ має підвідділи, а ті, в свою чергу, підрозділи. Кожен запис у таблиці представляє певний рівень в ієрархії організації. У вебсервісах ієрархічні структури часто застосовуються для категоризації товарів. Для того щоб користувачі могли легко знайти потрібний продукт, товари організовані в категорії та підкатегорії, які утворюють деревовидну структуру. Для представлення генеалогічної інформації, наприклад, родоводів чи сімейних дерев, така структура також є корисною. Кожна особа може мати батьків, дітей, братів та сестер, і ця інформація може бути організована в таблиці з ієрархічною структурою. У бібліотеці використовують ієрархічну структуру для організації книжкових колекцій, що дозволяє легко знаходити книги за їх тематикою та підтематикою. У ресторані може бути організована ієрархічна структура меню, де головні категорії включають підкатегорії страв, а ті, в свою чергу, можуть включати окремі страви.

Зберігання ієрархічних структур (деревовидних структур) у реляційних базах даних є складним завданням, оскільки система управління реляційними базами даних і мова SQL не мають вбудованих механізмів для ефективного зберігання та управління такими структурами.

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.

Для досягнення мети потрібно виконати наступні завдання:

- розглянути основні моделі подання ієрархічних структур у реляційних базах даних;
- описати переваги та недоліки кожної моделі у контексті різних типів операцій та запитів;
- розробити та реалізувати запити для виконання основних операцій управління даними для кожної з моделей;
- провести експериментальний аналіз часу виконання основних операцій та запитів для кожної з моделей на різних наборах даних;
- провести порівняльний аналіз ефективності кожної моделі для різних типів операцій та запитів;
- визначити оптимальні моделі для конкретних сценаріїв використання.

Об’єкт дослідження: методи та моделі зберігання і обробки деревоподібних структур у реляційних базах даних.

Предмет дослідження: ефективність операцій управління даними та запитів на вибірку даних у різних моделях подання деревоподібних структур.

Публікації:

1. Булатецька Л. В., Булатецький В. В. Методи моделювання ієрархічних структур в реляційних базах даних. *Прикладні проблеми комп’ютерних наук, безпеки та математики.* 2024. № 1. С. 47–65. URL: https://apcssm.vnu.edu.ua/index.php/Journalone/article/view/126

РОЗДІЛ І
ПОДАННЯ ІЄРАРХІЧНИХ СТРУКТУР В РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ

1.1. Теоретичні концепції графів.

Графи широко використовуються для моделювання різних концепцій теорії програмування. Основні поняття теорії графів описані у спеціальній літературі [1–10]. У цій роботі перераховані лише визначення, що використовуються для моделювання ієрархічних структур в реляційних базах даних.

Дерево – це зв'язаний ациклічний граф. Зв'язаність означає наявність шляхів (простого ланцюга) між будь-якою парою вершин, ациклічність – відсутність циклів, тобто між будь-якими парами вершин є тільки один шлях. Формально деревом називають скінченну множину вузлів T, з'єднаних гілками (ребрами), серед яких є один виділений вузол t, який називають коренем, а інші вузли розбиті на $m \geq 0$ множин, що не перетинаються $T_1,T_2,...,T_m$. Кожна така множина T_i, в свою чергу, є деревом та називається піддеревом вузла t. [5–10]

Граф є орієнтованим (орграф), якщо всі його ребра орієнтовані, тобто суміжні вершини поділені на початкову та кінцеву. Орієнтоване ребро називається дугою. Ступенем вузла називають кількість нащадків даного вузла. Ступенем дерева називають найбільшу ступінь всіх його вузлів. В орієнтованому дереві є лише одна вершина з нульовим ступенем входу (у неї не ведуть дуги), вона називається коренем дерева, а всі інші вершини мають ступінь входу 1 (в них веде рівно по одній дузі). Вершини з нульовим ступенем результату (з яких не виходить ні одна дуга) називаються кінцевими вершинами або листками. Якщо в дереві існує шлях з вершини a до вершини b, то вершина a називається предком вершини b, а вершина b – нащадком вершини a. Шляхом з вершини n_i до вершини n_j називається послідовність вершин, де для всіх k, $i \leq k \leq j$, вузол n_k є предком вузла n_{k+1}. Довжиною шляху називається число, на одиницю менше числа кількості вершин, які утворюють цей шлях. Висотою вершини називається

довжина найдовшого шляху з цієї вершини до якогось листка. Висота дерева збігається з висотою кореня. Глибина вершини визначається як довжина шляху (який єдиний у дереві) від кореня до цієї вершини. *N*-арне дерево (орієнтоване) – це орієнтоване дерево, в якому число вихідних дуг для будь-якої вершини не перевищує *N*. Вершини графа також часто називають «вузлом» дерева.

Граф порядку *N* називається навантаженим (зваженим графом), якщо його вершинам, або дугам присвоєно деякі мітки, наприклад, номери 1, 2, …, *N*, які називаються вагою або навантаженням. Ваги це додаткова інформація, що міститься у вузлі дерева та не впливає на його структуру. Ваги ребер можуть представляти різні значення в залежності від контексту застосування, такі як відстань, час, вартість, пропускна здатність тощо. Навантажені графи використовуються для моделювання відстаней між містами або часу, необхідного для подорожі між ними. Ваги можуть відповідати пропускній здатності з'єднань або затримкам передачі даних у комп'ютерних мережах, а також відображати силу зв'язків або частоту взаємодії між людьми в соціальних мережах. Пара вершин *u* та *v* графа є суміжною, якщо множина {*u, v*} є ребром. [2]

Довільний навантажений граф порядку *N* можна подати у вигляді квадратної матриці *N*×*N*, де на перетині *i*-го рядка і *j*-го стовпця стоїть значення, яке відповідає мітці (вазі) ребра між вершинами *i* і *j*, якщо вони суміжні. Якщо вершини *i* і *j* не суміжні, на перетині стоїть 0 або інше значення, яке позначає відсутність ребра. Таке подання називається матрицею суміжності [1-10]. Якщо граф ненавантажений, то значення комірки рівне 1, якщо між вершинами існує ребро. Якщо граф неорієнтований, то така матриця буде симетричною відносно основної діагоналі. Для орієнтованого графа матриця втрачає симетричність. Такий метод подання не є оптимальним з точки зору економії пам'яті. Проте він дозволяє миттєво (за O(1)) перевірити наявність ребра між вершинами *i* та *j*. У випадку навантаженого графа, значення у відповідній комірці відображатиме вагу ребра.

Також навантажений граф порядку *N* можна подати у вигляді списку суміжності. Де для кожної вершини зберігається список всіх суміжних вершин разом із вагою ребра до кожної з них. Наприклад, якщо вершина *i* з'єднана з вершиною *j* ребром з вагою *w*, то це записується як пара (*j*,*w*) у списку для вершини *i*. Цей спосіб зберігання є набагато економнішим щодо використання пам'яті, але він не дозволяє перевірити наявність ребра між двома вершинами за O(1). Водночас, його перевагою є швидкий доступ до суміжних вершин. У цій роботі розглянуто орієнтовані дерева.

Розглянемо простий орієнтований граф (A, B, C, D, E, F, G, H, I, J, K, L) рис. 1.1. Дерево, приведене на рис 1.1., є деревом четвертого ступеня та висоти 3. Коренем дерева є вершина A. Листками дерева є вершини E, F, G, J, K, L. Вузол D є коренем піддерева, що складається із вузлів H, I, J, K, L. Рівень кореня (вершина A) рівний 0. Кожний нащадок кореневої вершини (B,C,D) є вузлом першого рівня, а наступні нащадки (E, F, G, H, I) – вузлами другого рівня. Дерево є рекурсивною структурою даних, так як кожне піддерево є також деревом. Дії з такими структурами даних простіше всього описувати за допомогою рекурсивних алгоритмів. [1–8]

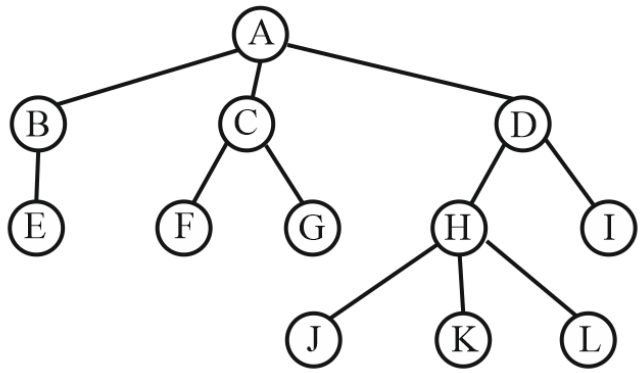


Рисунок 1.1. – Приклад представлення дерева

На рис 1.2. подана матриця суміжності для графа, поданого на рис.1.1 (1 означає наявність ребра, 0 – відсутність). На рис 1.3. подано список суміжності для того ж графа.

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	1	1	0	0	0	0	0	0	0	0
B	0	0	0	0	1	0	0	0	0	0	0	0
C	0	0	0	0	0	1	1	0	0	0	0	0
D	0	0	0	0	0	0	0	1	1	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	0	0	0	1	1	1
I	0	0	0	0	0	0	0	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 1.2. – Матриця суміжності графа поданого на рис 1.1

A: B, C, D

B: E

C: F, G

D: H, I

H: J, K, L

Рисунок 1.3. – Список суміжності для графа зображеного на рис 1.1.

1.2. Подання ієрархічних структур у реляційних базах даних

1.2.1. Реляційна модель даних

Реляційні бази даних відрізняються від інших своєю теоретичною основою, що базується на теорії відношень, що в свою чергу ґрунтується на теорії множин. Це теоретичне підґрунтя надає основу для організації даних і

здійснення операцій з ними в реляційних базах даних. У своїй основі, кожна таблиця в реляційній базі даних представляє собою відношення, яке в термінах теорії відношень є множиною кортежів. Кортежі у цьому контексті є елементами цих множин. Кожен стовпець таблиці відповідає атрибуту відношення, а кожен рядок – конкретному кортежу, що містить значення атрибутів. Операції, які виконуються над даними в реляційних базах даних, інтерпретуються як операції алгебри множин та числення предикатів. Наприклад, запити на вилучення, вставку, оновлення або видалення даних формулюються як комбінації проекцій, об’єднань, перетинів та інших операцій, які відомі з алгебри множин. [11–13]

Такий підхід забезпечує велику гнучкість та продуктивність у роботі з даними, оскільки дозволяє виконувати складні операції шляхом простих алгебраїчних маніпуляцій. В результатах реляційні бази даних знаходять широке застосування у різних сферах, від підприємницької аналітики та обробки транзакцій до наукових досліджень та адміністрування систем.

В реляційній моделі даних відношення є основною структурою для організації даних. Відношення в реляційній моделі даних представляють собою множину кортежів, де кожен кортеж складається з елементів інших множин. Кортеж у відношенні представляє собою упорядковану послідовність елементів, а не просто множину. Іншими словами, відношення представляють собою двовимірні таблиці, де дані організовані у вигляді рядків (кортежів) і стовпців (атрибути). Кожен стовпець відповідає певному типу даних (наприклад, цілі числа, рядки, дата тощо), а кожен рядок таблиці представляє конкретний запис або кортеж даних. Сукупність всіх можливих значень у стовпці називається доменом.

Наведемо властивості відношень реляційної моделі даних:

- кожен рядок в таблиці є унікальним;
- кожен стовпець таблиці містить тільки одне значення з атомарного домену (наприклад, ціле число, стрічка);
- кожне відношення має унікальне ім’я, яке ідентифікує його в межах бази даних;

- рядки в таблиці не мають фіксованого порядку, але можуть впорядковуватись за допомогою операцій запитів. [9–11]

Основна ідея відношень полягає в тому, що вони дозволяють структурувати дані у вигляді таблиць з різними атрибутами, що спрощує їх зберігання, оновлення та операції з ними. Реляційна модель даних, запропонована Едгаром Коддом, є основою сучасних систем управління базами даних (СУБД) і забезпечує ефективність та надійність обробки даних у широкому спектрі застосувань.

Технологічний прийом використання таблиць для відображення відношень є дуже ефективним, що сприяє широкому застосуванню баз даних у практиці. Факт того, що відношення представляють собою просто множини, дозволяє досліджувати проблеми реляційних баз даних на теоретичному рівні в математиці, зокрема в алгебрі множин і численні предикатів, а не тільки в контексті технологій. Це створює міцний теоретичний фундамент для розробки систем управління базами даних, конкретних баз даних і мови SQL, які використовуються для обробки даних. [11]

1.2.2. Підходи до моделювання ієрархічних структур в реляційній базі даних

Деревовидні структури в реляційних базах даних можна організувати кількома способами. Вибір конкретного методу для розв’язання задачі залежить від швидкості виконання основних операцій. Розглянемо найбільш поширені методи організації деревовидних структур в реляційних базах даних.

Списки суміжних вершин (Adjacency List). Метод моделювання ієрархічних структур даних у вигляді списків суміжності (Adjacency List) один із найпоширеніших способів побудови дерева в реляційній базі даних [15-18]. При такому моделюванні, кожен запис в таблиці відповідає вузлу дерева і зберігає його унікальний ідентифікатор (поле id, яке являється первинним ключем) та посилання на батьківський вузол (поле parent_id, яке описується як зовнішній ключ, що посилається на первинний ключ цієї ж таблиці). Якщо вузол немає

батьківського елемента, тобто він є коренем дерева, то атрибут parent_id має значення NULL. Такі ієрархічні відношення ще називають відношеннями предок-нащадок. Подання ієрархічної структури у вигляді списку суміжності демонструє найпростіший або базовий тип ієрархії, коли кожний нащадок має тільки одного предка. В загальному випадку у кожного нащадка може існувати декілька предків. У теорії графів списку суміжності з кількістю нащадків не більше ніж *n* відповідає орієнтований ациклічний граф.

Наведемо приклад запиту SQL на створення таблиці CATEGORY, яка представляє описану деревовидну структуру для реляційної бази даних Oracle (рис. 1.4). [17-18]

Використання зовнішніх ключів забезпечує збереження посилкової цілісності бази даних при зміні та видаленні записів, тому виникає проблема при видаленні вузла дерева, яке має нащадків, що пов’язані з даним вузлом зовнішніми ключами.

```
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

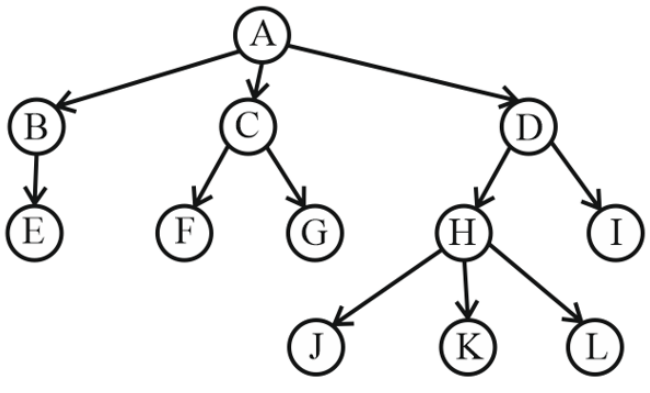


Рисунок 1.4. – SQL запит на створення таблиці, яка представляє деревовидну структуру списків суміжних вершин (Adjacency List) та дерево змодельоване у вигляді списків суміжності з елементами від A до L, які зберігаються в таблиці CATEGORY_ Adjacency_List.

При видаленні вузла такого дерева, для забезпечення автоматичного видалення всього піддерева потрібно використати правило ON DELETE CASCADE. На рис. 1.4 подано заповнене дерево з елементами від A до L, які зберігаються в таблиці CATEGORY_ Adjacency_List.

Ця модель має ряд переваг:

- посилкова цілісність даних;
- проста операція видалення вузла з усіма його нащадками завдяки цілісності посилання (ON DELETE CASCADE);
- проста операція переміщення вузла в дереві (достатньо оновити тільки одне поле parent_id у вузла, що переміщується);
- проста операція додавання вузла в дерево (необхідно додати вузол із нащадками до таблиці та оновити parent_id);
- проста операція отримання нащадків 1-го рівня вкладеності;
- проста операція отримання безпосереднього батьківського вузла.

Також ця модель має ряд недоліків [2, 5]. Для цієї моделі потрібні складні обмеження для підтримки цілісності даних. Для того, щоб виконати запит, потрібно пройти через усі вузли дерева. Складність обумовлена тим, що для отримання батьків або нащадків усіх рівнів вкладеності потрібно зробити безліч ітерацій, використовуючи для цього рекурсивний запит або збережену процедуру. Кожна ж ітерація – це перехід по дереву донизу (отримання нащадків), або вгору (отримання батьків), залежно від умови зв’язку.

Модель вкладених множин (Nested Sets Model of Hierarchies). Інший спосіб представлення дерев полягає в їх відображенні у вигляді вкладених множин. Оскільки SQL є мовою, орієнтованою на множини, цей підхід є кращою моделлю для моделювання ієрархічних структур.

Давайте визначимо таблицю організаційної схеми для представлення ієрархії категорій (рис. 1.5). Кожен вузол дерева має 2 значення: Left (значення ліворуч від вузла) та Right (значення праворуч від вузла) (рис. 1.5). Процедура визначення цих значень дуже проста і полягає у обході дерева ліворуч праворуч і нарощуванні лічильника на 1 під час проходження вузла. На рис 1.5

пунктирними стрілками показано процес обходу дерева. Результат обходу відображено у таблиці на рис 1.5. Слова Left та Right є зарезервованими і тому в таблиці CATEGORY_Nested_Sets вони замінені на LFT та RGT.

При такі організації деревовидної структури дуже легко отримати всіх нащадків для деякого вузла. На рис. 1.6. подано приклад запиту та його виконання, щоб отримати нащадків для вузла D, у якого Left = 12 і Right = 23 [19-22].

```
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

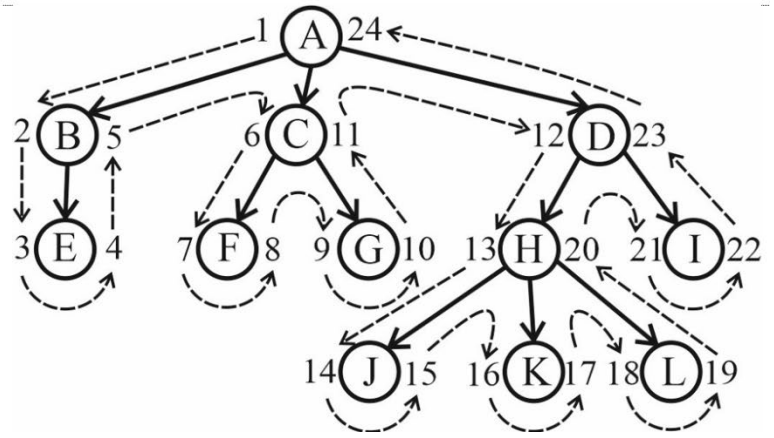


Рисунок 1.5. – Дерево змодельоване у вигляді вкладених множин з елементами від A до L, які зберігаються в таблиці CATEGORY

```
SELECT * FROM CATEGORY_Nested_Sets
WHERE LFT>12 AND RGT<23;
```

ID	TITLE	LFT	RGT
8	H	13	20
9	I	21	22
10	J	14	15
11	K	16	17
12	L	18	19

Рисунок 1.6. – Результат виконання запиту для отримання нащадків вузла D

А для отримання повного списку батьків для вузла D необхідно виконати запит поданий на рис 1.7.

```
SELECT * FROM CATEGORY_Nested_Sets
WHERE LFT<12 AND RGT>23;

 ID TITLE        LFT        RGT
--- ----- ---------- ----------
  1 A              1         24
```

Рисунок 1.7 – результат виконання запиту для отримання батьків вузла D

Перевагами цієї моделі є проста операція отримання нащадків без рівня вкладеності та проста операція отримання батьків. 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 «Вкладених множин» 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.

Таблиці зв’язків (Closure Table). Спосіб подання дерева у вигляді таблиці зв’язків показаний на рис. 1.8. При такому моделюванні список вузлів можна зберігати в одній таблиці, а зв’язки в іншій, (рис. 1.8). Перша таблиця зберігає тільки назви вузлів в полі title та ідентифікатор вузла.

Друга таблиця зберігатиме зв’язки між кожним вузлом дерева і вузлами, що знаходяться на шляху від нього до вершини, включаючи саму вершину дерева. Зв’язок задається трьома числами: ідентифікатором вузла, від якого починається шлях, ідентифікатором кінцевого вузла на шляху та відстанню –

цілим числом, що дорівнює кількості дуг між ними. Тобто, відстань між нащадком і батьком дорівнює 1, між нащадком і батьком батька дорівнює 2 і т.д. аж до самого кореня.

Як видно з рис. 1.8. одним з недоліків такого подання деревовидної структури – це багато записів у таблиці CATEGORY_link, які необхідні для опису всіх зв'язків.

```
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

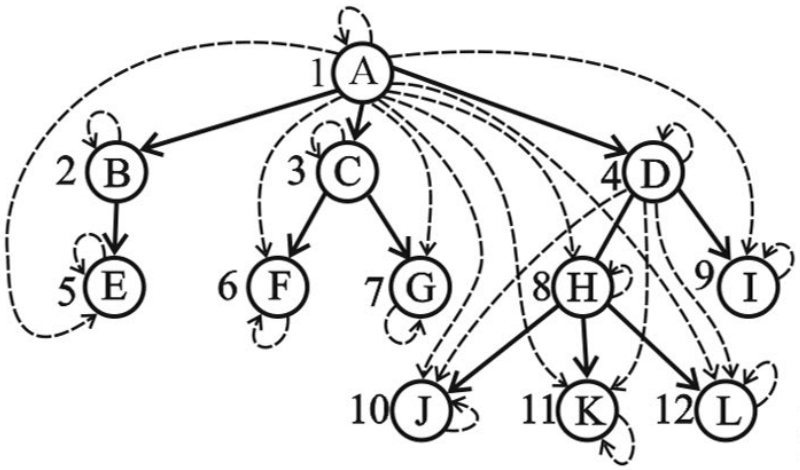


Рисунок 1.8. – Подання дерева у вигляді таблиці зв'язків

Якщо за зразок прийняти список суміжності, який 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

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

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, переміщення вузла з усіма його нащадками. 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.

Матеріалізований шлях (Materialized Path). На рис.1.9 показано приклад моделі «Матеріалізований шлях». 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 «Матеріалізований шлях» 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

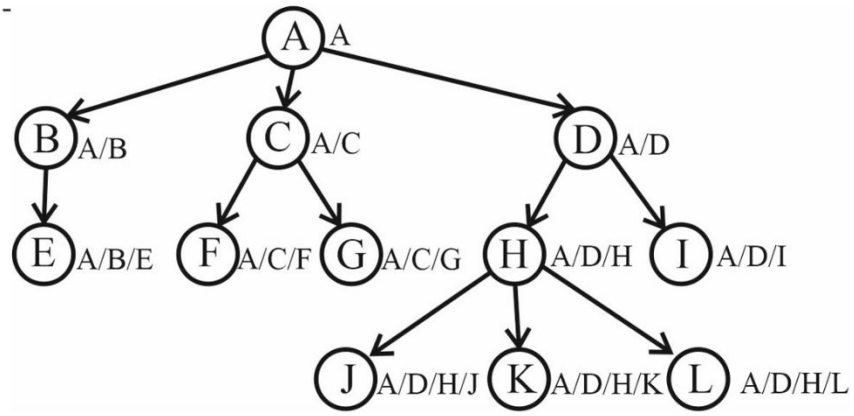


Рисунок 1.9 – Подання дерева у вигляді матеріалізованих шляхів

Серед переваг даної моделі можна виділити простоту написання запиту на отримання нащадків та батьків. 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.

Поєднання моделей матеріалізованих шляхів і списків суміжності може забезпечити гнучкість і ефективність роботи з ієрархічними даними в реляційних базах даних. Матеріалізовані шляхи дозволяють легко отримувати всіх нащадків або всіх батьків вузла, тоді як списки суміжності допомагають зберігати посилкову цілісність і забезпечують швидкий доступ до сусідніх вузлів. Наведемо приклад запиту на створення таблиці, яка поєднує ці дві моделі (рис.1.10).

```
CREATE TABLE CATEGORY (
id INTEGER PRIMARY KEY,
title VARCHAR2(5) NOT NULL,
path VARCHAR(255),
parent_id INTEGER REFERENCES CATEGORY ON DELETE CASCADE
);
```

Рисунок 1.10 – Подання дерева у вигляді поєднання моделей матеріалізованих шляхів і списків суміжності

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.

1.3. Рекурсивні запити в реляційних базах даних

При роботі з базами даних виникли проблеми, такі як специфікації матеріалів чи корпоративна ієрархія, які не можна ефективно вирішити за допомогою ранніх стандартів SQL. Через потребу в роботі з ієрархічними даними кілька постачальників СУБД ввели власні способи надання такої функціональності для своїх користувачів. Наприкла в Oracle було надано ієрархічні запити за допомогою конструкції Start With ... Connect By у своїй СУБД версії 5.0. Першою СУБД, що забезпечила рекурсивні загальні табличні вирази (Common Table Expressions), була IBM's DB2 v.7 у 1997 році. Пізніше рекурсивні запити додано у стандарт SQL:1999 у вигляді рекурсивних загальних

табличних виразів (Common Table Expressions, CTE). З кожним роком проблема рекурсії ставала все більш популярною серед постачальників баз даних. Сьогодні більшість основних СУБД підтримують рекурсивні CTE.

Common Table Expressions (CTE), загальний табличний вираз, також відомий як оператор WITH, є функцією SQL, яка повертає тимчасовий набір даних, який можна використовувати в іншому запиті. Цей тимчасовий результат не зберігається, але його можна використовувати, як будь-яку іншу таблицю. CTE часто використовується для спрощення складних запитів або розбиття логіки на менші, більш керовані частини.

Існують два типи CTE: нерекурсивні та рекурсивні. Загальний синтаксис нерекурсивного CTE виглядає так, як показано на рис. 1.11, 1.12.

```
WITH cte_name (column1, column2, ...) AS (
--Запит
)
SELECT * FROM cte_name;
```

Рисунок 1.11 – Синтаксис нерекурсивного CTE, де cte_name – це ім’я CTE, а column1, column2, ... - назви стовпців, які повертаються запитом у CTE.

CTE дозволяють спрощувати складні запити та покращують їх читабельність [26-28].

Рекурсивний запит у SQL – це запит, який викликає сам себе неодноразово, доки не досягне умови завершення. Він посилається на себе під час свого виконання, дозволяючи виконувати ієрархічні операції або операції з самопосиланням на ієрархічні дані або дані з рекурсивним зв’язком. Рекурсивний CTE посилається на самого себе. Він повертає підмножину результатів, а потім повторно (рекурсивно) посилається на себе, і зупиняється, коли повертає всі результати (рис .1.13).

```
WITH RECURSIVE cte_name (column1, column2, ...) AS (
-- Початковий підзапит
UNION ALL
-- Рекурсивний підзапит
)
SELECT * FROM cte_name;
```

Рисунок 1.12 – Синтаксис нерекурсивного CTE

Синтаксис рекурсивного CTE не надто відрізняється від синтаксису нерекурсивного CTE.

Альтернативним синтаксисом є нестандартна конструкція CONNECT BY, яку було впроваджено Oracle.

```
SELECT список_вибірки
FROM табличний_вираз
[ WHERE … ]
[ START WITH початковий_вираз ]
CONNECT BY [NOCYCLE] { PRIOR дочірній_вираз =
                батьківський_вираз | батьківський_вираз = PRIOR дочірній_вираз }
[ ORDER SIBLINGS BY колонка1 [ ASC | DESC ] [, колонка2 [ ASC | DESC ] ] …
[ GROUP BY … ]
[ HAVING … ]
…
```

Рисунок 1.13 – Синтаксис нерекурсивної конструкції CONNECT BY

Для деревовидної структури Adjacency List для виведення нащадків нащадка батьківського вузла, або щоб знайти шлях від певного місця дерева до кореня чи відображення всього дерева потрібно з’єднати таблицю саму з собою стільки разів, скільки рівнів має наше дерево (рис. 1.14).

Якщо ж глибина дерева не відома, або не відомо на якому рівні знаходиться заданий елемент, то запит не може бути побудований стандартними засобами оператора SELECT, потрібно створювати рекурсивну процедуру, або писати рекурсивний запит (рис. 1.15). Для Oracle рекурсивні запити можна писати використовуючи конструкція CONNECT BY [13, 14] (рис. 1.16), так, як CTE підтримується тільки починаючи з версії Oracle12c і вище.

```
SQL> SELECT t1.title AS level_1, t2.title AS level_2, t3.title AS level_3, t4.title AS level_4
  2  FROM CATEGORY_Adjacency_List t1
  3  LEFT JOIN CATEGORY t2 ON (t2.parent_id=t1.id)
  4  LEFT JOIN CATEGORY t3 ON (t3.parent_id=t2.id)
  5  LEFT JOIN CATEGORY t4 ON (t4.parent_id=t3.id)
  6  WHERE t1.title='A';
```

LEVEL_1	LEVEL_2	LEVEL_3	LEVEL_4
A	D	H	J
A	D	H	K
A	D	H	L
A	C	F	
A	C	G	
A	B	E	
A	D	I	

7 rows selected.

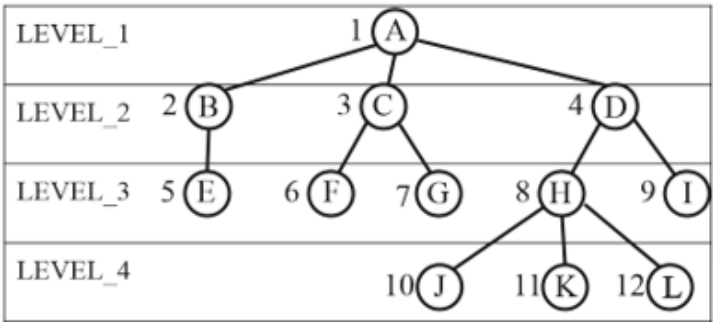


Рисунок 1.14 – Відображення всього дерева для моделі списки суміжності, якщо відома його глибина.

```
WITH RECURSIVE CategoryCTE AS (
-- Anchor member
SELECT id, title, parent_id, 1 AS level
FROM CATEGORY_Adjacency_List
WHERE parent_id IS NULL
 UNION ALL
-- Recursive member
SELECT c.id, c.title, c.parent_id, p.level + 1 AS level
FROM CATEGORY_Adjacency_List c JOIN CategoryCTE p ON c.parent_id = p.id )
SELECT id, title, parent_id, level FROM CategoryCTE ORDER BY level, id;
```

Рисунок 1.15 – Рекурсивний запит на відображення всього дерева для моделі списків суміжності, якщо не відома його глибина.

1.1. Аналіз останніх досліджень організації ієрархічних структур в реляційних базах даних

Сучасні дослідження рекурсивних запитів зосереджені на їх оптимізації. У роботі [23] розглянуто та проведено порівняльний аналіз реалізації рекурсивних запитів, для роботи з ієрархічними даними, визначених стандартами SQL:1999 до SQL:2008, які пропонуються провідними постачальниками СУБД, так як існують певні відмінності в реалізації рекурсивних запитів різними постачальниками. Це порівняння охоплює функції, синтаксис та продуктивність.

У цій статті автори дослідили реалізацію рекурсивних CTE у більшості популярних СУБД (MS SQL Server 2008, SQL Anywhere, PostgreSQL, DB2, RDBMS X, Firebird). Використання CTE у різних SQL-запитах дещо відрізняється для кожного SQL-діалекту. Крім того, автори роботи [23] провели тести продуктивності на системах MS Windows і Linux, а також дослідили вплив індексів на продуктивність виконання запитів. Найкращу продуктивність показав PostgreSQL та SQL Anywhere.

а)

```
SQL> SELECT level, id, parent_id, title
  2  FROM CATEGORY_Adjacency_List
  3  START WITH parent_id is NULL
  4  CONNECT BY PRIOR id=parent_id;

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

12 rows selected.
```

б)

```
SQL> SELECT lpad(' ',3*level)||title as TREE
  2  FROM CATEGORY_Adjacency_List
  3  START WITH parent_id IS NULL
  4  CONNECT BY PRIOR id = parent_id
  5  ORDER SIBLINGS BY title;

TREE
------------------------------------
   A
      B
         E
      C
         F
         G
      D
         H
            J
            K
            L
         I

12 rows selected.
```

Рисунок 1.16 – Рекурсивні запити на відображення всього дерева (*а*, *б*) для моделі списки суміжності.

У роботах [15, 16] розглянуто метод моделювання ієрархічних структур даних у вигляді списків суміжності та розглянуто можливості мови SQL по формулюванню рекурсивних запитів при зверненні до ієрархічних структур даних, задано формальну семантику рекурсивних загальних табличних виразів. Рекурсивні запити при роботі з ієрархічними структурами, дозволяють створювати складні запити, зберігаючи набагато простіший синтаксис. CTE-оператор працює на рівні мультимножинних аналогів теоретико-множинних

операцій, що дозволяє писати запити в компактній та наочній формі. Водночас CTE-оператор непридатний, якщо результат запиту залежить від порядку обходу дерева або іншої ієрархічної структури. Зрозуміло, що в цьому випадку треба використовувати процедурно-орієнтовані методи написання запитів.

Приклади написання рекурсивних запитів та їх різновиди, розглядаються в авторами робіт [24, 25]. В цих роботах показано можливості рекурсивного SQL на прикладах. У роботі [29] авторами проведено порівняння різних моделей подання ієрархічних структур у реляційних базах даних, що дозволило синтезувати найбільш цікаві з цих.

В роботі [32] авторами розглянуто представлення структур у вигляді списків суміжності (Adjacency List) та вкладених множин (Nested Sets) в реляційній базі даних. Отримані кількісні показники часу для вибірки даних, які представлені розглянутими методами. Згідно з результатами аналізу авторами було зроблено висновок, що якщо таких даних досить багато, то доцільно використовувати метод Nested Sets, так як вибірка таких даних займає менше часу, що знижує навантаження на сервер. Разом з тим операції додавання, переміщення та видалення вузлів вимагають додаткові затрати часу на оновлення ключів вузлів, які зачіпає ця операція. Якщо не передбачено, що деревовидна структура у реляційній моделі даних буде мати в перспективі великі об’єми даних і рівень вкладеності не буде суттєвим, то доцільно представляти такі дані у вигляді Adjacency List. Також якщо кожен елемент пов’язаний тільки з безпосереднім батьківським вузлом, то операції додавання, переміщення, та видалення вузлів відбувається без додаткових дій для оновлення ключів, як у випадку з Nested Sets.

У роботі [33] розглядаються проблеми, які виникають в моделі списку суміжності і розглядаються способи вирішення даних проблем для моделі вкладених множин. Також в цій роботі подано алгоритм для перетворення моделі списку суміжності в модель вкладених множин. В роботі [34] розроблено додаток, для тестування методів роботи з різними деревоподібними структурами в SQL. Для визначення оптимального методу проводився аналіз швидкості

роботи, складності вставки нового вузла, переміщення вузла в структурі, підтримка цілісності даних, видалення вузла. В роботі [35] Розглянуто задачу збереження, модифікації та видобування ієрархічних даних у реляційних базах даних MS SQL Server. Проведено аналіз ефективності ієрархічних структур на підставі списку суміжних вершин і модифікованої моделі вкладених множин. У роботі [36] детально описано створення та використання ключів упорядкування вузлів дерева в одній таблиці реляційної бази даних. Ключі для кожного вузла обчислюються з ключів його батьківського вузла таким чином, що порядок сортування розміщує кожен вузол у дереві перед усіма його нащадками і після всіх братів і сестер, які мають нижчий індекс.

У роботі [30] розглянуто модель вкладених інтервалів, яка є узагальненою версією моделі вкладених множин. Ця модель є стійкішою до проблем організації ієрархії. Оскільки кодування вкладених множин за допомогою цілих чисел допускає лише обмежені розриви для вставки нових вузлів, автори роботи [30] пропонують використовувати неперервну область, таку як раціональні числа, зокрема, кодування за допомогою двійкових та Фарейових дробів. У цій роботі показано, чому обидва методи є доречним вибором, і описано взаємозв’язок між ними для кодування дерев. Також досліджено різні способи створення інтервальної структури. Основний результат полягає у введенні матриць шляхів і дослідженні їхніх властивостей.

У роботі [37] автор обґрунтовує переваги використання рекурсивних CTE для створення ієрархії фінансових рахунків, на противагу вирішенню цієї проблеми за допомогою Entity Framework.

У роботі [38] показано, як загальні табличні вирази (CTE) у SQL Server підходять для навігації по деревах, наприклад для пошуку найдовшого шляху або діаметра дерева. Подано простий скрипт для пошуку найдовшого шляху в будь-якому дереві.

У роботі [39] подано можливості SQL-запитів, які дозволяють розрізняти та класифікувати різні типи вузлів, присутні в ієрархічному дереві.

РОЗДІЛ 2
АНАЛІЗ ОПЕРАЦІЙ УПРАВЛІННЯ ДАНИМИ ДЛЯ ІЄРАРХІЧНИХ СТРУКТУР У РЕЛЯЦІЙНИХ БАЗАХ ДАНИХ

2.1. Постановка задачі

Дослідження та використання деревовидних структур у реляційних базах даних є необхідним для ефективного зберігання та маніпулювання ієрархічними даними. Це дозволяє забезпечити швидкий доступ до даних, полегшити підтримку та масштабування систем, а також оптимізувати складні запити до ієрархічних даних. Розуміння різних моделей представлення деревовидних структур допомагає вибрати найкраще рішення для конкретних завдань і вимог.

З метою забезпечення ефективної роботи з деревовидними даними, важливо зберігати їх у форматі, який мінімізує час вибірки та ресурси, необхідні для цього процесу. У цій роботі розглянемо та проаналізуємо найбільш поширені підходи до зберігання деревовидних структур даних. Вивчимо переваги і недоліки кожного з цих методів та проведемо порівняльний аналіз їхньої продуктивності, враховуючи кількісні показники часу вибірки даних, що використовують ці методи. Для цього створимо таблиці у вигляді списків суміжних вершин (Adjacency List), ієрархічних вкладених множин (Nested Sets Model of Hierarchies), таблиці зв’язків (Closure Table), моделі матеріалізованих шляхів (Materialized Path) та заповнимо ці таблиці однаковими даними, щоб отримати однакові структури дерев. Проведемо аналіз часу виконання запитів та показників, які надають інформацію про продуктивність бази даних та потенційні області для оптимізації. На основі результатів аналізу визначимо оптимальні моделі для конкретних сценаріїв їх використання.

2.2. Методологія дослідження

Для проведення аналізу оцінки продуктивності кожного методу потрібно створити тестові сценарії генерації даних для створення деревовидних структур з різною глибиною та кількістю вузлів. Для цього потрібно написати процедуру, яка заповнить таблицю побудовану у вигляді списків суміжних вершин (Adjacency List) випадковими даними. Для того, щоб у всіх чотирьох таблицях були однакові дані потрібно написати процедури, які заповнюють таблиці інших ієрархічних моделей (ієрархічних вкладених множин, таблиці зв'язків, моделі матеріалізованих шляхів) даними з таблиці моделі списків суміжних вершин.

Наступним етапом дослідження є написання скриптів типових запитів для кожного методу:

- знайти всі дочірні елементи вузла;
- знайти батьківський елемент вузла;
- знайти всіх нащадків (дітей та їхніх дітей) вузла;
- додати новий елемент в таблицю (при вставці нового запису потрібно вказати не лише його власні дані, але й дані про батьківський елемент, до якого він буде належати);
- видалити елемент із таблиці (при видаленні може виникнути потреба перенести або видалити його дочірні елементи).
- додати новий рівень в ієрархію шляхом додавання нових дочірніх елементів до існуючих елементів;
- скоротити дерево шляхом видалення піддерев або підкатегорій (це може стати необхідним, якщо певна гілка ієрархії втратила актуальність чи більше не потрібна).

Після створення таблиць відповідно до обраних методів зберігання відбувається заповнення таблиць тестовими даними. Для повноти експерименту дослідження проведемо з деревовидними структурами, які мають 100, 1000 і 10000 вузлів. Враховуючи те, що дані генеруються випадковим чином кількість рівнів дерева не контролюється.

Наступним етапом буде виконання запитів для вимірювання часу відповіді та отримання статистики завантаженості системи для кожного методу.

Далі відбувається проведення аналізу отриманих результатів та вивчення складності реалізації і підтримки кожного методу.

2.3 Обґрунтування вибору інструментальних засобів

Найпопулярнішими на сьогодні серед серверних систем управління базами даних є DB2, Oracle, Microsoft SQL Server, Informix. Такі СУБД мають свої переваги та недоліки, які пов’язані з продуктивністю, масштабованістю, доступністю даних, функціональними можливостями сервера, відкритістю СУБД, наявністю засобів розробки й адміністрування [40]. Наприклад вважають, що перевагами використання СУБД Oracle є її орієнтація на Internet, підтримка великої кількості апаратних і програмних платформ, багаті можливості для розробників (об’єктно-реляційна БД, мова PL/SQL тощо) [40].

Обладнання Oracle – це повний набір масштабованих систем, серверів і систем зберігання даних, які забезпечують високу продуктивність, надійність та безпеку для підприємств будь-якого розміру. Ці рішення дозволяють ефективно управляти та обробляти великі обсяги даних, забезпечуючи безперебійну роботу критичних додатків та сервісів. Завдяки інтеграції з програмним забезпеченням Oracle, таке обладнання дозволяє оптимізувати інфраструктуру та спрощує адміністрування баз даних і додатків, що дає змогу оптимізувати продуктивність програм і баз даних, захищати критично важливі дані й знижувати витрати [41].

Oracle Database пропонує широкий набір функцій для роботи з даними, таких як: досить розвинута мова програмування PL/SQL, механізми тригерів, збережених процедур і функцій тощо [42]. Підтримує різноманітні типи даних та формати, включаючи JSON, XML, spatial та graph дані.

Одна із загальновизнаних переваг сучасних серверів Oracle – його високий ступінь масштабованості, як «горизонтальної», так й «вертикальної». Сервер Oracle у будь-якій конфігурації підтримує паралелізм під час виконання потоку

операцій (він архітектурно спроектований для цього), в SMP-архітектурі для паралельного виконання окремих запитів потрібна інсталяція Parallel Query Option. Для кластерів й MPP-систем Oracle пропонує архітектуру, що дозволяє всім вузлам цих систем паралельно здійснювати доступ до однієї БД: щоб домогтися цього, досить установити Parallel Server Option [40]. Oracle Database має розширені функції безпеки, включаючи шифрування даних, контроль доступу та аудит. Oracle Database легко інтегрується з іншими продуктами Oracle, а також з великою кількістю сторонніх рішень. Підтримка стандартів SQL забезпечує зручність у перенесенні та інтеграції даних з інших СУБД.

Одним із недоліків цієї СУБД є її відносно висока ціна та складне адміністрування, так як широкий функціонал сервера вимагає високу кваліфікацію розробників й адміністраторів.

В цій роботі для моделювання ієрархічних структур в реляційних базах даних було обрано Oracle Database 18c XE (Express Edition). Oracle Database 18c XE є вільно поширюваною версією[43]. Oracle Database 18c XE сумісна з різними платформами, що дозволяє її інтегрувати з різноманітними інструментами та програмами.

Вибір Oracle Database 18c XE може бути вигідним рішенням для розробників, малих підприємств та тих, хто бажає використовувати потужну СУБД без значних витрат, з можливістю масштабування та інтеграції з екосистемою Oracle.

Сервер баз даних був розгорнутий на платформі віртуалізації Oracle VirtualBox 7.0.18 [44-46].

Гостьова система мала наступні характеристики: 2xCPU, 4GB RAM, PIIX3 Chipset, 50GB Storage (vdi), Bridget NET adapter Intel Ethernet (14) I219-V, MS Windows 10 Pro (22H2).

Базова система складалася з процесора Intel Core i3-10100F (3.6GHz), чипсета Intel H510 Comet-Lake-S, 16GB RAM DDR4-2666, SSD Transcend TS512GMTE400S NVMe 512GB, графічного адаптера NVidia GTS450 (1GB),

мережевого адаптера Intel Ethernet (14) I219-V, і операційної системи MS Windows 11 Pro (23H2).

Робоча станція була представлена ноутбуком Lenovo ThinkPad T450 з процесором Intel Core i5-5300U (2.7GHz), чипсетом Intel Point-LP Broadwell, 8GB RAM DDR3-1600, SSD Intel SSD5C2BF 180GB SATA-3, графічним адаптером Intel HD 5500 (1GB), бездротовим модулем Intel Dual band 7265 і операційною системою MS Windows 10 Pro (22H2).

Комунікаційне середовище складалося з Mesh WiFi на базі Linksys MR9000 і Linksys Velop WHW0302 (2 пристрої) до 3Gbps, з робочою швидкістю 866Mbps.

В якості клієнта для сервера баз даних виступав Oracle SQL Developer 22.2.1.234. Oracle SQL Developer – це безкоштовне інтегроване середовище розробки, яке спрощує розробку та управління базами даних Oracle. Воно надає потужні інструменти для створення, редагування та відлагодження SQL-запитів і PL/SQL-коду. За допомогою Oracle SQL Developer можна легко переглядати дані, виконувати скрипти, імпортувати та експортувати дані, а також адмініструвати бази даних, роблячи процес роботи з базами даних більш ефективним і зручним. [47]

2.4. Методи маніпуляції та доступу до ієрархічних даних у реляційних базах

2.4.1 Списки суміжних вершин (Adjacency List)

На рис 2.1 подано запити, які повертають всі вузли дерева, які є листками, тобто не мають нащадків.

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;
```

Рисунок 2.1. – Запити, які повертають всі вузли дерева, які є листками

Підзапит (рис. 2.1, *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. 2.1, *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.

На рис 2.2 подано запит виведення дочірніх елементів та батьківського вузла для вузла title= 'H'.

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';
```

Рисунок 2.2. – Запит на виведення дочірніх (*а*) та батьківського (*б*) елементів вузла з вершиною title= 'H'

Виведення нащадків нащадка батьківського вузла здійснюється шляхом з'єднання таблиці самої на себе. Відображення всього дерева є досить складним, оскільки ми повинні об'єднати таблицю саму з собою, стільки разів, скільки рівнів має наше дерево.

Щоб знайти кількість рівнів ієрархії потрібно виконати запит поданий на рис 2.3.

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

Рисунок 2.3. – Запит для визначення кількості рівнів в ієрахії

Для згенерованого дерева кількість рівнів виявилось рівним 12, тому, щоб вивести все дерево, потрібно з’єднати таблицю саму з собою 12 разів (рис. 2.4)

```
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;
```

Рисунок 2.4 – Виведення всього дерева способом з’єднання таблицю саму з собою стільки разів скільки рівнів має дерево.

Щоб знайти шлях від певного місця дерева до іншого (рис. 2.5), необхідно запустити запит, побудований так само, як і запит, який відображає все дерево.

```
SELECT  t1.title AS level_1, t2.title AS level_2,
        t3.title AS level_3, t4.title AS level_4,
        t5.title AS level_5, t6.title AS level_6,
        t7.title AS level_7, t8.title AS level_8,
        t9.title AS level_9, t10.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  t5 ON (t5.parent_id=t4.id)
LEFT JOIN CATEGORY_Adjacency_List  t6 ON (t6.parent_id=t5.id)
LEFT JOIN CATEGORY_Adjacency_List  t7 ON (t7.parent_id=t6.id)
LEFT JOIN CATEGORY_Adjacency_List  t8 ON (t8.parent_id=t7.id)
LEFT JOIN CATEGORY_Adjacency_List  t9 ON (t9.parent_id=t8.id)
LEFT JOIN CATEGORY_Adjacency_List  t10 ON (t10.parent_id=t9.id)
WHERE t1.title='D' AND t10.title='K';
```

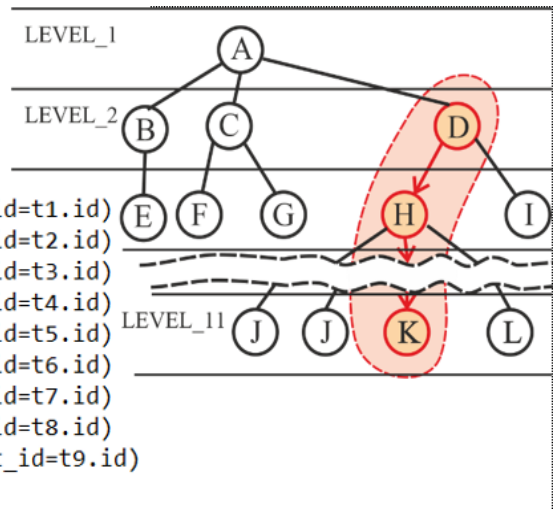


Рисунок 2.5 – Виведення шляху від одного вузла дерева до іншого.

В якості умови ми вказуємо елемент від якого і до якого ми шукаємо, тобто діапазон пошуку і з’єднуємо таблицю саму з собою стільки разів, скільки рівнів знаходиться між вузлами.

Запити на відображення всього дерева і відображення шляху від одного вузла до іншого не є універсальними, так, як кожен раз, коли змінюється кількість рівнів дерева, цей запит потрібно буде налаштовувати. Якщо ж глибина дерева не відома, або не відомо на якому рівні знаходиться шуканий елемент, то запит не може бути побудований стандартними засобами оператора SELECT, потрібно створювати рекурсивну процедуру, або писати рекурсивний запит [1] (рис. 2.6). Аналогічним способом можна вивести все піддерево (рис. 2.7).

```
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;
```

Рисунок 2.6 – рекурсивний запит на виведення шляху від вузла title='D' на 2 рівні до вузла title='K', який знаходиться на 11 рівні.

а)

```
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;
```

б)

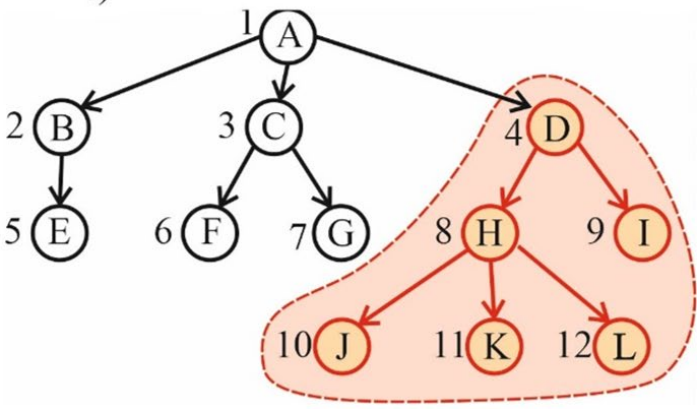


Рисунок 2.7. – Виведення всього піддерева з заданою вершиною

Додавання нового вузла вкінець дерева, відбувається з використанням простої команди SQL на додавання (INSERT). Для цього потрібно додати новий запис до таблиці CATEGORY, де як parent_id ми передаємо ідентифікатор батьківського елемента.

Наприклад, щоб додати елемент 'M' до вузла з id=6 ми виконуємо відповідні запити INSERT (рис. 2.8 (*а*)).

При видаленні будь-якого вузла, враховуючи правило ON DELETE CASCADE буде видалено все піддерево (рис. 2.8 (б)).

Додатковою перевагою є можливість переміщення всієї гілки, переміщаючи тільки один елемент. Це пов’язано з тим, що кожен елемент знає тільки свого батька, і тому при переміщенні будь-якого елемента дерево не втрачає цілісності.

а)

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

б)

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

с)

```
UPDATE CATEGORY_Adjacency_List SET parent_id=5 WHERE id=8;
```

Рисунок 2.8 – Додавання (*а*) та видалення (*б*) вузла дерева з усіма дочірними елементами та переміщення всієї гілки до іншого батьківського вузла(*с*).

2.4.2. Модель вкладених множин (Nested Sets Model of Hierarchies)

На рис 2.9 подано запити, які повертають всі вузли дерева, які є листками, тобто не мають нащадків.

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

Рисунок 2.9. – запит який повертає всі вузли дерева моделі вкладених множин, які є листками

Цей запит (рис. 2.9) вибирає 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 «Вкладені множини» потрібно виконати один із запитів поданих на рис 2.10.

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
   );
```

Рисунок 2.10 – запити, знаходять всі прямі дочірні елементи вузла з назвою 'H'

Запити (рис. 2.10) 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. 2.10, *а*) виконує з’єднання таблиці з самою собою (JOIN), щоб знайти вузол 'D' і одразу використовує його межі (lft і rgt) для фільтрації дочірніх елементів. Другий запит (рис. 2.10, *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 для визначення меж одного разу, тоді як перший запит об’єднує всі умови в основному запиті.

На рис. 2.11 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;
```

Рисунок 2.11 – Запит, який знаходить батьківський вузол для вузла 'D'

Перший запит (рис. 2.11, *а*) використовує JOIN для з'єднання таблиці саму з собою, щоб знайти всі батьківські вузли для вузла з title='D', потім фільтрує ці вузли за умовами p.lft < c.lft і p.rgt > c.rgt, та вибирає верхній вузол за допомогою ORDER BY p.lft DESC FETCH FIRST 1 ROWS ONLY.

Другий запит робить 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 саму з собою, тоді як другий використовує вкладені підзапити для знаходження меж значень.

На рис. 2.12 подано запит, який виводить шлях в моделі «Вкладені множини» (Nested Sets) 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.

```
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;
```

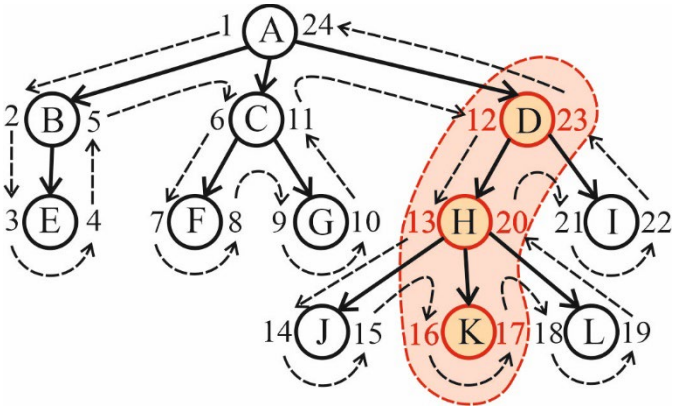


Рисунок 2.12. – Запит, який знаходить шлях від вузла 'D' до вузла 'K'

На рис. 2.13 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.

```
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;
```

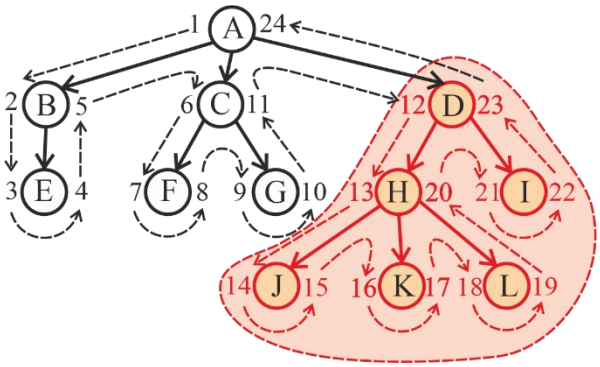


Рисунок 2.13. – Запит, який виводить піддерево вузла 'D'

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. 2.14).

```
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;
/
```

Рисунок 2.14 – Процедура додавання нового вузла в дерево

Для видалення вузла з усіма його нащадками в моделі ієрархічних вкладених множин (Nested Sets) необхідно виконати такі кроки:

- 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. 2.15).

```
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;
/
```

Рисунок 2.15. – Процедура видалення вузла з усіма його нащадками

2.4.3. Таблиці зв’язків (Closure Table)

Запит поданий на рис 2.16 поверне всі вузли дерева, які є листками, тобто не мають нащадків. (підзапит перевіряє, чи існує запис у таблиці CATEGORY_link, де Id_from дорівнює id з таблиці CATEGORY_Closure_Table де немає самопосилань. 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);
```

Рисунок 2.16. – Запит, який повертає всі вузли дерева, які є листками

Запити, що подано на рис 2.17 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;
```

Рисунок 2.17. – 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. 2.18).

```
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'
);
```

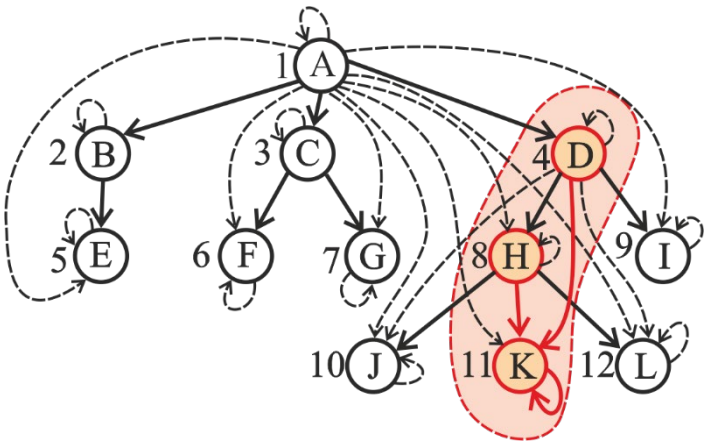


Рисунок 2.18. – Запит, який знаходить шлях від вузла 'D' до вузла 'K'

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 ріне id вершини піддерева (рис. 2.19).

```
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');
```

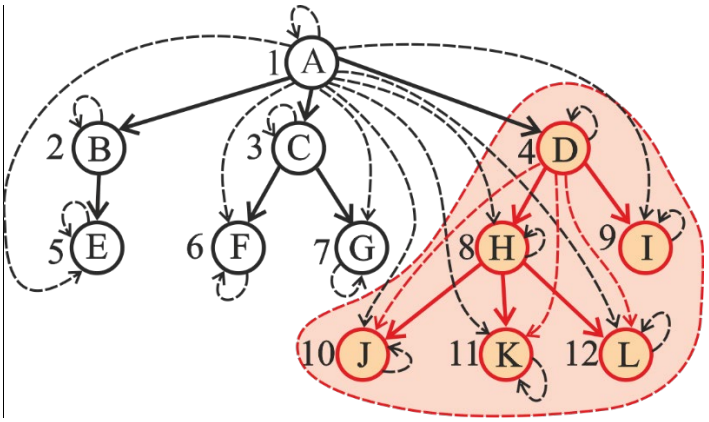


Рисунок 2.19 – Запит, який виведе піддерево вузла 'D'

Для додавання даних потрібно буде добавляти дані в дві таблиці (рис. 2.20). Замість запиту на заповнення другої таблиці (рис. 2.20, *а*), можна створити тригер, який буде створювати зв’язки між доданим вузлом і батьківським вузлом та з усіма вузлами, що знаходяться вище батьківського.

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);
```

Рисунок 2.20 – запити додавання (*а*) та видалення (*б*)

Операція видалення вузла виконується за допомогою операції DELETE, при цьому всі його нащадки стають нащадками видаленого батьківського вузла, але ще потрібно змінити відстань між новим батьківським вузлом і нащадками. Тому пропонується перед тим, як видаляти вузол виконати оновлення відстаней (поле distance) для нащадків. Для цього потрібно створити відповідний тригер, або написати запит на оновлення. Для того, щоб видалити все піддерево вузла з усіма його нащадками потрібно виконати запит поданий на рис. 2.20, *б*. Враховуючи використання правила ON DELETE CASCADE в таблиці CATEGORY_link будуть видалені всі зв'язки вузлів піддерева.

2.4.4. Матеріалізований шлях (Materialized Path)

На рис 2.21 подано запит, який повертає всі вузли дерева, які є листками, тобто ті, 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 || '/%' );
```

Рисунок 2.21 – Запит, який повертає всі вузли дерева, які є листками

На рис 2.22 подано запити на виведення дочірніх (*а,б*) та батьківського (*в, г*) елементів вузла з вершиною title= 'H'. Перший запит (рис. 2.22, *а*) виведення дочірніх елементів знаходить прямі дочірні елементи вузла з title='H', використовуючи JOIN і перевірку шляху з допомогою функції INSTR. Другий запит (рис. 2.22, *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.

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;
```

Рисунок 2.22 – запити на виведення дочірніх (*а*,*б*) та батьківського (*в*, *г*) елементів вузла з вершиною title= 'H'

Перший запит виведення батьківського елементу (рис. 2.22, *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. 2.22, *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. 2.23). 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 з title вузла з початкового вузла.

Для виведення піддерева потрібно перевірити для всіх рядків, де значення path починається з path батьківського вузла піддерева (рис. 2.24). Використання функції INSTR (рис. 2.24) 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. Для додавання нового вузла потрібно об'єднати шлях path з батьківського вузла з і нового title (рис.2.25, *а*). Для видалення вузла дерева з усіма його нащадками потрібно видалити всі рядки з таблиці CATEGORY_Materialized_Path, де шлях містить шлях з рядка з батьківського вузла піддерева (рис.2.25, *б*).

В таблиці 2.1. подано порівняльну характеристику запаитів вибірки оновлення даних для різних методів подання ієрархічних структур в реляційній базі даних.

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

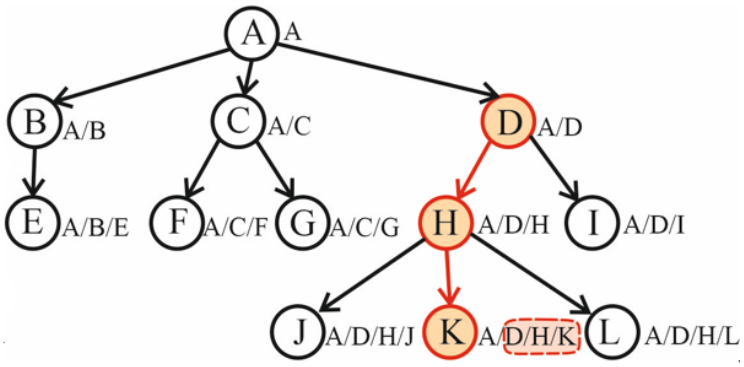
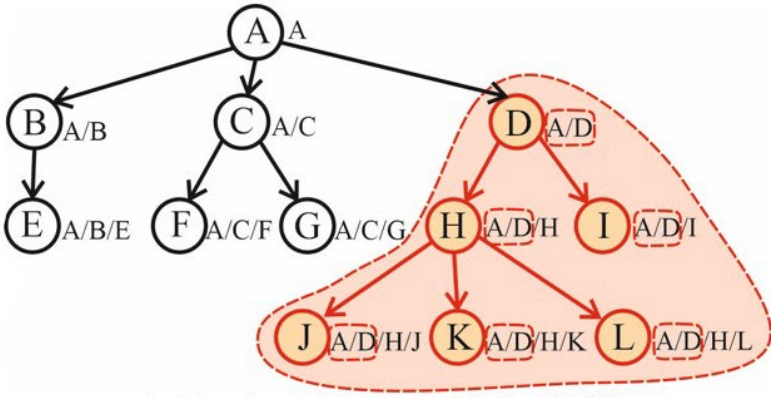


Рисунок 2.23 – Запит, який знаходить шлях від вузла 'D' до вузла 'K'




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

Рисунок 2.24 – Запит, який виведе піддерево вузла 'D'

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;
```

Рисунок 2.25 – Додавання (*а*) та видалення (*б*) вузла дерева з усіма дочірніми елементами.

Таблиця 2.1. Порівняльна характеристика методів подання деревовидних структур

Критерії порівняння	**Adjacency List**	**Nested Sets**	**Closure Table**	**Materialized Path**
Запити на вибірку даних (кількість з'єднань)				
вибірка всіх вузлів дерева, які не мають нащадків	2	2	2	1
виведення прямих дочірніх вузлів	2	2	2	2
виведення батьківського вузла	2	2	2	2
виведення шляху від одного вузла до іншого	N (*)	1	2	1
виведення піддерева	H – N + 1 (*)	1	2	1
Запити на оновлення даних				
додавання нового листка дерева	Один запит додавання	Вимагає оновлення меж для всіх вузлів, що знаходяться праворуч від батьківського вузла	Додатково потрібно додати всі записи про зв'язки до таблиці зв'язків	потрібно об'єднати шлях path з батьківського вузла і значення нового title
видалення піддерева	Один запит видалення	Вимагає оновлення меж для всіх вузлів, що знаходяться праворуч від батьківського вузла	Один запит видалення	потрібно видалити всі рядки, де шлях містить шлях рядка з батьківського вузла піддерева

Примітка: (*) — рекурсивний запит, H — висота дерева, N — поточний рівень

2.5. Сценарії генерації тестових даних для створення ієрархічних структур з різною глибиною та кількістю вузлів

Для аналізу виконання запитів до ієрархічної структури у вигляді списків суміжних вершин (Adjacency List) було побудовано тестові дерева зі стрічкових елементів, які збережені в полі title (рис. 1.4) і складаються з 100, 1000 та 10000 записів згенерованих випадковим чином (табл 2.2). Для цього під користувачем sys as sysdba створено відповідну процедуру Generate_Category_Data (Додаток А, рис. А.1). Після виконання процедури потрібно змінити батьківський елемент вершини на значення Null, або врахувати зміну при побудові самої процедури.

Для моделей ієрархічних вкладених множин (Nested Sets Model of Hierarchies), таблиці зв'язків (Closure Table) та матеріалізованих шляхів (Materialized Path) побудовано тестові дерева зі стрічкових елементів, які зберігаються в полі title і складаються з 100, 1000 та 10000 записів вибраних з дерева згенерованих випадковим чином для моделі списків суміжних вершин. Для цього створено відповідні процедуру Populate_Nested_Sets, Populate_Closure_Table, Populate_Materialized_Path (Додаток А, рис. А.2-А4).

Отримані дерева мали глибину 12, 13 та 18 рівнів для таблиць які складаються з 100, 1000 та 10000 записів відповідно. При моделюванні дерева для моделей списків суміжних вершин (Adjacency List), ієрархічних вкладених множин (Nested Sets Model of Hierarchies) та матеріалізованих шляхів (Materialized Path) дерева моделюються за допомогою однієї таблиці які складаються з 100, 1000 та 10000 записів. Для моделювання дерева моделі таблиці зв'язків (Closure Table) використано дві таблиці. В таблиці з даними знаходяться дані які складаються з 100, 1000 та 10000 записів, а в таблиці зв'язків кількість записів залежить від згенерованої структури дерева. В даному випадку при випадковим чином згенерованих деревовидних структурах які складаються з 100, 1000 та 10000 записів в таблиці з даними ми отримали в таблиці зв'язків 546, 7129 та 93110 записів відповідно (табл 2.2.).

Таблиця 2.2. 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

2.6 Порівняльний аналіз кількісних показників часу виконання вибірки.

Проведемо оцінювання затрат часу на вибірку всіх листків дерева, які зберігаються в базі даних всіма розглянутими методами. Було проведено 10 вибірок для кожного методу та знайдено їх середнє значення.

На рисунку 2.26 наведені результати експерименту. 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. 2.26 вибірка всіх вузлів дерева, які не мають нащадків деревовидної структури даних, які зберігаються в базі даних методом 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. 2.27 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.2.27 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.

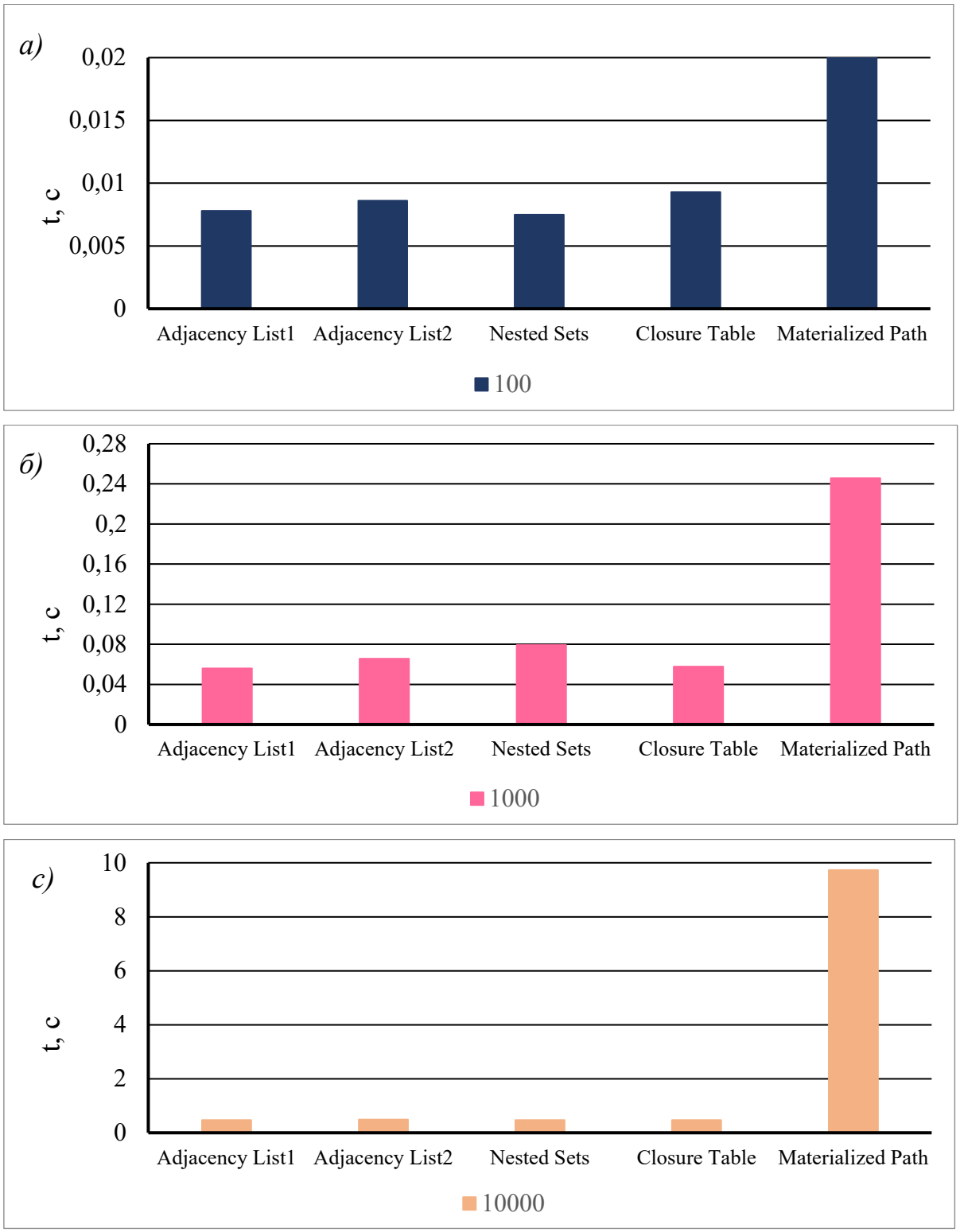


Рисунок 2.26 – Час виконання запиту виведення листків дерева (Adjacency List1: запит на рис. 2.1, *а*; Adjacency List2: запит на рис.2.1, *б*; Nested Sets: запит на рис. 2.9; Closure Table: запит на рис. 2.13; Materialized Path: запит на рис. 2.18).

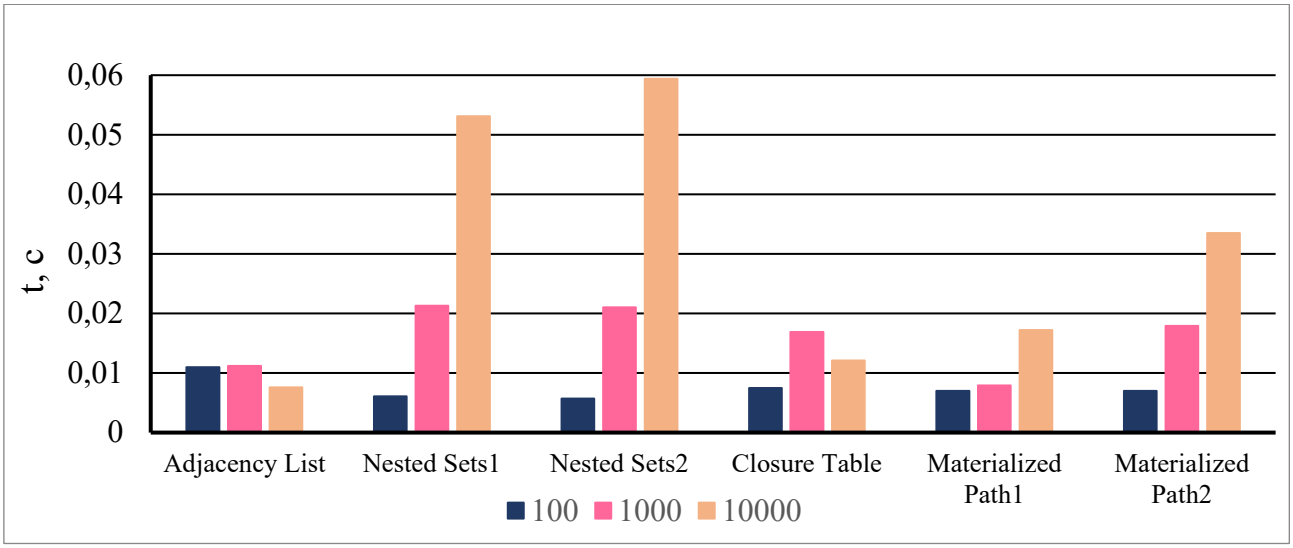


Рисунок 2.27 – Час виконання запиту виведення прямих дочірніх вузлів, батьківського вузла, який знаходиться на 5 рівні (Adjacency List1: запит на рис. 2.2, *а*; Nested Sets1: запит на рис. 2.10,*а*; Nested Sets1: запит на рис. 2.10, *б*; Closure Table: запит на рис. 2.17, *а*; Materialized Path1: запит на рис. 2.22, *а*; Materialized Path2: запит на рис. 2.22, *б*)

Для малої кількості даних (100 записів) час виконання запиту на виведення прямих дочірніх елементів 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 Nested Sets для визначення дочірніх вузлів необхідно виконувати діапазонні операції на значеннях "ліва" і "права" межа. Це призводить до більших витрат на обчислення, особливо коли потрібно відфільтрувати лише прямі дочірні елементи серед усіх нащадків. У Adjacency List кожен вузол просто містить посилання на свого батька, 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.2.27) обмежують діапазон пошуку за допомогою перевірки на префікс у шляху (LIKE p.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. 2.28) для Adjacency List мало залежить від кількості записів в таблиці, для інших моделей час дещо зростає.

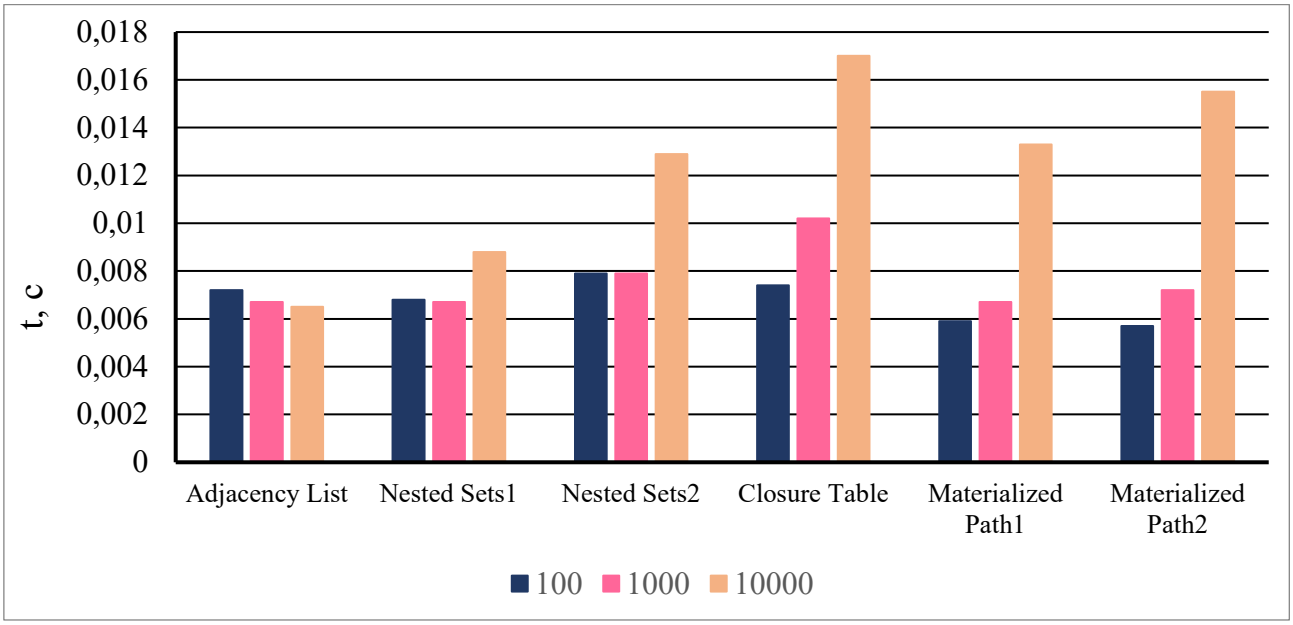


Рисунок 2.28 – Час виконання запиту виведення батьківського вузла, для вузла який знаходиться на 5 рівні (Adjacency List: запит на рис. 2.2,*б*; Nested Sets1: запит на рис. 2.11, *а*; Nested Sets1: запит на рис. 2.11, *б*; Closure Table: запит на рис. 2.17,*б*; Materialized Path1: запит на рис. 2.22, *в*; Materialized Path2: запит на рис. 2.22, *г*)

На рис. 2.29 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. 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.

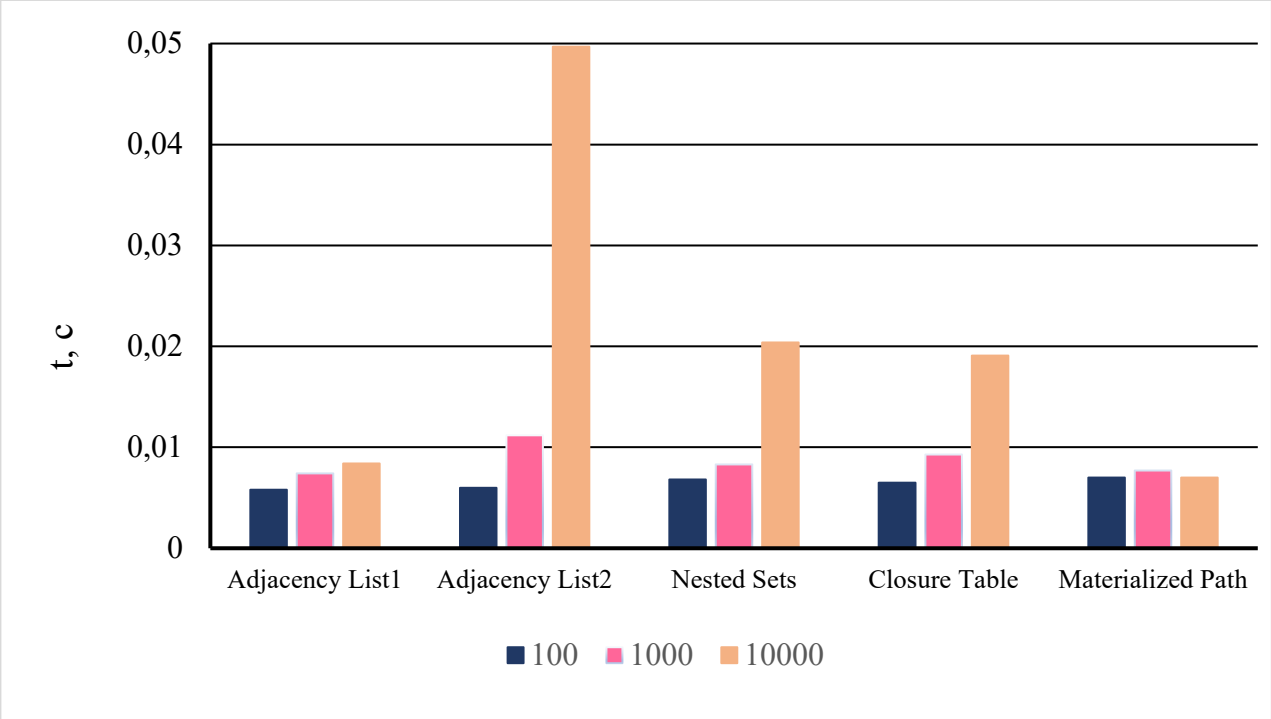


Рисунок 2.29 – Час виконання запиту виведення шляху від вузла який знаходиться на 2 рівні до вузла, який знаходиться на 11 рівні (Adjacency List1: запит на рис. 2.5; Adjacency List2: запит на рис. 2.6; Nested Sets: запит на рис. 2.12; Closure Table: запит на рис. 2.18; Materialized Path: запит на рис. 2.23)

Для виведення всього піддерева (рис. 2.30) 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.

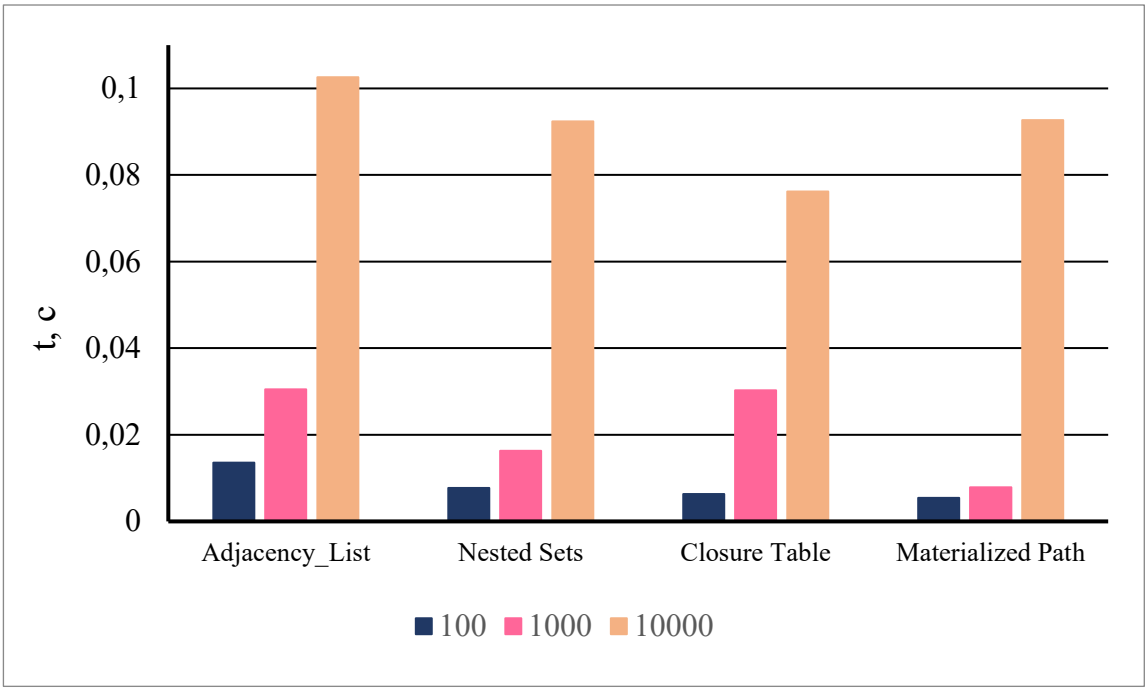


Рисунок 2.30 – Час виконання запиту виведення піддерева з вершиною, яка знаходиться на 5 рівні (Adjacency List: запит на рис. 2.7; Nested Sets: запит на рис. 2.13; Closure Table: запит на рис. 2.19; Materialized Path: запит на рис. 2.24)

Час виконання запитів на додавання або видалення листка в деревоподібній структурі даних майже не залежить від кількості рядків у таблиці (рис.2.31; рис. 2.32), 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). Операції на додавання або видалення листка мають низьку алгоритмічну складність (O(1) або O(log N)), оскільки вони не вимагають сканування всієї таблиці або перегляду всіх рядків.

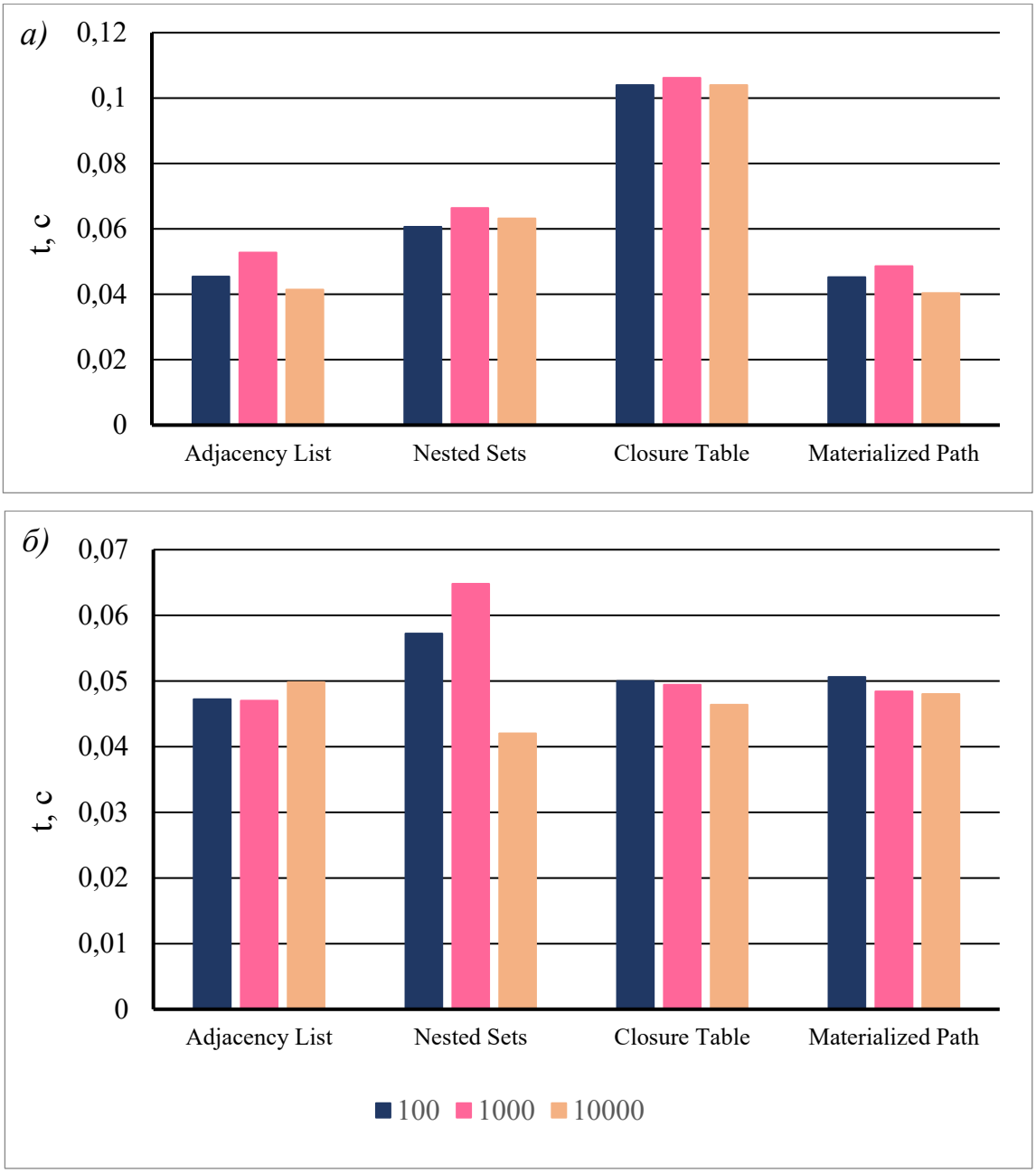


Рисунок 2.31 – Час виконання запиту на додавання нового листка дерева (*а*) та видалення вузла дерева, яке є листком (*б*)

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. 2.32 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.

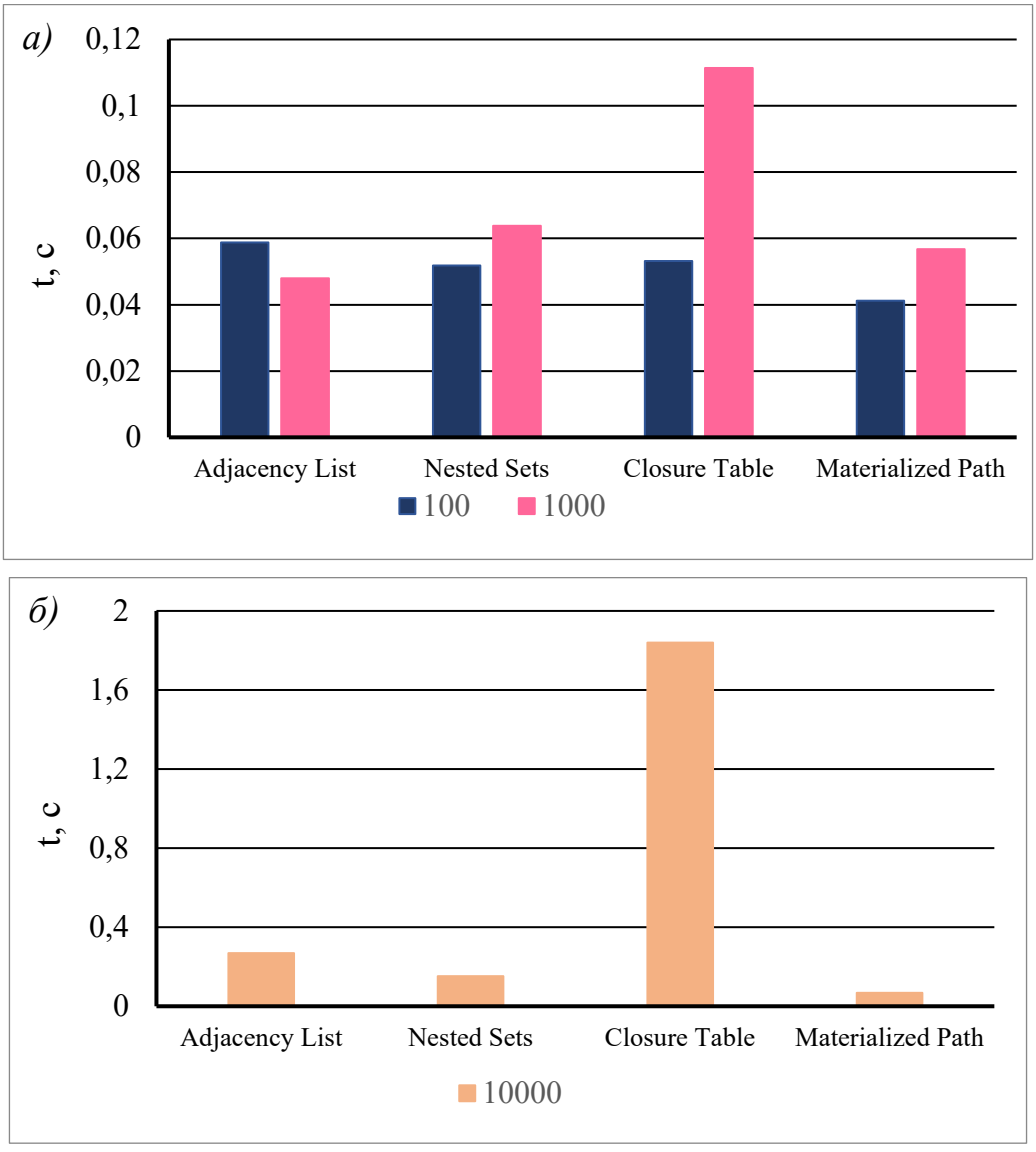


Рисунок 2.32 – Час виконання запиту на видалення піддерева з вершиною, яка знаходиться на 5 рівні.

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 за префіксом шляху (наприклад, LIKE p.path || '/%'), тому видалення піддерева відбувається найшвидше, порівняно з іншими моделями.

ВИСНОВКИ

У ході дослідження, присвяченого аналізу операцій роботи з даними для деревоподібних структур у реляційних базах даних, було зроблено кілька важливих висновків, що стосуються ефективності різних моделей подання ієрархічних даних, а також їхнього застосування в різних сценаріях.

У реляційних базах даних використовуються кілька основних моделей для представлення деревоподібних структур: Adjacency List, Nested Sets, Closure Table та Materialized Path. Кожна з цих моделей має свої переваги і недоліки залежно від типу операцій, що виконуються. Adjacency List є простим у реалізації і зручним для операцій на рівні окремих вузлів, таких як додавання або видалення листка, але не оптимальним для складних запитів, пов’язаних із отриманням піддерева або шляхів між вузлами. Nested Sets показує високу ефективність при виконанні запитів на отримання піддерева, проте операції додавання та видалення є складними та ресурсомісткими. Closure Table забезпечує високу гнучкість для запитів на визначення предків і нащадків, але потребує значних ресурсів для підтримки актуальності даних при частих оновленнях. Materialized Path відрізняється ефективністю у запитах на отримання шляху між вузлами та піддерев, але також вимагає уваги до оптимізації індексів для забезпечення високої продуктивності.

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 можуть потребувати 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.

Вибір моделі для зберігання деревоподібних структур у реляційних базах даних залежить від специфіки запитів та частоти операцій. Загалом, дослідження підкреслює важливість ретельного аналізу вимог до системи перед вибором моделі для роботи з деревоподібними структурами в реляційних базах даних. Ефективність операцій суттєво залежить від правильного підбору моделі відповідно до специфіки завдань і типів запитів, що виконуються в базі даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Комп'ютерна дискретна математика : Підручник / М.Ф. Бондаренко, Н.В. Білоус, А.Г. Руткас. – Харків : «Компанія СМІТ», 2004. – 480 с.
2. Бардачов Ю.М., Соколова Н.А., Ходаков В.Є. Дискретна математика : Підручник – За ред. В.Є. Ходакова. – К.: Вища шк., 2002. – 287 с.
3. Боднарчук Ю.В., Олійник Б.В. Основи дискретної математики (для студентів-інформатиків) Київ – Національний університет «Києво-Могилянська академія» – 2007. – 138 с. URL: https://www.ukma.edu.ua/~bogd/Discrete%20Mathematics/PosibnykNew.pdf
4. Борисенко О.А. Дискретна математика: Підручник. - Суми: ВТД «Університетська книга», 2007. – 255 с
5. Wirth, N. (1986). *Algorithms and data structures.* Prentice-Hall.
6. Narasimha Karumanchi. (2017). *Data Structures And Algorithms Made Easy.* CareerMonk.
7. Алгоритми і структура даних: Навчальний посібник / В.М.Ткачук. - Івано-Франківськ : Видавництво Прикарпатського національного університету іменіВасиля Стефаника, 2016.-286 с.
8. Креневич А.П. Алгоритми і структури даних. Підручник. – К.: ВПЦ "Київський Університет", 2021. – 200 с
9. Коротєєва Т.О. Алгоритми та структури даних. - Навчальний посібник. Львів: Видавництво Львівської політехніки, 2014. 280 с.
10. Курс лекцій з дисципліни «Алгоритми та структури даних» для студентів спеціальності 014 Середня освіта. Інформатика [Електронний ресурс] / Т.О. Гришанович; ВНУ імені Лесі Українки. Електронні текстові данні (1 файл: 1,33 МБ). Луцьк : ВНУ імені Лесі Українки, 2021. – 110 с.
11. Булатецька Л. В., Булатецький В. В. Бази даних та розподілені інформаційно-аналітичні системи (Частина 1) : електронний курс навчальної дисципліни, затверджений НМР ВНУ імені Лесі Українки, протокол № 4 від 16.12.2020.

Луцьк: ВНУ ім. Лесі Українки, 2020. URL: https://moodle-cs.vnu.edu.ua/course/view.php?id=35

12. Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Копитчук І. М. Організація баз даних: навч. посібник. 2-ге вид. виправ. і доповн. Одеса: Фенікс, 2019. 246 с
13. Булатецька Л. В.,БулатецькийВ. В. Реляційна алгебра. Реляційне числення : методичні вказівки для підготовки до контрольної роботи з нормативних навчальних дисциплін “Бази даних та розподілені інформаційно-аналітичні системи”, “Організація баз даних та знань”. Луцьк: ВНУ ім. Лесі Українки, 2020. 36 с. URI: https://evnuir.vnu.edu.ua/handle/123456789/18857.
14. Булатецька Л. В., Булатецький В. В. Мова запитів SQL : текст лекцій нормативної навчальної дисципліни “Бази даних та розподілені інформаційно-аналітичні системи”. Луцьк : СНУ ім. Лесі Українки, 2018. 92 с. URI: http://evnuir.vnu.edu.ua/handle/123456789/17722
15. Буй,Д., & Поляков,С. (2010). Рекурсивні запити в SQL-подібних мовах: приклади, змістова і формальна семантика. Проблеми програмування, (2-3).
16. 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.
17. 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.
18. 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
19. Joe Celko's Trees and Hierarchies in SQL for Smarties. (2004). Elsevier. https://doi.org/10.1016/b978-1-55860-920-4.x5000-4
20. Celko J. Trees and Hierarchies in SQL for Smarties. – Morgan-Kaufmann, 2012. – 296 p.

21. https://open2web.com.ua/blog/derevo-katalogov-nested-sets-vlozhennye-mnozhestva-i-upravlenie-im.html
22. Є.В. Павловський, І.В. Ярош, Аналіз методів представлення деревовидних структур у реляційній моделі даних. Наукові праці ДонНТУ №2 (35), 2022-№1(36), 2023 Серія "Інформатика, кібернетика та обчислювальна техніка"
23. Recursive Query Facilities in Relational Databases: A Survey. Authors : Piotr Przymus, Aleksandra Boniewicz, Marta Burzańska, Krzysztof Stencel Published in: Database Theory and Application, Bio-Science and Bio-Technology. Publisher: Springer Berlin Heidelberg. DOI:10.1007/978-3-642-17622-7_10
24. Резниченко, В. (2010). Рекурсивный SQL. *Інженерія програмного забезпечення*, (4), 63–81.
25. Sarmiento, E. (2008, June 16). *Recursive Queries using Common Table Expressions (CTE) in SQL Server*. SQL Server Tips, Techniques and Articles. https://www.mssqltips.com/sqlservertip/1520/recursive-queries-using-common-table-expressions-cte-in-sql-server/
26. Koidan K. SQL CTEs Explained with Examples. *LearnSQL.com*. URL: https://learnsql.com/blog/cte-with-examples/ (date of access: 09.11.2024).
27. Babic T. What Is a Recursive CTE in SQL?. *LearnSQL.com*. URL: https://learnsql.com/blog/sql-recursive-cte/ (date of access: 09.11.2024).
28. Ilic M. What Is a CTE?. *LearnSQL.com*. URL: https://learnsql.com/blog/what-is-cte/ (date of access: 09.11.2024).
29. Abdelali E., Mohammed Cherkaoui Malki O., Mazouz S. Different Approaches to Modeling the Trees Data in Relational Database. *International Journal of Computer Applications*. 2012. Vol. 53, no. 18. P. 38–42. URL: https://doi.org/10.5120/8524-2551 (date of access: 02.07.2024).
30. Tropashko V. Nested intervals tree encoding in SQL. *ACM SIGMOD Record*. 2005. Vol. 34, no. 2. P. 47–52. URL: https://doi.org/10.1145/1083784.1083793 (date of access: 02.07.2024).
31. Pavlovskyi E., Yarosh I. Analysis of methods for representing tree-like structures in a relational data model. *Scientific papers of Donetsk National Technical*

University. Series: Informatics, Cybernetics and Computer Science. 2023. Vol. 2 - №1, no. 35-36. P. 17–21. URL: https://doi.org/10.31474/1996-1588-2023-1-36-17-21 (date of access: 02.07.2024).

32. Павловський Є.В., Ярош І.В. Аналіз методів представлення деревовидних структур у реляційній моделі даних. Наукові праці ДонНТУ, Серія "Інформатика, кібернетика та обчислювальна техніка". №2 (35), 2022-№1(36), 2023. DOI: 10.31474/1996-1588-2023-1-36-17-21
33. *Trees in SQL. Joe Celko*. (n.d.). Firebird, InterBase, HQbird: replication, monitoring, technical support. http://www.ibase.ru/files/articles/programming/dbmstrees/sqltrees.html.
34. Шрамченко, Б. Л., & Ахматов, В. В. (2021). Обробка ієрархічних даних у реляційній базі даних. In *нформаційні технології в науці, виробництві та підприємництві* (pp. 152–155). Освіта України.
35. Голуб, В. (2010). Ієрархічна модель вкладених множин у реляційних базах даних. *ВІСНИК ЛЬВІВ. УН-ТУ. Серія прикл. матем. інформ.*, (16), 106–113.
36. Hazel, D. (2008). Using rational numbers to key nested sets. Article DocSetID-311997. https://doi.org/10.48550/arXiv.0806.3115
37. Adler C. Generating Trees in SQL with Common Table Expressions and Cross Apply. *LinkedIn: Log In or Sign Up*. URL: https://www.linkedin.com/pulse/generating-trees-sql-common-table-expressions-cross-apply-corey-adler (date of access: 03.08.2024).
38. Navigating Trees With CTE – SQLServerCentral. *SQLServerCentral*. URL: https://www.sqlservercentral.com/articles/navigating-trees-with-cte (date of access: 03.08.2024).
39. vishnutr. Tree Node: Identify the Node Type with SQL. *Medium*. URL: https://medium.com/@vishnutrvishnutr/tree-node-identify-the-node-type-with-sql-6e9d1668fda3 (date of access: 03.08.2024).
40. Коваленко, В. В. Аналіз найактуальніших серверних систем управління базами даних / В. В. Коваленко, В. О. Гуменюк // Вісник Національного

університету "Львівська політехніка". – 2007. – № 603 : Комп'ютерні системи та мережі. – С. 65-72.

41. Обладнання | Oracle Україна. *Oracle | Cloud Applications and Cloud Platform*. URL: https://www.oracle.com/ua/it-infrastructure/ (дата звернення: 03.08.2024).
42. Кэри Мілсап Джефф Хольт Oracle. Оптимізація продуктивності // Символ-Плюс. – 2006 – С. 129–143.
43. Oracle Database Express Edition (XE) 18c Downloads. *Oracle | Cloud Applications and Cloud Platform*. URL: https://www.oracle.com/database/technologies/xe18c-downloads.html (date of access: 03.08.2024).
44. Index of http://download.virtualbox.org/virtualbox/7.0.18. *Software Download | Oracle*. URL: https://download.virtualbox.org/virtualbox/7.0.18/ (date of access: 03.08.2024).
45. Oracle VM VirtualBox. *Oracle* 2024. URL: https://download.virtualbox.org/virtualbox/7.0.18/UserManual.pdf
46. Oracle VM VirtualBox User Guide for Release 7.0. *Oracle* 2024. URL: https://docs.oracle.com/en/virtualization/virtualbox/7.0/user/EN-VBOX-7-0-USER.pdf
47. SQL Developer. *Oracle | Cloud Applications and Cloud Platform*. URL: https://www.oracle.com/database/sqldeveloper/ (date of access: 06.08.2024).

ДОДАТКИ

Додаток А

```
CREATE OR REPLACE PROCEDURE Generate_Category_Data (n in INTEGER) AS
BEGIN
    FOR i IN 1..n LOOP
        DECLARE
            title_hash VARCHAR2(32);
            title CATEGORY_Adjacency_List.title%TYPE;
            parent_id INTEGER;
        BEGIN
            title_hash := DBMS_CRYPTO.HASH
                        (UTL_RAW.CAST_TO_RAW('Category' || TO_CHAR(i)),
                                    SYS.DBMS_CRYPTO.HASH_MD5);
            title := SUBSTR(title_hash, 1, 5);
            parent_id := ROUND(DBMS_RANDOM.VALUE(1, i - 1));

            INSERT INTO CATEGORY_Adjacency_List (id, title, parent_id)
                                        VALUES (i, title, parent_id);

            IF MOD(i, 100) = 0 THEN
                COMMIT;
                DBMS_OUTPUT.PUT_LINE('Згенеровано ' || i || ' записів.');
            END IF;
        END;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('Генерація даних завершена.');
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Сталася помилка: ' || SQLERRM);
        ROLLBACK;
END;
/
```

Рисунок А.1 – Процедура генерування випадковим чином дерева побудованого методом списків суміжних вершин (Adjacency List).

```
CREATE OR REPLACE PROCEDURE populate_nested_sets AS
    lft_counter INTEGER := 1;

    PROCEDURE process_node(node_id INTEGER, node_title VARCHAR2) IS
        child_cursor SYS_REFCURSOR;
        child_id INTEGER;
        child_title VARCHAR2(5);
        current_lft INTEGER;
    BEGIN
        -- Встановлення значення lft для поточного вузла
        current_lft := lft_counter;
        lft_counter := lft_counter + 1;

        -- Відкриття курсора для обробки дочірніх вузлів
        OPEN child_cursor FOR
            SELECT id, title
            FROM CATEGORY_Adjacency_List
            WHERE parent_id = node_id;

        LOOP
            FETCH child_cursor INTO child_id, child_title;
            EXIT WHEN child_cursor%NOTFOUND;
            process_node(child_id, child_title);
        END LOOP;
        CLOSE child_cursor;

        -- Встановлення значення rgt для поточного вузла
        INSERT INTO CATEGORY_Nested_Sets (id, title, lft, rgt)
        VALUES (node_id, node_title, current_lft, lft_counter);
        lft_counter := lft_counter + 1;
    END process_node;

BEGIN
    -- Ініціалізація процесу з кореневих вузлів
    FOR root_node IN (SELECT id, title
                      FROM CATEGORY_Adjacency_List
                      WHERE parent_id IS NULL) LOOP
        process_node(root_node.id, root_node.title);
    END LOOP;
END populate_nested_sets;
/
```

Рисунок А.2 – Процедура заповнення даними таблиці CATEGORY_Nested_Sets із заповненого випадковими значеннями дерева побудованого методом списків суміжних вершин (Adjacency List).

```
INSERT  INTO CATEGORY_Closure_Table
        SELECT id, title  FROM CATEGORY_Adjacency_List;

CREATE OR REPLACE PROCEDURE Populate_Closure_Table IS
BEGIN
    -- Видаляємо всі записи з таблиці CATEGORY_Closure перед заповненням
    DELETE FROM CATEGORY_link;
    -- Вставка початкових зв'язків (вузли самі з собою)
    INSERT INTO CATEGORY_link (id_from, id_to, distance)
    SELECT id, id, 0 FROM CATEGORY_Adjacency_List;
    -- Рекурсивне заповнення таблиці зв'язків
    DECLARE
        CURSOR c1 IS SELECT id FROM CATEGORY_Adjacency_List;
        v_id CATEGORY_Adjacency_List.id%TYPE;
    BEGIN
        OPEN c1;
        LOOP
            FETCH c1 INTO v_id;
            EXIT WHEN c1%NOTFOUND;
            -- Рекурсивне оброблення кожного вузла
            INSERT INTO CATEGORY_link (id_from, id_to, distance)
            SELECT id_from, v_id, distance + 1
            FROM CATEGORY_link
            WHERE id_to = (SELECT parent_id
                           FROM CATEGORY_Adjacency_List
                           WHERE id = v_id)
              AND id_to IS NOT NULL;
        END LOOP;
        CLOSE c1;
    END;
END;
/
```

Рисунок А.3 – Процедура заповнення даними таблиці CATEGORY_Closure із заповненого випадковими значеннями дерева побудованого методом списків суміжних вершин (Adjacency List).

```
CREATE OR REPLACE PROCEDURE populate_materialized_path AS
    PROCEDURE process_node(node_id INTEGER, node_title VARCHAR2, parent_path VARCHAR2) IS
        child_cursor SYS_REFCURSOR;
        child_id INTEGER;
        child_title VARCHAR2(50);
        current_path VARCHAR2(100);
    BEGIN
        -- Встановлення шляху для поточного вузла
        current_path := parent_path || node_title || '/';

        -- Вставка поточного вузла у CATEGORY_Materialized_Path
        INSERT INTO CATEGORY_Materialized_Path (id, title, path)
        VALUES (node_id, node_title, current_path);

        -- Відкриття курсора для обробки дочірніх вузлів
        OPEN child_cursor FOR
            SELECT id, title
            FROM CATEGORY_Adjacency_List
            WHERE parent_id = node_id;

        LOOP
            FETCH child_cursor INTO child_id, child_title;
            EXIT WHEN child_cursor%NOTFOUND;
            process_node(child_id, child_title, current_path);
        END LOOP;
        CLOSE child_cursor;
    END process_node;

BEGIN
    -- Ініціалізація процесу з кореневих вузлів
    FOR root_node IN    (SELECT id, title
                         FROM CATEGORY_Adjacency_List
                         WHERE parent_id IS NULL) LOOP
        process_node(root_node.id, root_node.title, '/');
    END LOOP;
END populate_materialized_path;
/
```

Рисунок А.4 – Процедура заповнення даними таблиці CATEGORY_Materialized_Path із заповненого випадковими значеннями дерева побудованого методом списків суміжних вершин (Adjacency List).

Запит	Кількість вузлів в ієрахії	Adjacency List	Nested Sets	Closure Table	Materialized Path
Виведення вузлів дерева, які не мають нащадків	100	+	+	+	_
	1000	+	+	+	_
	10000	+	+	+	_
виведення прямих дочірніх вузлів	100	+	+	+	+
	1000	+	-	+	+
	10000	+	-	+	-
виведення батьківського вузла	100	+	+	+	+
	1000	+	+	-	-
	10000	+	-	-	-
виведення шляху від одного вузла до іншого	100	+	+	+	+
	1000	+	+	+	+
	10000	+	-	-	+
виведення піддерева	100	-	+	+	+
	1000	-	+	-	+
	10000	-	-	+	-
додавання нового листка дерева	100	+	-	-	+
	1000	+	-	-	+
	10000	+	-	-	+
видалення вузла дерева, яке є листком	100	+	-	+	+
	1000	+	-	+	+
	10000	+	+	+	+
видалення піддерева	100	+	+	+	+
	1000	+	+	-	+
	10000	-	-	-	+

Abstract.

Bulatetska L.V. – Analysis of methods for modeling hierarchical structures in relational databases. Manuscript. Qualification work for the degree of "Master" in the field of Computer Science, educational program "Computer Science and Information Technologies." – Lesya Ukrainka Volyn National University. – 2024.

The study examines the main methods for representing hierarchical structures in relational databases: Adjacency List, Nested Sets, Closure Table, and Materialized Path. Each of these models has its advantages and disadvantages depending on the type of operations performed. Working with tree-like structures in relational databases is a complex task, as relational database management systems (RDBMS) and the SQL language do not provide specialized tools for efficiently managing and storing such data types. Quantitative metrics of data retrieval times for the methods under consideration were obtained. Based on these metrics, an analysis was conducted to determine the feasibility of using these methods depending on the characteristics of data nesting and volume. The 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 suboptimal for complex queries involving subtree retrieval or paths between nodes. The Nested Sets method demonstrates high efficiency for subtree retrieval queries; however, operations like adding and deleting nodes are complex and resource-intensive. The Closure Table offers high flexibility for queries involving ancestors and descendants but requires significant resources to maintain data integrity during frequent updates. The Materialized Path is efficient for queries that involve path retrieval between nodes and subtrees, though it necessitates additional operations to ensure data consistency during structural changes.

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

Анотація

Булатецька Л.В. – Аналіз методів моделювання ієрархічних структур в реляційних базах даних. – Рукопис.

Кваліфікаційна робота на здобуття освітнього ступеня «магістр» за спеціальністю 122 Комп'ютерні науки, освітньої програми Комп'ютерні науки та інформаційні технології. – Волинський національний університет імені Лесі Українки. – 2024 р.

У роботі розглянуто основні способи представлення ієрархічних структур у реляційних базах даних: Adjacency List, Nested Sets, Closure Table та Materialized Path. Кожна з цих моделей має свої переваги і недоліки залежно від типу операцій, що виконуються. Робота з деревовидними структурами в реляційних базах даних є складним завданням, оскільки реляційні СУБД і мова SQL не передбачають спеціалізованих засобів для ефективного управління та зберігання таких типів даних. Були отримані кількісні показники часу вибірки даних, які представлені в базі даних розглянутими методами. На основі цих показників проведено аналіз доцільності представлення даних розглянутими методами, у залежності від характеристик вкладеності та обсягів даних. Adjacency List є простою у реалізації і зручною для операцій на рівні окремих вузлів, таких як додавання або видалення листка, але не оптимальною для складних запитів, пов'язаних із отриманням піддерева або шляхів між вузлами. Nested Sets показує високу ефективність при виконанні запитів на отримання піддерева, проте операції додавання та видалення є складними та ресурсомісткими. Closure Table забезпечує високу гнучкість для запитів на визначення предків і нащадків, але потребує значних ресурсів для підтримки актуальності даних при частих оновленнях. Materialized Path відрізняється ефективністю у запитах на отримання шляху між вузлами та піддерев.

Ключові слова: Adjacency List, Nested Sets, Closure Table, Materialized Path ієрархічні структури, рекурсивна вибірка, реляційна модель даних.