Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего профессионального образования «Уральский федеральный

университет имени первого президента России Б. Н. Ельцина»

Институт фундаментального образования

Кафедра интеллектуальных информационных технологий

ДЕРЕВЬЯ

**Пояснительная записка**

|  |  |
| --- | --- |
| Руководитель | Свинцов Д.В. |
| Студент гр. ФО-350005 | Савинова М.А |

Екатеринбург 2017

Оглавление

[Подготовка 4](#_Toc496822550)

[Adjacency list 5](#_Toc496822551)

[Nested set 7](#_Toc496822552)

**ВВЕДЕНИЕ**

Довольно часто разработчики сталкиваются с такой проблемой как хранение иерархических данных или деревьев. Когда дело доходит до работы с иерархическими структурами данных, он берет готовую библиотеку, которая реализует одни из способов хранение информации и начинают ее использовать.

Однако существуют алгоритмы, которые позволяют хранить деревья в базе данных. Иерархические данные имеют связь “родитель-ребенок”, что усложняет хранение. В отличие от иерархических данных, реляционные же структуры представляют собой простые списки. Их удобнее хранить.

## Adjacency list

Список смежности (adjacency list) – один из способов представления графа в виде коллекции списков вершин. Каждой вершине графа соответствует список, состоящий из "соседей" этой вершины.[[1]](#footnote-1)

Для того чтобы хранить иерархические структуры методом adjacency list в базе данных необходимо хранить связь «родитель - ребенок». Пример:

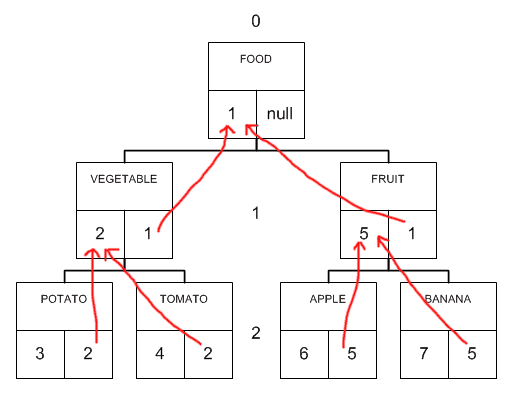


Рисунок 1 - Пример списка смежности

Имеем дерево, в котором в каждой ячейке хранится информация о названии, содержимом и о родителе. Так первый элемент корень дерева “Food” хранит в себе собственное значение 1 и значение родителя null. Второй элемент дерева “Vegetable” хранит в себе собственное значение 2, а значение родителя – 1.

В базе данных такая информация будет записываться так:

CREATE TABLE al\_tree (

id INT NOT NULL,

parent\_id INT NULL,

name VARCHAR(50) NOT NULL

);

CREATE INDEX fk\_tree\_tree ON al\_tree (parent\_id);

ALTER TABLE al\_tree ADD CONSTRAINT fk\_tree\_tree

FOREIGN KEY (parent\_id) REFERENCES al\_tree (id) ON UPDATE CASCADE ON DELETE CASCADE;

INSERT INTO al\_tree VALUES

(1, NULL, 'FOOD'),

(2, 1, 'VEGETABLE'),

(3, 2, 'POTATO'),

(4, 2, 'TOMATO'),

(5, 1, 'FRUIT'),

(6, 5, 'APPLE'),

(7, 5, 'BANANA');

То есть создаем таблицу с колонками id, id родителей и названием. Далее делаем возможность создания и редактирования родительских индексов. То же самое проделываем и для таблицы, используя конструкцию ALTER TABLE. Затем заполняем таблицу.

Для удобства чтения, можем воспользоваться запросом:

SELECT \* FROM al\_tree ORDER BY id;

Результат:

+----+-----------+-----------+

| id | parent\_id | name |

+----+-----------+-----------+

| 1 | NULL | FOOD |

| 2 | 1 | VEGETABLE |

| 3 | 2 | POTATO |

| 4 | 2 | TOMATO |

| 5 | 1 | FRUIT |

| 6 | 5 | APPLE |

| 7 | 5 | BANANA |

+----+-----------+-----------+

Данный способ помогает легко вносить изменения в таблицу, удалять заданные узлы или ветви. Особых недостатков я не заметила.

# Nested set

Вложенное множество (Nested set) – способ представления дерева, в котором в каждой вершине дерева указывается диапазон чисел.[[2]](#footnote-2)

Для построения дерева должны обойти дерево слева направо.

Пример:

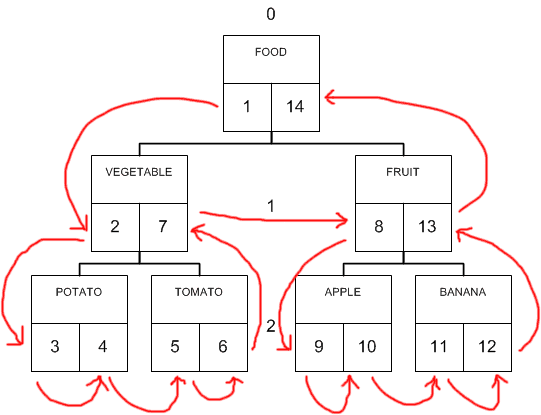


Рисунок 2 - Пример вложенного множества

Каждый узел получил 2 значения. Это диапазон тех значений, которые имеют его дети. Так мы точно знаем, что у корня дерева “Food” его потомки входят в интервал от 1 до 14. Также мы можем понять, что всего элементов в дереве 14/2 = 7. Ветка “Fruit” получила числа 8 и 13, которые включают в свой диапазон числа всех детей. Аналогично для других ветвей.

В базе данных такой запрос будет выглядеть вот так:

CREATE TABLE ns\_tree (

id INT NOT NULL,

name VARCHAR(50) NOT NULL,

lft INT NOT NULL,

rgt INT NOT NULL,

level INT NOT NULL

);

CREATE INDEX nslrl\_idx ON ns\_tree (lft, rgt, level);

INSERT INTO ns\_tree VALUES

(1, 'FOOD', 1, 14, 0),

(2, 'VEGETABLE', 2, 7, 1),

(3, 'POTATO', 3, 4, 2),

(4, 'TOMATO', 5, 6, 2),

(5, 'FRUIT', 8, 13, 1),

(6, 'APPLE', 9, 10, 2),

(7, 'BANANA', 11, 12, 2);

Для начала мы создаем таблицу с колонками “id” – универсальный ключ, “name” – название, “lft” – левый ключ, “rgt” – правый ключ и “level” – уровень. В “level” будет храниться информация об уровне вложенности. Хотя этот пункт не обязателен.

Для чтения дерева воспользуемся запросом:

SELECT node.id, node.name, node.level

FROM ns\_tree AS node,

ns\_tree AS parent

WHERE node.lft BETWEEN parent.lft AND parent.rgt

AND parent.id = 1

ORDER BY node.lft;

Результат будет таким:

+----+-----------+-------+

| id | name | level |

+----+-----------+-------+

| 1 | FOOD | 0 |

| 2 | VEGETABLE | 1 |

| 3 | POTATO | 2 |

| 4 | TOMATO | 2 |

| 5 | FRUIT | 1 |

| 6 | APPLE | 2 |

| 7 | BANANA | 2 |

+----+-----------+-------+

Таким образом, мы можем считывать дерево целиком или же ту ветку, которая нам нужна.

Пример: вывести часть ветки, в корневом узле которого находится “Vegetable”.

SELECT node.id, node.name, node.level

FROM ns\_tree AS node, ns\_tree AS parent

WHERE node.lft BETWEEN parent.lft AND parent.rgt AND parent.id = 2

ORDER BY node.lft;

Результат:

+----+-----------+-------+

| id | name | level |

+----+-----------+-------+

| 2 | VEGETABLE | 1 |

| 3 | POTATO | 2 |

| 4 | TOMATO | 2 |

+----+-----------+-------+

Если же мы хотим добавить новую ветку, выполняем запрос:

BEGIN;

SELECT @treeRight := rgt FROM ns\_tree

WHERE id = 2; /\* справа от ветки VEGETABLES, у которой id = 2 \*/

UPDATE ns\_tree SET rgt = rgt + 2 WHERE rgt > @treeRight;

UPDATE ns\_tree SET lft = lft + 2 WHERE lft > @treeRight;

INSERT INTO ns\_tree VALUES(0, 'SEA FOOD', @treeRight + 1, @treeRight + 2, 1);

COMMIT;

Удаление нужной ветки и ее содержимое:

BEGIN;

SELECT @treeLeft := lft, @treeRight := rgt, @treeWidth := rgt - lft + 1

FROM ns\_tree

WHERE id = 8; /\* здесь идентификатор новой записи с именем 'SEA FOOD' \*/

DELETE FROM ns\_tree WHERE lft BETWEEN @treeLeft AND @treeRight;

UPDATE ns\_tree SET rgt = rgt - @treeWidth WHERE rgt > @treeRight;

UPDATE ns\_tree SET lft = lft - @treeWidth WHERE lft > @treeRight;

COMMIT;

В данном случае мы удаляем не по id, а полную ветку со всем ее содержимым.

Вложенное множество - хороший способ для описания деревьев в базе данных любого объема. Однако если добавили ветку или удалили приходиться заново обходить дерево и назначать каждому узлу новое значение.

# Materialized Path

Существует еще один способ представления деревьев в базе данных. Это материализованный путь. Суть его заключается в том, что каждая вершина хранит полный путь. Пример:

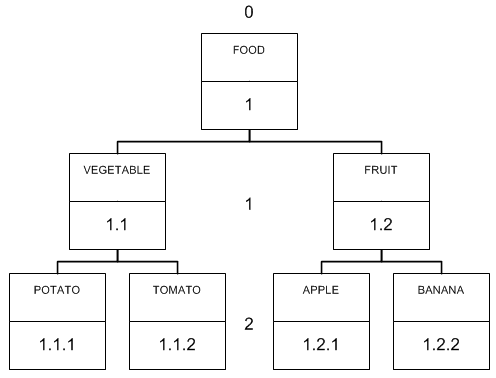


Рисунок 3 - Пример материализованного пути

Корень дерева, находящийся на 0 уровне, имеет значение 1. Его потомки хранят в себе значения родителя и свое собственное.

Также можно рассчитать насколько длинным будет адрес узла. Он будет соответствовать значению уровня +1.

Вывод дерева с помощью запроса:

CREATE TABLE mp\_tree (

id INT NOT NULL,

name VARCHAR(50) NOT NULL,

path VARCHAR(100) NOT NULL

);

CREATE INDEX mpp\_idx ON mp\_tree (`path`);

INSERT INTO mp\_tree VALUES

(1, 'FOOD', '1'),

(2, 'VEGETABLE', '1.1'),

(3, 'POTATO', '1.1.1'),

(4, 'TOMATO', '1.1.2'),

(5, 'FRUIT', '1.2'),

(6, 'APPLE', '1.2.1'),

(7, 'BANANA', '1.2.2');

Получаем результат:

+---------------+-------+

| name | path |

+---------------+-------+

| FOOD | 1 |

| VEGETABLE | 1.1 |

| POTATO | 1.1.1 |

| TOMATO | 1.1.2 |

| FRUIT | 1.2 |

| APPLE | 1.2.1 |

| BANANA | 1.2.2 |

+---------------+-------+

Поиск по узлу

SELECT t1.name from mp\_tree t1, mp\_tree t2

WHERE t2.path like CONCAT( t1.path, '.%')

AND t2.id = 3; /\* Поиск пути к узлу с именем 'POTATO' \*/

Выводит имя из таблиц t1, t2, у которого id = 3.

Результат:

+-----------+

| name |

+-----------+

| FOOD |

| VEGETABLE |

+-----------+

Также выполняется запрос, если мы хотим найти путь к узлу. В данном случае путь будет выглядеть так: ‘Food’/’Vegetable’/’Potato’.

Можем также найти уровень вложенности. Он будет равен количеству точек или количеству цифр – 1.

CREATE FUNCTION STRFIND (str VARCHAR(255), delimtr CHAR(1)) RETURNS INT

BEGIN

DECLARE \_cnt INT;

DECLARE \_start INT;

SET \_cnt = -1;

SET \_start = 1;

WHILE \_start > 0 DO

SET \_start = LOCATE( delimtr, str);

SET str = SUBSTRING( str, \_start + 1);

SET \_cnt = \_cnt + 1;

END WHILE;

RETURN \_cnt;

END

Нет трудности в алгоритмах поиска в материализованном пути. Чтение дерева гораздо удобнее, чем у вложенного множества и списка смежности. Но могут возникнуть сложности в вставки узла в уже существующую ветвь (например, в середину), потому что это повлечет за собой изменение узлов, которые находятся на нижних уровнях. Если же добавляем в конец, то запрос довольно простой.

1. Электронный ресурс: https://ru.wikipedia.org/wiki/Список\_смежности [↑](#footnote-ref-1)
2. Электронный ресурс: https://phpclub.ru/faq/Tree/Ns [↑](#footnote-ref-2)