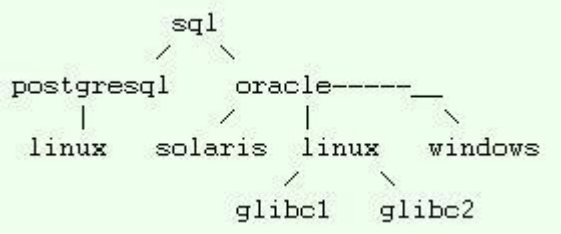
Drzewa w SQL

Część I

Dane testowe

We wszystkich przykładach będę się odwoływał do poniższego drzewa:



Metoda 1

Oddzielne table na każdy poziom zagnieżdżenia danych. W tym przypadku byłoby to:
tree_level_1 (id, name) :
(1,sql);
tree_level_2 (id, parent_id, name) :
(1, 1, postgresql),
(2, 1, oracle);
tree_level_3 (id, parent_id, name) :
(1, 1, linux),
(2, 2, solaris);
(3, 2, linux);
(4, 2, windows);
tree_level_4 (id, parent_id, name) :
(1, 3, glibc1),
(2, 3, glibc2);
Metoda ta wybierana jest najczęściej przez początkujących programistów. Bardziej doświadczeni bazo-danowcy, odrzucają ją ze względu na konieczność modyfikowania *struktury bazy danych* przy zmianie danych!!!
Praktycznie rzecz biorąc nie powinna być nigdy stosowana.

Cechy:
+ trywialne w implementacji
+ dosyć szybkie tworzenie pełnej nazwy (z zastrzeżeniem o znajdywaniu danych konkretnego (początkowego) ID)
- skomplikowane wyszukiwanie danych na podstawie konkretnego "ID"
- ograniczona ilość poziomów zagnieżdżeń
- bardzo wolne znajdowanie np. wszystkich książek w kategorii "informatyka" i podkategoriach informatyki (dowolnie głęboko)
- wolne wyszukiwanie podkategorii niebezpośrednich.

Metoda 2

Tabela typu:

```
create table kategorie (
id serial,
parent_id int8,
name text not null default '',
primary key (id)
);
create unique index ui_kategorie_pin on kategorie (parent_id, name);
alter table kategorie add foreign key (parent_id) references kategorie (id);
```

Cechy:
+ trywialne w implementacji
+ nieograniczona ilość poziomów zagnieżdżeń
+ błyskawiczne wyszukiwanie podkategoriii danej kategorii
- wolne tworzenie pełnej nazwy (chyba, że przechowujesz ją w polu name, ale to marnotrawstwo miejsca)
- bardzo wolne znajdowanie np. wszystkich książek w kategorii "informatyka" i podkategoriach informatyki (dowolnie głęboko)
- wolne wyszukiwanie podkategorii niebezpośrednich.

Metoda 3

Układ typu nested sets (vide książka "SQL - zaawansowane programowanie", autor: Joe Celko).
Tabela plus minus jak wyżej, ale zamiast parent_id jest left_mark i right_mark. Left_mark i right_mark są wyliczane.
Przykład (w nawiasach przy nazwie odpowiednio: left_mark, right_mark).
(1, 18) sql

(2, 5) sql/postgresql
(6, 17) sql/oracle
(3, 4) sql/postgresql/linux
(7, 8) sql/oracle/solaris
(9, 14) sql/oracle/linux
(15, 16) sql/oracle/windows
(10, 11) sql/oracle/linux/glibc1
(12, 13) sql/oracle/linux/glibc2

Wartości left/right mark są nadawane w następujący sposób: zaczynając od konkretnego miejsca w drzewie, nadajemy mu kolejny numer (np. 1), i znajdujemy pierwsze "dziecko". "Pierwsze" definiujemy w dowolny sposób - tu akurat użyłem odwzorowania graficznego. Jeśli dany element nie posiada "dzieci", zwiekszam licznik o jeden, ustawiam right-mark oraz wraz z numerowaniem jeden poziom w górę.

Przykład:

Zaczynam od "sql". Ustawiam mu left-mark na 1, znajduję pierwsze "dziecko" - czyli postgresql, ustawiam mu left-mark na 2, znajduję kolejne pierwsze "dziecko" czyli "linux", ustawiam mu left-mark na 3.

Ponieważ "linux" nie ma "dzieci", ustawiam mu right-mark na 4, po czym wracam poziom wyżej.

"postgresql" nie ma innych dzieci, więc dostaje right-mark 5 i idę do kolejnego "dziecka" sql -> czyli "oracle".

Proces powtarzam, aż wrócę do "sql" i nadam mu right-marka.

Cechy:

+ niesamowicie szybkie wyszukiwanie w dół drzewa od określonego elementu (zakładając wyszukiwanie bez limitu głębokości)
+ nieograniczona ilość poziomów zagnieżdżeń
- nietrywialne w implementacji
- dużo więcej pracy przy wstawianiu i usuwaniu elementów
- utrudnione wyszukiwanie bezpośrednich podkategorii danej kategorii.

Metoda 4

Metoda 1 + metoda 2, czyli wykorzystanie i parent_id i left/right mark-ów. Opisu nie będę robił, bo jest oczywisty.

Cechy:

+ nieograniczona ilość poziomów zagnieżdżeń
+ błyskawiczne wyszukiwanie podkategoriii danej kategorii
+ niesamowicie szybkie wyszukiwanie w dół drzewa od określonego elementu (zakładając wyszukiwanie bez limitu głębokości)
- sporo pracy przy wstawianiu i usuwaniu elementów

- nietrywialne w implementacji
- nadmiarowość informacji (choć w bazach danych to raczej standard)

Metoda 5

Wymyśliliśmy ją z współpracownikami w trakcie pracy nad jednym z wspólnych projektów.
2 tabele:
create table kategorie (
id serial,
name text,
primary key (id)
);
-- ważne: name nie jest unique !!!!
create table powiazania (
first_id int8,
second_id int8,
depth int8,
primary key (first_id, second_id)
);
alter table powiazania add foreign key (first_id) references kategorie (id);
alter table powiazania add foreign key (second_id) references kategorie (id);
Pole depth oznacza, o ile poziomów "głębiej" jest kategoria second od first.
Np. dla sytuacji z metody 2:
kategorie:

```
id | name

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

1 | sql

2 | postgresql

3 | oracle

4 | linux

5 | solaris

6 | linux
```

```
7 | windows

8 | glibc1

9 | glibc2
```

powiazania:

```
first_id | second_id | depth

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

1 | 2 | 1

1 | 3 | 1

1 | 4 | 2

1 | 5 | 2

1 | 6 | 2

1 | 7 | 2

1 | 8 | 3

1 | 9 | 3

2 | 4 | 1

3 | 5 | 1

3 | 6 | 1

3 | 7 | 1

3 | 8 | 2

3 | 9 | 2

6 | 8 | 1
```

```
6 | 9 | 1
```

Ze względów wydajnościowych oraz ułatwiajacych programowanie polecam dopisywanie dodatkowo powiazań z głębokością 0, czyli w naszym przypadku 9 rekordów:

```
first_id | second_id | depth

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

1 | 1 | 0

2 | 2 | 0

3 | 3 | 0

4 | 4 | 0

5 | 5 | 0

6 | 6 | 0

7 | 7 | 0

8 | 8 | 0

9 | 9 | 0
```

Może nie wydaje się to bardzo potrzebne, ale wiele razy okazało się, że taka dodatkowa ilość danych bardzo ułatwia późniejsze oprogramowywanie.
Zwrócić należy uwagę, że ilość dodatkowych danych jakie trzeba przechowywać jest zależna od maksymalnej głębokości zagnieżdżenia drzewa.
W skrajnym przypadku ilość danych które trzeba zapisać jest zbliżona do (n^2)/2 * wielkość rekordu w tablicy powiązania.
W praktyce jednak zazwyczaj nie przekracza się wartości 2n do 3n, co oznacza, że do zapisania drzewa ze 100 elementami potrzebujemy (zazwyczaj) około 200-300 rekordów w tablicy powiązania. A dzięki temu, że dane tam są proste (3 pola typu INT), wielkość pojedyńczego rekordu wynosi 12 (lub 24 dla INT8) bajtów - co nie jest wielką wartością.
Cechy:
+ relatywnie proste w implementacji
+ nieograniczona ilość poziomów zagnieżdżeń
+ błyskawiczne wyszukiwanie podkategoriii danej kategorii

+ niesamowicie szybkie wyszukiwanie w dół rzewa od określonego elementu (niezależnie od tego, czy z limitem, czy bez limitu głębokości)
+ proste wstawianie i usuwanie rekordów
+ szybkie (dużo szybsze niż w jakiejkolwiek innej implementacji) tworzenie pełnych nazw
+ szybkie wyszukiwanie kategorii niebezpośrednich (np. znajdź wszystkie podkategorie podkategorii "informatyki". czyli np. informatyka/podręczniki/sql tak, ale informatyka/sql już nie)
- nadmiarowość informacji (choć w bazach danych to raczej standard).

Więcej o drzewach w SQL:

- A Look at SQL Trees - http://www.dbmsmag.com/9603d06.html
- Nested Model Sets - http://www.dbmsmag.com/9604d06.html
- Trees in SQL - http://www.dbmsmag.com/9605d06.html

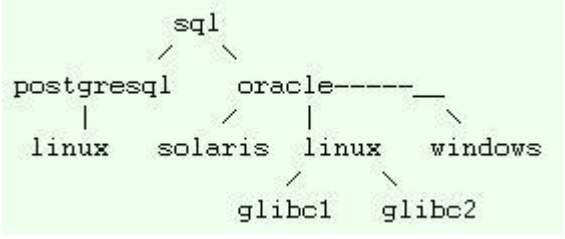
Część II

wprowadzenie

po tym jak napisałem tekst (część I) nt. różnych metod zapisu drzew w sqlu trafiło do mnie sporo pytań jak zrobić to czy tamto, jak przyspieszyć, jak uzdatnić, jak skonwertować.
w ten sposób zostałem zmotywowany do napisania poniższego tekstu nt. implementacji drzew w/g metody "5" ze wspomnianego wprowadzenia.
ostrzegam jedynie: wszystkie podane przykłady napisałem w oparciu o bazę postgresql w wersji 7.3.4 i nie daję żadnej gwarancji, że podane zapytania zadziałają pod czymkolwiek innym - innej wersji postgresa czy z innym silnikiem bazy danych. w szczególności tyczy się to uruchamiania funkcji napisanych w pl/pgsql który to język jest dostępny tylko i wyłączenie w bazie postgresql.
tymniemniej, jeśli używasz innej bazy - po przeczytaniu tego tekstu będziesz miał przynajmniej ogólne pojęcia jak powinny wyglądać odpowiednie zapytania i z mniejszym lub większym nakładem czasu/pracy dasz radę przekonwertować je do swojej bazy danych.

dane testowe

we wszystkich przykładach będę się odwoływał do poniższego drzewa:



struktura tabel

tworzymy dwie podstawowe tabelki - ponieważ jedyną daną nt. elementu drzewa jest jego nazwa, możemy użyć takiej struktury:
CREATE TABLE kategorie (
id serial,
name TEXT,
PRIMARY KEY (id)
);

CREATE TABLE powiazania (
parent_id integer NOT NULL references kategorie (id),
child_id integer NOT NULL references kategorie (id),
depth integer
);

wstawianie rekordów

wstawianie rekordów do przygotowanej struktury jest proste i strasznie powtarzalne.wstawmy kilka pierwszych kategorii:
zaczniemy od (oczywiście) rekordu pierwszego: **sql**:
INSERT INTO kategorie (name) VALUES ('sql');
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (currval('kategorie_id_seq'), currval('kategorie_id_seq'), 0);
w drugim insercie wykorzytujemy funkcję postgresql currval - zwraca ona aktualną wartość sekwencji. wartość ta została ustalona pierwszym insertem - po szczegóły odsyłam do manuala.
po obu insertach mamy taką sytuację:
SELECT * FROM kategorie; id | name

```
----+------

1 | sql

(1 row)
```

SELECT * FROM powiazania;

```
parent_id | child_id | depth
```

```
-----------+----------+-------

  1 | 1 | 0
```

(1 row)
jako uprzejmy użytkownik postgresql'a zajmę się teraz wstawieniem kategorii **oracle**:
INSERT INTO kategorie (name) VALUES ('oracle');
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (currval('kategorie_id_seq'), currval('kategorie_id_seq'), 0);
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (1, currval('kategorie_id_seq'), 1);
po wykonaniu tych poleceń uzyskujemy:
SELECT * FROM kategorie;

```
  id | name

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

  1 | sql

  2 | oracle

  (2 rows)
```

SELECT * FROM powiazania;

```
  parent_id | child_id | depth

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

  1 | 1 | 0

  2 | 2 | 0

  1 | 2 | 1
```

(3 rows)
zwracam uwagę na to, że aby dodać rekord z "depth" większym niż 0 - musimy znać id "przodków" danego elementu/kategorii.
jak do tej pory wszystko było miłe i łatwe. teraz zaczną się "schody":
dodajemy **solaris**:

INSERT INTO kategorie (name) VALUES ('solaris');
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (currval('kategorie_id_seq'), currval('kategorie_id_seq'), 0);
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (2, currval('kategorie_id_seq'), 1);
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (1, currval('kategorie_id_seq'), 2);
dodaliśmy, tabelki wyglądają tak:
SELECT * FROM kategorie;

```
  id | name

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

  1 | sql

  2 | oracle

  3 | solaris
```

(3 rows)

SELECT * FROM powiazania;

```
  parent_id | child_id | depth

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

  1 | 1 | 0

  2 | 2 | 0

  1 | 2 | 1

  3 | 3 | 0

  2 | 3 | 1

  1 | 3 | 2
```

(6 rows)
dalsze wstawianie może się odbywać analogicznie, lub ...

optymalizacja wstawiania rekordów

jak widać ilośc wykonywanych insertów rośnie wraz z poziomem zagnieżdżenia danego elementu/kategorii. nie jest to może największy problem, ale już fakt, że musimy znać id wszystkich "przodków" danego elementu powoduje, że musimy robić dodatkowe selecty, pętle itp. przekłada się to na zwiększenie kodu a więc i szans na zrobienie błędu. czy da się tego uniknąć? oczywiście.

pierwsza metoda jest całkiem niezła i jednocześnie bardzo przenośna. zobaczymy jak wygląda na przykładzie "windows":

```
INSERT INTO kategorie (name) VALUES ('windows');
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (currval('kategorie_id_seq'), currval('kategorie_id_seq'), 0);
INSERT INTO powiazania (parent_id, child_id, depth)
SELECT parent_id, currval('kategorie_id_seq'), depth + 1 FROM powiazania WHERE child_id = 2;
```

wynik wykonania tych poleceń?

```
# SELECT * FROM kategorie;

id | name

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

1 | sql

2 | oracle

3 | solaris

4 | windows

(4 rows)

# SELECT * FROM powiazania;

parent_id | child_id | depth

-----------+----------+-------
```

1 | 1 | 0

2 | 2 | 0

1 | 2 | 1

3 | 3 | 0

2 | 3 | 1

1 | 3 | 2

4 | 4 | 0

2 | 4 | 1

1 | 4 | 2

(9 rows)

jak to działa?

pierwszy insert jest oczywisty - musimy mieć kategorię by ją jakoś "podpiąć" do drzewa. drugi - kategoria musi wiedzieć, że jest sobie sama ojcem - może nie wydaje się to oczywiste, ale jak dojdę do odczytywania danych z tego drzewa zrozumiecie, że znakomicie upraszcza to parę zapytań. a co z trzecim insertem?

zobaczmy najpierw co robi select użyty przy trzecim insercie:
SELECT parent_id, currval('kategorie_id_seq'), depth + 1 FROM powiazania WHERE child_id = 2;

parent_id | currval | ?column?

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

2 | 4 | 1

1 | 4 | 2

(2 rows)

jak widać tworzy on automatycznie wszystkie rekordy potrzebne od ojca "windows" (czyli "oracle") wzwyż. jak on to robi? zobaczmy:

zaczynamy od prostego selecta który wyciągnie informacje o wszystkich przodkach danego id (w tym przypadku "oracle"):
SELECT parent_id, child_id, depth FROM powiazania WHERE child_id = 2;

parent_id | child_id | depth

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

2 | 2 | 0

1 | 2 | 1

(2 rows)

można łatwo zauważyć, że jak ktoś był "przodkiem" "ojca" to jest także "przodkiem" aktualnego elementu, tyle, że odległość (depth) jest większa o 1:
SELECT parent_id, child_id, depth + 1 FROM powiazania WHERE child_id = 2;

parent_id | child_id | ?column?

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

2 | 2 | 1

1 | 2 | 2

(2 rows)

mając to wystarczy tylko podmienić child_id na id naszego nowego elementu (currval('kategorie_id_seq')) i uzyskujemy to o co chodziło:
SELECT parent_id, currval('kategorie_id_seq'), depth + 1 FROM powiazania WHERE child_id = 2;

parent_id | currval | ?column?

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

2 | 4 | 1

1 | 4 | 2

(2 rows)

natomiast konstrukcja INSERT INTO tabelka (pola) SELECT ... powoduje, wstawienie wyselectowanych rekordów do wskazanej tabelki - w naszym przypadku oszczędzając nam pisania wielu insertów - a więc skracając kod i zmniejszając szanse na wystąpienie błędu w nim.

co istotne: o ile do tej pory ilość insertów rosła z każdym poziomem, o tyle teraz widzimy, że jest ona stała - niezależnie od tego czy dodajemy coś na 2 poziomie zagnieżdżenia czy na 101 - zawsze do wykonania są tylko 2 inserty do tabeli z powiazaniami. czy da się lepiej?

oczywiście. druga metoda optymalizacji wstawiania - najlepsza jaką teraz znam, ale niestety nie do końca przenośna - wymaga użycia procedur/funkcji składowanych i triggera. większość baz danych posiada obie te cechy, ale nadal są też bazy danych gdzie w aktualnych wersjach będzie to niemożliwe.

kod wygląda tak:

```
CREATE OR REPLACE FUNCTION trg_powiazania_i () RETURNS TRIGGER AS '
DECLARE
BEGIN
IF NEW.depth <> 1 THEN
RETURN NEW;
END IF;
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (NEW.child_id, NEW.child_id, 0);
INSERT INTO powiazania (parent_id, child_id, depth)
SELECT parent_id, NEW.child_id, depth + 1 FROM powiazania WHERE child_id = NEW.parent_id AND depth > 0;
RETURN NEW;
END;
' LANGUAGE 'plpgsql';
```

CREATE TRIGGER trg_powiazania_i AFTER INSERT ON powiazania FOR EACH ROW EXECUTE PROCEDURE trg_powiazania_i ();

analizę działania tej funkcji zostawię sobie na chwilę na boku, natomiast pokażę jak to działa. dodajmy kategorię "linux":
INSERT INTO kategorie (name) VALUES ('linux');
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (2, currval('kategorie_id_seq'), 1);

i to wszystko - jeden insert (do powiazań) wystarczył. wynik:
SELECT * FROM kategorie;
id | name

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

1 | sql

2 | oracle

3 | solaris

4 | windows

5 | linux

(5 rows)

SELECT * FROM powiazania;
parent_id | child_id | depth
-----------+----------+-------
1 | 1 | 0
2 | 2 | 0
1 | 2 | 1
3 | 3 | 0
2 | 3 | 1
1 | 3 | 2
4 | 4 | 0
2 | 4 | 1

1 | 4 | 2
2 | 5 | 1
5 | 5 | 0
1 | 5 | 2
(12 rows)

jak widać są wszystkie potrzebne rekordy. jak to zadziałało?

przypomnę kod który tworzył i aktywował trigger:

```
CREATE OR REPLACE FUNCTION trg_powiazania_i () RETURNS TRIGGER AS '
DECLARE
BEGIN
IF NEW.depth <> 1 THEN
RETURN NEW;
END IF;
INSERT INTO powiazania (parent_id, child_id, depth) VALUES (NEW.child_id, NEW.child_id,
0);
INSERT INTO powiazania (parent_id, child_id, depth)
SELECT parent_id, NEW.child_id, depth + 1 FROM powiazania WHERE child_id =
NEW.parent_id AND depth > 0;
RETURN NEW;
END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER trg_powiazania_i AFTER INSERT ON powiazania FOR EACH ROW
EXECUTE PROCEDURE trg_powiazania_i ();
```

tłumaczenie co co robi zacznę od końca: linia CREATE TRIGGER ... stworzyła (aktywowała) triggera który po (AFTER) każdym wstawieniu rekordu (INSERT) wykona pewną procedure składowaną (EXECUTE PROCEDURE trg_powiazania_i). trigger będzie miał nazwę taką samą jak procedura - nie jest to konieczne, natomiast ułatwia późniejszą analizę bazy.

użyliśmy triggera "AFTER" a nie "BEFORE" aby oszczędzić bazie danych pracy w sytuacji gdyby się okazało, że wstawiane dane są niepoprawne - trigger "AFTER" jest uruchamiany po wszystkich testach poprawności: constraint'ach, check'ach, warunkach unikalności itp.

wiemy już, że insert wykona procedurę, ale co ona zrobi?

w pierwszej linii (tzn. IF NEW.depth) sprawdzamy czy nowo wsawiany rekord (dostępny jako

zmienna NEW) ma ustawioną głębokość "1". jeśli nie, to kończymy działanie procedury i zwracamy przekazany rekord - jest to przyjęty sposób "zaakceptowania" wstawienia rekordu, bez wykonywania jakichkolwiek dodatkowych poleceń.

następnie wstawiemy rekord powiązań, który w obu polach (parent_id i child_id) ma wstawioną wartość NEW.child_id - czyli naszego nowo wstawianego elementu drzewa. ponieważ parent_id == child_id, więc depth będzie równa 0.

następnie korzystając z opisanej wyżej metody wstawiamy wszystkie brakujące "powiązania" - istotnym jest, że w warunku WHERE do SELECTa eliminujemy rekord w którym depth == 0. eliminujemy go, gdyż byłby on duplikatem rekordu, którego wstawienie wywołało trigger.

należy zwrócić uwagę, iż wszystkie rekordu wstawiane przez trigger będą miały depth <> 1 - trigger wstawi jeden rekord z depth == 0, oraz pewną ilość rekordów z depth >= 2 - a ponieważ każde wstawienie uruchomi ten sam trigger - okazuje się, że pierwszy warunek (IF NEW.depth <> 1) jest konieczny aby nie wpaść w nieskończoną pętlę.

uwagi końcowe do wstawiania rekordów

podane motody nie wyczerpują tematu - można np. dodać do tabeli "kategorie" pole "parent_id" przechowujące "ojca", i ograniczyć wstawianie rekordów zamiast 2 insertów (do kategorii i do powiązań) do tylko jednego, a całe wypełnianie tabeli powiązania zostawić triggerowi (nelaży przy tym tylko pamiętać by usunąć warunek co do pola depth z select'a przy insert'cie w procedurze - gdyż inaczej tej informacji będzie brakować w powiązaniach - tworząc trudną do oprogramowania sytuację. można także użyć w ogóle w całości procedury skłądowanej wywoływanej np. tak:
SELECT addItemToTree('nazwa nowego elementu', id_rodzica);
i pozostawić całą modyfikację obu tabel w ramach tej procedury.

wybrana metoda zależy tylko i wyłącznie od potrzeb konkretnego projektu.

wstęp do odczytu danych

zanim zacznę odczytywać dane, pozwolę sobie załadować do bazy pozostałe rekordy z testowego drzewa. po wykonaniu odpowiednich insertów uzyskuję następującą sytuację:

```
# SELECT * FROM kategorie;
 id |    name
----+------------
  1 | sql
  2 | oracle
  3 | solaris
  4 | windows
  5 | linux
  6 | glibc1
  7 | glibc2
  8 | postgresql
  9 | linux
(9 rows)
```

```
# SELECT * FROM powiazania;
 parent_id | child_id | depth
-----------+----------+-------
         1 |        1 |     0
         2 |        2 |     0
         1 |        2 |     1
         3 |        3 |     0
         2 |        3 |     1
         1 |        3 |     2
         4 |        4 |     0
         2 |        4 |     1
         1 |        4 |     2
         2 |        5 |     1
         5 |        5 |     0
         1 |        5 |     2
         5 |        6 |     1
         6 |        6 |     0
         2 |        6 |     2
         1 |        6 |     3
         5 |        7 |     1
         7 |        7 |     0
         2 |        7 |     2
         1 |        7 |     3
         1 |        8 |     1
         8 |        8 |     0
         8 |        9 |     1
         9 |        9 |     0
         1 |        9 |     2
(25 rows)
```

odczytywanie danych

w większości poniższych przykładów nie będę pokazywał wyników poleceń - możecie państwo uwierzyć mi, że działają (nie polecam), lub spróbować samemu korzystając z przedstawionych powyżej metod wstawiania danych (polecam).
prześledźmy kilka "scenariuszy" w których potrzeba odczytać dane z drzewa:
wyświetlenie wszystkich kategorii głównych
aby to zrobić wystarczy zauważyć, że kategoriia główna (top-levelowa) to taka która nie ma "ojca", a więc nie istnieje dla niech powiązanie z depth == 1. zapytanie:

```
SELECT
k.name
FROM
kategorie k
WHERE
NOT exists (
SELECT *
FROM powiazania p
WHERE k.id = p.child_id AND p.depth = 1
);
```

wyświetlenie wszystkich podkategorii danej kategorii

to dosyć proste zadanie wymaga jedynie skojarzenia, że wszystkie podkategorie będą miały powiązanie do "ojca" z depth==1, tak więc (dla kategorii "oracle"):

```
SELECT
k.name
FROM
kategorie k join powiazania p on k.id = p.child_id
WHERE
p.parent_id = 2 AND
p.depth = 1;
```

pobranie "ścieżki" do danej kategorii, tak aby np. wyświetlić ją w postaci klikalnej "mapy"

zakładam, że generowanie "mapy" jest poza bazą i wymaga nazwy (by ją wyświetlić) oraz id (aby użyć przy kliknięciu). najlepiej by było gdyby dane były posortowane od "korzenia" aż do aktualnej kategorii. dla kategorii glibc1 będzie to:

```
SELECT
k.id,
k.name
FROM
kategorie k join powiazania p on k.id = p.parent_id
WHERE
p.child_id = 6
ORDER BY
p.depth desc;
```

wyciągnięcie danych w kolejności rysowania drzewa

tu może wyjaśnię co mam na myśli - chodzi o takie wyciągnięcie danych, aby rekordy na wyjściu były w takiej kolejności (dla całego drzewa):

sql, postgresql, linux, oracle, solaris, linux, glibc1, glibc2, windows

aby to zrobić należy użyć jakiegoś sortowania - jakiego - pytanie jest dosyć trudne i możliwe do odpowiedzenia jedynie w sytuacji gdy mamy do dyspozycji procedury składowane. z pomocą przyjdzie nam taka procedura:

```
CREATE OR REPLACE FUNCTION createtreepath(integer) RETURNS TEXT AS '
declare
in_leaf_id ALIAS FOR $1;
temprec RECORD;
reply TEXT;
BEGIN
reply := "/";
FOR temprec IN SELECT k.name FROM kategorie k join powiazania p on k.id = p.parent_id
WHERE p.child_id = in_leaf_id ORDER BY p.depth desc LOOP
reply := reply || temprec.name || "/";
END LOOP;
```

RETURN reply;
END;
' LANGUAGE 'plpgsql';
funkcja ta zwraca ścieżkę (w postaci tekstu) dla danego elementu drzewa. przykładowe wywołanie:

```
# select *, createtreepath(id) from kategorie ;
 id |    name    |      createtreepath
----+------------+--------------------------
  1 | sql        | /sql/
  2 | oracle     | /sql/oracle/
  3 | solaris    | /sql/oracle/solaris/
  4 | windows    | /sql/oracle/windows/
  5 | linux      | /sql/oracle/linux/
  6 | glibc1     | /sql/oracle/linux/glibc1/
  7 | glibc2     | /sql/oracle/linux/glibc2/
  8 | postgresql | /sql/postgresql/
  9 | linux      | /sql/postgresql/linux/
(9 rows)
```

tak wygenerowaną ścieżkę można użyć do sortowania:

```
# select *, createtreepath(id) from kategorie order by 3;
 id |    name    |      createtreepath
----+------------+--------------------------
  1 | sql        | /sql/
  2 | oracle     | /sql/oracle/
  5 | linux      | /sql/oracle/linux/
  6 | glibc1     | /sql/oracle/linux/glibc1/
  7 | glibc2     | /sql/oracle/linux/glibc2/
  3 | solaris    | /sql/oracle/solaris/
  4 | windows    | /sql/oracle/windows/
  8 | postgresql | /sql/postgresql/
  9 | linux      | /sql/postgresql/linux/
(9 rows)
```

jak widać powstała struktura nie jest dokładnie zgodna z oczekiwaniami - dokładniej: postgresql jest za oraclem, a i kolejność podkategorii kategorii oracle nie jest taka jak chcieliśmy. aby to ominąć, należy dodać do tabeli kategorii dodatkowe pole (o nazwie np. "ordering") zawierające liczbę która będzie używana do generowania ścieżki (a więc i sortowania):
ALTER TABLE kategorie add column ordering integer;
UPDATE kategorie SET ordering = id;
ALTER TABLE kategorie ALTER column ordering SET NOT NULL;
jak widać ustawiłem pole ordering jako not null - chodzi o uproszczenie budowy funkcji createtreepath oraz pewność, że dane do sortowania tam będą. mając te dane, możemy przebudować funkcję createtreepath:
CREATE OR REPLACE FUNCTION createTreePath(integer) RETURNS TEXT AS '
declare
in_leaf_id ALIAS FOR $1;
temprec RECORD;
reply TEXT;

```
BEGIN
reply := "/";
FOR temprec IN SELECT k.ordering FROM kategorie k join powiazania p on k.id =
p.parent_id WHERE p.child_id = in_leaf_id ORDER BY p.depth desc LOOP
reply := reply || temprec.ordering::TEXT || "/";
END LOOP;
RETURN reply;
END;
' LANGUAGE 'plpgsql';
```

oczywiście w tym momencie nasuwa się pytanie - czemu po prostu nie użyć wartości id zamiast wprowadzać dodatkowe pole. przyczyna jest prosta - zmiana wartości klucza głównego jest praktycznie zawsze złym pomysłem. podczas gdy zmiana wartości pola "ordering" niczego nie psuje.

select z orderem wygląda teraz tak:

```
# select *, createtreepath(id) from kategorie order by 4;
 id |    name    | ordering | createtreepath
----+------------+----------+----------------
  1 | sql        |        1 | /1/
  2 | oracle     |        2 | /1/2/
  3 | solaris    |        3 | /1/2/3/
  4 | windows    |        4 | /1/2/4/
  5 | linux      |        5 | /1/2/5/
  6 | glibc1     |        6 | /1/2/5/6/
  7 | glibc2     |        7 | /1/2/5/7/
  8 | postgresql |        8 | /1/8/
  9 | linux      |        9 | /1/8/9/
(9 rows)
```

coś się zmieniło, ale nie do końca tak jak byśmy chcieli. zobaczmy więc ...

```
UPDATE kategorie SET ordering=1 WHERE id = 8;
UPDATE kategorie SET ordering=100 WHERE id = 4;
```

pierwszy update powoduje, że postgresql otrzymuje niższy ordering niż oracle, więc pojawi się przed oracle. drugi update natomiast przydziela windows'om ordering 100, co powinno wyrzucić je na koniec listy podkategorii oracle. ale:

```
# select *, createtreepath(id) from kategorie order by 4;
 id |    name    | ordering | createtreepath
----+------------+----------+----------------
  1 | sql        |        1 | /1/
  8 | postgresql |        1 | /1/1/
  9 | linux      |        9 | /1/1/9/
  2 | oracle     |        2 | /1/2/
  4 | windows    |      100 | /1/2/100/
  3 | solaris    |        3 | /1/2/3/
  5 | linux      |        5 | /1/2/5/
  6 | glibc1     |        6 | /1/2/5/6/
  7 | glibc2     |        7 | /1/2/5/7/
(9 rows)
```

pojawił się problem. porównywanie przy orderze jest leksykalne, a nie numeryczne. czy można to naprawić? tak:

```
CREATE OR REPLACE FUNCTION createTreePath(integer) RETURNS TEXT AS '
declare
in_leaf_id ALIAS FOR $1;
temprec RECORD;
reply TEXT;
BEGIN
reply := "/";
FOR temprec IN SELECT k.ordering FROM kategorie k join powiazania p on k.id =
p.parent_id WHERE p.child_id = in_leaf_id ORDER BY p.depth desc LOOP
reply := reply || to_char(temprec.ordering,"000000") || "/";
END LOOP;
RETURN reply;
END;
' LANGUAGE 'plpgsql';
```

jak widać wprowadzona zmiana oznacza zapisywanie liczb z wiodącymi zerami - tak aby miały po 6 cyfr. jest to rozwiązanie nie do końca eleganckie, ale ma jedną bardzo ważną zaletę: działa:

```
# select *, createtreepath(id) from kategorie order by 4;
 id |    name    | ordering |          createtreepath
----+------------+----------+-----------------------------------
  1 | sql        |        1 | / 000001/
  8 | postgresql |        1 | / 000001/ 000001/
  9 | linux      |        9 | / 000001/ 000001/ 000009/
  2 | oracle     |        2 | / 000001/ 000002/
  3 | solaris    |        3 | / 000001/ 000002/ 000003/
  5 | linux      |        5 | / 000001/ 000002/ 000005/
  6 | glibc1     |        6 | / 000001/ 000002/ 000005/ 000006/
  7 | glibc2     |        7 | / 000001/ 000002/ 000005/ 000007/
  4 | windows    |      100 | / 000001/ 000002/ 000100/
(9 rows)
```

problem ten nie pojawiłby się gdyby zawsze stosować do pola ordering wartości o tej samej długości w znakach. ponieważ jednak upilnowanie tego może być trudne - takie obejście przy pomocy to_char w 100% nas zabezpiecza (no, chyba, że ktoś użyje liczby 7 lub więcej cyfrowej, ale to z kolei można ominąć dodając dodatkowe zera do to_char'a).
zwrócić należy uwagę iż mimo, że nie ma formalengo nakazu unikalności wartości ordering to, należy jednak dbać o to by w ramach kategorii na tym samym poziomie zagnieżdżenia nie było dwóch o tym samym orderingu - może to prowadzić do dziwnych i trudnych do wykrycia błędów. najbezpieczniej więc jest po prostu założyć na to pole indeks unikalny i nie zawracać sobie tym głowy.

sprawdzenie na jakim poziomie zagnieżdżenia jest dana kategoria

aby to sprawdzić wystarczy spojrzeć na to, że wartość ta jest tożsama z największą "głębokością" (depth) zarejestrowaną w tabeli powiazania. mimo, iż teoretycznie wystarczy (dla kategorii glibc1):

```
SELECT
depth
FROM
powiazania
WHERE
child_id = 6
ORDER BY
depth desc
LIMIT 1;
```

to jednak sugeruję włożyć to w procedurę składowaną - pozwoli to na kilka dosyć przydatnych rzeczy. np. taka procedura:

```
CREATE OR REPLACE FUNCTION getitemlevel(integer) RETURNS integer AS '
DECLARE
in_item_id ALIAS FOR $1;
reply INTEGER;
BEGIN
SELECT depth INTO reply FROM powiazania WHERE child_id = in_item_id ORDER BY
depth desc LIMIT 1;
RETURN reply;
END;
' LANGUAGE 'plpgsql';
```

mając taką funkcję możemy np zrobić coś takiego:

```
# SELECT id, repeat(' ', getitemlevel(id))||'+- '||name FROM kategorie ORDER BY
createtreepath(id);
```

```
 id |     ?column?
----+-----------------
  1 | +- sql
  8 |   +- postgresql
  9 |     +- linux
  2 |   +- oracle
  3 |     +- solaris
  5 |     +- linux
  6 |       +- glibc1
  7 |       +- glibc2
  4 |     +- windows
(9 rows)
```

co jest już dosyć bliskie graficznej reprezentacji drzewa.

to już jest koniec

na chwilę obecną nie przychodzi mi do głowy nic więcej co mógłbym napisać - jeśli macie jakieś pytania -

nie wahajcie się - piszcie na newsy - czytam codziennie i staram się odpowiadać w miarę moich umiejętności.

Przedruk za zgodą autora

Autor: Hubert Lubaczewski