Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki
Zakład Baz danych

Krzysztof Kosyl
nr albumu: 187411

Praca magisterska
na kierunku informatyka

Przechowywanie danych hierarchicznych w relacyjnych bazach danych

Opiekun pracy dyplomowej
dr hab. Krzysztof Stencel, prof. UMK
Wydział Matematyki i Informatyki

Toruń 2010

Pracę przyjmuję i akceptuję

.....
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....
data i podpis pracownika dziekanatu

Spis treści

1 Wstęp **2**
1.1 Cel pracy 2

2 Wprowadzenie do tematu **4**
2.1 Podstawowe definicje 4
2.2 Przyjęte założenia 5
2.3 Metodyka testów 6

3 Podstawowe reprezentacje **8**
3.1 Metoda krawędziowa 9
3.2 Metoda zagnieżdżonych zbiorów 13
3.3 Metoda pełnych ścieżek 19
3.4 Metoda zmaterializowanych ścieżek 24

4 Konstrukcje języka **28**
4.1 Wspólne Wyrażenia Tabelowe 29
4.2 `CONNECT BY` 32

5 Typy danych **34**
5.1 PostgreSQL `ltree` 35
5.2 Microsoft SQL Server `hierarchyid` 38

6 Podsumowanie **42**

Bibliografia **44**

Rozdział 1

Wstęp

Hierarchie są bardzo użyteczną metodą reprezentacji danych. Zawdzięczają to głównie temu, że są bardzo intuicyjne, naturalne dla ludzi. Nic dziwnego, że z tej abstrakcji korzystają mapy myśli, systemy plików, kategorie produktów w sklepach czy katalogi książek w bibliotece. Używane są również do reprezentacji struktur organizacyjnych, przykładowo oddziałów firm czy wydziałów na uniwersytetach. Nawet układy prac naukowych są hierarchiczne.

Strukturą danych najlepiej nadającą się do przechowywania danych hierarchicznych są drzewa.

Każda współczesna baza danych korzysta z drzew. Indeksy są implementowane jako *B-drzewa*, *R-drzewa*, itd. Zapytania SQL są przetwarzane do *drzew składniowych* (ang. *AST — Abstract Syntax Tree*). Następnie są przetwarzane do postaci planu zapytania, który również jest drzewem. Lecz to są drzewa zastosowane w implementacji, mające na celu zwiększenie szybkości działania bazy danych i nie są bezpośrednio widoczne dla użytkownika bazy danych. Z tego powodu nie będą tu omawiane.

Warto wspomnieć, że istnieją modele danych wspierające przechowywanie danych hierarchicznych. Oczywistym przykładem jest *hierarchiczny model danych*. Korzysta on z dwóch struktur danych: rekordów oraz związków nadrzędny-podrzędny. Najważniejszym jego reprezentantem jest *IMS* (ang. *Information Management System*) firmy IBM. Innymi bazami danych przechowującymi dane hierarchiczne są LDAP (ang. *Lightweight Directory Access Protocol*) oraz rejestr systemu operacyjnego Windows.

Alternatywnym podejściem do przechowywania danych hierarchicznych jest umieszczenie ich w wnętrzu pojedynczego rekordu. Można do tego zastosować różne formaty danych. Obecnie najpopularniejszym używanym w tym celu jest XML. Korzystają z niego zarówno *natywne bazy danych XML* (ang. *NXD — Native XML databases*) jak i relacyjne bazy danych z stosownymi rozszerzeniami. Innym formatem jest JSON stosowany w obecnie zdobywających popularność bazach *NoSQL*, takich jak *CouchDB* czy *MongoDB*.

Dominujące obecnie relacyjne bazy danych nie zostały stworzone z myślą o obsłudze danych hierarchicznych. Nie oznacza to bynajmniej, że jest to niemożliwe. Już Edgar Frank Codd zaproponował jedno z rozwiązań. Miało być ono obroną relacyjnego modelu danych przed zarzutami, że w takich bazach nie można przechowywać danych hierarchicznych.

1.1 Cel pracy

Celem tej pracy jest:

- przedstawienie popularnych lub godnych zainteresowania metod przechowywania danych hierarchicznych
- porównanie tych metod pod względem łatwości użycia, dostępności, możliwości

- sprawdzenie ich wydajności w najpopularniejszych bazach danych
- ocena metod, wskazanie ich mocnych i słabych stron
- dostarczenie programu umożliwiającego własnoręczne sprawdzenie wymienionych metod oraz ułatwiającego sprawdzenie własnych

Zagadnienie przechowywania danych hierarchicznych w relacyjnych bazach danych nie jest tematem nowym. Jest często poruszany w Internecie, wiele książek omawiających bazy danych zawiera rozdziały jemu poświęcone. Joe Celko napisał całą książkę poruszającą to zagadnienie[Cel04]. Mogło by się więc wydawać, że temat jest dobrze opracowany.

Więc dlaczego powstała ta praca? Otóż dostępne materiały często zawierają opisy tylko kilku wybranych metod (zwykle nie więcej niż czterech). Ponadto nie zawierają testów wydajności a jeśli już je podają to tylko w jednej bazie. Celem tej pracy jest uzupełnienie tych braków w sposób spójny i jednolity.

Rozdział 2

Wprowadzenie do tematu

2.1 Podstawowe definicje

Drzewa można traktować i definiować na wiele równoważnych sposobów. Przykładowo w teorii grafów jest to acykliczny i spójny graf. W informatyce częściej stosuje się definicję rekurencyjną. Wynika to z częstego w informatyce wymogu by drzewo było uporządkowane. Taki twór formalnie nie jest grafem.

Poniżej zostaną przedstawione najważniejsze, używane w tej pracy definicje (na podstawie [Knu02]).

Drzewo (ang. *tree*) definiujemy jako zbiór T zawierający jeden lub więcej elementów $t \in T$ zwanych **węzłami** (ang. *node*), takich że:

1. istnieje jeden wyróżniony węzeł zwany **korzeniem** drzewa, $root(T)$
2. pozostałe węzły (z wyłączeniem korzenia) są podzielone na $m \geqslant 0$ rozłącznych zbiorów $T_1, \ldots, T_m$, będących drzewami. Są one nazywane **poddrzewami** (ang. *subtree*) korzenia.

Każdy węzeł drzewa jest korzeniem pewnego poddrzewa zawartego w większym drzewie. Ilość poddrzew węzła jest nazywana **stopniem** (ang. *degree*) tego węzła. Węzeł o stopniu zero nazywamy **liściem** (ang. *leaf*) lub **węzłem zewnętrznym** (ang. *terminal node*). Węzeł nie będący liściem nazywamy **węzłem wewnętrznym** (ang. *branch node*).

Przodkami (ang. *descendant*) węzła nazywamy korzenie wszystkich drzew, w których się zawiera. **Potomkami** (ang. *ancestor*) węzła nazywamy wszystkie węzły należące do poddrzewa, którego jest korzeniem. Wynika z tego, że każdy węzeł jest jednocześnie swym przodkiem i potomkiem. Jeśli węzeł x jest przodkiem y oraz $x \neq y$ to x jest **przodkiem właściwym**. Analogiczna, jeśli węzeł x jest potomkiem y oraz $x \neq y$ to x jest **potomkiem właściwym**.

Korzenie poddrzew $T_1, \ldots, T_m$ nawywamy **dziećmi** (ang. *children*) węzła. Węzeł jest **rodzicem** (ang. *parent*) węzłów jeśli są one jego dziećmi.

Poziom (ang. *level*) węzła jest zdefiniowany rekurencyjnie: poziom korzenia $root(T)$ równa się zero, a poziom każdego innego węzła jest o jeden większy niż poziom korzenia w najmniejszym zawierającym go poddrzewie. Maksymalny poziom wszystkich węzłów nazywa się **wysokością** (ang. *height*) drzewa.

Jeśli względny porządek poddrzew $T_1, \ldots, T_m$ w części (2) definicji jest istotny, to mówimy, że drzewo jest **uporządkowane** (ang. *ordered tree*). Jeśli nie chcemy rozróżniać drzew, które różnią się jedynie kolejnością poddrzew, mówimy o drzewach **zorientowanych**.

Lasem (ang. *forest*) nazywamy zbiór zera lub więcej rozłącznych drzew. Abstrakcyjnie drzewa i lasy różnią się nieznacznie. Usuwając korzeń drzewa, otrzymamy las. Jeśli zaś

wszystkie drzewa w lesie uczynimy poddrzewami jednego dodatkowego węzła, to otrzymamy drzewo. Z tego powodu przy nieformalnym omawianiu struktur danych słowa „drzewa" i „lasy" są używane niemal wymiennie.

2.2 Przyjęte założenia

Terminologia angielska Literatura dotycząca informatyki jest zdominowana przez publikacje w języku angielskim. Aby ułatwić czytelnikowi korzystanie z tych źródeł wiedzy przy każdym ważnym lub nietrywialnym w tłumaczeniu terminie została podana jego angielska wersja.

Drzewa uporządkowane Nie wymagane jest by metoda odpowiadała za przechowywanie porządku poddrzew. Jeśli zaistnieje taki wymóg to można dodać kolumnę przeznaczoną do przechowywania wartości odpowiedzialnej za relację porządku. Czasem istnieje też możliwość sortowania po istniejących już kolumnach, przykładowo nazwie kategorii.

Tabela zawiera tylko jedną kolumnę z danymi użytkownika Bez problemu można dodać więcej kolumn do tabeli. Obecność wyłącznie kolumny `name` zwiększa czytelność przykładów.

Przyjęty interface Aby móc porównywać różne metody przechowywania danych hierarchicznych należy stworzyć każdej te same warunki. Dla tego został ustalony poniższy interface.

- `create_table()` Tworzy w bazie danych wymagane struktury. Zwykle sprowadza się to do wykonania sekwencji zapytań SQL z podzbioru *definicji danych* (ang. *DDL — Data Definition Language*). Każda tabela kolumnę identyfikatora o nazwie `id`. Jest to numeryczny klucz główny, którego wartości są automatycznie pobierane z sekwencji. Jest ona bardzo ważny gdyż wszystkie metody interface odwołują się do węzłów za jego pomocą
- `insert(parent, name)` Dodaje nowy węzeł o wartości `name` jako potomka węzła o identyfikatorze `parent`. Zwraca identyfikator nowo wstawionego rekordu.
- `get_roots()` Zwraca rekordy *korzeni* wszystkich drzew w lesie.
- `get_parent(id)` Zwraca rekord *rodzica* węzła, którego identyfikator został podany jako parametr.
- `get_ancestors(id)` Zwraca listę rekordów *przodków* węzła, którego identyfikator został podany jako parametr. Lista jest posortowana malejąco wedle poziomu rekordu, czyli korzeń jest jej ostatnim elementem.
- `get_children(id)` Zwraca listę rekordów *dzieci* węzła, którego identyfikator został podany jako parametr.
- `get_descendants(id)` Zwraca listę rekordów *właściwych potomków* węzła, którego identyfikator został podany jako parametr. Metoda nie wymaga konkretnej kolejności — jest ona zależna od zastosowanego algorytmu. W praktyce najczęściej jest to kolejność specyficzna dla przeszukiwania wszerz

Jak widać są to operacje wyłącznie związane z hierarchiczną strukturą danych.

Przedstawione fragmenty kodu W opisach metod znajdują się fragmenty kodu SQL implementujące operacje na nich. Tam gdzie to jest możliwe (czyli metoda jest dostępna) została zaprezentowana składnia PostgreSQL. Jeśli operacja potrzebuje danych wejściowych zostają one podane jako nazwa parametru poprzedzona dwukropkiem — przykładowo `:id`.

Część operacji nie daje się wykonać wyłącznie za pomocą jednego (lub więcej) zapytań SQL. W takiej sytuacji fragmenty kodu zawierają kod języka *Python*. Został on wybrany do tego zadania ze względu na jego ekspresyjność, zwięzłość oraz wysoki poziom abstrakcji. Z powodu tych cech jest często nazywany *wykonywalnym pseudokodem*.

Aby polepszyć zwięzłość fragmenty kodu korzystają z biblioteki PADA, stworzonej aby uprościć pisanie implementacji opisanych metod. Została ona dołączona do tej pracy.

2.3 Metodyka testów

Ważnym elementem tej pracy jest sprawdzenie wydajności przedstawionych metod w bazach danych. W tym rozdziale zostaną przedstawione warunki w jakich odbywały się testy.

Wybrane SZDB

Do testów zostały wybrane popularne, dostępne bezpłatnie bazy danych. Dla baz komercyjnych zostały wybrane ich darmowe edycje. Posiadają one ograniczenia co do wielkości danych, wykorzystania zasobów oraz zmniejszoną funkcjonalność. Specyfika testów sprawia jednak, że te ograniczenia nie miały podczas nich znaczenia.

Tabela 2.1: Użyte wersje baz danych

Baza danych	Wersja	Edycja
PostgreSQL	8.4	
MySQL	5.1	
SQLite	3	
Oracle Database	10g	Express Edition
IBM DB2	9.7	Express-C
Microsoft SQL Server	2008 R2	Express

Dane testowe

By móc porównać metody przechowywania danych hierarchicznych należy zapewnić jednorodne warunki testów. Oznacza to konieczność wykonywania dokładnie tych samych operacji, z takimi samymi parametrami, w tej samej kolejności. Aby to osiągnąć testy są przechowywane wewnątrz plików XML (w podkatalogu `src/data`).

Dane użyte podczas zaprezentowanych testów zostały wygenerowane w sposób automatyczny. Opisują one las o wysokości 10, zawierający dwa drzewa, z każdym węzłem wewnętrznym o stopniu 2. W efekcie całe drzewo ma 4094 węzły.

Środowisko testowe

Bazy danych były testowane na systemie operacyjnym *Microsoft Windows 7 32bit*. Wybór ten był podyktowany faktem, że jedna z testowanych baz (SQL Server) posiadała wersję wyłącznie na systemy z rodziny Windows.

Sprzęt, na którym wykonywane były testy, składał się z:

- procesora Intel Core2 Quad 2.4 Ghz
- 4 GB pamięci DDR2 (3,25 GB pamięci dostępnej dla systemu)
- dysku HDD 1TB klasy ekonomicznej

Serwer bazy danych działał na tej samej maszynie co program testujący. Zmniejsza to opóźnienia sieciowe, ale zwiększa zużycie procesora[1].

Optymalizacje

Aby nie faworyzować żadnej z testowanych baz ustanowione zostały poniższe zasady:

- Bazy nie były w żaden sposób dostrajane (ang. *database tuning*), czyli pracują z domyślnymi ustawieniami.
- Nie zastosowano indeksów[2] oraz kluczy obcych.
- Zapytania były pisane z myślą by być czytelne i dobrze ilustrować koncepcje danej reprezentacji. Została więc zastosowana maksyma Donalda Knutha: *„przedwczesna optymalizacja jest źródłem wszelkiego zła”*.

Prezentacja wyników

Wyniki wydajności są prezentowane na końcu opisu każdej z opisywanych metod. Są podawane zarówno w postaci tabeli (umożliwiającej dokładną analizę wyników) jak i wykresu słupkowego (dającego możliwość szybkiego przejrzenia wyników).

Zastosowaną miarą wydajności jest *przepustowość*, w tej pracy zastosowano zapytania na milisekundę. Takie podejście jest intuicyjne dla ludzi — większa wartość oznacza lepszy wynik.

[1] W praktyce na maszynie czterordzeniowej czas procesora nie był problemem.
[2] Jawnych indeksów — klucze główne były stosowane

Rozdział 3

Podstawowe reprezentacje

W tym rozdziale zostaną przedstawione podstawowe reprezentacje drzew. Reprezentacje — czyli sposoby przedstawienia struktury drzewa. Wiele reprezentacji zostało wymienione w [Knu02]. Tu zostaną opisane te najbardziej przydatne w bazach danych.

Tym co różni wymienione tu metody to sposób w jaki przetwarzają dane przed zapisaniem ich do bazy danych. Tylko pierwsza z wymienionych metod wstawia dane w najprostszej postaci. Pozostałe stosują (zwykle dosyć kosztowne) wstępne przetwarzanie, mające na celu przyśpieszenie późniejszych zapytań. Wstawianie danych jest zwykle o wiele rzadziej wykonywane niż ich pobieranie. W efekcie takie zachowanie jest opłacalne gdyż zmniejsza koszt średni operacji.

Metody wymienione w tym rozdziale są uniwersalne. Daje się je zaimplementować w każdej relacyjnej bazie danych. Można również (przy odrobinie wysiłku) korzystać z nich również w *odwzorowaniach obiektowo-relacyjnych*. Nic nie stoi na przeszkodzie aby wykorzystać te metody również w innych niż relacyjne bazach danych.

3.1 Metoda krawędziowa

Najprostszą, najbardziej intuicyjną i zapewne najpopularniejszą metodą jest metoda krawędziowa (ang. *adjacency*). Została ona pierwszy raz zaprezentowana przez Edgara Franka Codda.

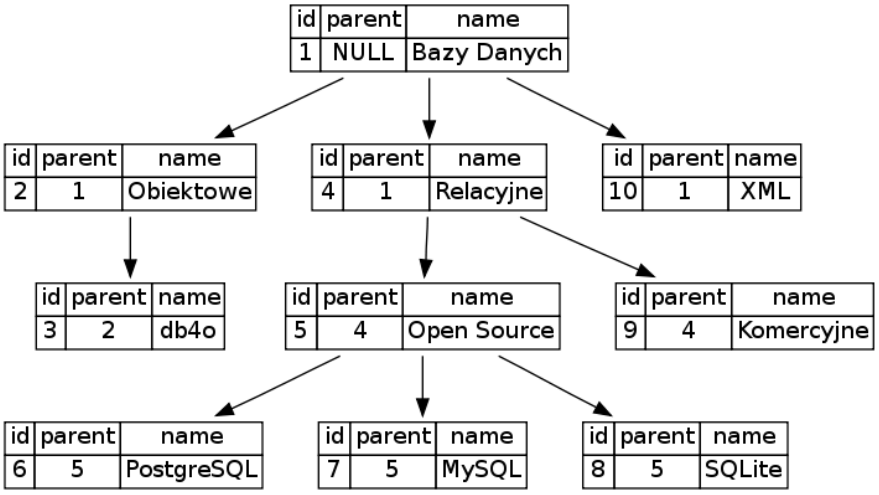
Zasługa spopularyzowania tej metody przypada Oracle. Dołączył on do swojego produktu przykładową bazę danych, nazywaną „Scott/Tiger" [1] korzystającą z tej metody.

Na popularność metody przekłada się również jej znaczące podobieństwo do używanego między innymi w językach C i C++ sposobu przechowywania list i drzew. Mianowicie każdy węzeł zawiera wskaźnik na rodzica.

```
1  typedef struct item {
2      struct item* parent;
3      char*        name;
4  } treeitem;
```

W relacyjnych bazach danych odpowiednikiem wskaźnika jest klucz obcy.

Ta metoda jest na tyle popularna, że w bazach danych pojawiły się specjalne konstrukcje do jej obsługi. Zostaną one przedstawione w rozdziale o konstrukcjach języka specyficznych dla systemu zarządzania bazą danych.



Reprezentacja w SQL

```
1  CREATE TABLE simple(
2    id     serial PRIMARY KEY,
3    parent int REFERENCES simple(id) ON DELETE CASCADE,
4    name   varchar(100)
5  )
```

Należy zwrócić uwagę na warunek `ON DELETE CASCADE`. Sprawia on, że w razie usunięcia węzła automatycznie zostaną usunięci wszyscy jego potomkowie.

Wstawianie danych

Ta metoda — jako jedyna w tym rozdziale — nie wymaga żadnego wstępnego przetwarzania danych. W efekcie wstawianie węzłów jest bardzo proste.

[1] Nazwa tej bazy danych pochodzi od metody autoryzacji w bazie Oracle (login/hasło). Login pochodził z nazwiska jednego z pierwszych pracowników Software Development Laboratories (przekształconych ostatecznie w Oracle) Bruca Scotta. Natomiast hasło to imię jego kota.

```
INSERT INTO simple (parent, name)
  VALUES (:parent, :name)
```

Pobranie korzeni

Cechą charakterystyczną korzenia jest to, że jego rodzic jest ustawiony na `NULL`. Więc pobranie rodziców sprowadza się do tego zapytania:

```
SELECT *
  FROM simple
  WHERE parent IS NULL
```

Pobranie rodzica

Identyfikator rodzica znajduje się w każdym węźle. Więc zapytanie musi pobrać najpierw rekord o podanym jako parametr identyfikatorze a następnie — korzystając z tego identyfikatora rodzica — wynikowy rekord. Użyte został SQL z podzapytaniem, lecz równie dobrze można by było zastosować złączenie.

```
SELECT *
  FROM simple
  WHERE id = (
    SELECT parent
      FROM simple
      WHERE id = :id
  )
```

Pobranie dzieci

Pobieranie dzieci jest bardzo prostą i szybką operacją — każdy rekord zawiera identyfikator rodzica.

```
SELECT *
  FROM simple
  WHERE parent = :parent
```

Pobranie przodków

Tu pojawia się po raz pierwszy największa wada tej reprezentacji — dla kilku operacji wymaga wykonania wielu zapytań w bazie danych.

W tym przypadku widać, że pobranie przodków sprowadza się do pobrania aktualnego węzła, a następnie w kolejnym zapytaniu jego rodzica, potem rodzica jego rodzica... aż dotrze się do korzenia drzewa.

```
result = []
while id is not None:
  row = self.db.execute('''
    SELECT id, parent, name
      FROM simple
      WHERE id = :id
    ''',
    dict(id=id)
  ).fetch_one()
  result.append(row)
```

```
    id = row.parent
  return result
```

Pobieranie potomków

Algorytm pobieranie potomków jest podobny do pobierania przodków — też pobiera się po jednym poziomie.

```
result = []
    ids = [id]

    while ids:
      rows = self.db.execute('''
        SELECT id, parent, name
          FROM simple
          WHERE (%s)''' % ' OR '.join(['parent = %d' % i for i in ids])
      ).fetch_all()
      result.extend(rows)
      ids = [row.id for row in rows]

    return result
```

Należy zwrócić uwagę na równoczesne pobieranie wszystkich węzłów na danym poziomie. Wybrany algorytm wymaga wykonania tylko tylu zapytań ile wynosi wysokość poddrzewa, którego korzeniem jest dany element.

Gdyby zastosować naiwny algorytm pobierania dzieci jednego węzła, a następnie, rekurencyjnie, dzieci jego dzieci to wymagał by on wykonania w sumie tylu zapytań ile węzłów ma poddrzewo, którego korzeniem jest dany element.

Uwagi

Czasem nie ma potrzeby pobierać wszystkich potomków a tylko tych różniących się poziomem o nie więcej niż N. Czyli dla $N = 1$ dzieci, dla $N = 2$, dzieci oraz wnuki, itd. Mając taką dodatkową wiedzę program można wygenerować bardziej optymalne zapytanie. Zasada działania się nie zmienia (dalej jest to przeszukiwanie wszerz) ale zamiast łączyć wyniki w wnętrzu programu są one łączone wewnątrz bazy danych za pomocą operatora `UNION ALL`.

Poniżej przykład dla $N = 3$.

```
    SELECT * FROM simple WHERE parent = :id
UNION ALL
    SELECT * FROM simple WHERE parent IN (
        SELECT id FROM simple WHERE parent = :id
    )
UNION ALL
    SELECT * FROM simple WHERE parent IN (
        SELECT id FROM simple WHERE parent IN (
            SELECT id FROM simple WHERE parent = :id
        )
    )
```

Podejście to może być bardzo przydatne w drzewach o ustalonej, małej wysokości. W takiej sytuacji może ono zastąpić ogólny algorytm.

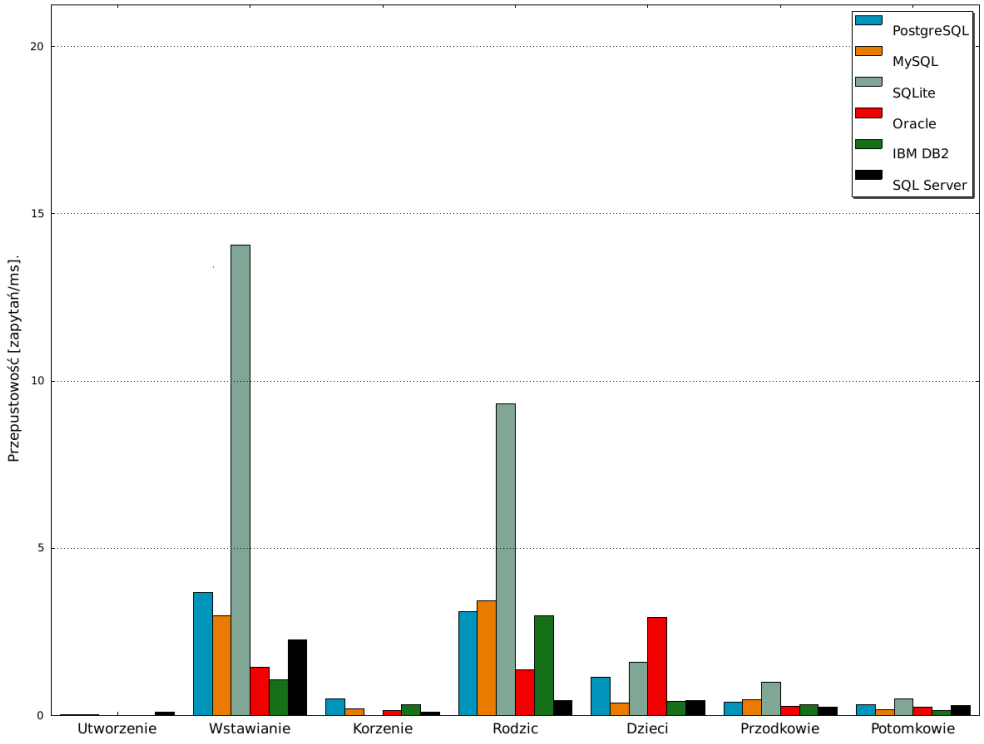
W analogiczny sposób można pobierać przodków.

Wydajność

Tabela 3.1: Wydajność reprezentacji krawędziowej

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
PostgreSQL	0,018	3,675	0,500	3,106	1,133	0,393	0,311
MySQL	0,016	2,982	0,200	3,417	0,367	0,483	0,162
SQLite	0,007	14,069	0,000	9,318	1,579	0,988	0,509
Oracle	0,008	1,430	0,143	1,367	2,942	0,273	0,240
IBM DB2	0,001	1,072	0,333	2,971	0,427	0,328	0,156
SQL Server	0,100	2,267	0,111	0,452	0,441	0,251	0,286

Rysunek 3.1: Wydajność reprezentacji krawędziowej



3.2 Metoda zagnieżdżonych zbiorów

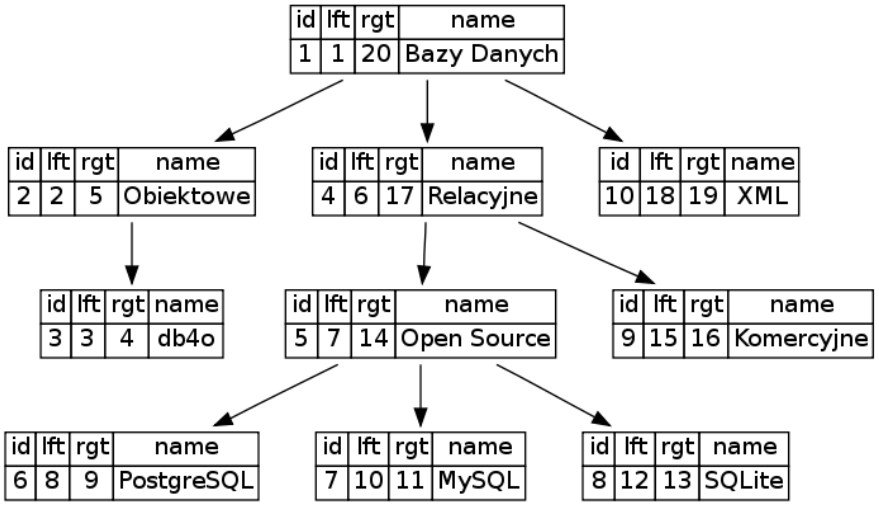
Reprezentacja *zagnieżdżonych zbiorów* (ang. *nested sets*) została spopularyzowana przez Joe Celko[Cel00]. Dlatego czasem bywa nazywana „metodą Celko". Który, jak sam przyznaje, oparł się na opisie Donalda Knutha[Knu02].

W tej metodzie każdy węzeł drzewa traktowany jest jako zbiór. Muszą one spełniać wymóg by **każde dwa zbiory były rozłączne albo jeden był podzbiorem drugiego**. W efekcie jeśli zbiór nie ma podzbiorów to jest liściem. Natomiast jeśli ma podzbiory to staje się ich przodkiem. Jeśli dodatkowo jest najmniejszym z nadzbiorów danego węzła to jest jego rodzicem. W efekcie każdy zbiór jest *„zagnieżdżony"* w swoim rodzicu[2]. Stąd pochodzi nazwa tej reprezentacji.

W bazie danych najprostszą metodą implementacji tej metody jest myślenie o zbiorach jako o odcinkach na prostej[3]. Odcinek taki jest opisywany przez parę liczb. Jedna z nich określa lewy a druga prawy koniec odcinka. Jeśli dodatkowo przyjmiemy, że zbiór tych wartości liczbowych końców odcinków jest zbiorem kolejnych liczb naturalnych to zyskujemy możliwość łatwego wykonywania wielu operacji.

Ponieważ `left` oraz `right` są słowami kluczowymi SQL-92, w kodzie zostaną zastosowane ich skróty, odpowiednio `lft` oraz `rgt`.

Reprezentacja pozwala na przechowywanie drzew uporządkowanych.



Operacje

Reprezentacja w SQL

```
1  CREATE TABLE nested_sets(
2    id   serial PRIMARY KEY,
3    lft  int,
4    rgt  int,
5    name varchar(100)
6  )
```

[2] Za przykład z realnego świata może posłużyć metoda pakowania przedmiotów w celu przewozu. W kontenerach znajdują się palety, na których znajdują się kartony, wewnątrz których znajdują się opakowania indywidualne produktów. Innym przykładem są matrioszki czy jednostki podziału administracyjnego (województwa, gminy, miejscowości, ...).

[3] W tym celu można by było użyć również rozszerzeń przestrzennych, przykładowo *PostGIS* lub *Oracle Spatial*

Wstawianie danych

```
if parent is None:
      right = self.db.execute('''
        SELECT max(rgt) AS max_rgt
          FROM nested_sets
        '''
      ).fetch_single('max_rgt')
      right = right or 0
      pid = self.db.insert_returning_id(
        'nested_sets',
        dict(
          lft=(right + 1),
          rgt=(right + 2),
          name=name
        )
      )
    else:
      right = self.db.execute('''
        SELECT rgt
          FROM nested_sets
          WHERE id = :parent
        ''',
        dict(parent=parent)
      ).fetch_single('rgt')
      self.db.execute('''
        UPDATE nested_sets
          SET lft = lft + 2
          WHERE lft >  :val
        ''',
        dict(val=right)
      )
      self.db.execute('''
        UPDATE nested_sets
          SET rgt = rgt + 2
          WHERE rgt >= :val
        ''',
        dict(val=right)
      )
      pid = self.db.insert_returning_id(
        'nested_sets',
        dict(
          lft=(right),
          rgt=(right + 1),
          name=name
        )
      )

    return pid
```

Wstawianie danych w tej metodzie jest jej najsłabszą stroną. Wymaga ono zmiany wartości `lft` i `rgt` wielu rekordów.

W powyższym przypadku — jako że wstawiamy dane w kolejności przeszukiwania w głąb — wymagana jest modyfikacja tylko tylu rekordów na jakim poziomie aktualnie wstawiamy nowy węzeł. Lecz należy się liczyć z przypadkiem pesymistycznym — dodaniem nowego drzewa jako pierwszego elementu lasu. W takiej sytuacji wymagane jest zmodyfikowanie wszystkich istniejących rekordów.

Uwaga: Jeśli chce się jednorazowo załadować całe drzewo do bazy danych można zrobić to bardziej wydajnie. (powyższy kod tego nie robi gdyż ma być ogólny).

Mianowicie przed załadowaniem danych do bazy danych wstępnie przetworzyć dane. W takiej sytuacji należy się posłużyć algorytmem *przeszukiwania w głąb* i kolejno numerować wartości `lft` węzła po wejściu do niego oraz `rgt` przed jego opuszczeniem. Przykładowa implementacja tego algorytmu:

```
1  def preprocess(node):
2    node.lft = get_next_int()
3    for n in children(node):
4      preprocess(n)
5    node.rgt = get_next_int()
```

Pobranie węzłów

Wszystkie operacje pobierające węzły zbudowane na wspólnych koncepcjach. Warto wymienić pośród nich kilka:

- Węzeł, którego potomków chcemy poznać znamy tylko z identyfikatora, a potrzebne są jego końce. Problem rozwiązuje złączenie `JOIN nested_sets AS box` oraz warunek `box.id = :id` umieszczony w klauzuli `WHERE`.
- Mając już końce przedziału można pobierać jego podzbiory, przyładowo stosując warunek `box.lft < result.lft AND result.rgt < box.rgt`. Poniewarz w tej reprezentacji dwa zbiory są rozłączne albo jeden jest podzbiorem drugiego to można uprościć ten warunek do postaci `result.lft BETWEEN box.lft AND box.rgt`.
- Chcąc pobrać potomków właściwych należy zmniejszyć przedział. Czyli przykładowo zamienić `box.lft` na `box.lft + 1`.

Analogiczna sytuacja zachodzi dla przodków (czyli nadzbiorów)

Pobranie korzeni

W tej reprezentacji korzeń to węzeł, który nie zawiera się w żadnym innym. Zapytanie próbuje pobrać rodzica danego rekordu, a dzięki złączeniu zewnętrznemu tam gdzie on nie istnieje pojawi się wartość `NULL`.

```
1  SELECT result.*
2    FROM nested_sets result
3      LEFT OUTER JOIN nested_sets box
4        ON (box.lft < result.lft AND result.rgt < box.rgt)
5    WHERE
6      box.lft IS NULL
```

Pobranie rodzica

```
SELECT result.*
  FROM nested_sets box
    JOIN nested_sets result
      ON (box.lft BETWEEN result.lft + 1  AND result.rgt)
  WHERE
    box.id = :id
  ORDER BY result.lft DESC
  LIMIT 1
```

To zapytanie różni się od pobrania przodków w dwóch miejscach. Po pierwsze `result.lft + 1` w warunku złączenia powoduje, że spełniają go wyłącznie przodkowie właściwi. Natomiast klauzula `LIMIT 1` sprawia, że zapytanie pobiera tylko jeden rekord.

Pobranie dzieci

Pobieranie dzieci to de facto pobieranie potomków z dodatkowym warunkiem sprawdzającym czy potomek nie jest „wnukiem".

```
SELECT d.id, d.name, d.rgt as rgt
  FROM nested_sets d
  WHERE d.lft = :left;

SELECT result.*
  FROM nested_sets box
    JOIN nested_sets result
      ON (result.lft BETWEEN box.lft + 1 AND box.rgt)
  WHERE
    box.id = :id AND
    NOT EXISTS (
      SELECT *
        FROM nested_sets ns
        WHERE
          (ns.lft BETWEEN box.lft + 1 AND box.rgt) AND
          (result.lft BETWEEN ns.lft + 1 AND ns.rgt)
    )
```

Pobranie przodków

Przodkowie to wszystkie węzły zawierające (w całości) dany węzeł. Oznacza to, że wartość z węzła będącego parametrem znajduje się pomiędzy ich lewym i prawym końcem.

```
SELECT result.*
  FROM nested_sets box
    JOIN nested_sets result
      ON (box.lft BETWEEN result.lft AND result.rgt)
  WHERE
    box.id = :id
  ORDER BY result.lft DESC
```

Aby uzyskać wymaganą kolejność można by zastosować sortowanie `ORDER BY (result.rgt - result.lft) ASC`. Jednak można zauważyć, że lewy koniec przodka ma mniejszą wartość liczbową niż u potomka. Prowadzi to do prostszego `ORDER BY result.lft DESC`.

Pobieranie potomków

Pobieranie potomków jest tą operacją, dla której stosuje się tą reprezentację. Metoda pozwala na bardzo proste i szybkie wykonywanie tej operacji.

```
1 SELECT result.*
2   FROM nested_sets box
3     JOIN nested_sets result
4       ON (result.lft BETWEEN box.lft + 1 AND box.rgt)
5   WHERE
6     box.id = :id
7   ORDER BY result.lft ASC
```

Dzięki sortowaniu `ORDER BY result.lft ASC` otrzymuje się potomków w kolejności przeszukiwania w głąb.

Uwagi

Metoda umożliwia bardzo proste pobieranie liści drzewa. Jest ono możliwe dzięki temu, że $node.rgt - node.lft = 1$[4] tylko i wyłącznie dla liści.

```
1 SELECT *
2   FROM nested_sets
3   WHERE lft + 1 = rgt
```

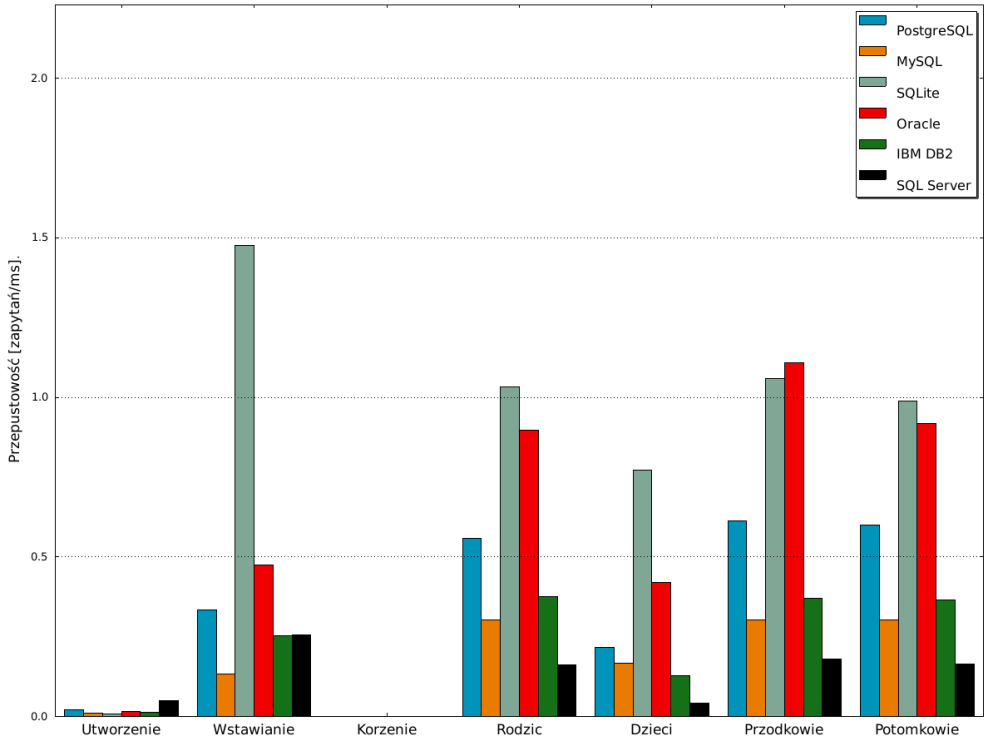
Wydajność

Tabela 3.2: Wydajność reprezentacji zagnieżdżonych zbiorów

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
PostgreSQL	0,021	0,333	0,001	0,559	0,216	0,613	0,600
MySQL	0,011	0,132	0,000	0,303	0,167	0,302	0,302
SQLite	0,008	1,477	0,000	1,033	0,772	1,060	0,988
Oracle	0,016	0,474	0,001	0,897	0,420	1,108	0,919
IBM DB2	0,013	0,252	0,000	0,375	0,127	0,370	0,365
SQL Server	0,050	0,255	0,000	0,162	0,041	0,179	0,164

[4] Warto zauważyć, że warunek podany jako `node.rgt - node.lft = 1` mimo, że jest matematycznie równoważny z `node.lft + 1 = node.rgt` może być mniej wydajny. Wynika to z tego, że drugim przypadku optymalizator może wykorzystać indeks.

Rysunek 3.2: Wydajność reprezentacji zagnieżdżonych zbiorów



3.3 Metoda pełnych ścieżek

Ta reprezentacja jest najmniej rozpowszechniona spośród zaprezentowanych w tym rozdziale. Przy czym w Polsce jest bardziej popularna niż poza granicami kraju. Jest to zasługa *Huberta Lubaczewskiego* lepiej znanego pod pseudonimem *depesz*. Stąd w polskim internecie często można spotkać się z tą metodą pod nazwą *metoda depesza*. On sam nazywa tę reprezentację *metodą pełnych ścieżek*.

Idea metody jest prosta. Głowna tabela z danymi nie zawiera żadnych informacji o hierarchii danych. Jedynym wymogiem jest istnienie w niej klucza głównego.

Cała informacja potrzebna do operowania na drzewie zawiera się w dodatkowej tabeli. Zawiera ona informacje o odległości pomiędzy każdym elementem a wszystkimi jego przodkami.

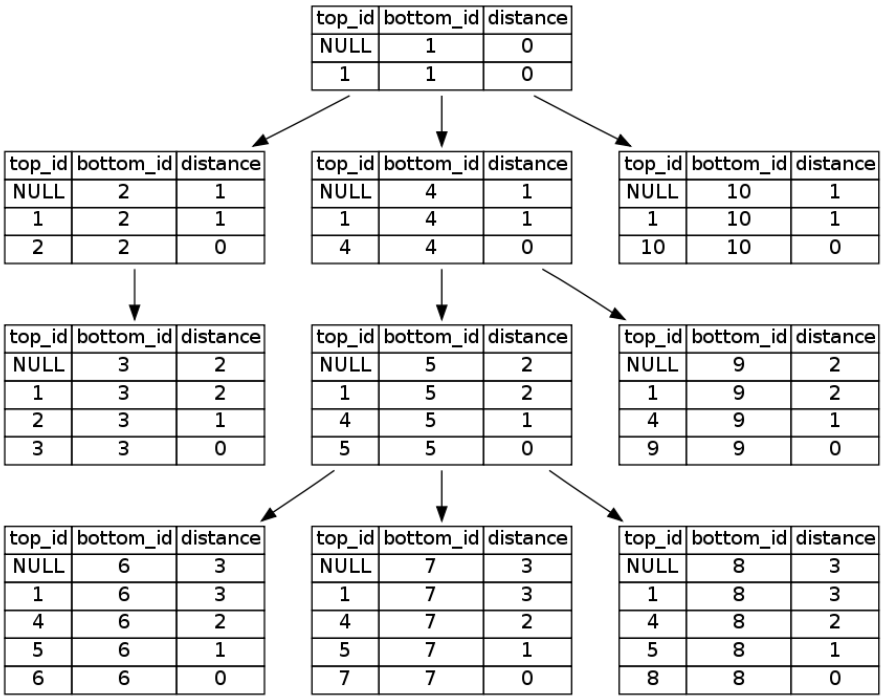
Przedstawiona tutaj implementacja została lekko zmieniona względem oryginalnej „metody depesza". Mianowicie zawiera dodatkowo jeden specjalny rekord dla każdego węzła. Przechowuje on odległość pomiędzy węzłem a korzeniem. Posiada on atrybut `top_id` ustawiony na `NULL`.

Ilość wierszy drugiej tabeli wynosi:

$$\sum_{t \in T} level(t) + 2 \leqslant |T|(height(T) + 2)$$

Jak widać, metoda wymaga znacząca ilości rekordów (które trzeba wstawić i przechowywać). Jednak nie jest to duży problem gdyż pojedyńczy rekord zawiera wyłącznie trzy liczby całkowite, więc rozmiar rekordu jest mały.

Poniższy graf pokazuje wyłącznie zawartość dodatkowej tabeli zawierającej strukturę drzewa.



Operacje

Reprezentacja w SQL

```
1 CREATE TABLE full_data(
2   id   serial PRIMARY KEY,
3   name varchar(100)
```

```
);

CREATE TABLE full_tree(
  top_id    int,
  bottom_id int,
  distance  int
)
```

Wstawianie danych

W bazach dających możliwość wstawiania danych wygenerowanych przez podzapytanie ta operacja jest bardzo łatwa do wykonania.

```
INSERT INTO full_tree(top_id, bottom_id, distance)
    SELECT top_id, :pid, distance + 1
      FROM full_tree
      WHERE bottom_id = :parent
  UNION ALL
    SELECT NULL, :pid, 0
      WHERE :parent IS NULL
  UNION ALL
    SELECT :pid, :pid, 0
```

Jak widać podzapytanie składa się z 3 części, których wyniki są łączone za pomocą operatora `UNION ALL`. Pierwszy fragment przetwarza rekordy odpowiadające za położenie rodzica w drzewie. Drugi odpowiada za sytuację gdy węzeł jest korzeniem i trzeba wstawić specjalny rekord z `top_id = NULL`. Natomiast trzeci odpowiada za rekord z `distance = 0`.

Pobranie węzłów

W tej metodzie wszystkie analizowane operacje są do siebie bardzo podobne. By nie omawiać ich wielokrotnie zostanie tu zaprezentowany ogólny mechanizm.

```
SELECT d.*
    FROM full_data d
        JOIN full_tree t
            ON (d.id = $JOIN-ATTR$)
    WHERE
        $WHERE-ATTR$ = :id AND
        t.distance $DISTANCE$
```

Jak widać w powyższym kodze pojawiły się „bloki” (zapisywane `$NAZWA-BLOKU$`). Należy je zastąpić odpowiednimi fragmentami SQL.

Bloki `$JOIN-ATTR$` i `$WHERE-ATTR$` są od siebie zależne. Gdy jeden z nich jest postaci `t.top_id` drugi musi być `t.bottom_id`. Jeśli `$JOIN-ATTR$` ma wartość `t.bottom_id` to zapytanie będzie pobierało potomków (węzły „pod”). Analogicznie przodków w przeciwnej sytuacji.

Natomiast blok `$DISTANCE$` pozwala sprecyzować z których poziomów zostaną pobrane węzły. I tak dla `>= 0` zostaną pobrani przodkowie/potomkowie[5], dla `> 0` zostaną pobrani przodkowie/potomkowie właściwi. Natomiast `= 1` spowoduje pobranie rodzica/potomków.

[5] W tej sytuacji można by było pominąć ten warunek, gdyż wszystkie wartości atrybutu `distance` są większe lub równe zero

Oczywiście można zastosować dowolny warunek lub ich kombinację, co stanowi siłę tej reprezentacji.

Pobranie korzeni

```
SELECT d.*
  FROM full_data d
    JOIN full_tree t
      ON (d.id = t.bottom_id)
  WHERE
    t.top_id IS NULL AND
    t.distance = 0
```

Pobranie rodzica

```
SELECT d.*
  FROM full_data d
    JOIN full_tree t
      ON (d.id = t.top_id)
  WHERE
    t.bottom_id = :id AND
    t.distance = 1
```

Pobranie dzieci

```
SELECT d.*
  FROM full_data d
    JOIN full_tree t
      ON (d.id = t.bottom_id)
  WHERE
    t.top_id = :id AND
    t.distance = 1
```

Pobranie przodków

```
SELECT d.*
  FROM full_data d
    JOIN full_tree t
      ON (d.id = t.top_id)
  WHERE
    t.bottom_id = :id AND
    t.distance >= 0
  ORDER BY t.distance ASC
```

Pobieranie potomków

```
SELECT
  d.*
  FROM full_data d
    JOIN full_tree t
      ON (d.id = t.bottom_id)
  WHERE
    t.top_id = :id AND
```

```
    t.distance > 0;

SELECT
  d.*,
  (SELECT tmp.top_id
    FROM full_tree AS tmp
      WHERE
        tmp.bottom_id = d.id AND
        tmp.distance = 1 AND
        tmp.top_id IS NOT NULL
  ) AS parent
  FROM full_data d
    JOIN full_tree t
      ON (d.id = t.bottom_id)
  WHERE
    t.top_id = :id AND
    t.distance > 0
```

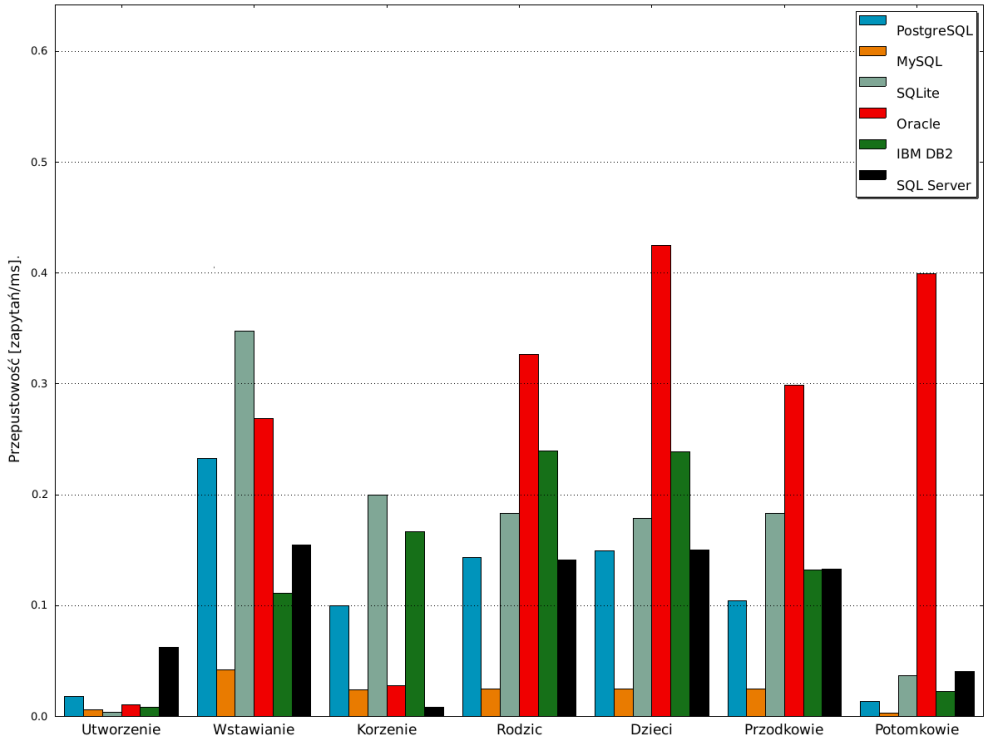
Jak widać w klauzuli `SELECT` pojawiło się podzapytanie zwracające identyfikator rodzica bieżącego węzła. Jeśli operacja ma pobierać tylko zbiór potomków to można je pominąć. Natomiast jeśli wynik zapytania ma zostać przekształcony w drzewo to jest ono konieczne. Wynika to z faktu, że sama tabela `full_tree` nie zawiera informacji o strukturze hierarchicznej.

Wydajność

Tabela 3.3: Wydajność reprezentacji pełnych ścieżek

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
PostgreSQL	0,018	0,233	0,100	0,143	0,150	0,105	0,014
MySQL	0,006	0,042	0,024	0,025	0,025	0,025	0,003
SQLite	0,003	0,348	0,200	0,183	0,179	0,183	0,037
Oracle	0,010	0,269	0,028	0,327	0,425	0,299	0,400
IBM DB2	0,008	0,111	0,167	0,239	0,239	0,132	0,023
SQL Server	0,062	0,155	0,008	0,141	0,150	0,133	0,041

Rysunek 3.3: Wydajność reprezentacji pełnych ścieżek



3.4 Metoda zmaterializowanych ścieżek

Koncepcja stojąca za tą reprezentacją jest bardzo stara i powszechnie znana. Przykładowo książka w bibliotece może zostać zaklasyfikowana jako *Epika / Powieść / Powieść fantastyczna*. Biolodzy dla opisu różnorodności życia na Ziemi stosują drzewo filogenetyczne. Za pomocą niego organizm w nim może zostać opisany jako *eukarionty / zwierzęta / strunowce / kręgowce / ssaki / ssaki żyworodne / łożyskowce / drapieżne / kotowate / felis / kot domowy*. Stąd też wywodzi się popularna angielska nazwa tej reprezentacji — *lineage* — oznaczająca rodowód, pochodzenie.

Ta reprezentacja może wyglądać podobnie do ścieżek dostępu w systemach plików. Jest to złudne podobieństwo. W tej metodzie z wartości każdej części ścieżki można wywnioskować wszystkich jej przodków. Przykładowo, mając do czynienia z kotem domowym wiemy, że jest ssakiem. Natomiast nazwa katalogu `bin` nie daje nam informacji o tym czy jesteśmy w katalogu `/bin`, `/usr/bin` czy `/home/user/bin`.

Metoda ta polega na przechowywaniu ciągu identyfikatorów wszystkich węzłów pomiędzy korzeniem a danym węzłem. Są one przechowywane w pojedynczym atrybucie danego rekordu. Czyli de facto przechowuje się wynik zapytania o przodków czyli ścieżkę. Stąd pochodzi najpopularniejsza jej nazwa: *metoda zmaterializowanych ścieżek* (ang. *materialized path*)[6].

Identyfikatorem może być dowolny, unikalny atrybut tabeli. Dobrym rozwiązaniem jest numeryczny klucz główny, lecz często bywa też stosowana unikalna nazwa węzła.

Ten opis nie wymusza konkretnej implementacji. Listę elementów można przechowywać na wiele sposobów. Jedynym wymogiem jest to by można było dokonać prostego pobrania identyfikatorów poszczególnych węzłów. Przykładowo ciąg identyfikatorów całkowitoliczbowych `4, 8, 15, 16, 23, 42` można przechować jako:

- napis z elementami rozdzielonymi separatorami. Przykładowo dla separatora `'.'` będzie to napis `'4.8.15.16.23.42'`. Ważne jest aby znak lub ciąg znaków będący separatorem nie mogły występować w identyfikatorze.

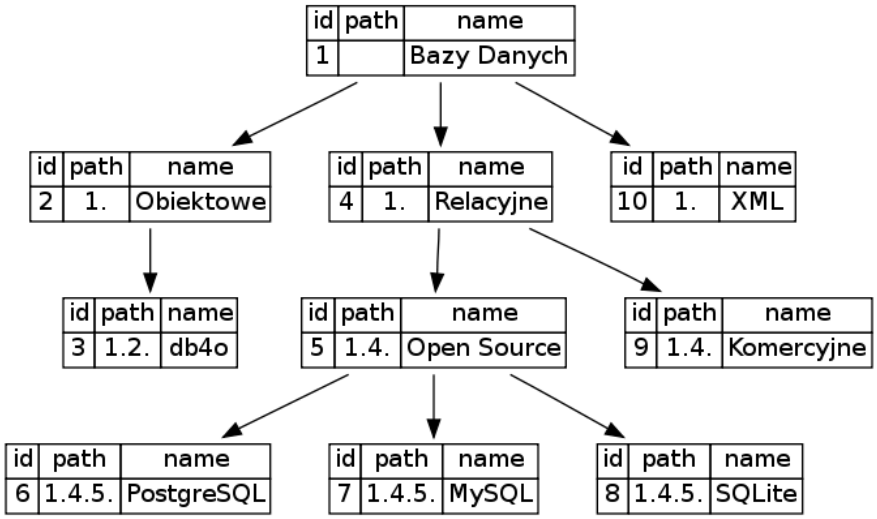
- napis z elementami przekształconymi na napis stałej długości. Przykładowo dla długości 3 będzie to `'004008015016023042'`. Wadą tej metody jest konieczność jednorazowego wyboru długości napisu dla identyfikatora. Dla długości n można przechować tylko $10^n - 1$ węzłów. Natomiast zastosowanie dużej n zwiększa zużycie zasobów SZDB oraz spowalnia działanie.

- tablicę np. `{4, 8, 15, 16, 23, 42}` Możliwość przechowywania tablic wewnątrz rekordu nie jest powszechna wśród implementacji relacyjnych baz danych. Jako pozytywny wyjątek może posłużyć PostgreSQL oferujący typ danych `ARRAY`.

W praktyce najczęściej stosowanym podejściem jest pierwsze z wymienionych, więc to ono zostanie przedstawione poniżej.

Dla ułatwienia implementacji i zmniejszenia duplikacji danych została wprowadzona mała zmiana. Skoro w każdym węźle jest już atrybut zawierający jego identyfikator to można go pominąć w ścieżce. Dzięki temu nie trzeba ręcznie pobierać z sekwencji nowego `id` a można pozostawić to zadanie bazie danych. W efekcie ścieżka jest w postaci `'4.8.15.16.23.'` a `42` znajduje się w atrybucie `id`.

[6] Sama nazwa sugeruje podobieństwo do *zmaterializowanych widoków* (ang. *materialized views*). W obu przypadkach wynik zapytania jest przechowywany by przyśpieszyć kolejne operacje.



Operacje

Reprezentacja w SQL

W tej metodzie ważne jest by właściwie oszacować wymaganą długość atrybutu `path`. W prakryce nie powinna być ona mniejsza niż:

$$(\lceil log_{10}(|T|)\rceil + 2) \times height(T)$$

```
1  CREATE TABLE pathenum(
2    id   serial PRIMARY KEY,
3    path varchar(100),
4    name varchar(100)
5  )
```

Wstawianie danych

Podczas wstawiania potrzebne są informacje o wszystkich właściwych przodkach wstawianego węzła. Na szczęście znajdują się one w pojedynczym rekordzie — rodzicu. Wystarczy więc je pobrać i odpowiednio sformatować.

```
1  INSERT INTO pathenum (path, name) VALUES (
2    (SELECT path || id ||  '.' FROM pathenum WHERE id = :parent),
3    :name
4  )
```

Pobranie korzeni

Jako, że korzenie nie posiadają przodków to ich ścieżka jest pusta.

```
1  SELECT *
2    FROM pathenum
3    WHERE path = ''
```

Pobranie rodzica

```
1  SELECT result.*
2    FROM
```

```
3       pathenum AS arg,
4       pathenum AS result
5     WHERE
6       arg.id = :id AND
7       (result.path || result.id || '.') = arg.path
```

Ta implementacja nie jest ekstremalnie szybka, ale ma dwie zalety — daje się zaimplementować w każdej z obsługiwanych baz danych oraz wymagane zapytanie jest wszędzie takie samo.

Pobranie dzieci

Podczas wstawiania danych ścieżka rodzica jest konkatenowana z jego identyfikatorem. Wynik jest zapisywany w ścieżce dziecka. Czyli z węzła można wyliczyć wartość ścieżki jego dzieci.

```
1  SELECT *
2    FROM pathenum
3    WHERE path = (
4      SELECT path || id ||  '.'
5        FROM pathenum
6        WHERE id = :parent
7    )
```

Pobranie przodków

```
1  SELECT result.*
2    FROM
3      pathenum AS arg,
4      pathenum AS result
5    WHERE
6      arg.id = :id AND
7      arg.path LIKE (result.path || result.id || '%')
8    ORDER BY result.path DESC
```

Pobieranie potomków

```
1  SELECT *
2    FROM pathenum
3    WHERE path LIKE (
4      SELECT path || id || '.' || '%'
5        FROM pathenum
6        WHERE id = :parent
7    )
```

Uwagi

Dostosowania metody do przyjętego interface zmniejsza jej wydajność. Praktycznie wszystkie zapytania zawierają podzapytanie które dla danego identyfikatora węzła pobierają jego ścieżkę. Ta operacja jest w prawdzie bardzo szybka (kolumna `id` jest kluczem głównym) ale mimo wszystko mają wpływ na ogólną wydajność. W razie stosowania tej metody warto się zastanowić nad używaniem ścieżki (`path`) jako parametrów metod.

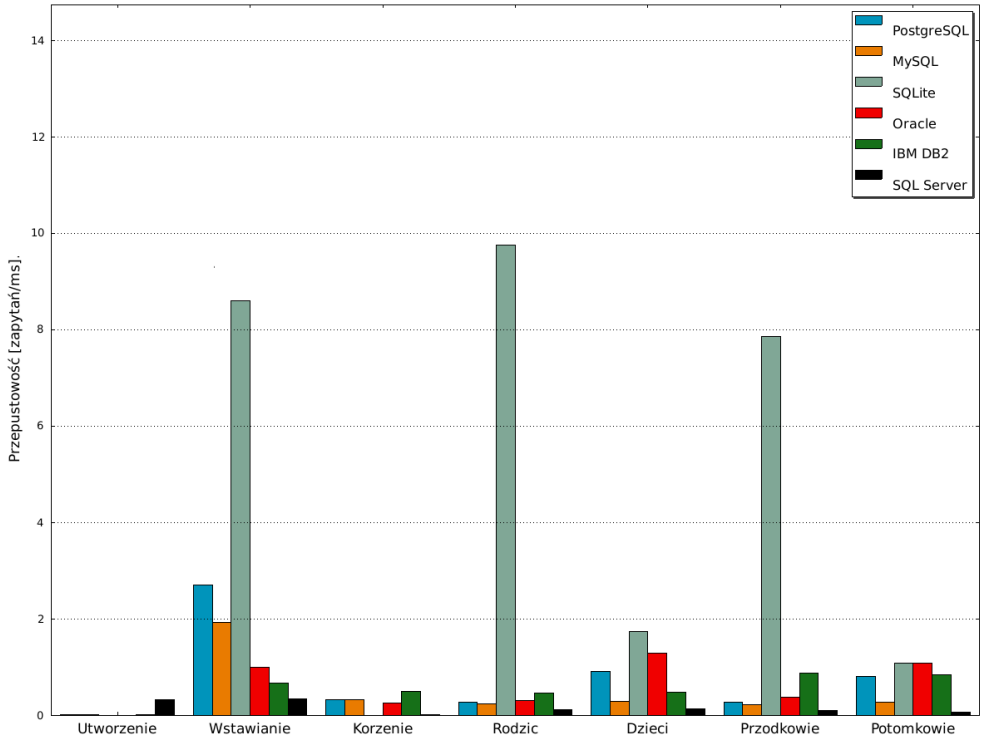
Dodatkowo odwoływanie się do węzłów za pomocą ścieżek a nie identyfikatorów daje większe możliwości. W takiej sytuacji można wykonać dodatkowe operacje bez konieczności wykonywania zapytań w bazie danych.

Wydajność

Tabela 3.4: Wydajność reprezentacji zmaterializowanych ścieżek

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
PostgreSQL	0,020	2,701	0,333	0,277	0,919	0,279	0,816
MySQL	0,011	1,933	0,333	0,244	0,289	0,222	0,278
SQLite	0,007	8,601	0,000	9,762	1,740	7,865	1,082
Oracle	0,007	1,002	0,250	0,309	1,294	0,388	1,094
IBM DB2	0,012	0,668	0,500	0,462	0,482	0,880	0,836
SQL Server	0,333	0,337	0,023	0,125	0,131	0,108	0,062

Rysunek 3.4: Wydajność reprezentacji zmaterializowanych ścieżek



Rozdział 4

Konstrukcje języka

Opisana w poprzednim rozdziale metoda krawędziowa ma szereg zalet. Niestety ma też poważną wadę — aby pobrać przodków lub potomków należy wykonać wiele zapytań. W praktyce oznacza to większą czasochłonność takiej operacji. Ponadto wymaga bardziej skomplikowanego kodu.

Obejściem tego problemu jest pełniejsze wykorzystanie możliwości dawanych przez bazy danych. Nie są one zwykłymi składami danych, lecz udostępniają coraz większe możliwości operowania na nich.

Wystarczy więc przepisać te dwie powolne metody tak by wykonywane były jako pojedyncze zapytanie w bazie danych. Pozostałe operacje w tych metodach korzystają z implementacji *metody krawędziowej*.

4.1 Wspólne Wyrażenia Tabelowe

W standardzie SQL:1999 dodano *Wspólne Wyrażenia Tabelowe* (ang. *CTE — Common Table Expressions*). Głównym celem ich wprowadzenia było umożliwienie pisania bardziej zwięzłego, czytelnego a przede wszystkim prostszego kodu.

Ich działanie przypomina stworzenie tymczasowego widoku. Po zdefiniowaniu jest on dostępny dla zapytania. Jednak w odróżnieniu od widoku jest on tworzony w samym zapytaniu i istnieje tylko na czas jego wykonywania.

Przykładowym zastosowaniem CTE jest uniknięcie wielokrotnego używania tego samego podzapytania[WCR+08]. Nie tylko zwiększy to czytelność kodu, ale poprawi jego wydajność.

Wspólne Wyrażenia Tabelowe mają też inne, ciekawsze z punktu widzenia tej pracy, zastosowanie. Pozwalają one na rekurencyjne wykonywanie zapytań w bazie danych.

W chwili obecnej to rozszerzenie jest dostępne w:

- **IBM DB2** począwszy od wersji 8
- **Microsoft SQL Server** począwszy od SQL Server 2005
- **PostgreSQL** począwszy od 8.4

Opis Wspólnych Wyrażeń Tabelowych

Składnia Rekurencyjnych Wspólnych Wyrażeń Tabelowych prezentuje się następująco:

```
1 WITH <<nazwa widoku>>(<<nazwy kolumn>>) AS
2 (
3   <<zapytanie startowe>>
4 UNION ALL
5   <<zapytanie rekurencyjne>>
6 )
7 SELECT <<nazwy kolumn>> FROM <<nazwa widoku>>
```

Działanie tej konstrukcji można streścić jako:

1. Na początku wykonywane jest `zapytanie startowe`. Jego wynik jest widoczny pod nazwą `nazwa widoku`.
2. Korzystając z wygenerowanych w poprzednim kroku rekordów (wyłącznie w ostatnim kroku, wyniki poprzednich wywołań nie są tu dostępne) `zapytanie rekurencyjne` zwraca nowe wyniki. Jeśli jest ich więcej niż zero to ten etap się powtarza.
3. Po zakończeniu pracy części rekurencyjnej jej *wszystkie* wyniki są dostępne dla zapytania znajdującego się po nim.

Ten algorytm można przedstawić również w poniższy sposób:

```
1 widok = []
2 tmp = zapytanie_startowe()
3 while tmp:
4   widok.extend(tmp)
5   tmp = zapytanie_rekurencyjne(tmp)
6 return widok
```

Najprostszy przykład wykorzystania zapytań rekurencyjnych to wygenerowanie ciągu liczb. W tym przykładzie ciągu arytmetycznego zaczynającego się od 0 (co wynika z zapytania startowego `SELECT 0 AS i`), różnica ciągu wynosi 1 (`SELECT i + 1 AS i`)

```
WITH RECURSIVE a(i) AS (
    SELECT 0 AS i
  UNION ALL
    SELECT i + 1 AS i
      FROM a
      WHERE i < 10
)
SELECT i FROM a;
```

Operacje

Niestety implementacje CTE różnią się od siebie. PostgreSQL wymaga jawnego podania, że chodzi o zapytanie rekurencyjne czyli `WITH RECURSIVE`. Pozostałe bazy wymagają samego `WITH` a słowo `RECURSIVE` powoduje zgłoszenie błędu. Ponadto DB2 nie zezwala na jawne użycie złączenia (czyli z `JOIN`). Czyli trzeba przenieść warunek złączenia do klauzuli `WHERE`.

Pobranie przodków

```
WITH RECURSIVE temptab(id, parent, name) AS
(
  SELECT root.id, root.parent, root.name
    FROM simple AS root
    WHERE root.id = :id
UNION ALL
  SELECT s.id, s.parent, s.name
    FROM simple AS s
      JOIN temptab AS t
        ON (s.id = t.parent)
)
SELECT id, parent, name
  FROM temptab
```

Pobieranie potomków

```
WITH RECURSIVE temptab(level, id, parent, name) AS
(
  SELECT 0, root.id, root.parent, root.name
    FROM simple AS root
    WHERE root.id = :id
UNION ALL
  SELECT t.level + 1, s.id, s.parent, s.name
    FROM simple AS s
      JOIN temptab AS t
        ON (s.parent = t.id)
)
SELECT level, id, parent, name
  FROM temptab
  WHERE level > 0
```

By zapytanie zwracało potomków właściwych dodano dodatkową kolumnę `level`. Jej wartość odpowiada poziomowi węzła w pobieranym poddrzewie.

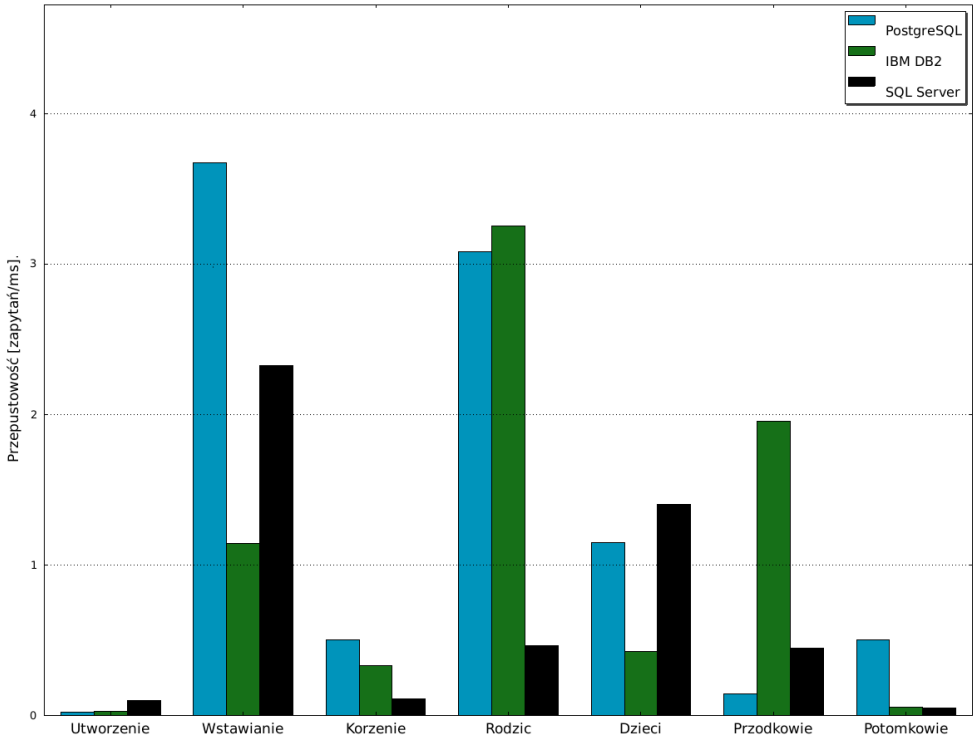
Pomijając tą kolumnę, kod jest niemal identyczny jak w pobieraniu przodków. Tym co je różni jest warunek złączenia.

Wydajność

Tabela 4.1: Wydajność metody Wspólnych Wyrażeń Tabelowych

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
PostgreSQL	0,021	3,675	0,500	3,083	1,149	0,141	0,504
IBM DB2	0,027	1,144	0,333	3,254	0,425	1,957	0,056
SQL Server	0,100	2,325	0,111	0,465	1,405	0,448	0,051

Rysunek 4.1: Wydajność metody Wspólnych Wyrażeń Tabelowych



4.2 CONNECT BY

System zarządzania bazą danych Oracle nie posiada możliwości przetwarzania danych hierarchicznych za pomocą klauzuli `WITH`. W prawdzie jest ona dostępna od wersji *Oracle 9i release 2*, ale służy wyłacznie do pracy z podzapytaniami.

W to miejsce *Oracle* udostępnia własne rozszerzenie `CONNECT BY`. Jest ono dobrze opisane w [Lon09], wiec tu zostaną przedstawione tylko podstawowe i najczęściej używane jego możliwości.

Opis klauzuli `CONNECT BY`

```
1 SELECT expression [,expression]...
2     FROM [user.]table
3     WHERE condition
4     CONNECT BY [PRIOR] expression = [PRIOR] expression
5     START WITH expression = expression
6     ORDER BY expression
```

Sposób działania da się w skrócie opisać regułami:

- `START WITH` wskazuje w którym miejscu drzewa rozpocząć działanie. Klauzula może się znajdować zarówno przed jak i po `CONNECT BY`.
- kierunek przechodzenia po węzłach zależny jest od tego przed którym wyrażeniem stoi `PRIOR`. Nie ma różnicy `CONNECT BY PRIOR a = b` a `CONNECT BY b = PRIOR a`.
- klauzula `WHERE` pozwala na wyeliminowanie z wyniku pojedyńczych rekordów, ale nie usuwa rekordów, które są dostępne po przejściu przez ten pominięty rekord.
- zastosowanie zwykłej klauzuli `ORDER BY` niszczy hierarchiczny układ danych (więc w praktyce rzadko bywa używana). Aby umożliwić kontrolowanie kolejności odwiedzania węzłów, w *Oracle 9i* wprowadzono klauzulę `ORDER SIBLINGS BY`.

Przydatne mechanizmy:

`LEVEL` pseudokolumna, jest równa 1 dla korzenia (lub węzła wskazanego przez `START WITH`), dla dzieci tego węzła jest równa 2, dla dzieci tych dzieci 3, i tak dalej.

`CONNECT_BY_ISLEAF` pseudokolumna o wartości 1 jeśli rekord jest liściem, 0 w przeciwnym wypadku

`CONNECT_BY_ISCYCLE` pseudokolumna o wartości 1 jeśli rekord ma potomka, który też jest jego przodkiem, 0 w przeciwnym wypadku

`SYS_CONNECT_BY_PATH(kolumna_wartosci, znak_separujacy)` funkcja zwraca złączoną w listę wartości z kolumny `kolumna_wartosci` wchodzące w skład ścieżki pomiędzy korzeniem a aktualnym węzłem. Każda wartość jest poprzedzona znakiem `znak_separujacy`. Dla przykładu `SYS_CONNECT_BY_PATH(name, '/')` może zwrócić `/Bazy danych/Obiektowe/db4o`.

Operacje

Pobranie przodków

```
1 SELECT level, id, parent, name
2   FROM simple
3   START WITH id = :id
4   CONNECT BY id = PRIOR parent
```

Pobieranie potomków

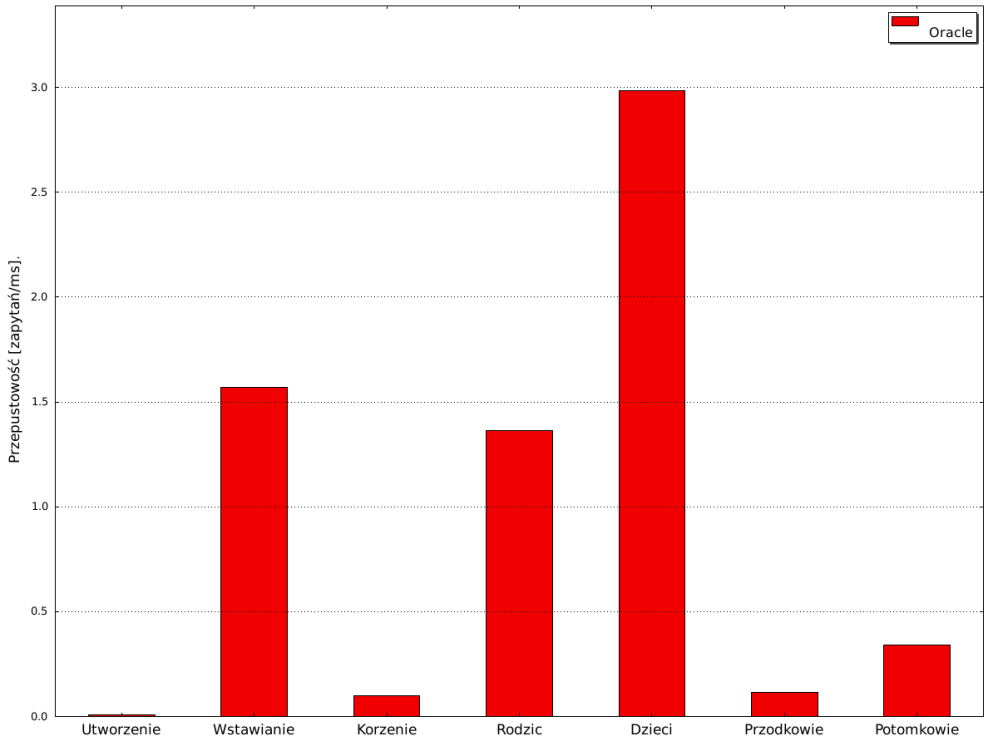
```
1 SELECT level, id, parent, name
2   FROM simple
3   START WITH id = :id
4   CONNECT BY parent = PRIOR id
```

Wydajność

Tabela 4.2: Wydajność metody `CONNECT BY`

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
Oracle	0,007	1,568	0,100	1,362	2,985	0,116	0,341

Rysunek 4.2: Wydajność metody `CONNECT BY`



Uwagi

Oracle w wersjach wcześniejszych od 11g wykonując zapytanie korzystające z klauzuli `CONNECT BY` korzysta z *full table scan*. Dlatego wydajność tych metod nie jest dobra.

Rozdział 5

Typy danych

Standardowo bazy danych umożliwiają przechowywanie wielu typów danych. Do powszechnie występujących można zaliczyć typy[1]:

- numeryczne (`integer`, `numeric`, `real`, ...)
- znakowe (`char`, `varchar`, `text`, ...)
- daty i czasu (`date`, `time`, `timestamp`, `interval`, ...)
- binarne (`bytea`)

Ponadto bazy danych oferują własne, specyficzne dla implementacji typy. Wśród nich warto wymienić typy tablicowe, sieciowe, logiczne, monetarne, wyliczeniowe oraz złożone. Coraz częściej stosowane są też typy umożliwiające przechowywanie dokumentów XML.

Co najważniejsze — z punktu widzenia tej pracy — istnieją typy ułatwiające przechowywanie danych hierarchicznych.

[1] Wymienione nazwy konkretnych typów pochodzą z bazy PostgreSQL

5.1 PostgreSQL `ltree`

Moduł `ltree` zawiera implementacje typu danych oraz funkcji umożliwiających przechowywanie danych hierarchicznych. Jest on dostępny w bazie PostgreSQL począwszy od wersji 7.2 (wydanej w roku 2002).

Oparty jest na *reprezentacji zmaterializowanych ścieżek*. Jednak, w odróżnieniu od metody opisanej w jednym z poprzednich rozdziałów tej pracy, oferuje wiele udogodnień. Chodzi tu przede wszystkim o kilkadziesiąt funkcji i operatorów ułatwiających pracę z tą metodą. Są one dobrze opisane w oficjalnej dokumentacji [ltr], więc tutaj zostaną przedstawione wyłącznie najważniejsze funkcjonalności.

Jedną z metod korzystania z `ltree` jest wykorzystanie `lquery`. Są to wyrażenia ścieżkowe, przypominające trochę wyrażenia regularne. Poszczególne elementy zapytania rozdziela się za pomocą kropki. Chcąc dopasować konkretną etykietę węzła po prostu się ją podaje. Można też zastosować znak `*`, który pasuje do każdego ciągu węzłów. Jeśli zastosuje {`n`}, {`n,`}, {`n,m`} lub {`,m`} (działające jak w wyrażeniach regularnych) można określić ile węzłów ma zostać dopasowanych. Mając zbudowane zapytanie, używa się go następująco: `ltree ~ lquery`. Przykładowo `t.node ~ '*.5.*'` spowoduje dopasowanie wszystkich węzłów, których przodkiem jest węzeł o identyfikatorze `5`.

Ponadto dostępnych jest wiele funkcji, z których najpotrzebniejsze są:

`ltree subpath(ltree, offset[, len])` pobiera część `ltree` zawierającą się pomiędzy elementami o pozycjach pomiędzy `offset` a `len`.

`ltree text2ltree(text)` **oraz** `text ltree2text(ltree)` pozwalają dokonywać konwersji pomiędzy `ltree` a łańcuchami znaków

operator `ltree <@ ltree` sprawdza, czy lewa strona jest potomkiem prawej

operator `ltree @> ltree` sprawdza, czy lewa strona jest przodkiem prawej

Operacje

Reprezentacja w SQL

```
1  CREATE TABLE ltreetab(
2    id   serial PRIMARY KEY,
3    path ltree,
4    name varchar(100)
5  )
```

Wstawianie danych

```
1  INSERT INTO ltreetab (path, name) VALUES (
2    text2ltree('' || currval('ltreetab_path_seq')),
3    :name
4  ) RETURNING id
```

Pobranie korzeni

```
1  SELECT *
2    FROM ltreetab
3    WHERE path ~ '*{1}'
```

Pobranie rodzica

```
SELECT *
  FROM ltreetab
  WHERE path = (
    SELECT subpath(path, 0, -1)
      FROM ltreetab
      WHERE id = :id
  )
```

Pobranie dzieci

```
SELECT *
  FROM ltreetab
  WHERE path ~ ((
    SELECT ltree2text(path)
      FROM ltreetab
      WHERE id = :parent
  ) || '.*{1}')::lquery
```

Pobranie przodków

```
SELECT *
  FROM ltreetab
  WHERE path @> (
    SELECT path
      FROM ltreetab
      WHERE id = :id
  )
```

Pobieranie potomków

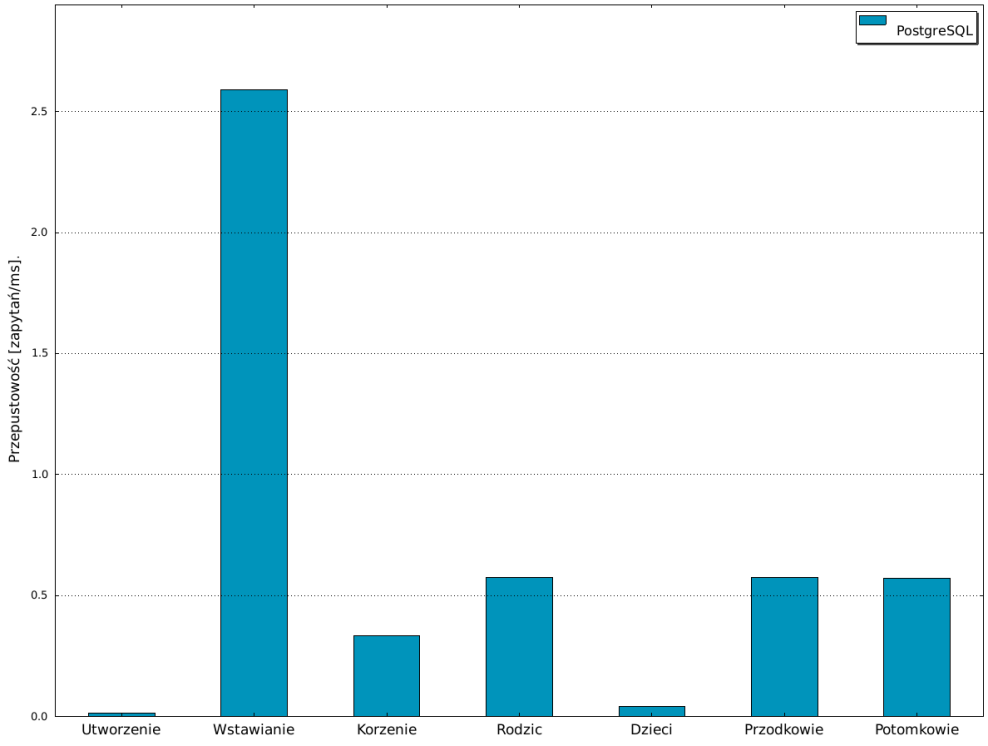
```
SELECT *
  FROM ltreetab
  WHERE path <@ (
    SELECT path
      FROM ltreetab
      WHERE id = :parent
  )
```

Wydajność

Tabela 5.1: Wydajność metody `ltree`

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
PostgreSQL	0,012	2,591	0,333	0,575	0,040	0,574	0,571

Rysunek 5.1: Wydajność metody `ltree`



5.2 Microsoft SQL Server `hierarchyid`

Jedną z najciekawszych nowości jakie Microsoft wprowadził w SQL Server 2008 jest nowy typ danych `hierarchyid`. Pozwala on na wygodne przechowywanie danych hierarchicznych.

Reprezentacja pozwala na przechowywanie drzew uporządkowanych.

Opis typu `hierarchyid`

Typ `hierarchyid` koncepcyjnie przypomina *reprezentację zmaterializowanych ścieżek*. Różnica polega na tym, że zamiast zapisywać pełne identyfikatory węzłów przodków zapisuje numery porządkowe określające, którym z kolei dzieckiem swojego rodzica jest dany węzeł.

Typ `hierarchyid` jest przechowywany wewnętrznie jako `VARBINARY`. Dodatkowo by osiągnąć mały rozmiar atrybutu zastosowano algorytm OrdPath[OOP+04, KC07]. Pierwotnie skonstruowano go, aby ułatwić pracę z danymi XML pozbawionymi schematu. Następnie został on zastosowany w implementacji typu `hierarchyid`.

Użyteczne funkcje

`hierarchyid::GetRoot()` Zwraca korzeń drzewa.

`parent.GetDescendant(child_after, child_before)` Zwraca wartość ścieżki dla nowo wstawianego węzła. Zmienne oznaczają odpowiednio:

`parent` węzeł rodzica nowo wstawianego rekordu

`child_after` dziecko, za którym nowy węzeł ma być wstawiony

`child_before` dziecko, przed którym nowy węzeł ma być wstawiony

W przypadku drzew zorientowanych `child_before` należy ustawić na `NULL`. Natomiast `child_after` ustawić na największą z wartości ścieżki braci nowego elementu.

`node.GetLevel()` Zwraca liczbę całkowitą będącą poziomem danego węzła w drzewie. Jako, że typ `hierarchyid` nie obsługuje lasów, to aby to kompensować tworzy się sztuczny korzeń, a korzenie obsługiwanych drzew znajdują się na poziomie 1.

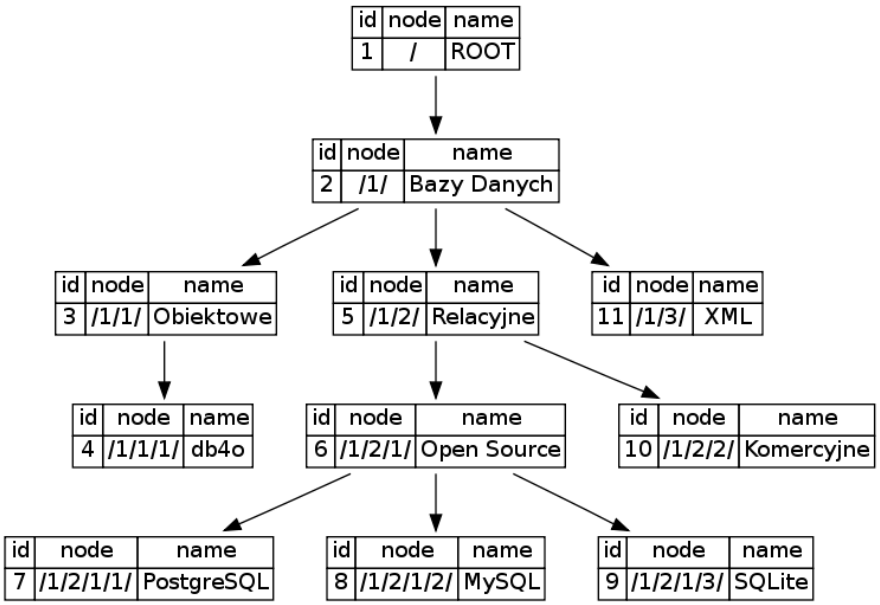
`child.GetAncestor(n)` Zwraca węzeł będący `n-tym` przodkiem danego węzła. Dla $n = 0$ zwraca tem sam element, dla $n = 1$ rodzica, dla $n = 2$ dziadka, itd.

`child.IsDescendantOf(parent)` Zwraca wartość logiczną określającą czy `child` jest potomkiem `parent`

`node.ToString()` Zwraca tekstową reprezentację danych.

Reprezentacja

Typ `hierarchyid` jest przeznaczony do przechowywania pojedyńczego drzewa. Aby umożliwić mu przechowywanie lasu należy połączyć korzenie wszystkich drzew z dodatkowym węzłem. Stanie się on korzeniem przechowywanego drzewa.



Operacje

Reprezentacja w SQL

```
CREATE TABLE herid (
  id   int IDENTITY PRIMARY KEY,
  node hierarchyid,
  name varchar(100)
);

INSERT INTO herid (node, name)
  VALUES (hierarchyid::GetRoot(), 'ROOT')
```

Po stworzeniu tabeli wstawiamy do niej pierwszy rekord — sztuczny korzeń całego lasu.

Wstawianie danych

```
INSERT INTO herid (node, name) VALUES (
  (SELECT node
    FROM herid
    WHERE id = :parent
  ).GetDescendant(
    (SELECT max(node) node
      FROM herid
      WHERE node.GetAncestor(1) = (
        SELECT node
          FROM herid
          WHERE id = :parent
      )
    ),
    NULL
  ),
  :name)
```

Pobranie korzeni

Mając do dyspozycji metodę `node.GetLevel()` można łatwo pobrać korzenie. Wystarczy tylko pamiętać, że na poziomie 0 znajduje się sztuczny korzeń, a właściwe korzenie są ma poziomie 1.

```
1 SELECT node.ToString() AS path, name
2   FROM herid
3   WHERE node.GetLevel() = 1
```

Pobranie rodzica

Dzięki metodzie `node.GetAncestor(n)` można wyliczyć ścieżkę rodzica.

```
1 SELECT node.ToString() AS path, name
2   FROM herid
3   WHERE node = (
4     SELECT node.GetAncestor(1)
5       FROM herid
6       WHERE id = :id
7   )
```

Pobranie dzieci

Ta operacja jest de facto odwróconym pobieraniem rodzica.

```
1 SELECT node.ToString() AS path, name
2   FROM herid
3   WHERE node.GetAncestor(1) = (
4     SELECT node
5       FROM herid
6       WHERE id = :id
7   )
```

Pobranie przodków

```
1 SELECT node.ToString() AS path, node.GetLevel() AS lvl, name
2   FROM herid
3   WHERE (
4     SELECT node n1 FROM herid WHERE id = :id
5   ).IsDescendantOf(node) = 1 AND
6   node.GetLevel() > 0
7   ORDER BY lvl DESC
```

Po raz kolejny należy pamiętać o pominięciu sztucznego korzenia.

Pobieranie potomków

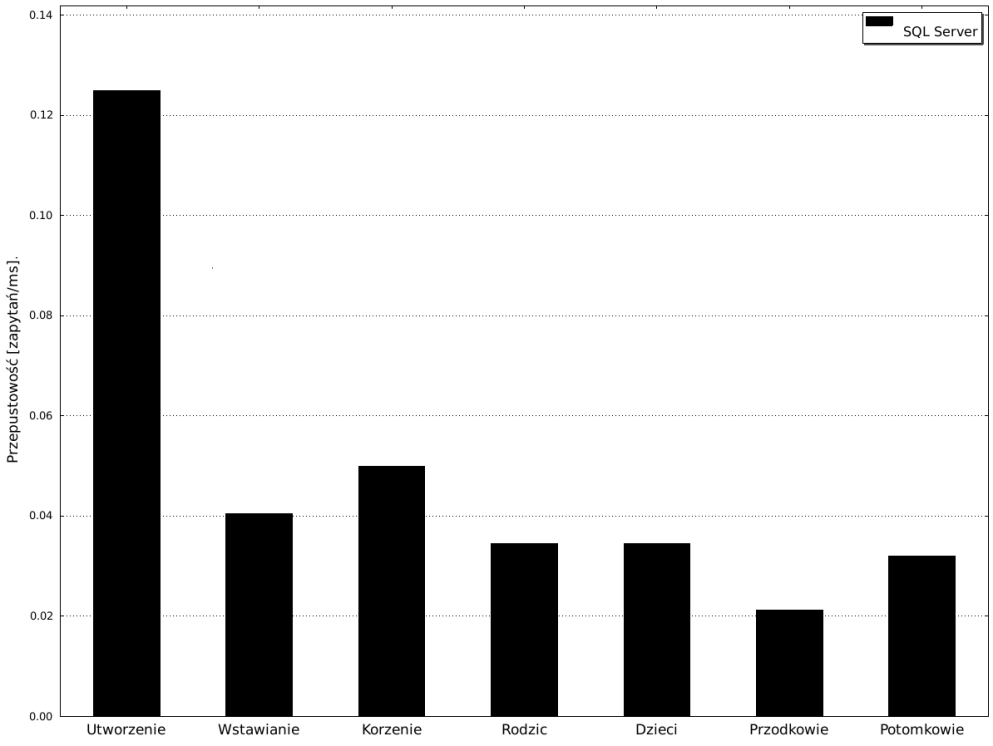
```
1 SELECT node.ToString() AS path, name
2   FROM herid
3   WHERE node.IsDescendantOf((
4     SELECT node FROM herid WHERE id = :id
5   )) = 1 AND
6   id <> :id
```

Wyniki

Tabela 5.2: Wyniki `hierarchyid`

	Utworzenie	Wstawianie	Korzenie	Rodzic	Dzieci	Przodkowie	Potomkowie
SQL Server	0,125	0,041	0,050	0,035	0,035	0,021	0,032

Rysunek 5.2: Wyniki `hierarchyid`

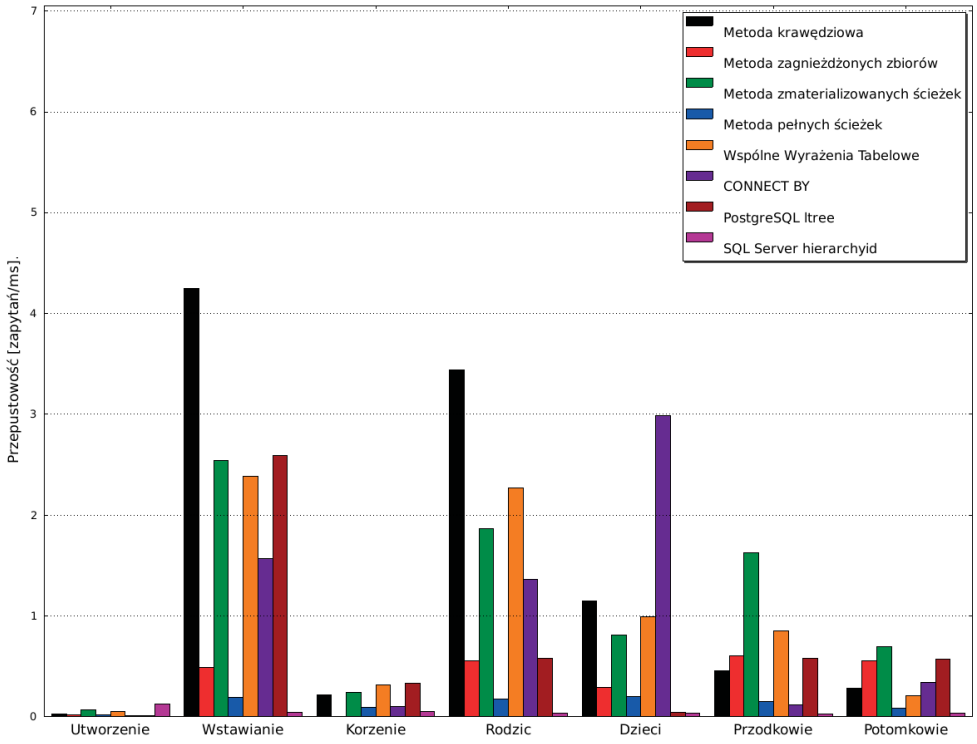


Rozdział 6

Podsumowanie

Do tej pory wyniki testów były prezentowane dla każdej metody oddzielnie, pokazując różnice pomiędzy systemami baz danych. By móc porównać metody między sobą warto je ująć w jednym miejscu. Przedstawione tu wartości powstały poprzez uśrednienie wyników wszystkich SZBD umożliwiających implementację danej metody.

Rysunek 6.1: Porównanie wydajności metod



Którą metodę wybrać?

W poprzednich rozdziałach zostały przedstawione popularne oraz warte zainteresowania metody przechowywania danych hierarchicznych. Zaprezentowano ich implementacje oraz wyniki wydajności.

Jak można się domyślić, nie istnieje metoda najlepsza w wszystkich sytuacjach[1]. Czyli programista musi przewidzieć [2] jakie operacje będą najczęściej wykorzystywane.

Poniżej zostaną podane rady ułatwiające wybór metody dla konkretnego, praktycznego zastosowania.

Częste wstawianie węzłów Jeśli struktura hierarchiczna jest często modyfikowana to należy wybrać metodę oferującą szybką operację wstawiania. Zdecydowanie najlepsza jest w tym metoda krawędziowa. Ewentualnie można by użyć reprezentacji zmaterializowanych ścieżek. Przy okazji warto zauważyć, że oparty na niej typ `ltree` oferuje równie dobrą wydajność.

Pobieranie przodków i potomków Są to operacje, w których reprezentacja krawędziowa radzi sobie słabo. Z tego wynika popularność metody zagnieżdżonych zbiorów, uważanej za najlepszą w tych operacjach. Jak pokazały wyniki, niesłusznie. Wprawdzie oferuje dobrą wydajność, ale metoda zmaterializowanych ścieżek jest w tej sytuacji jeszcze lepsza.

Jeśli nie jest możliwe przewidzenie jak będą używane dane hierarchiczne to najlepiej zastosować reprezentację krawędziową. Przemawia za tym:

- Jest najprostsza w implementacji.
- Metoda ma dobrą wydajność. W kilku operacjach jest najszybsza, nigdzie nie jest bardzo powolna.
- W razie potrzeby można (bez zmian w strukturze tabel, wystarczy zastosować inne zapytania) przekształcić ją w metodę Wspólnych Wyrażeń Tabelowych lub `CONNECT BY`.
- Opiera się na bardzo prostej koncepcji, więc przeniesienie danych z niej do dowolnej z omówionych w tej pracy metod powinno być proste.

[1] Dowód nie wprost: jeśli by istniała perfekcyjna metoda to tylko ona była by stosowana

[2] Jeszcze lepiej było by sprawdzić założenia, przeprowadzając symulację

Bibliografia

[Cel00] Joe Celko. *SQL. Zaawansowane techniki programowania.* MIKOM, 2000.

[Cel04] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties.* Morgan Kaufmann, 2004.

[CLRS04] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Wprowadzenie do algorytmów.* Wydawnictwa Naukowo-Techniczne, Warszawa, 2004.

[Dro04] Adam Drozdek. *C++. Algorytmy i struktury danych.* Helion, Gliwice, 2004.

[KC07] A. Kumaran, Peter Carlin. Multilingual Semantic Matching with OrdPath in Relational Systems. Raport instytutowy, Microsoft Corporation, 2007.

[Knu02] Donald E. Knuth. *Sztuka programowania.* Wydawnictwa Naukowo-Techniczne, Warszawa, 2002.

[Lon09] Kevin Loney. *Oracle Database 11g: The Complete Reference.* McGraw-Hill, 2009.

[ltr] Postgresql: Documentation: Manuals: Postgresql 8.4: ltree. `http://www.postgresql.org/docs/8.4/interactive/ltree.html`. [dostęp: 2010-09-27].

[OOP+04] Patrick O'Neil, Elizabeth O'Neil1, Shankar Pal, Istvan Cseri, Gideon Schaller, Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. Raport instytutowy, University of Massachusetts Boston, 2004.

[WCR+08] Robert E. Walters, Michael Coles, Robert Rae, Fabio Ferracchiati, Donald Farmer. *Accelerated SQL Server 2008.* Apress, 2008.