Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Aleksandra Boniewicz

Optymalizacja warstwy dostępu do danych w aplikacjach korzystających z odwzorowań obiektowo-relacyjnych

Rozprawa doktorska

Promotor

prof. dr hab. Krzysztof Stencel
Instytut Matematyki
Uniwersytet Warszawski

Promotor pomocniczy

dr hab. Piotr Wiśniewski
Zakład Baz Danych
Uniwersytet Mikołaja Kopernika w Toruniu

Luty 2016

Oświadczenie autora rozprawy:
Oświadczam, że niniejsza rozprawa została napisana przeze mnie samodzielnie.

15 Luty 2016 .....
data *Aleksandra Boniewicz*

Oświadczenie promotora rozprawy:
Oświadczam, że niniejsza rozprawa jest gotowa do oceny recenzentów.

15 Luty 2016 .....
data *prof. dr hab. Krzysztof Stencel*

Streszczenie

Współczesne aplikacje są budowane za pomocą obiektowych języków programowania. Do składowania danych trwałych najczęściej stosuje się jednak bazy danych o relacyjnym modelu danych, który istotnie różni się od obiektowego. Utrwalanie danych aplikacji w bazie danych wymaga więc niebanalnego odwzorowania obiektów. Chociaż takie odwzorowanie można zaprogramować od nowa na użytek danej aplikacji, warto jednak także rozważyć zastosowanie gotowych bibliotek do odwzorowania obiektowo-relacyjnego (*ORM*, *Object-Relational Mapping*).

Funkcjonalność istniejących systemów ORM jest istotnie mniejsza niż dostępnych systemów zarządzania bazami danych (SZBD). Programista, który chce skorzystać z zaawansowanych funkcji SZBD, musi to zrobić z pominięciem warstwy ORM, co burzy architekturę aplikacji. Uważamy, że tak nie musi być.

Celem niniejszej rozprawy doktorskiej jest zbadanie możliwości zaoferowania zaawansowanych funkcji systemów zarządzania bazami danych na poziomie warstwy odwzorowania obiektowo-relacyjnego. Cel ten został osiągnięty. Wykazaliśmy też, że ORM może realizować odpowiednią funkcjonalność nawet wtedy, gdy nie ma jej w użytkowanym SZBD. Zaawansowaną funkcjonalnością SZBD rozważaną w niniejszej rozprawie jest realizacja zapytań rekurencyjnych zgodnych ze standardem SQL:1999. Przedstawiliśmy jej założenia, prototypową implementację oraz wyniki testów wydajnościowych. Do budowy prototypu wybrano Hibernate, tj. jedno z najpopularniejszych narzędzi ORM dla Javy.

Słowa kluczowe: zapytania rekurencyjne, optymalizacja zapytań, SQL, HQL, Hibernate, redundancja danych

Klasyfikacja według ACM: H.2.2, H.2.3, H.2.4

Abstract

Contemporary applications are developed in object-oriented programming languages. However, persistent data storage is usually realized by databases with the relational data model that significantly differs from the object-oriented data model. Therefore, persisting application data in a database requires a non-trivial mapping. Although such a mapping can be coded from scratch for a given application, the usage of off-the-shelf object-relational mapping (ORM) libraries is worth consideration.

The functionality of existing ORM middleware is notably smaller than the functionality of available database management systems (DBMS). An application programmer who wants to use sophisticated features of a DBMS is forced to bypass the ORM layer. This obviously ruins the architecture of the application. In our opinion it is not necessary.

The goal of this thesis is to examine the possibility to offer advanced functions of database management systems at the level of object-relational mapping middleware. This goal has been reached. We have also shown that ORM layers can also offer a particular DBMS functionality even when the underlying DBMS does not support it. The advanced DBMS functionality considered in this thesis is the execution of SQL:1999 recursive queries. We have presented the concept of this feature, a prototype implementation and results of thorough performance evaluation. The prototype has been realized as an extension to Hibernate, i.e. probably the most popular object-relational mapping library for Java.

Keywords: recursive queries, query optimization, SQL, HQL, Hibernate, redundant data

Classification according to ACM: H.2.2, H.2.3, H.2.4

Podziękowania

Chciałabym bardzo serdecznie podziękować mojemu promotorowi Krzysztofowi Stenclowi oraz promotorowi pomocniczemu Piotrowi Wiśniewskiemu za cierpliwość, pomoc oraz naukowe wsparcie przy tworzeniu niniejszej rozprawy.

Dziękuję również moim Rodzicom za wsparcie oraz pomoc w życiu codziennym.

Na koniec pragnę podziękować mojemu Mężowi za wiarę we mnie, cierpliwość, wsparcie i miłość.

Pracę dedykuję
mojej Rodzinie.

Spis treści

Rozdział 1

Wprowadzenie

1.1 Cel prowadzonych badań

W większości powstających dziś aplikacji za przechowywanie i dostarczanie danych odpowiada baza danych. Nadal najbardziej popularnymi bazami danych są relacyjne bazy danych [30], natomiast najczęściej wybieranymi językami do tworzenia aplikacji są języki obiektowe [36]. Model relacyjny oraz model obiektowy są dwoma zupełnie różnymi światami [38].

Model relacyjny stworzony przez E.F. Codda [11] jest oparty na logice pierwszego rzędu oraz teorii zbiorów. Informacja w tym modelu jest opisana przez predykaty. Relacyjny model danych obejmuje relacje, atrybuty, wartości oraz zmienne relacyjne. Relacja jest pojęciem w sensie teorii zbiorów. Przykładową relacją może być relacja `Osoba` o następującym schemacie:

```
Osoba: [Imie, Nazwisko, Data urodzenia, Adres]
```

Wartości atrybutów są powiązane przez relacje. Każdy atrybut ma przypisaną dziedzinę (np. `integer`, `string`, `date`) oraz nazwę. W przykładzie powyżej atrybutami są: `Imię`, `Nazwisko`, `Data urodzenia` i `Adres`. Przez wartości relacyjne rozumiemy zbiór krotek spełniających relację. Przykładowo:

```
{['Jan', 'Kowalski', '20.01.1980', 'ul. Piekarska 32'],
 ['Anna', 'Nowak', '30.04.1985', 'ul. Podrzeczna 15a']}
```

Natomiast przez zmienną relacyjną rozumiemy zmienną zawierającą wartość relacyjną.

Z drugiej strony model obiektowy jest próbą usystematyzowania informacji w byty zwane obiektami. Model obiektowy obejmuje następujące pojęcia:

Tożsamość każdego obiektu jest niepowtarzalna. Nie ma dwóch identycznych obiektów. Obiekty są rozróżnialne nawet wtedy, gdy mają ten

sam stan.

Stan to zbiór wartości atrybutów obiektu w ustalonej chwili.

Interfejs to zbiór operacji (metod), które można wykonać na obiekcie.

Hermetyzacja to reguła mówiąca, że stan obiektu może się zmienić jedynie w wyniku uruchomienia jego metod. Hermetyzacja ogranicza też możliwości bezpośredniego odczytu stanu obiektu.

Klasa to projekt (implementacja) zbioru obiektów.

Dziedziczenie pozwala definicjom klas korzystać z właściwości innych klas.

Problemy związane z wykorzystaniem modelu relacyjnego w aplikacjach tworzonych w modelu obiektowym określa się wziętym z elektrotechniki terminem *niezgodność impedancji* (ang. *impedance mismatch* [39])[1]. Niezgodność impedancji odnosi się m.in. do różnic między modelami pojęciowymi języka zapytań SQL i obiektowych języków programowania [27]. Problem ten został szeroko opisany w literaturze [31]. Problem niezgodności impedancji powoduje obniżenie jakości oprogramowania, zmniejszenie poziomu abstrakcji oraz zwiększenie nakładu pracy podczas tworzenia aplikacji [31].

Wyróżniamy pięć podstawowych aspektów niezgodności impedancji. Są to [38, 13]:

Identyczność obiektów w świecie relacyjnym sprowadza się do identyczności bytów mających ten sam stan. W relacyjnej bazie danych dwie krotki o tych samych wartościach są nierozróżnialne, podczas, gdy dwa obiekty mające identyczny stan są rozróżnialne.

Dziedziczenie nie występuje w modelu relacyjnym.

Stan w modelu relacyjnym jest składowany w tabelach. Aplikacje operują jedynie na tymczasowych kopiach, które mogą stać się nieaktualne, jeśli w międzyczasie zostanie wykonana transakcja zmieniająca stan bazy danych. Nawet jeśli aplikacja skończy działanie, stan bazy danych nie zostanie usunięty. W modelu obiektowym stan obiektów zależy od wykonywanej na nim aplikacji. Gdy aplikacja się kończy, stan obiektu również przestaje istnieć.

[1] Warto zwrócić uwagę, że w języku polskim jest to termin wyłącznie informatyczny, ponieważ polscy elektrotechnicy przyjęli inne nazewnictwo.

Zachowanie W odróżnieniu od relacyjnego modelu danych w modelu obiektowym nie można bezpośrednio manipulować stanem obiektu. Można to robić jedynie poprzez metody. Zbiór operacji na obiekcie nazywany jest interfejsem lub zachowaniem obiektu.

Te zagadnienia dały początek badaniom nad technikami odwzorowania obiektów i relacji [25, 24]. Narzędzia *odwzorowania obiektowo-relacyjnego* (ang. *Object/Rational Mapping* - w skrócie *ORM*) dostarczają obiektowym systemom oprogramowania mechanizmów do przechowywania trwałych danych obiektów w bazie danych z pełną kontrolą transakcyjności. Jednocześnie obiekty w aplikacji [38] pozostają obiektami w rozumieniu języka programowania. Pierwszym tego typu narzędziem był Hibernate stworzony przez Gavina Kinga w 2002 roku [1]. Obecnie większość popularnych języków obiektowych ma narzędzia ORM, np. Hibernate dla Javy, LinQ i ADO dla .NET, SQLObject dla języka Phyton [33].

Narzędzia ORM zapewniają wygodną obsługę automatycznej trwałości obiektów biznesowych aplikacji. Pozwalają programistom skupić się na pisaniu kodu aplikacji, a nie na zaawansowanych niuansach języka SQL. Na podstawie metadanych ORM generuje zapytania w dialekcie języka SQL odpowiednim dla aktualnie stosowanej bazy danych. Umożliwia podstawowe operacje CRUD na obiektach klasy zapewniając trwałość i poprawność przetwarzania transakcji. Systemy ORM nie są związane z dostawcami baz danych. Uniezależniają więc programistów od znajomości odpowiedniego dialektu SQL. Zwiększa to przenośność aplikacji. Pozwala wykorzystać podejście, w którym programiści w fazie budowy i testowania aplikacji używają lekkich, lokalnych baz danych. Wdrażają natomiast system działający na innej, bardziej złożonej bazie danych [1]. Narzędzia ORM przyczyniają się również do większej czytelności tworzonej aplikacji. To z kolei ułatwia jej konserwację. Narzędzia te są cały czas rozwijane. Twórcy ORM skupili się jednak głównie na podstawowych problemach trwałości danych. Systemy zarządzania bazami danych (SZBD) oferują natomiast wiele skomplikowanych funkcji i możliwości optymalizacji przetwarzania zapytań realizowanych po stronie bazy danych [3]. Użycie tych funkcjonalności w projektach opartych o ORM rujnuje fundamentalną architekturę proponowaną w ORM. Korzystanie z zaawansowanych funkcji SZBD wymaga bowiem bezpośredniego (z pominięciem warstwy ORM) kontaktu z serwerem bazy danych. Funkcje te w znaczący sposób ułatwiają jednak pracę programistom tworzącym aplikacje obiektowe.

Warto pamiętać, że warstwa odwzorowania obiektowo-relacyjnego stanowi dodatkową abstrakcję składowiska danych. Można ją więc wykorzystać do przezroczystej integracji funkcji SZBD w celu udostępnienia ich progra-

mistom aplikacyjnym. To nie tylko ułatwi pracę programistów, ale również poprawi wydajność aplikacji. Programiści nie będą bowiem musieli stosować nieeleganckich obejść pozwalających skorzystać z niedostępnych w warstwie ORM, ale bardzo potrzebnych im funkcji SZBD. Użycie tych funkcji z pominięciem interfejsu ORM powoduje nieczytelność kodu, wykonywanie nadmiarowych operacji i wysyłanie zbędnych zapytań do bazy danych.

1.1.1 Cel rozprawy

Celem prezentowanej rozprawy doktorskiej jest zbadanie możliwości zaoferowania zaawansowanych funkcji systemów zarządzania bazami danych na poziomie warstwy odwzorowania obiektowo-relacyjnego. Rozważane są transakcyjne bazy danych. Proponowane rozwiązania powinny być zgodne lub rozszerzać standard JPA (Java Persistence API). Cel ten został osiągnięty. Wykazaliśmy też, że ORM może realizować odpowiednią funkcjonalność nawet wtedy, gdy nie ma jej w użytkowanym SZBD. Skupiliśmy się na zbadaniu jednej funkcjonalności: wykonywaniu zapytań rekurencyjnych. Dla wybranej funkcjonalności przedstawiliśmy jej założenia, prototypową implementację oraz wyniki testów wydajnościowych. Ponieważ nie istnieją aktualnie żadne biblioteki czy narzędzia wspierające proponowaną funkcjonalność dla istniejących narzędzi ORM do budowy prototypu wybrano Hibernate, tj. jedno z najpopularniejszych narzędzi ORM dla obiektowego języka programowania - Javy. Wybór ten był podyktowany tym, że jest to po pierwsze najpopularniejsze narzędzie ORM, a po drugie ma posłużyć jedynie do implementacji i testów wydajnościowych naszej propozycji. Sposoby realizacji oraz rozszerzenia standardu JPA wybranej funkcjonalności (zależnie od możliwości użytkowanego SZBD) opisano w kolejnych rozdziałach.

1.1.2 Znaczenie zapytań rekurencyjnych

Modele baz danych większości aplikacji biznesowych zawierają dane hierarchiczne i grafowe. Przykładami takich danych są hierarchie pracowników, rozkłady jazdy, rozkłady lotów, drzewa kategoryzacji produktów w sklepach internetowych, itp. Jakkolwiek składowanie takich danych w relacyjnych strukturach danych nie nastręcza wielu trudności, to do ich przetwarzania niezbędne są środki wykraczające poza SQL:1992. O takie środki wzbogacono standard SQL w 1999 roku. W SQL:1999 pojawiły się tzw. zapytania rekurencyjne, pozwalające przemierzać hierarchie dowolnej głębokości i grafy o dowolnych średnicach. W 2010 opublikowaliśmy przeglądowy artykuł [29] o implementacjach zapytań rekurencyjnych w komercyjnych systemach zarządzania bazami danych. Przedstawiliśmy też wyniki eksperymentów nad

wydajnością tych implementacji. Artykuł ten spotkał się z zainteresowaniem: w chwili obecnej ma trzy zewnętrzne cytowania, w tym jedno w artykule z konferencji VLDB w 2014 roku.

USOS (Uniwersytecki System Obsługi Studiów) jest przykładem aplikacji bogatej w dane hierarchiczne. Jest on przeznaczony do kompleksowej obsługi spraw związanych ze studiami, studentami, doktorantami, słuchaczami studiów podyplomowych oraz z pracownikami naukowo-dydaktycznymi [21]. USOS Powstał na Uniwersytecie Warszawskim w 2000 roku w ramach programu TEMPUS. Od tamtej pory jest cały czas rozwijany i eksploatowany. Używa go 30 uczelni w Polsce, w tym Uniwersytet Mikołaja Kopernika w Toruniu. USOS obsługuje 20% łącznej liczby uczelni publicznych w Polsce i zarazem 40% łącznej liczby studentów uczelni publicznych w Polsce [22]. Dane w tym systemie są przechowywane w relacyjnej bazie danych Oracle. Schemat bazy danych USOS obejmuje 418 tabel (stan na lipiec 2014), z czego 15 (3.5%) tabel przechowuje dane hierarchiczne. W kodzie źródłowym USOS znajduje się 50 zapytań rekurencyjnych (stan na lipiec 2014). Wykaz tabel przechowujących dane hierarchiczne wraz z liczbą ich wierszy oraz przykładowe zapytania rekurencyjne zostały zamieszczone w Dodatku B.

1.2 Wyniki przedstawione w rozprawie

- Opracowano rozszerzenia narzędzi odwzorowań obiektowo-relacyjnych o obsługę zapytań rekurencyjnych zgodnych ze standardem SQL:99. Obsługa ta pozwoli istotnie uprościć kod aplikacji oraz zwiększyć ich wydajność.

- Opracowano i zanalizowano mechanizmy realizacji zapytań rekurencyjnych w narzędziach ORM dla tych SZBD, które nie wspierają tych zapytań. Mechanizmy te uproszczą pracę programistów, którzy dotąd musieli zapytania rekurencyjne wykonywać w sposób iteracyjny. Obciążało to bazę danych i sieć oraz komplikowało kod aplikacji.

- Opracowano i zanalizowano rozmaite mechanizmy realizacji zapytań rekurencyjnych wykorzystujących dane redundantne w narzędziach ORM dla tych SZBD, które nie wspierają tych zapytań. Mechanizmy te oraz ich analiza pozwolą programistom aplikacyjnym wybrać tę metodę wykonywania zapytań rekurencyjnych, która jest najbardziej właściwą dla danej aplikacji. To będzie skutkować zwiększeniem jej wydajności i uproszczeniem jej kodu źródłowego.

1.3 Struktura pracy

Rozprawa jest podzielona na 7 rozdziałów. W rozdziale 2 opisano podstawowe pojęcia i użyte technologie. W rozdziale 3 przedstawiono propozycję benchmarku do testów wydajnościowych implementacji zapytań rekurencyjnych SQL:1999. W rozdziale 4 przedstawiono rozszerzenie narzędzi odwzorowań obiektowo-relacyjnych o możliwość wykonywania zapytań rekurencyjnych w bazach, które te zapytania wspierają, czyli np. IBM DB2, Oracle oraz PostgreSQL. W rozdziale 5 zaproponowano metody wykonywania zapytań rekurencyjnych w aplikacjach korzystających z ORM nad bazami danych, które tego mechanizmu nie mają, np. MySQL. W rozdziale 6 rozważono metody wykonywania zapytań rekurencyjnych w oparciu o nadmiarową materializację danych. Rozdział 7 zawiera podsumowanie wyników przedstawionych w niniejszej rozprawie.

Rozdział 2

Używane technologie

2.1 JPA

JPA (Java Persistence API) jest standardem ORM (Object-Relational Mapping) dla języka Java będącym elementem EJB 3.0 (Enterprise JavaBeans 3.0).

Pierwsza wersja standardu (1.0) została zdefiniowana w JSR 220 (Java Specification Requests) w ramach programu JCP (Java Community Process). Akceptacja nastąpiła 11 maja 2006 roku. Kolejna wersja (2.0) została zdefiniowana JSR 317 ([26]) i zaakceptowana 10 grudnia 2009 roku. Ostatnia wersja JPA została zdefiniowana w JSR 338 i zaakceptowana 22 kwietnia 2013 roku.

JPA powstało głównie na podstawie Hibernate; twórca Hibernate Gavin King wszedł w skład grupy ekspertów projektujących ten standard, jednak twórcy inspirowali się również projektami JDO oraz TopLink ([2]).

JPA operuje na obiektach zwanych encjami (entity). Są one pobierane i zapisywane w relacyjnej bazie danych za pomocą EntityManagera. Mapowania encji, które są podstawą działania każdego ORM'a, opisywać można poprzez pliki XML lub adnotacje. Do wykonywania zapytań można wykorzystać metody z EntityManager'a oraz JPA Query Language - język zapytań podobny do SQL'a.

W skład JPA wchodzi ponadto wiele dodatkowych elementów znacznie ułatwiających pracę. Są to między innymi walidatory czy konwertery ([23]).

2.2 Hibernate

Hibernate stanowi warstwę pośredniczącą między bazą danych a aplikacją. Jest oparty o standard JPA. Nie zależy od żadnej konkretnej bazy danych.

Umożliwia odwzorowanie obiektów aplikacji na tabele w relacyjnej bazie danych. Hibernate jest oprogramowaniem o otwartym kodzie źródłowym. Hibernate jest podzielony jest na kilka podprojektów:

Hibernate Core jest centralną częścią projektu.

Hibernate Annotations obsługuje adnotacje do definiowania odwzorowań między obiektami i relacjami.

Hibernate EntityManager realizuje komunikację z bazą danych.

Hibernate Shards ułatwia stosowanie Hibernate Core w przypadku jednoczesnego podłączenia wielu baz danych.

Hibernate Validator ułatwia walidację odwzorowywanych pól.

Hibernate Search umożliwia wyszukiwanie pełnotekstowe.

Odwzorowanie obiektów na relacje może odbywać się na dwa różne sposoby. Pierwszy z nich polega na użyciu pliku konfiguracyjnego napisanego w języku XML. Drugi sposób polega na dodaniu adnotacji do klas trwałych. Plik `hibernate.cfg.xml` zawiera podstawową konfigurację Hibernate, czyli wszystkie potrzebne informacje do połączenia się z wybraną bazą danych. Wymienia się w nim także nazwy plików z odwzorowaniami klas obiektów. Przykładowy plik konfiguracyjny może wyglądać tak, jak na Listingu 2.1. Zawiera on następujące informacje [1]:

- parametry połączenia z bazą danych: sterownik (linijka 7), port (linijka 9), nazwa i hasło użytkownika (linijki 11 i 13), rodzaj bazy danych (linijka 15),
- właściwości puli połączeń: minimalna liczba oczekujących na działanie połączeń JDBC (linijka 17), maksymalna liczba połączeń w puli (linijka 19), okres bezczynności (po tym okresie połączenie zostaje usunięte z puli - linijka 21), maksymalna liczba zbuforowanych poleceń (linijka 23),
- określenie, czy automatycznie generować polecenia DDL SQL na podstawie istniejącego dokumentu odwzorowań Hibernate (linijka 25),
- wskazanie, czy włączyć wyświetlanie poleceń SQL na konsolę (linijka 27),
- nazwy plików z odwzorowaniami klas obiektów na relacyjny schemat (linijka 28).

Listing 2.1: Przykład pliku konfiguracyjnego dla Hibernate.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE hibernate-configuration SYSTEM
3    "http://hibernate.sourceforge.net/
4  ␣␣hibernate-configuration-3.0.dtd">
5  <hibernate-configuration>
6    <session-factory>
7      <property name="hibernate.connection.driver_class">
8        org.postgresql.Driver </property>
9      <property name="hibernate.connection.url">
10       jdbc:postgresql://localhost </property>
11     <property name="hibernate.connection.username">
12       postgres </property>
13     <property name="hibernate.connection.password">
14       postgres </property>
15     <property name="dialect">
16       org.hibernate.dialect.PostgreSQLDialect</property>
17     <property name="hibernate.c3p0.min_size">
18       5 </property>
19     <property name="hibernate.c3p0.max_size">
20       20 </property>
21     <property name="hibernate.c3p0.timeout">
22       300 </property>
23     <property name="hibernate.c3p0.max_statements">
24       50 </property>
25     <property name="hibernate.hbm2ddl.auto">
26       update </property>
27     <property name="hibernate.show.sql">true</property>
28     <mapping resource="Emp.hbm.xml"/>
29   </session-factory>
30 </hibernate-configuration>
```

Do odwzorowania obiektów służą pliki z rozszerzeniem `hbm.xml`. W powyższej konfiguracji mamy jeden taki plik: `Emp.hbm.xml`. Może on wyglądać tak, jak pokazano na Listingu 2.2. Plik konfiguracyjny odwzorowujący obiekty na relacje zawiera następujące informacje:

- nazwa tabeli do przechowywania obiektów klasy (linijka 8),
- sposób odwzorowania identyfikatorów obiektów (linijki 9-11),
- odwzorowanie pól klasy na kolumny w tabeli (linijki 12-14).

Listing 2.2: Przykład pliku odwzorowania klasy trwałej w Hibernate.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC
3      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4      "http://hibernate.sourceforge.net/hibernate
5  ---mapping-3.0.dtd">
6
7      <hibernate-mapping package="pkg">
8        <class name="Emp" table="emp">
9          <id name="id" column="id">
10           <generator class="native"/>
11         </id>
12         <property name="name" column="ename"/>
13         <property name="job" column="job"/>
14         <property name="chiefid" column="chiefid"/>
15       </class>
16     </hibernate-mapping>
```

Klasę trwałą deklarują programiści aplikacji. Może ona wyglądać tak, jak na Listingu 2.3. Pola klasy muszą być zgodne z plikiem konfiguracyjnym odwzorowania klasy. Listing 2.3 zawiera przykład klasy w języku Java dla tabeli `Emp`.

Listing 2.3: Przykład pliku klasy w Javie

```
public class Emp {
    public long id;
    public String name;
    public String job;
    public long chiefId;

    public Empl() {};
    ...
}
```

Druga metoda odwzorowania obiektów na relacje polega na umieszczeniu adnotacji w plikach źródłowych klas. W metodzie tej nie jest wymagany plik konfiguracyjny `hbm.xml`. Dzięki adnotacjom wybrana klasa i jej pola są odwzorowywane na tabele w bazie danych. Przykład takiego odwzorowania pokazano na Listingu 2.4.

Listing 2.4: Przykład pliku źródłowego klasy z adnotacjami używanymi przez Hibernate.

```
@Entity
@Table(name = "emp")
public class Emp {
    @Id
    @GeneratedValue(
      strategy=GenerationType.IDENTITY
    )
    @Column(name = "id")
    public long id;
    @Column(name = "ename")
    public String name;
    @Column(name = "job")
    public String job;
    @Column(name = "chiefid")
    public long chiefid;

    public Emp() {}
}
```

Adnotacja *Entity* wskazuje, że klasa ma być odwzorowana na tabelę. Adnotacja *Table* ustala nazwę tej tabeli, natomiast adnotacje *Column* określają nazwy kolumn.

2.3 Zapytania rekurencyjne

W niniejszej rozprawie przedstawiam wyniki badań nad zapytaniami rekurencyjnymi w kontekście ich generowania oraz wykonywania w narzędziach typu ORM. W tym punkcie pokrótce przedstawię jak zapytania rekurencyjne są zdefiniowane w standardzie SQL i niektórych jego dialektach.

Aplikacje biznesowe często przechowują i przetwarzają dane hierarchiczne i grafowe. Są nimi np. hierarchia pracowników w firmie, rozkład jazdy pociągów czy rozkład lotów. W standardzie SQL:1999 zapytania, które przetwarzają takie dane, nazwano *zapytaniami rekurencyjnymi* [29]. Wcześniejsze standardy SQL niestety nie obejmowały takich zapytań. Dostawcy systemów zaradzania bazami danych próbowali sobie z tym radzić wprowadzając rozmaite elementy do swoich dialektów SQL. Jednym z nich było rozwiązanie firmy Oracle, która w swojej bazie w wersji 5 wprowadziła zapytanie postaci `START WITH ... CONNECT BY`. W 1997 roku firma IBM w swojej bazie DB2 w wersji 7 wprowadziła wyrażenia tabelaryczne - CTE (*Common*

Table Expression) do przetwarzania zapytań rekurencyjnych. Dopiero jednak w 1999 roku zapytania rekurencyjne zostały oficjalnie wprowadzone do standardu SQL. Zaproponowano w nim wyrażenia tabelaryczne do przechowywania tymczasowych rezultatów przetwarzanych zapytań rekurencyjnych. Składnia tych zapytań wygląda następująco:

WITH RECURSIVE cte_0 $(A_{01}, \cdots, A_{0n})$ **AS**
(*seed_query* **UNION ALL** *recursive_query* additional_clauses),
[RECURSIVE] $cte_1(A_{11}, \cdots, A_{1n})$ ...
outer query with cte_i $(i \geq 0)$

Wyrażenie tabelaryczne CTE do generowania zapytań rekurencyjnych zaczyna się od słów kluczowych `WITH RECURSIVE`. Następnie podana jest lista kolumn, które będą występować w obliczanej rekurencyjnie pomocniczej relacji. Zarówno zapytanie podstawowe (*seed query*), jak i rekurencyjne (*recursive query*) mogą składać się z wielu zapytań typu `SELECT` połączonych operatorem `UNION ALL`. Zapytanie podstawowe generuje wyjściowy zbiór rekordów, które zostaną użyte w zapytaniu rekurencyjnym. Wyrażenie tabelaryczne może zawierać uzupełniające klauzule, które zapewniają dodatkową funkcjonalność. Za pomocą klauzuli `SEARCH` można określić kolejność wyszukiwania. Jeśli ma ona wartość `DEPTH FIRST`, przeszukiwanie będzie prowadzone w głąb. Jeśli natomiast wybierzemy `BREADTH FIRST`, procesor zapytań zastosuje metodę wszerz.

Obecnie zapytania rekurencyjne oparte na CTE zaimplementowano w IBM DB2 (od 1997), SQL Anywhere (od 2003), Microsoft SQL Server (od 2005), Firebird (od 2008), PostgreSQL (od 2009) oraz Oracle (także od 2009). Faktycznie zaimplementowana funkcjonalność i składnia zapytań rekurencyjnych dość istotnie się różni w zależności od konkretnego systemu zarządzania bazą danych. Różnice skatalogowano w przeglądowym artykule [29]. Tabele 2.1 oraz 2.2 zawierają podsumowanie tych informacji. Warto także zaznaczyć, że istnieją bardzo popularne systemy relacyjnych baz danych, w których nie zaimplementowano zapytań rekurencyjnych. Jednym z nich jest MySQL.

Tabela 2.1 zawiera informacje o właściwościach zapytań rekurencyjnych oraz o wsparciu ich w wybranych bazach danych ('T' - oznacza Tak, 'N' - oznacza Nie, '-' - niedostępne, '*' - opisane dokładniej w Tabeli 2.2). Porównanie to zostało przeprowadzone w oparciu o dokumentacje, wspieraną informację o błędach oraz testy przeprowadzone na konkretnych systemach zarządzania bazami danych.

Tablica 2.1: Porównanie implementacji zapytań rekurencyjnych

Funkcjonalność		MS SQL	SQL Anywhere	PostgreSQL	DB2	Oracle	Firebird
Wiele zapytań w	kroku inicjującym	T	T	T	T	N*	T
	kroku rekurencyjnym	T	N	N	T	N	T
Inne (niż UNION ALL) operacja na zbiorach w CTE	między inicjującymi zapytaniami	T*	T*	T*	T	-*	T*
	między inicjującymi a rekurencyjnymi zapytaniami	N	N	T*	N	N	N
	między rekurencyjnymi zapytaniami	N	-	-	N	-	N
Odwołanie do CTE	Rekurencyjne CTE do innego rekurencyjnego CTE	T	T*	T	T	T	T
	Rekurencyjne CTE do nierekurencyjnego CTE	T	T	T	T	T	T
	Nierekurencyjne CTE do rekurencyjnego CTE	T	T	T	T	T	T
	Rekurencja wzajemna CTE	N	N	N	N	N	N
Użycie rekurencyjnego CTE w	SELECT	T*	T*	T	T	T	T
	INSERT	T	T*	N	T	T	N
	UPDATE	T	N	N	T*	T	N
	DELETE	T	N	N	N	T	N
	CREATE VIEW	T	T*	N	N	T	N
	OTHER	T*	N	N	N	T*	N
Różne	Zabezpieczenie przed zapętleniem	T*	T*	N	N	T	T*
	Klauzula Search	N	N	N	N	T	N
	Klauzula Cycle	N	N	N	N	T	N
Ograniczenia w kroku rekurencyjnym	Group by	N	N	N	N	N	N
	Having	N	N	N	N	N	N
	Order by	N	N	N	N	T	T
	Distinct	N	N	T	N	N	N
	Funkcje nieagregujące	T*	T	T	N	T*	T
	Funkcje agregujące	N	N	N	N	N	N
	Rekurencja w podzapytaniach	N	N	N	N	N	N
	Left outer join	N	T*	T	N	T*	N
	Right outer join	N	T*	T	N	T*	N
	Full outer join	N	N*	N	N	N	N
	Limiting results	N	T	N	T	T	T
Możliwości użycia w kroku inicjującym	Group by, Order by, Distinct, Having, agregacje i funkcje nieagregujące	T	T	T	T	T	T
Składnia	Obowiązkowe słowo kluczowe RECURSIVE	N	T	T	N	N	T
	Obowiązkowa lista kolumn po słowie kluczowym WITH	N	T	N	T	T	N

Tablica 2.2: Porównanie implementacji zapytań rekurencyjnych - wyjaśnienia odnośników z Tabeli 2.1

Baza danych	**Funkcjonalność**	**Opis**
MS SQL Server 2008	Ochrona przed nieskończoną pętlą	Wskazówka MAXRECURSION przyjmująca wartości: od 0 do 32767
	Inne operacje na zbiorach (oprócz UNION ALL) w CTE w kroku inicjującym	UNION, EXCEPT, lub INTERSECT
	Użycie rekurencji w podzapytaniu - Funkcje nieagregujące	Funkcje niedozwolone: funkcje z parametrami, funkcje zwracające wartość.
	Użycie rekurencyjnego CTE w	MERGE
SQL Anywhere	Ochrona przed nieskończoną pętlą	Parametry bazy max_recursive_iterations (domyślnie 100). Dodatkowo dwa różne typy połączenia MAX_TEMP_SPACE zapobieganie użyciu nieograniczonej ilości miejsca dla plików tymczasowych przez serwer. TEMP_SPACE_LIMIT_CHECK ogranicza rozmiar plików tymczasowych na poziomie połączenia z bazą danych.
	Inne operacje na zbiorach (oprócz UNION ALL) w CTE w kroku inicjującym	UNION, EXCEPT, lub INTERSECT
	LEFT, RIGHT OUTER JOIN	Odwołanie do tabeli rekurencyjnej nie może być po stronie zezwalającej `null` w złączeniu zewnętrznym.
	Odwołanie z rekurencyjnego CTE do innego rekurencyjnego CTE	Działa, ale dokumentacja różnie o tym mówi.
PostgreSQL	Inne operacje na zbiorach (oprócz UNION ALL) w CTE w kroku inicjującym	UNION, EXCEPT, lub INTERSECT
	Inne operacje na zbiorach (oprócz UNION ALL) w CTE w obu krokach	UNION
	LEFT,RIGHT OUTER JOIN	Odwołanie do tabeli rekurencyjnej nie może być po stronie zezwalającej `null` w złączeniu zewnętrznym.
DB2	Użycie rekurencyjnego CTE w UPDATE	Nie ma nic oficjalnego na ten temat (ale jest sugerowane przez twórców bazy DB2) WITH v AS (..) SELECT COUNT(∗) FROM OLD TABLE(UPDATE ..)
Oracle	Funkcje analityczne są dozwolone w kroku rekurencyjnym	Funkcje analityczne są dozwolone.
	Wiele zapytań w kroku inicjującym	Opisane w dokumentacji ale nie zaimplementowane.
	Inne operacje na zbiorach (oprócz UNION ALL) w CTE w kroku inicjującym	Właściwość opisana w dokumentacji, ale nie zaimplementowana. Dokumentacja wspomina UNION, INTERSECT and MINUS.
	LEFT, RIGHT OUTER JOIN	Odwołanie do tabeli rekurencyjnej nie może być po stronie zezwalającej `null` w złączeniu zewnętrznym.
	Użycie rekurencyjnego CTE w	CREATE TABLE.
Firebird	Inne operacje na zbiorach (oprócz UNION ALL) w CTE w kroku inicjującym	UNION
	Ochrona przed nieskończoną pętlą	Maksymalny poziom rekurencji 1024.

Rozdział 3

Benchmark

W niniejszym rozdziale opiszemy benchmark służący do pomiaru wydajności implementacji zapytań rekurencyjnych zgodnych ze standardem SQL:1999. Istnieje wiele firm oraz organizacji, które definiują standardy benchmarków, m.in. SPEC (*Standard Performance Evaluation Cooperation*) [40]. W dziedzinie baz danych najpopularniejsze benchmarki definiuje konsorcjum TPC (*Transaction Processing Performance Council*) [12, 17]. Niestety żaden z benchmarków dla relacyjnych baz danych (także tych definiowanych przez TPC) nie uwzględnia zapytań rekurencyjnych wg standardu SQL:1999.

Postanowiliśmy zapełnić te lukę. W ramach prac nad niniejszą rozprawą przygotowaliśmy odpowiedni benchmark. Jego celem jest porównanie implementacji zapytań rekurencyjnych SQL:1999 a także alternatywnych rozwiązań zaproponowanych w niniejszej rozprawie doktorskiej. Pierwszą wersję tego benchmarku wraz z wynikami jego ewaluacji dla wybranych komercyjnych systemów zarządzania bazami danych przedstawiono w [29].

3.1 Dane

Schemat danych wykorzystywany w omawianym benchmarku obejmuje cztery tabele. Jedna z nich (`EMP`) służy do testowania wydajności zapytań do struktur hierarchicznych. Pozostałe trzy (`CITIES`, `FLIGHTS` i `TRAINS`) reprezentują dane sieciowe (grafowe). Stanowią więc źródło danych do testowania wydajności zapytań do struktur grafowych. Poniżej wymieniono kolumny każdej z tych tabel. Skrypt tworzący strukturę tych tabel znajduje się w Dodatku C.

`EMP`:

> `empno, mgr, ename, lname, job, hiredate, sal, comm, deptno, sex.`

```
CITIES:
    cid, city.

FLIGHTS:
    departure, arrival, carrier, fid, price.

TRAINS:
    departure, arrival, railline, tid, price.
```

Przewidzieliśmy 20 wariantów danych w tabeli `EMP` i stworzyliśmy generator danych dla wszystkich tych wariantów. Liczba tych wariantów wynika z zastosowania wszystkich kombinacji następujących parametrów:

- 4 poziomy zagłębienia hierarchii: 5, 10, 15, 20, oraz
- 5 liczb rekordów: 1000, 10000, 100000, 1000000, 10000000.

Trzy pozostałe tabele (służące do testowania zapytań do struktur grafowych) mają stałą zawartość o następujących liczbach rekordów.

- `CITIES` - 200 rekordów,
- `TRAINS` - 800 rekordów,
- `FLIGTS` - 800 rekordów,

Jakkolwiek te liczby rekordów wydają się nierealistycznie małe, jednak jak wynika z [29] są one zupełnie wystarczające, aby dokładnie sprawdzić wydajność istniejących komercyjnych i niekomercyjnych implementacji zapytań rekurencyjnych SQL:1999 do struktur grafowych. Każde zapytanie z benchmarku opisanego w niniejszym rozdziale wybiera tylko jedną hierarchię.

3.2 Zapytania

3.2.1 Q1: podwładni danej osoby

Listing 3.1: Zapytanie Q1

```
WITH employees(ename, empno, mgr) AS
  ( SELECT ename, empno, mgr
      FROM emp
     WHERE ename = 'SMITH'
UNION ALL
    SELECT e.ename, e.empno, e.mgr
      FROM emp e, employees p
     WHERE e.mgr = p.empno
)
SELECT ename
FROM   employees;
```

3.2.2 Q2: główny szef danej osoby

Listing 3.2: Zapytanie Q2

```
WITH boss(ename, empno, mgr) AS (
  SELECT empno, mgr, ename
    FROM emp
   where ename = 'SMITH'
   union all
  SELECT e.empno, e.mgr, e.ename
    FROM emp e, boss p
   WHERE e.empno = p.mgr
)
select * from boss
where mgr is null;
```

3.2.3 Q3: połączenia lotnicze z danego miasta

Listing 3.3: Zapytanie Q3

```
WITH destinations (origin, departure, arrival, connections) AS
  (SELECT a.departure, a.departure, a.arrival, 1
   FROM flights a, cities c
   WHERE a.departure = c.cid AND c.city = 'Toronto'
  UNION ALL
   SELECT r.origin, b.departure, b.arrival,
          r.connections + 1
   FROM destinations r, flights b
   WHERE r.arrival = b.departure
     AND r.flight_count < I )
SELECT count(*) FROM destinations
```

Występujący w tym zapytaniu parametr I to ograniczenie liczby możliwych przesiadek. W czasie ewaluacji benchmarku przyjmuje on wszystkie wartości całkowite z przedziału od 1 do 10.

3.2.4 Q4: połączenia lotnicze z danych dwóch miast

Listing 3.4: Zapytanie Q4

```
WITH destinations (departure, arrival, connections, cost) AS
 (SELECT a.departure, a.arrival, 0, price
  FROM flights a, cities c
  WHERE a.departure = c.cid
    AND c.city = 'Toronto' OR c.city = 'Warsaw'
 UNION ALL
  SELECT r.departure, b.arrival,
    r.connections + 1, r.cost + b.price
  FROM destinations r, flights b
  WHERE r.arrival = b.departure
    AND r.connections < I)
SELECT count(*) FROM destinations
```

Podobnie jak w przypadku zapytania Q3 parametr I to ograniczenie liczby możliwych przesiadek przyjmujący wszystkie wartości całkowite z przedziału od 1 do 10.

3.2.5 Q5: połączenia lotnicze i kolejowe z danego miasta

Listing 3.5: Zapytanie Q5

```
WITH destinations(departure, arrival, connections,
     flights, trains, cost) AS
 (SELECT f.departure, f.arrival, 0, 1, 0, price
  FROM flights f, cities c
  WHERE f.departure = c.cid AND c.city = 'Toronto'
 UNION ALL
  SELECT t.departure, t.arrival, 0, 0, 1, price
  FROM trains t, cities c
  WHERE t.departure = c.cid AND c.city = 'Toronto'
 UNION ALL
  SELECT r.departure, b.arrival, r.connections+1 ,
         r.flights+1, r.trains,
         r.cost + b.price
  FROM  destinations r, flights b
  WHERE r.arrival = b.departure
    AND r.connections<I
 UNION ALL
  SELECT r.departure, v.arrival, r.connections+1 ,
         r.flights, r.trains + 1, r.cost + v.price
  FROM destinations r, trains v
  WHERE r.arrival = v.departure
    AND r.connections<I )
SELECT count(*) FROM destinations
```

Znaczenie i wartości parametru I są takie same jak w zapytaniach Q3 i Q4.

3.2.6 Q6: połączenia lotnicze z danego miasta bez ograniczenia przesiadek

Listing 3.6: Zapytanie Q6

```
WITH destinations (departure, arrival, connections, cost) AS
 (SELECT a.departure, a.arrival, 0, price
  FROM flights a, cities c
  WHERE a.departure = c.cid
    AND c.city = 'Toronto' OR c.city = 'Warsaw'
 UNION ALL
  SELECT r.departure, b.arrival, r.connections+1,
         r.cost + b.price
  FROM destinations r, flights b
  WHERE r.arrival = b.departure)
SELECT count(*) FROM destinations
```

3.2.7 Q7: suma kolejnych liczba naturalnych

Listing 3.7: Zapytanie Q7

```
WITH t(n) AS (
   VALUES (1)
 UNION ALL
   SELECT n+1 FROM t WHERE n < 100*I
)
SELECT sum(n) FROM t;
```

Parametr I stanowi ograniczenie przedziału sumowania. Podobnie jak dla pozostałych zapytań przyjmuje on wszystkie wartości całkowite z przedziału od 1 do 10.

Rozdział 4

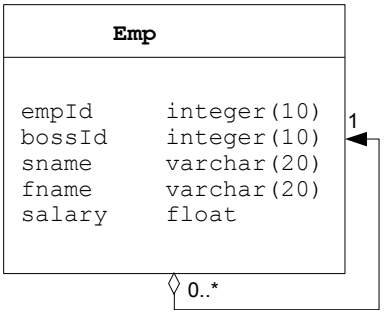
Interfejs programisty ORM dla zapytań rekurencyjnych

4.1 Motywacja

Interfejs programisty Hibernate nie ma udogodnień do odpytywania struktur rekurencyjnych. Dane hierarchiczne i grafowe są jednak bardzo często przetwarzane przez aplikacje. Przykładami takich danych są hierarchia pracowników firmy, dane genealogiczne, rozkłady jazdy pociągów, czy harmonogramy lotów samolotów. Rozważmy dwa przykłady takich danych. Pierwszy z nich to dane osobowe pracowników firmy z najbardziej nas interesującą informacją o hierarchii zależności. Schemat tych danych pokazano na rys. 4.1, a przykładowe dane zawarto w tabeli 4.1. Drugi z nich zawiera dane grafowe o połączeniach lotniczych, których schemat znajduje się na rys. 4.2, a przykładowe rekordy widać w tabeli 4.2.

Listing 4.1: Zapytanie rekurencyjne o podwładnych Smitha

```
WITH RECURSIVE rcte (
  SELECT fname, sname, empId, False as isSubordinate
    FROM Emp
    WHERE sname = 'Smith'
  UNION
    SELECT e.fname, e.sname, e.empId,
           True as isSubordinate
    FROM Emp e JOIN rcte r ON (e.bossId = r.empId)
  )
SELECT name, surname
FROM rcte WHERE rcte.isSubordinate = True
```

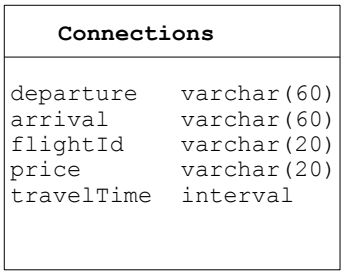


Rysunek 4.1: Przykład schematu danych hierarchicznych

Tablica 4.1: Przykładowe dane hierarchiczne składowane w tabeli `Emp`

empId	**bossId**	**sname**	**fname**	**salary**
7369	7902	Green	John	100
7499	7698	Brown	George	210
7521	7698	Christie	Andrew	210
7566	7839	Jones	Brandon	360
7654	7698	Ford	Carl	210
7698	7839	Blake	Ernest	360
7782	7839	Bell	Gordon	360
7788	7839	Willis	James	360
7839		Smith	John	500
7844	7698	Turner	Johnathan	210
7902	7698	Adams	Trevor	210
7900	7566	Miller	Kyle	150

Na listingu 4.1 przedstawiono zapytanie rekurencyjne do tabeli `Emp` (rys. 4.1) o bezpośrednich i pośrednich podwładnych Smitha. Niestety programista aplikacyjny korzystający z Hibernate nie ma możliwości zadania zapytania rekurencyjnego. Aby poznać podwładnych Smitha, musi on po stronie klienta zaprogramować pętlę, która dla każdego pracownika będzie wysyłać zapytania o jego bezpośrednich podwładnych. Przykład takiej pętli pokazano na listingu 4.2. Chociaż sam kod tej pętli nie jest skomplikowany, jednak czas jej przetwarzania jest nie do zaakceptowania. Dla każdego zwróconego obiektu należy bowiem sprawdzić, czy istnieją w bazie pracownicy, których danych pracownik jest szefem. To właśnie jest przyczyną długiego czasu przetwarzania tej pętli. W efekcie oznacza to, że zapytanie jest wysyłane do bazy danych tyle razy, ilu pracowników jest w strukturze, o którą pytamy. Z testów na tabeli zawierającej dane 900 pracowników wynika, że zapytanie



Rysunek 4.2: Przykład schematu danych grafowych

Tablica 4.2: Przykładowe dane grafowe składowane w tabeli `Conns`

departure	**arrival**	**flightId**	**price**	**travelTime**
Paris	Phoenix	TW 123	120	6h 15min
Paris	Houston	TW 120	130	8h 30min
Phoenix	Huston	PW 230	100	3h 10min
Huston	Chicago	RW 121	90	2h 45min
Huston	Dallas	RW 122	80	3h 00min
Dallas	Chicago	DW 80	110	2h 30min
Chicago	Atlanta	CH 542	220	2h 45min
Chicago	Berlin	CH 543	360	7h 15min
Paris	Berlin	TW 118	300	1h 10min
Dallas	Berlin	DW 90	350	5h 45min
Berlin	Boston	YW 421	100	6h 00min
Chicago	Boston	CH 544	250	2h 15min

rekurencyjne (por. listing 4.1) wykonuje się po stronie systemu zarządzania bazą danych 20 razy szybciej niż kod z listingu 4.2.

Listing 4.2: Pętla pobierająca dane podwładnych wskazanego pracownika (kod natywny Hibernate)

```
while (!stack.isEmpty()) {
  Empl emp = (Empl) stack.firstElement();
  List<Empl> emps = session.createQuery(
    "from Empl e where e.bossId = " +
        emp.getId()).list();
  for (int i = 0; i < emps.size() ; i++)
    stack.push(emps.get(i));
  stack.remove(0);
}
```

4.2 Proponowane rozwiązanie

Interfejs programistyczny będący przedmiotem niniejszego rozdziału po raz pierwszy zaprezentowano w artykule [33]. Szczegóły techniczne jego prototypowej implementacji można znaleźć w pracy magisterskiej [32]. Hibernate korzysta z dwóch sposobów odwzorowania obiektów na tabele: za pomocą plików XML i adnotacji (por. p. 2.2). Analogicznie potraktowaliśmy zatem odwzorowanie zapytań rekurencyjnych. Można je udostępnić programiście aplikacyjnemu za pomocą pliku XML lub odpowiednich adnotacji. Prototypowa implementacja obsługuje trzy systemy zarządzania bazami danych: IBM DB2, Oracle i PostgreSQL. Na listingu 4.3 pokazano zawartość pliku `ncu-extras-mapping`, tj. definicję typu dokumentu (DTD) dla plików konfiguracyjnych XML z listingów 4.4 i 4.5.

Listing 4.3: Plik `ncu-extras-mapping.dtd` zawierający definicję typu dokumentu dla pliku konfiguracyjnego

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rcte[
  <!ELEMENT rcte(rcteTable, tables, recursive-condition,
                 contats*, constans*, summands*, filter,
                 outer*)>
  <!ELEMENT rcteTable (#PCDATA)>
    <!ATTLIST rcteTable name CDATA #REQUIRED
                        max-level CDATA #REQUIRED
                        cycle CDATA #IMPLIED>
  <!ELEMENT tables(table, join-on*)>
  <!ELEMENT table (#PCDATA)>
  <!ELEMENT join-on (#PCDATA)>
  <!ELEMENT recursive-condition(on, to)>
  <!ELEMENT on (#PCDATA)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT contats(concat)>
  <!ELEMENT concat (#PCDATA)>
  <!ELEMENT constants(conc)>
  <!ELEMENT conc (#PCDATA)>
  <!ELEMENT summands(sum)>
  <!ELEMENT sum (#PCDATA)>
  <!ELEMENT filter (#PCDATA)>
    <!ATTLIST filter section (seed|loop)>
  <!ELEMENT outer(filter, property*)>
  <!ELEMENT filter (#PCDATA)>
```

```
  <!ELEMENT property (#PCDATA)>
    <!ATTLIST property concat*|constant*|summand CDATA>
]>
```

Zgodnie z definicją typu dokumentu z listingu 4.3 plik konfiguracyjny XML składa się z elementu `<rcte>`. Jego elementy podrzędne określają postać zapytania rekurencyjnego. Są to [32]:

`rcteTable` ma atrybuty takie, jak `name` (nazwa tworzonej tabeli), `max-level` (liczba zagnieżdżeń części rekurencyjnej), oraz `cycle` (czy mogą wystąpić cykle?).

`tables` wskazuje tabele (`table`), które są wykorzystane podczas generowania zapytania rekurencyjnego. Może także zawierać warunki ich łączenia (`join-on`).

`recursive-condition` określa warunek łączący część inicjującą zapytanie z jego częścią rekurencyjną.

- `on` wskazuje atrybut klasy bazowej, dzięki któremu możliwe jest złączenie jej z warunkiem rekurencyjnym CTE.
- `to` wskazuje atrybut powstałego rekurencyjnego CTE, który umożliwia złączenie z częścią bazową.

`concats`, `conc` podają atrybuty napisowe, które należy sklejać w trakcie obliczania zapytania rekurencyjnego.

`constants`, `const` określają pola stałe, których wartość będzie pojawiać się w wyniku zapytania.

`summands`, `sum` podają atrybuty liczbowe, które należy sumować w trakcie obliczania zapytania rekurencyjnego.

`filter` określa warunki, z jakimi tworzone są rekurencyjne wyrażenia tabelaryczne. Można w nim użyć funkcji `$Param`, która przekazuje parametry użytkownika do kodu aplikacji Javy. Ma atrybut `section`, który przyjmuje wartość `seed` lub `loop`, oznaczający, w jakiej części zapytania, odpowiednio inicjującej lub rekurencyjnej, ma znaleźć się dany warunek.

`outer` definiuje zewnętrzny `SELECT` powstałego polecenia SQL.

- `filter` działa podobnie, jak powyżej opisany znacznik o takiej samej nazwie, jednak warunki tu określone dotyczą tylko zewnętrznej części zapytania rekurencyjnego.

`property` wskazuje pola, jakie mają znaleźć się w rezultacie wywołania zapytania SQL.

Przykładowy plik konfiguracyjny XML pokazano na listingu 4.4. Na jego podstawie generowane jest zapytanie rekurencyjne o funkcjonalności zapytania z listingu 4.1. Z kolei listing 4.5 zawiera przykładowy plik konfiguracyjny XML dla danych grafowych o schemacie takim jak na rys. 4.2.

Listing 4.4: Plik konfiguracyjny (`subordinate.rcte.xml`) do generowania zapytań rekurencyjnych dla danych hierarchicznych w Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ncu-extras
  SYSTEM "ncu-extras-mapping.dtd">
<rcte>
  <rcteTable name="subordinates" max-level="4"/>
  <tables>
    <table>Emp</table>
  </tables>
  <recursive-condition>
    <on>Emp.bossId</on>
    <to>Emp.empId</to>
  </recursive-condition>
  <concats>
    <conc>Emp.empId</conc>
    <conc>Emp.sname</conc>
  </concats>
  <filter section="seed">
    Emp.sname = $Param(sname)
  </filter>
</rcte>
```

Listing 4.5: Plik konfiguracyjny do generowania zapytań rekurencyjnych dla danych grafowych w Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ncu-extras
  SYSTEM "ncu-extras-mapping.dtd">
<rcte>
  <rcteTable name="travel" max-level="4"/>
  <tables>
    <table>Conns </table>
  </tables>
```

```
  <recursive-condition>
    <on>Conns.cityStart </on>
    <to>Conns.cityEnd </to>
  </recursive-condition>
  <concats>
    <conc>Conns.id </conc>
    <conc>Conns.cityEnd </conc>
  </concats>
  <constants>
    <const>Conns.cityStart </const>
  </constants>
  <summands>
    <sum>Conns.travelTime </sum>
  </summands>
  <filter section="seed">
    Conns.cityStart = 'Torun' </filter>
  <outer>
    <filter>
      Conns.cityEnd = 'Zakopane' </filter>
    <property concat="sciezka_id">
      Conns.id </property>
    <property concat="sciezka">
      Conns.cityEnd </property>
    <property constant="miasto_poczatkowe">
      Conns.cityStart </property>
    <property summand="czas_podrozy">
      Conns.travelTime </property>
  </outer>
</rcte>
```

Druga metoda konfiguracji zapytań rekurencyjnych polega na dodaniu adnotacji do pliku w Javie. Nazwy adnotacji są podobne do nazw znaczników z pliku konfiguracyjnego XML:

`RecursiveQuery` wskazuje, że dana klasa zawiera dane, na których podstawie będzie można budować zapytanie rekurencyjne. Jej atrybut `maxLevel` ustala maksymalną liczbę zagnieżdżeń części rekurencyjnej, a `cycle` włącza wykrywanie cykli w danych.

`Tables` określa tabele, które są wykorzystane podczas generowania zapytania rekurencyjnego. Nazwy tabel są podane w metadanej `name`. W `joinOn` zapisane są złączenia tabel.

`RecursiveCondition` wskazuje warunek łączący część inicjującą zapytanie z jego częścią rekurencyjną. Pola, z których warunek ten jest budowany należy podać w atrybutach `on` oraz `to`.

`Concats` podaje atrybuty napisowe, które należy sklejać w trakcie obliczania zapytania rekurencyjnego.

`Constants` określa pola stałe, których wartość będzie pojawiać się w wyniku zapytania.

`Summands` podaje atrybuty liczbowe, które należy sumować w trakcie obliczania zapytania rekurencyjnego.

`Filter` zawiera listy predykatów, które mają się znaleźć w jednej z trzech części rekurencyjnego CTE (zgodnie z atrybutami `seed`, `loop` i `outer`).

`Column` ustala pola wynikowe zapytania rekurencyjnego.

Podobnie jak w pliku konfiguracyjnym zapytanie można parametryzować za pomocą funkcji `$Param`. Adnotacja następującej postaci pozwala na podanie nazwy parametru `name` jako atrybutu odpowiedniej metody.

```
@Filter(seed = "Emp.sname = $Param(name)")
```

Adnotacja `Column` określa, jakie kolumny mają znaleźć się w wyniku zapytania rekurencyjnego. W tej metadanej znajdują się atrybuty takie, jak `name` (nazwa kolumny klasy bazowej), `concat`, `constant` i `summand` (nazwa wybranego pola, jeśli wystąpiło ono w jednej z adnotacji `Concats`, `Constants` lub `Summands`). Jeśli w danej klasie żadne z pól nie będzie oznaczone tą metadaną, to w wyniku znajdą się obiekty z wszystkimi polami wygenerowanymi automatycznie.

Na listingu 4.6 przedstawiono wykorzystanie adnotacji w klasie Javy. Na ich podstawie zostanie wygenerowane zapytanie rekurencyjne dla danych hierarchicznych z tabeli 4.1. Listing 4.7 zawiera podobne adnotacje dla danych grafowych z tabeli 4.2.

Listing 4.6: Przykład użycia adnotacji w klasie Javy do generowania zapytań rekurencyjnych dla danych hierarchicznych

```
import org.ncu.hibernate.annotations.*;

@RecursiveQuery(cycle = false)
@Tables(name = "Emp")
@RecursiveCondition(
  on = "Emp.bossId",
```

```
  to = "Emp.empId"
)
@Filter(
  seed = "Empl.sname = 'Travolta'"
)
public class Subordinates {}
```

Listing 4.7: Przykład użycia adnotacji w klasie Javy do generowania zapytań rekurencyjnych dla danych grafowych

```
import org.ncu.hibernate.annotations.*;

@RecursiveQuery(maxLevel = 4)
@Tables(name = "Conns")
@RecursiveCondition(
  on = "Conns.cityStart",
  to = "Conns.cityEnd"
)
@Concats(
  field = {
    "Conns.id",
    "Conns.cityEnd",
    "Conns.cityStart"
  }
)
@Summands(
  field = "Conns.travelTime"
)
@Constants(
  field = "Conns.cityStart"
)
@Filter(
  seed = "Conns.cityStart = $Param(cityStart)",
  outer = "Conns.cityEnd = 'Zakopane'"
)

public class Travel {
    @Column(concat = "Conns.id")
    public String sciezkaId;
    @Column(concat = "Conns.cityEnd")
    public String sciezka;
    @Column(constant = "Conns.cityStart")
```

```
    public String miastoPoczatkowe;
    @Column(summand = "Conns.travelTime")
    public double czasPodrozy;
}
```

Pakiet `org.ncu.hibernate` zawiera implementację generatorów zapytań rekurencyjnych w Hibernate. W jego skład wchodzą klasy i metody odczytujące zadaną konfigurację i budujące zapytanie rekurencyjne SQL. Zapytanie to wysyłane jest do bazy danych, skąd pobierany jest jego wynik. Rozwiązanie to współpracuje z bazami danych Oracle, PostgreSQL oraz IBM DB2. Dodatkowo należy zdefiniować plik XML, w którym znajdą się wszystkie dane dotyczące schematów. Plik ten nazywa się `ncu_extras.cfg.xml`. Na listingu 4.8 przedstawiono definicję typu dokumentu dla tego pliku. Definicja ta znajduje się w pliku `ncu-extras-configuration.dtd`.

Listing 4.8: Zawartość pliku `ncu-extras-configuration.dtd` z definicją typu dokumentu

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ncu-extras[
  <!ELEMENT ncu-extras(log-value,
                       print-query-only,
                       recursive-mapping+)>
  <!ELEMENT log EMPTY>
    <!ATTLIST log value false|true #REQUIRED>
  <!ELEMENT print-query-only EMPTY>
    <!ATTLIST print-query-only value false|true #REQUIRED>
  <!ELEMENT recursive-mapping EMPTY>
    <!ATTLIST recursive-mapping resource CDATA #REQUIRED>
]>
```

Zgodnie z definicją typu dokumentu z listingu 4.8 głównym elementem pliku XML jest `ncu-extras`. Znacznik `recursive-mapping` zawiera informacje, na których podstawie będzie można generować zapytania rekurencyjne. W atrybucie `resource` podaje się nazwę pliku konfiguracyjnego lub klasę z adnotacjami. Dodatkowo można ustawić dwa następujące znaczniki:

`log` powoduje zapisanie zapytania rekurencyjnego i czasu jego wykonania w dzienniku aplikacji, jeśli jego atrybut `value` jest ustawiony na `true`.

`print-query-only` wskazuje, czy zapytanie ma zostać wykonane, czy tylko wypisane na standardowe wyjście.

Przykład pliku XML zgodnego z definicją typu dokumentu z listingu 4.8 pokazano na listingu 4.9.

Listing 4.9: Przykład pliku `ncu_extras.cfg.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ncu-extras
  SYSTEM "ncu-extras-configuration.dtd">
<ncu-extras>
  <log value="true" />
  <print-query-only value="false" />
  <recursive-mapping resource="subordinate.rcte.xml" />
  <recursive-mapping resource="travel.rcte.xml" />
</ncu-extras>
```

Plik konfiguracyjny jest wczytywany przez metodę `loadRCTEConfiguration` z klasy `NCUExtrasLoader` z pakietu `org.ncu.hibernate`. Metoda ta (`loadRCTEConfiguration`) z kolei jest wywoływana w metodzie `configure` klasy `Configuration`. Metoda `configure` służy do wczytywania plików konfiguracyjnych `hibernate.cfg.xml` do Hibernate. Jest to jedna z dwóch modyfikacji oryginalnej metody Hibernate. Dzięki tej modyfikacji można tworzyć zapytania rekurencyjne za pomocą interfejsu opisywanego w niniejszym rozdziale.

W kolejnym kroku w metodzie `loadRCTEConfiguration` plik konfiguracyjny z listingu 4.9 jest odczytywany i analizowany składniowo. Informacja o schemacie ze znacznika `recursive-mapping` jest przekazywana do konstruktora klasy `RecursiveManagerXML` lub `RecursiveManagerAnnotation`, w zależności czy dany schemat jest opisany przez plik XML, czy przez adnotacje. Obie te klasy mają metody do analizowania składni podanego źródła i na jej podstawie tworzą obiekt klasy `RecursiveQueryData`. Obiekt ten zawiera informacje potrzebne do wygenerowania zapytania rekurencyjnego. Ponadto, jest on składowany jako wartość w instancji słownika `HashMap`. Kluczem tego słownika jest nazwa tabeli, która zostanie wygenerowana w czasie wywołania stworzonego polecenia SQL. Aby zbudować już samo zapytanie rekurencyjne w języku SQL, zmieniliśmy oryginalną klasę Hibernate `SessionImpl` w pakiecie `org.hibernate.impl`. Dodano do niej metodę `prepareRcte`, która jako argument pobiera nazwę tabeli podanej w pliku konfiguracyjnym. To jest druga zmiana w oryginalnych klasach Hibernate. Na podstawie tej nazwy z obiektu klasy `HashMap` pobierana jest instancja klasy `RecursiveQueryData`, a następnie podawana jest ona jako argument w konstruktorze klasy. W klasie tej znajdują się metody `setParam` oraz `getRecursiveObjects`, które, odpowiednio, ustawiają parametr podany w schemacie poprzez funkcję `$Param` oraz zwracają rezultat wykonania powstałego zapytania rekurencyjnego. Na listingu 4.10 zobrazowano użycie tych metod.

W zależności od tego, z jaką bazą danych nastąpi połączenie (PostgreSQL, IBM DB2, Oracle), wywołanie metody `getRecursiveObject` spowoduje, że powstaną instancje jednej z klas odpowiadających za stworzenie rekurencyjnego zapytania. Są nimi `RecursiveQueryBuilderPostgres`, `RecursiveQueryBuilderDB2` lub `RecursiveQueryBuilderORACLE`. Każda z tych klas implementuje interfejs o nazwie `RecursiveQueryBuilder`, który zawiera definicje metod tworzących polecenie w języku SQL. Wygenerowane polecenie jest jednym z argumentów metody `prepareView` należącej do klasy `RecursiveViewBuilder`. Jest ona pośrednio wywoływana poprzez funkcję `getRecursiveObjects`. W niej rekurencyjne zapytanie jest przesyłane do bazy danych za pomocą `createSQLQuery` będącej metodą natywną samego Hibernate. Następnie zwracane są wyniki zapytania.

Listing 4.10: Generowanie zapytania rekurencyjnego

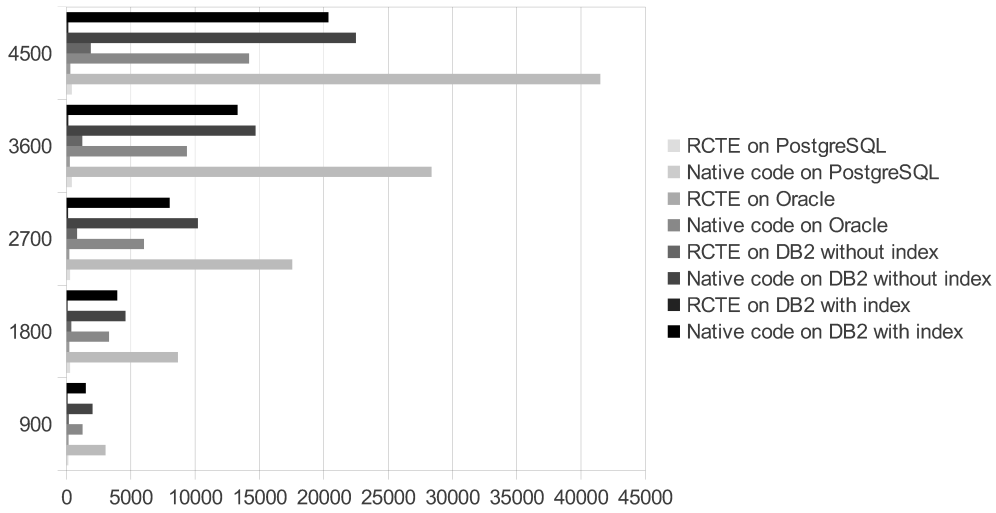
```
try {
  SessionFactory session = HibernateUtil
    .getSession();
  Session s = session.openSession();
  RecursiveQuery rqSubords = s
    .prepareRcte("Subordinate");
  rqSubords.setParam(1,"Smith");
  ArrayList<ArrayList<Object>> rcteList =
    rqSubords.getRecursiveObjects();
  Iterator<Object> iter = rcteList.get(0)
    .iterator();
  while(iter.hasNext()) {
    RcteObject obj = (RcteObject)iter.next();
    System.out.println(obj);
  }
  s.close();
} catch (Exception e) {
  e.printStackTrace();
}
```

4.3 Testy

Opisane powyżej modyfikacje Hibernate przetestowano dla trzech baz wspierających zapytania rekurencyjne: PostgreSQL, Oracle oraz IBM DB2. Testy przeprowadzono na tabeli zawierającej dane pracowników wybranej firmy. Były podzielone na pięć przypadków, w których tabela ta zawierała ko-

lejno 900, 1800, 2700, 3600 oraz 4500 rekordów z odpowiednio siedmioma, ośmioma, dziewięcioma, dziesięcioma oraz jedenastoma poziomami hierarchii. Zostało użyte zapytanie 3.1 z benchmarku z rozdziału 3 dla powyżej opisanej konfiguracji danych. Wyniki testów przedstawiono na rysunku 4.3. W tabelach 4.3, 4.4 oraz 4.5 zaprezentowano czasy wykonania zapytań rekurencyjnych w Hibernate z tymi trzema systemami zarządzania bazami danych. Zostały one porównane z czasem pobierania danych za pomocą natywnej metody Hibernate przedstawionej na listingu 4.2

Kolumna o nazwie *Procentowe przyspieszenie* zawiera zapisany w procentach stosunek uzyskanego rezultatu z kolumny drugiej do rezultatu z kolumny trzeciej. Umożliwia to oszacowanie faktycznego zysku ze stosowania jednej bądź drugiej metody.



Rysunek 4.3: Wyniki testów wykonania zapytań rekurencyjnych

Tablica 4.3: Wyniki testów Hibernate z bazą danych PostgreSQL

Liczba rekordów	Hibernate RCTE	Kod natywny Hibernate	Procentowe przyspieszenie
900	143 ms	3033 ms	4.71 %
1800	266 ms	8669 ms	2.61 %
2700	271 ms	17550 ms	1.54 %
3600	405 ms	28391 ms	1.43 %
4500	414 ms	41500 ms	1.00 %

Tablica 4.4: Wyniki testów Hibernate z bazą danych Oracle

Liczba rekordów	Hibernate RCTE	Kod natywny Hibernate	Procentowe przyspieszenie
900	162 ms	1237 ms	13.10 %
1800	201 ms	3303 ms	6.08 %
2700	232 ms	6012 ms	3.85 %
3600	256 ms	9354 ms	2.74 %
4500	289 ms	14192 ms	2.04 %

Tablica 4.5: Wyniki testów Hibernate z bazą danych IBM DB2

Liczba rekordów	Bez indeksu		
	RCTE	Kod natywny	Procentowe przyspieszenie
900	171 ms	2026 ms	8.44 %
1800	371 ms	4584 ms	8.09 %
2700	820 ms	10225 ms	8.02 %
3600	1233 ms	14698 ms	8.39 %
4500	1887 ms	22495 ms	8.39 %

Liczba rekordów	Z indeksem na `chief_id`		
	RCTE	Kod natywny	Procentowe przyspieszenie
900	78 ms	1498 ms	5.20 %
1800	86 ms	3936 ms	2.18 %
2700	111 ms	8011 ms	1.38 %
3600	135 ms	13291 ms	1.01 %
4500	139 ms	20356 ms	0.68 %

4.4 Analiza zaproponowanego rozwiązania

Cały czas trwają pracę nad optymalizacją zapytań rekurencyjnych. Polegają one m.in. na próbie przeniesienia warunku z kroku rekurencyjnego do kroku

podstawowego, na usuwaniu zduplikowanych wierszy z tabel pośrednich, zakładaniu indeksów na kolumnach tabeli podstawowej (tzn. tej, która zawiera dane hierarchiczne) i rekurencyjnej (tzn. tej, która powstaje w wyniku zapytania rekurencyjnego używającego tabeli podstawowej), na których opiera się warunek złączenia [28]. Inną z metod optymalizacji jest wykorzystanie techniki *przeniesienia predykatów* [8, 9]. W trakcie przetwarzania zapytania rekurencyjnego duże znaczenie ma przechowywanie struktur pośrednich. Technika przeniesienia predykatów polega na redukcji tych struktur poprzez odpowiednie przepisanie zapytania w taki sposób, aby przesunąć predykaty z zapytania zewnętrznego do zapytania inicjującego CTE. Jeszcze inna metoda [15, 16] polega na modyfikacji planu wykonania zapytania rekurencyjnego w każdym kroku iteracji. Komercyjne systemy zarządzania bazami danych ustalają bowiem plan wykonania w kroku początkowym i nie zmieniają go w krokach iteracyjnych.

Wiele systemów zarządzania bazami danych wspiera zapytania rekurencyjne [29]. Chociaż w narzędziach typu ORM również można takie zapytania wykonywać, jednak sposób ich wykonania nie jest optymalny. Z tego powodu opracowaliśmy opisane w tym rozdziale rozszerzenia ORM. Rozszerzenie to zostało prototypowo zaimplementowane i przetestowane w Hibernate, który jest obecnie najpopularniejszym ORM dla Javy. Można je również zrealizować w innych narzędziach odwzorowania obiektowo-relacyjnego [34, 7]. W pracy [35] narzędzie to zostało dodatkowo rozszerzone o możliwość tworzenia zapytań rekurencyjnych przy użyciu pozastandardowej klauzuli `CONECT BY` zaimplementowanej przez firmę Oracle.

Na rysunku 4.3 można zauważyć, że czas wykonania zapytań rekurencyjnych proponowaną metodą jest znacznie szybszy od metod natywnych dostępnych w Hibernate. Wynika to z faktu, że dla metod natywnych do bazy danych było wysyłanych tyle zapytań, ile obiektów ma dana struktura. Wyniki te potwierdzają wnioski z tabel 4.3, 4.4 i 4.5. Dodatkowo przeprowadzono testy na bazie DB2 z założonym indeksem na kolumnie złączenia w rekurencyjnym CTE i bez niego. W tabeli 4.5 przedstawiono wyniki tych testów. Okazało się, że dla bazy DB2 istnienie tego indeksu ma duże znacznie.

Rozdział 5

Metody wykonywania zapytań rekurencyjnych w ORM

5.1 Motywacja

W rozdziale 4 przedstawiono interfejs programisty ORM, który udostępnia funkcjonalność zapytań rekurencyjnych. Opisano także prototypową implementację tych udogodnień w popularnym systemie odwzorowań obiektowo-relacyjnych Hibernate, która polega na generowaniu zapytań rekurencyjnych SQL:1999 i przesyłaniu ich do wykonania przez system zarządzania bazą danych. Takie rozwiązanie jest skuteczne jedynie w przypadku systemów zarządzania bazami danych, w których zaimplementowano zapytania rekurencyjne. Istnieją jednak systemy zarządzania bazami danych, które takiej funkcjonalności nie mają. Niektóre z nich są bardzo popularne, np. MySQL. Postanowiliśmy więc zaprojektować i przetestować rozszerzenie narzędzi ORM o możliwość realizacji zapytań rekurencyjnych nawet wtedy, gdy użytkowany SZBD ich nie implementuje. Prototypową implementację opracowaliśmy dla Hibernate i MySQL. W niniejszym rozdziale zajmiemy się metodami, w których nie korzysta się ze zmaterializowanych pomocniczych struktur danych. W rozdziale 6 zajmiemy się natomiast metodami realizacji w ORM rekurencji z użyciem danych redundantnych.

5.2 Proponowane rozwiązanie

W tym rozdziale rozważymy trzy metody wykonywania zapytań rekurencyjnych: iterację bezpośrednią (ang. *direct loop*), rozwinięcie w głąb (ang. *vertical unrolling*) oraz rozwinięcie wszerz (ang. *horizontal unrolling*). Metody te opisano też w artykule [4]. W tym rozdziale jako wiodącego przykładu

Tablica 5.1: Przykładowe dane hierarchiczne składowane w tabeli `Emp`

empId	fname	sname	bossId
1	John	Travolta	
2	Bruce	Willis	1
3	Marilyn	Monroe	
4	Angelina	Jolie	3
5	Brad	Pitt	4
6	Hugh	Grant	4
7	Colin	Firth	3
8	Keira	Knightley	6
9	Sean	Connery	1
10	Pierce	Brosnan	3

użyjemy struktury hierarchicznej pracowników korporacji przedstawionej w tablicy 5.1.

5.2.1 Iteracja bezpośrednia

Metoda ta polega na wykonaniu po stronie klienta pętli, która dla każdego znalezionego elementu danych wysyła oddzielne zapytanie do bazy danych. Zapytanie to ma pobrać potomków lub sąsiadów tego elementu. Przykładowy pseudokod tej pętli przedstawiono w postaci algorytmu 1. Na początku inicjujemy listę L zbiorem identyfikatorów wszystkich pracowników noszących zadane nazwisko (w tym przykładzie jest to Travolta). Następnie w pętli dla każdego pracownika z listy L szukamy jego podwładnych. Identyfikatory znalezionych podwładnych są dodawane na koniec listy L.

```
L ← get(" SELECT * FROM Emp WHERE sname = 'Travolta' ");
i ← 0;
while i < len(L) do
    id ← L[i].empId;
    L ← L + get(" SELECT * FROM Emp WHERE bossId = $id ");
    i++;
end
return L;
```

Algorithm 1: Iteracja bezpośrednia w celu pobrania podwładnych zadanego pracownika.

Algorytm ten nie jest efektywny, ponieważ liczba zapytań wysłanych do bazy danych jest równa końcowej liczbie elementów listy L. To wymaga du-

żego zaangażowania ze strony serwera bazy danych. Każde zapytanie musi zostać przesłane, zanalizowane składniowo, zoptymalizowane oraz sprawdzone pod względem praw dostępu do danych. Chociaż można ten algorytm przyśpieszyć poprzez wprowadzenie *poleceń przygotowanych* (ang. *prepared statement*), jednak zmiana ta nie wpłynie znacząco na przyśpieszenie algorytmu. Zaletą tego algorytmu jest natomiast prostota i przenośność.

5.2.2 Rozwinięcie wszerz

Metoda ta ma zastosowanie, jeśli znamy głębokość danych rekurencyjnych lub chcemy ograniczyć głębokość przeszukiwania. Zamiast wysyłać wiele pojedynczych zapytań do bazy danych, budujemy jedno zapytanie złożone z samozłączeń lewostronnych tej samej tabeli. Aby zapytać o wszystkich podwładnych pracownika o zadanym nazwisku (tu: Travolta) z dopuszczalnym zagłębieniem 4, tworzymy zapytanie przedstawione na listingu 5.1.

Listing 5.1: Zapytanie o podwładnych Travolty do czwartego poziomu

```
SELECT *
FROM Emp l1
  LEFT JOIN Emp l2 ON (l1.empId = l2.bossId)
  LEFT JOIN Emp l3 ON (l2.empId = l3.bossId)
  LEFT JOIN Emp l4 ON (l3.empId = l4.bossId)
WHERE
  l1.sname = 'Travolta'
```

W wyniku tego zapytania otrzymujemy tabelę o szerokości równej liczbie dopuszczalnych poziomów zagłębienia pomnożonej przez liczbę kolumn wypisywanych z pojedynczego rekordu. Każdy wiersz tej tabeli reprezentuje pojedynczą pełną ścieżkę drzewa. W naszym przykładzie będzie miał 16 kolumn, czyli liczba kolumn tabeli `Emp` pomnożona przez maksymalny poziom zagłębienia. Stąd wzięła się nazwa tej metody: *rozwinięcie wszerz*. Poniżej przedstawiono fragment wyniku zapytania z listingu 5.1 ograniczony do 4 kolumn (po jednej z każdej łączonej tabeli) i 4 wierszy.

```
 l1.sname | l2.sname | l3.sname | l4.sname | ...
----------+----------+----------+----------+----------
Travolta  | Willis   | Schmidt  | Foremann | ...
Travolta  | Willis   | Schmidt  | Flinch   | ...
Travolta  | Willis   | Ivanov   | null     | ...
Travolta  | Connery  | null     | null     | ...
...       | ...      | ...      | ...      | ...
```

Wadą rozwinięcia wszerz jest szybki przyrost szerokości zwracanego wyniku oraz nadmiarowość danych. Otrzymujemy bowiem nie tylko ostateczne wyniki. Każdy rekord zawiera pełny opis ścieżki, której przejście pozwala dotrzeć do docelowego rekordu. Z eksperymentów wynika jednak, że jest to najbardziej wydajna metoda (por. p. 5.3).

Listing 5.2: Rozwinięcie w głąb zapytania o podwładnych zadanego pracownika do poziomu zagnieżdżenia N.

```
SELECT e_0.empId, e_0.fname, e_0.sname, e_0.bossId
FROM Emp e_0
WHERE e_0.sname = 'Travolta'

UNION

SELECT e_1.empId, e_1.fname, e_1.sname, e_1.bossId
FROM Emp e_1
  JOIN Emp e_0 ON (e_1.bossId = e_0.empId)
WHERE e_0.sname = 'Travolta'

UNION

SELECT e_2.empId, e_2.fname, e_2.sname, e_2.bossId
FROM Emp e_2
  JOIN Emp e_1 ON (e_2.bossId = e_1.empId)
  JOIN Emp e_0 ON (e_1.bossId = e_0.empId)
WHERE
  e_0.sname = 'Travolta'

UNION
...
UNION

SELECT e_N.empId, e_N.fname, e_N.sname, e_N.bossId
FROM Emp e_N
  JOIN Emp e_N-1 ON (e_N.bossId = e_N-1.empId)
  ...
  JOIN Emp e_1 ON (e_2.bossId = e_1.empId)
  JOIN Emp e_0 ON (e_1.bossId = e_0.empId)
WHERE
  e_0.sname = 'Travolta'
```

5.2.3 Rozwinięcie w głąb

Rozwinięcie wszerz niestety zmienia format wyniku zapytania. Jeśli chcemy tego uniknąć, a przy tym nie stosować iteracji bezpośredniej, możemy skorzystać z *rozwinięcia w głąb* omówionego w artykule [4]. W metodzie tej zamiast dodawać nowe kolumny (jak w rozwinięciu wszerz) dodajemy nowe wiersze uzyskiwane poprzez kolejne sumy mnogościowe z coraz głębszymi samozłączeniami. Otrzymujemy więc zapytanie, które jest sumą mnogościową złączeń tabeli samej ze sobą odpowiednio tyle razy, ile jest poziomów zagłębienia. Niestety oznacza to, że przy wyznaczaniu kolejnego poziomu trzeba od nowa liczyć złączenia z poziomów poprzednich.

Przypuśćmy na razie, że znamy N będące maksymalnym dopuszczalnym poziomem zagnieżdżenia rekurencji. Rozwinięcie w głąb zapytania o podwładnych zadanego pracownika (tu Travolty) pokazano na listingu 5.2.

Gdy jednak nie znamy maksymalnego poziomu zagnieżdżenia przetwarzanego zapytania rekurencyjnego, możemy skorzystać z tabel tymczasowych do przechowywania wyników częściowych na poszczególnych poziomach zagłębienia. Przy takiej metodzie zamiast jednego zapytania, wykonywane jest tyle zapytań, ile jest poziomów zagłębienia rekurencji. To oznacza i tak znacznie mniejszą liczbę niż przy iteracji bezpośredniej (por. p. 5.2.1). W celu uzyskania wyniku na kolejnym poziomie zagłębienia należy złączyć tabelę tymczasową z oryginalną tabelą `Emp`. Algorytm realizujący rozszerzenie zapytania rekurencyjnego w głąb w oparciu o tabele tymczasowe składa się z trzech następujących kroków.

Krok podstawowy W kroku tym wykonujemy zapytanie podstawowe zapytania rekurencyjnego. Rezultat umieszczamy w tabeli tymczasowej `tmp`(0). Krok podstawowy dla zapytania o podwładnych pracowników o ustalonym nazwisku przedstawiono na listingu 4.1.

Listing 5.3: Krok podstawowy algorytmu

```
CREATE TEMPORARY TABLE tmp(0)
  SELECT empId, fname, sname, bossId
  FROM Emp
  WHERE sname = 'Travolta';
```

Krok rekurencyjny Dla każdego $n = 1, 2, \ldots$, konstruujemy tabelę tymczasową `tmp`(n) używając w tym celu tabeli tymczasowej z poprzedniego kroku `tmp`($n-1$). Listing 5.4 zawiera zapytanie budujące tabelę tymczasową dla kroku n-tego.

Listing 5.4: Krok rekurencyjny algorytmu

```
CREATE TEMPORARY TABLE tmp(n)
  SELECT e.empId, e.fname, e.sname, e.bossId
  FROM Emp e
    JOIN tmp(n - 1) t ON (e.bossId = t.empId)
```

Jeśli podany jest maksymalny poziom zagłębienia L, wówczas krok rekurencyjny jest wykonywany L razy. Jeśli maksymalny poziom zagłębienia nie jest znany, wówczas krok rekurencyjny jest wykonywany tak długo, dopóki tymczasowa tabela jest niepusta.

Krok wynikowy W kroku tym wykonujemy zapytanie, które jest sumą zapytań z wszystkich tymczasowych tabel. Listing 5.5 zawiera to zapytanie. Jeśli maksymalny poziom zagłębienia był ustalony, to N jest mu równe. W przeciwnym przypadku N oznacza ostatni krok rekurencji, w którym tabela tymczasowa nie była pusta.

Listing 5.5: Krok wynikowy

```
SELECT sname, fname
FROM
  (
    SELECT * FROM tmp(0)
    UNION
    SELECT * FROM tmp(1)
    UNION
     ...
    UNION
    SELECT * FROM tmp(N)
  )
```

5.2.4 Realizacja w MySQL i Hibernate

Wykonywanie zapytań rekurencyjnych w Hibernate, gdy współpracujemy z bazą danych MySQL, wymagało dalszego rozszerzenia składni konfiguracji (por. listing 4.4 z rozdziału 4). Do znacznika `rcteTable` dodano parametr `unrollingType`, który domyślnie ma wartość `vertical`. Oznacza to, że domyślnie zostanie zastosowana metoda rozszerzenia w głąb z tabelami tymczasowymi. Kolejno wykonane będą zapytanie z kroku podstawowego i zapytania z kroku rekurencyjnego. Za każdym razem będzie sprawdzana liczba zwracanych rekordów. Jeśli Hibernate osiągnie podany maksymalny

poziom zagłębienia (parametr `max-level` znacznika `rcteTable`) lub nie zostanie zwrócony żaden rekord, wykonywanie kroków rekurencyjnych polegających na tworzeniu tabel tymczasowych zostanie zakończone. Następnie Hibernate pobierze wszystkie dane z tabel tymczasowych i na ich podstawie wygeneruje ostateczny wynik zapytania rekurencyjnego.

Jeśli parametr `unrollingType` ma wartość `horizontal` zostanie wygenerowane zapytanie z listingu 5.1. Obowiązkowym parametrem w tej metodzie jest `max-level`.

```
<rcteTable
  name="subordinates"
  unrolling="horizontal"
  max-level="3"
/>
```

W takim wypadku Hibernate wygeneruje dużo bardziej rozbudowany wynik zapytania, który wymaga od aplikacji odpowiedniego przetworzenia. Każdy rekord zawiera bowiem pełną ścieżkę w drzewie. Otrzymane wyniki są udostępniane aplikacji w postaci obiektów reprezentujących pojedyncze rekordy.

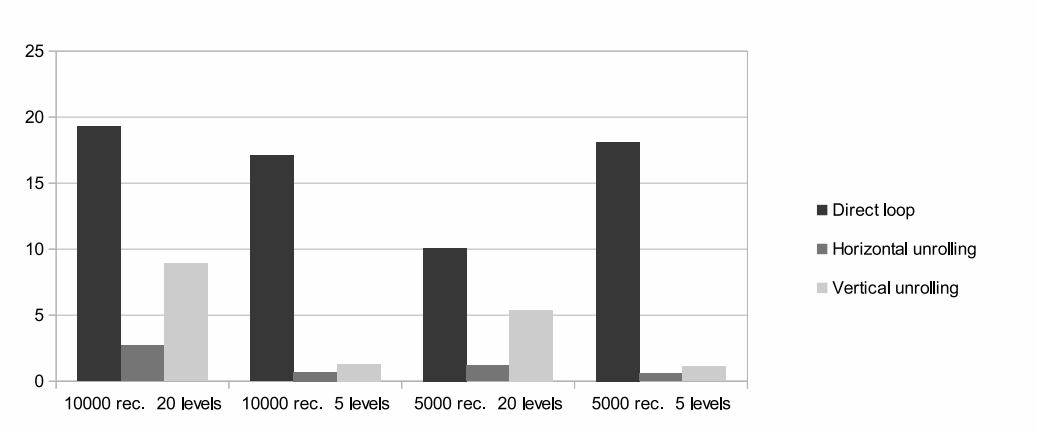
5.3 Testy

Przeprowadziliśmy eksperymenty w celu zmierzenia wydajności poszczególnych metod opisanych w niniejszym rozdziale: iteracji bezpośredniej, rozwinięcia wszerz i rozwinięcia w głąb. Dane w tabeli `Emp` miały cztery następujące warianty.

1. 5 000 rekordów i 5 poziomów zagłębienia,
2. 5 000 rekordów i 20 poziomów zagłębienia,
3. 10 000 rekordów i 5 poziomów zagłębienia,
4. 10 000 rekordów i 20 poziomów zagłębienia.

Testy były przeprowadzone na komputerze z procesorem AMD Phenom II 3.4 GHz, pamięcią 8 GB RAM oraz dwoma dyskami Caviar Black 7400 Rpm po 500 GB pojemności każdy. Wykorzystana została baza MySQL w wersji 5.5. Rozmiar bufora danych wynosił 2GB.

Dla każdego wariantu tabeli utworzono indeksy na kolumnach `empId` oraz `bossId`. Zostało użyte zapytanie 3.1 z benchmarku z rozdziału 3 dla powyżej



Rysunek 5.1: Wyniki testów Hibernate z bazą MySQL dla trzech opisywanych metod

Liczba rekordów	Głębokość hierarchii	Iteracja bezpośrednia	Rozwinięcie wszerz	Rozwinięcie w głąb
5000	5	18.1s	0.6s	1.1s
5000	20	10.1s	1.2s	5.4s
10000	5	17.1s	0.7s	1.3s
10000	20	19.3s	2.7s	8.9s

Tablica 5.2: Porównanie czasów wykonania zapytań rekurencyjnych przez Hibernate z bazą MySQL

opisanej konfiguracji danych. Rezultat testów przedstawiono w tabeli 5.2 oraz na rysunku 5.1.

Każde zapytanie zostało wykonane 5 razy. Czasy najlepszy i najgorszy zostały odrzucone. Z pozostałych trzech wyników została wzięta średnia czasów.

5.4 Analiza zaproponowanego rozwiązania

Chociaż zapytania rekurencyjne dołączono do standardu SQL w 1999 roku [29], jednak nadal powszechnie używa się SZBD, które nie mają implementacji takich zapytań. Jednym z takich systemów zarządzania bazą danych jest MySQL. Metody optymalizacji zapytań w bazach danych, które zapytania rekurencyjne wspierają, zostały omówione w punkcie 4.4. Dla baz nieposiadających mechanizmów wspierających zapytania rekurencyjne najbardziej

intuicyjnym sposobem korzystania z tych zapytań jest użycie prostej pętli (iteracja bezpośrednia). Niestety najbardziej intuicyjna metoda jest jednak też najbardziej kosztowną.

Z przeprowadzonych testów wynika, że metody rozwinięcia zapytania rekurencyjnego przedstawione w niniejszym rozdziale są istotnie szybsze niż iteracja bezpośrednia. Z tych dwóch metod szybszym okazało się rozwinięcie wszerz. Jest ono nawet czterokrotnie szybsze od rozwinięcia w głąb dla dwudziestu poziomów zagłębienia danych. Duża liczba kolumn w wyniku nie stanowi dla tej metody przeszkody.

Rozdział 6

Dane redundantne w optymalizacji zapytań rekurencyjnych w ORM

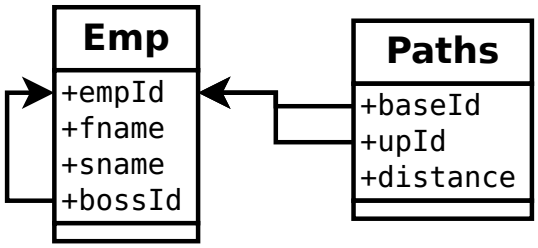
6.1 Motywacja

Jak wiemy z poprzednich rozdziałów, dane hierarchiczne są zwykle przechowywane w tabeli, która ma klucz obcy odwołujący się do jej klucza głównego. Takiej metody składowania danych o hierarchii użyto na przykład w bazie testowej Oracle [10]. W rozdziale 5 przedstawiono metody wykonywania zapytań do struktur hierarchicznych, w sytuacji gdy użytkowany system zarządzania bazą danych nie ma zaimplementowanych zapytań rekurencyjnych. Opisane tam metody korzystają wyłącznie z podstawowej tabeli z danymi. W odróżnieniu od metod z rozdziału 5 w niniejszym rozdziale zajmiemy się technikami korzystającymi z materializacji danych redundantnych. Celem składowania i pielęgnacji takich dodatkowych danych jest przyspieszenie wykonywania zapytań rekurencyjnych użytkownika. Metody opisane w niniejszym rozdziale mogą być oczywiście stosowane także wówczas, gdy użytkowany system zarządzania bazą danych umie sam wykonywać zapytania rekurencyjne SQL:1999. Dane redundantne wykorzystywane w niniejszym rozdziale mogą być przechowywane na dwa różne sposoby. Pierwszy sposób polega na utworzeniu dodatkowej tabeli, a drugi na dodaniu nowych kolumn do tabeli podstawowej. Przedstawimy cztery metody wykorzystywania danych redundantnych w realizacji zapytań rekurencyjnych, po dwa dla każdego sposobu składowania.

Trzy przedstawione w tym rozdziale metody są powszechnie znane w literaturze [10]. Ta czwarta metoda (ścieżki logarytmiczne) jest próbą znale-

Tablica 6.1: Przykładowe dane hierarchiczne składowane w tabeli `Emp`

empId	fname	sname	bossId
1	John	Travolta	
2	Bruce	Willis	1
3	Marilyn	Monroe	
4	Angelina	Jolie	3
5	Brad	Pitt	4
6	Hugh	Grant	4
7	Colin	Firth	3
8	Keira	Knightley	6
9	Sean	Connery	1
10	Pierce	Brosnan	3



Rysunek 6.1: Tabela podstawowa `Emp` oraz tabela z danymi redundantnymi `Paths`

zienia oszczędniejszego sposobu materializacji danych pomocniczych w celu przyspieszenia implementacji zapytań rekurencyjnych. W literaturze przedmiotu brakuje też porównania tych metod oraz sprawdzenia ich wydajności. Postaramy się uzupełnić tę lukę. Omówimy prototypową implementację wszystkich tych metod oraz wyniki eksperymentów nad wydajnością poszczególnych metod. Zanalizujemy te wyniki i wskażemy, kiedy warto stosować każdą z opisanych metod.

Metody wykonywania zapytań rekurencyjnych SQL:1999, które opisano w rozdziałach niniejszym oraz poprzednim, można stosować używając dowolnego systemu zarządzania bazą danych. Natywne metody wykonywania zapytań rekurencyjnych SQL:1999 są jednak zdecydowanie szybsze. Z tego powodu metody nienatywne są przeznaczone w głównej mierze dla tych systemów zarządzania bazą danych, które nie mają zaimplementowanych zapytań rekurencyjnych SQL:1999.

6.2 Proponowane rozwiązanie

6.2.1 Metoda ścieżek pełnych

Metoda ścieżek pełnych polega na wyznaczeniu i zmaterializowaniu wszystkich ścieżek w hierarchii. Informacje te będą przechowywane w dodatkowej tabeli. Tabela podstawowa nie ulega zmianom. W istocie jest to wykorzystanie znanej metody obliczania domknięcia przechodniego relacji. Metodę ścieżek pełnych opisano w [10] i wykorzystano w artykule [5].

Przyjmijmy, że nadmiarowe informacje o ścieżkach będą przechowywane w dodatkowej tabeli `Paths`. Dla każdego węzła z tabeli podstawowej gromadzimy informacje o wszystkich jego przodkach i odległościach do tych przodków. Rozważmy przykładową tabelę z danymi hierarchicznymi pokazaną na tablicy 6.1. Na rysunku 6.1 przedstawiono schemat tabeli podstawowej `Emp` oraz redundantnej tabeli `Paths` ze zmaterializowanymi ścieżkami.

Zawartość tabeli dodatkowej `Paths` jest obliczana w następujący sposób:

1. Zapytanie z listingu 6.1 wstawia wiersze z odległością każdego węzła od niego samego (czyli oczywiście zero).

2. Zapytanie z listingu 6.2 jest wykonywane dla kolejnych $n = 0, 1, 2, \ldots$, dopóki zwracany jest niepusty wynik. Zapytanie to wstawia do tabeli `Paths` przodków stopnia $n + 1$ każdego węzła.

Listing 6.1: Zapytanie inicjujące zawartość tabeli `Paths`.

```
INSERT INTO
  Paths
SELECT
  empId as baseId,
  empId as upId,
  0 as distance
FROM
  Emp
```

Listing 6.2: Zapytanie wstawiające do tabeli `Paths` przodków stopnia $n + 1$ każdego węzła przy założeniu, że są w niej już informacje o przodkach stopnia n.

```
INSERT INTO
  Paths
SELECT
  empId as baseId,
```

```
  upId,
  p.distance+1 as distance
FROM
  Emp e
JOIN
  Paths p
ON
  (e.bossId = p.baseId)
WHERE
  p.distance = n
```

baseId	upId	distance
8	8	0
8	6	1
8	4	2
8	3	3

Rysunek 6.2: Wiersze z informacją o węzłach na ścieżce od korzenia do węzła `(8, 'Keira', 'Knightley')`.

Na rysunku 6.2 przedstawiono wiersze, które w tabeli `Paths` zostaną utworzone dla wiersza o identyfikatorze 8 z tabeli `Emp` (por. tab. 6.1). Jak widać zapytanie z listingu 6.1 wstawiło wiersz

8	8	0

Zapytanie z z listingu 6.2 zostało wykonane trzy razy dla $n = 0, 1, 2, 3$. Zostały wprowadzone odpowiednio trzy kolejne wiersze.

8	6	1
8	4	2
8	3	3

Zapytanie z z listingu 6.2 dla $n = 4$ nie zwróci już żadnego wiersza, zatem nie będzie więcej wykonane.

Dla wiersza o identyfikatorze 7 z tabeli `Emp` zostaną utworzone wiersze w tabeli `Paths` przedstawione na rysunku 6.3.

baseId	upId	distance
7	7	0
7	3	1

Rysunek 6.3: Wiersze z informacją o węzłach na ścieżce od korzenia do węzła `(7, 'Colin', 'Firth')`.

Dane w tej tabeli zależą od zawartości tabeli podstawowej `Emp`. Każda operacja modyfikująca dane w `Emp` powinna również modyfikować dane w tabeli `Paths`. Aby zapewnić taką semantykę modyfikacji wierszy tabeli `Emp`, zostaną na niej założone wyzwalacze reagujące na wszelkie modyfikację danych (CRUD). Będą one dbały, aby w tabeli redundantnej `Paths` zawsze znajdowały się aktualne dane.

Przypuśćmy, że użytkownik systemu odwzorowania obiektowo-relacyjnego skonfigurował go tak, by korzystał on z metody ścieżek pełnych. Wówczas będzie można wywoływać metodę `getRecursive` (por. rozdział 4), która znajduje wszystkich potomków zadanego węzła. Gdy wywołamy ją z argumentem równym `Travolta`, system odwzorowania obiektowo-relacyjnego wyśle do bazy danych zapytanie z listingu 6.3.

Listing 6.3: Zapytanie wysyłane przez wywołanie `getRecursive('Travolta')`, gdy zastosowano metodę ścieżek pełnych.

```
SELECT
   e.empId,
   e.fname,
   e.sname,
   e.bossId
FROM
   Emp b
JOIN
   Paths p
ON
   (p.UpId = b.empId)
JOIN
   Emp e
ON
   (p.baseId = e.empId)
WHERE
   b.sname = 'Travolta'
```

6.2.2 Metoda ścieżek logarytmicznych

Metodę ścieżek logarytmicznych zaproponowano i opisano w artykule [6]. Ta metoda również korzysta z dodatkowej tabeli do przechowywania danych redundantnych. W odróżnieniu od metody ścieżek pełnych przedstawionej powyżej, metoda ścieżek logarytmicznych nie polega na przechowywaniu wszystkich węzłów na ścieżce do korzenia do każdego węzła. Warto bowiem zwrócić uwagę, że w ogólności hierarchia o n wierszach może mieć domknięcie przechodnie o wielkości $O(n^2)$. W przypadku metody ścieżek pełnych może to oznaczać nieakceptowany rozmiar tabeli pomocniczej `Paths`.

Metoda ścieżek logarytmicznych polega na przechowywaniu jedynie tych par węzłów, których odległość w hierarchii jest potęgą dwójki. Otrzymujemy w ten sposób rozsądny kompromis między wielkością pomocniczej struktury i kosztem jej budowy (teraz jest to tylko $O(n \log n)$), a kosztem zadawania zapytań o połączenia węzłów hierarchii (zamiast $O(1)$ musimy wysłać $O(\log n)$ zapytań). Dokładniejsze oszacowania złożoności są podane poniżej.

Przyjmijmy, że dane redundantne metody ścieżek logarytmicznych są przechowywane w tabeli pomocniczej `LogPaths`. Dla przykładowych danych z tablicy 6.1 rekordy w tabeli `LogPaths` dla węzła `Keira Knightley` (8) będą wyglądały następująco:

```
 baseId | upId  | distance |
--------+-------+----------+
   8    |   6   | 1        |
   8    |   4   | 2        |
```

W porównaniu z metodą ścieżek pełnych nie jest dodany wiersz:

```
8 |  3   | 3    |
```

ponieważ 3 nie jest potęga dwójki.

Dla przykładowych danych z tablicy 6.1 rekordy w tabeli `LogPaths` dla węzła `Colin Firtht` (7) będą wyglądały następująco:

```
 baseId | upId  | distance |
--------+-------+----------+
   7    |   3   | 1        |
```

Jeśli tabela podstawowa zawiera n rekordów, które przechowują dane hierarchiczne o głębokości m, wówczas tabela `LogPaths` będzie zawierała $O(n \log m)$ wierszy. Będziemy zatem przechowywać tylko $O(\log m)$ wierszy

dla każdego wiersza w tabeli podstawowej, podczas gdy metoda ścieżek pełnych przechowuje tych wierszy $O(m)$. Realizacja zapytania rekurencyjnego użytkownika przy metodzie ścieżek logarytmicznych powoduje wykonywanie $O(\log m)$ zapytań łączących tabelę `LogPaths` $O(\log m)$ razy.

Tabela `LogPaths` powstaje w następujący sposób:

1. Zapytanie z listingu 6.4 wstawia wiersze o odległości 1 na ścieżce do korzenia. Oznacza to, że wstawiane są wiersze zawierające dany węzeł wraz z jego rodzicem.

2. Dla $n = 1, 2, 4, 8, \ldots$ (potęgi dwójki) zapytanie z listingu 6.5 wstawia dla każdego węzła jego przodków stopnia $2n$. Wiersze są dodawane dopóki kolejne zapytanie nie zwróci pustego wyniku. W tym kroku wykonuje się $O(\log m)$ zapytań, gdzie m jest głębokością drzewa reprezentującego dane hierarchiczne.

Listing 6.4: Zapytanie inicjujące zawartość tabeli `LogPaths`.

```
INSERT INTO
  LogPaths
SELECT
  empId as baseId,
  bossId as upId,
  1 AS distance
FROM
  Emp
```

Listing 6.5: Zapytanie wstawiające do tabeli `LogPaths` przodków stopnia $2n$ każdego węzła przy założeniu, że są w niej już informacje o przodkach stopnia n.

```
INSERT INTO
  LogPaths
SELECT
  e.baseId,
  b.upId,
  2n AS distance
FROM
  LogPaths e
JOIN
  LogPaths b
ON
  (e.upId = b.baseId)
```

```
WHERE
  e.distance = n
AND
  b.distance = n
```

W celu uzyskania danych hierarchicznych należy wykonać zapytanie z listingu 6.6. To zapytanie rekonstruuje dane o ścieżkach, których długości mają dokładnie k jedynek w rozwinięciu dwójkowym. Takie k-krotne samozłączenie tabeli `LogPaths` zwróci tych potomków zadanego węzła, których odległość w drzewie wyrażona w postaci binarnej zawiera dokładnie k jedynek. Aby odtworzyć wszystkich podwładnych wystarczy uruchomić to zapytanie dla $k = 1, 2, ..., \log m$, a następnie zsumować mnogościowo wyniki. W ten sposób wygenerowane zostaną wszystkie ścieżki. Żadna ścieżka nie zostanie wygenerowana wielokrotnie, ponieważ reprezentacja każdej liczby naturalnej w systemie dwójkowym jest jednoznaczna.

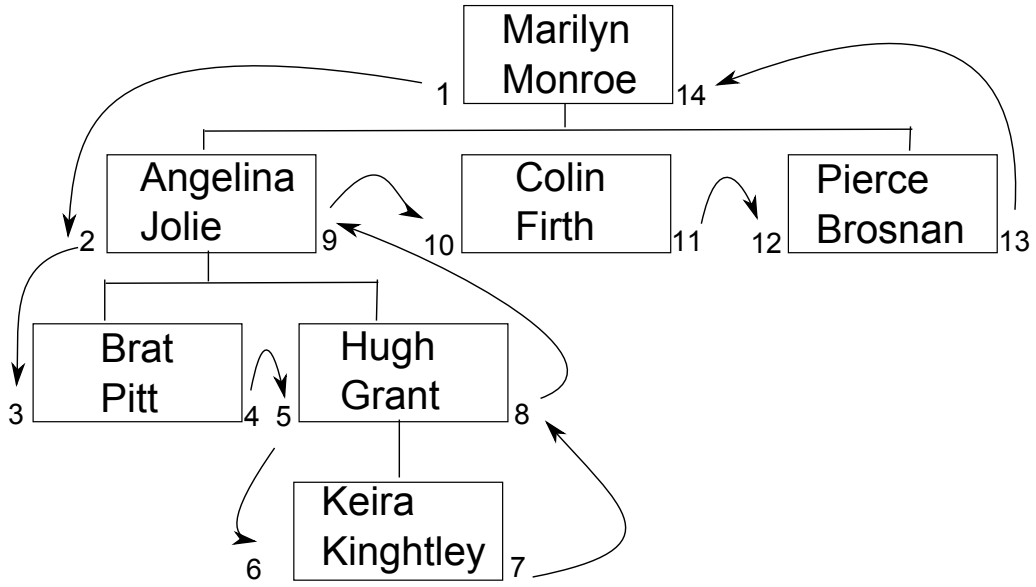
Podobnie jak w *metodzie ścieżek pełnych* dane w tabeli `LogPaths` zależą od zawartości tabeli bazowej `Emp`. Każda zmiana danych w tabeli bazowej musi mieć odzwierciedlenie w tabeli `LogPaths`. Na tabeli `Emp` należy zatem założyć odpowiednie wyzwalacze, które będą dbały, by tabela `LogPaths` zawsze zawierała wyłącznie aktualne dane.

Listing 6.6: Zapytanie, które odpytuje dane hierarchiczne używając tabeli `LogPaths`.

```
SELECT
  lp1.baseId,
  lpk.upId,
  lp1.distance + lp2.distance + ... + lpk.distance AS distance
FROM
    LogPaths lp1
JOIN LogPaths lp2 ON (lp1.upId = lp2.baseId)
...
JOIN LogPaths lpk ON (lp(k-1).upId = lpk.baseId)
WHERE
    lp1.distance < lp2.distance
AND lp2.distance < lp3.distance
AND ...
AND lp(k-1).distance < lpk.distance
```

6.2.3 Metoda zbiorów zagnieżdżonych

Dane redundantne w metodzie zbiorów zagnieżdżonych są przechowywane inaczej niż w dwóch wcześniej omówionych metodach. Nie jest tworzona



Rysunek 6.4: Wartości dodatkowych kolumn `left` i `right` w metodzie zbiorów zagnieżdżonych dla danych z tablicy 6.1.

żadna dodatkowa tabela. Do tabeli podstawowej dodaje się bowiem kolumny `left` oraz `right`. Metoda ta została opisana w [10]. Każdy węzeł jest traktowany jak zbiór. Każdy zbiór zawiera w sobie podzbiory odpowiadające jego bezpośrednim potomkom. Jeśli zbiór nie ma podzbiorów/potomków, to jest liściem. Każdy taki węzeł/zbiór jest reprezentowany przez dwie liczby (`left` oraz `right`), które spełniają następujące warunki:

- e.`left` $<$ e.`right` dla każdej krotki e,
- jeśli krotka e jest w poddrzewie wyznaczonym przez krotkę b, wtedy prawdą jest, że e.`left` $>$ b.`left` oraz e.`right` $<$ b.`right`.

Na rysunku 6.4 przedstawiono zawartość kolumn `left` i `right` dla danych przykładowych z tablicy 6.1 oraz w jaki sposób są wyliczane. Strzałki prezentują kierunek przeglądania poszczególnych węzłów oraz kolejność wstawiania wartości kolumn dodatkowych. W celu wypełnienia kolumn `left` i `right` należy przejść daną strukturę hierarchiczną rekurencyjnie metodą w głąb zaczynając od korzenia i wstawić odpowiednie wartości w dodatkowych kolumnach. Znalezienie wszystkich potomków zadanego węzła (tu `Travolty`) wymaga wykonania zapytania z listingu 6.7.

Podobnie jak we wcześniej omówionych metodach każda aktualizacja danych podstawowych wymaga poprawienia danych redundantnych. Zadbają

o to wyzwalacze, które należy nałożyć na tabelę podstawową.

Listing 6.7: Zapytanie o potomków węzła w *metodzie zbiorów zagnieżdżonych*.

```
SELECT
   e.empId,
   e.sname
FROM
   emp e,
   emp b
WHERE
     b.sname = 'Travolta'
AND e.left  BETWEEN b.left AND b.right
AND e.right BETWEEN b.left AND b.right
```

6.2.4 Metoda ścieżek zmaterializowanych

Metoda ścieżek zmaterializowanych również polega na dodaniu dodatkowej kolumny do tabeli bazowej. Jest to kolumna typu napisowego. W tej nowej kolumnie przechowywana jest lista wszystkich węzłów na ścieżce od danego węzła do korzenia. Poniżej przedstawiono przykładowe dane z rozszerzonej tabeli `Emp` z tablicy 6.1:

```
 empId | fname |   sname   | bossId | paths
-------+-------+-----------+--------+-----------
   7   | Colin | Firth     |   3    | 7,3
   8   | Keira | Knightley |   6    | 8,6,4,3
```

Węzeł o id 7 ma tylko jednego przodka o id 3. Natomiast węzeł o id 8 ma aż trzech przodków: $6, 4, 3$. Nie jest tu przechowywana odległość danego węzła od jego przodka, tylko lista przodków.

Wypełnienie dodatkowej kolumny odbywa się w dwóch następujących krokach:

1. Wstawienie wartości `empId` do kolumny `paths` w węzłach reprezentujących korzenie hierarchii (korzeniami są węzły spełniające warunek `bossId IS NULL`).

2. Wykonywanie aktualizacji z listingu 6.8 dopóki liczba zmodyfikowanych przez nią wierszy jest niezerowa.

Zapytanie, które znajduje wszystkich potomków węzła `Travolta` przedstawiono na listingu 6.9. Każda operacja modyfikująca tabelę bazową może mieć wpływ na dodatkową kolumnę, dlatego też na tabeli bazowej zakłada się wyzwalacze, które zadbają o aktualność zawartości dodatkowej kolumny.

Listing 6.8: Wypełnianie dodatkowej kolumny w *metodzie ścieżek zmaterializowanych*.

```
UPDATE
  Emp AS current
INNER JOIN
  Emp AS parent
ON
  (parent.eid = current.bID)
SET
  current.paths =
 CONCAT(parent.paths, ',', current.empno)
WHERE
  parent.paths IS NOT NULL
AND
  current.paths IS NULL;
```

Listing 6.9: Zapytanie zwracające potomków danego węzła w *metodzie ścieżek zmaterializowanych*.

```
SELECT
  *
FROM
  emp
WHERE
  path_string LIKE
  (
    SELECT
      concat(path_string, '%')
    FROM
      emp
    WHERE
      sname = 'Travolta'
  )
```

6.2.5 Implementacja w Hibernate

Opisane powyżej metody zaimplementowaliśmy jako rozszerzenie systemu odwzorowania obiektowo-relacyjnego Hibernate. Aby użyć jednej z tych metod do wykonywania zapytań do danych hierarchicznych, należy w rozszerzonej konfiguracji opisanej w p. 5.2.4 podać odpowiednią wartość parametru `unrollingType`. Może on przyjmować następujące wartości:

- `"full paths"` dla metody ścieżek pełnych,
- `"logarithmic paths"` dla metody ścieżek logarytmicznych,
- `"nested sets"` dla metody zbiorów zagnieżdżonych,
- `"materialized paths"` dla metody ścieżek zmaterializowanych.

To samo można skonfigurować za pomocą adnotacji. Przy wybranej klasie należy dopisać `@unrolling(method = "nazwa_metody")`. Zostaną wówczas wygenerowane odpowiednie redundantne struktury danych oraz metoda `getRecursive(String)`, która zbuduje i wyśle do bazy danych zapytanie dla właściwej metody.

6.3 Testy

W celu eksperymentalnej weryfikacji przydatności metod omówionych w niniejszym rozdziale przeprowadzono dwie niezależne grupy testów. Za pomocą pierwszej z nich porównaliśmy metodę ścieżek pełnych z rozwijaniem zapytań wszerz (por. rozdział 5). Chcieliśmy bowiem sprawdzić, czy użycie danych redundantnych rzeczywiście poprawia wydajność zapytań rekurencyjnych. Druga grupa przeprowadzonych testów miała na celu porównanie metod opisanych w niniejszym rozdziale między sobą.

Kolumna o nazwie *Procentowe przyspieszenie* (czasem skracanej do *Proc. przysp.*) zawiera zapisany w procentach stosunek uzyskanego rezultatu z kolumny drugiej do rezultatu z kolumny trzeciej. Umożliwia to oszacowanie faktycznego zysku ze stosowania jednej bądź drugiej metody.

6.3.1 Rozwinięcie wszerz a metoda ścieżek pełnych

Rozwinięcie wszerz okazało się być najbardziej wydajną metodą wykonywania zapytań rekurencyjnych bez użycia danych redundantnych (por. p. 5.3). Z tego powodu postanowiliśmy porównać właśnie tę metodę z metodą ścieżek pełnych. Testy przeprowadzono z użyciem systemu zarządzania bazą danych MySQL w wersji 5.5 z buforem danych 2 GB na komputerze z procesorem AMD Phenom II 3.4 GHz, pamięcią 8 GB RAM oraz dwoma dyskami Caviar Black 7400 Rpm po 500 GB pojemności każdy.

Każde zapytanie zostało wykonane 5 razy. Czasy najlepszy i najgorszy zostały odrzucone. Z pozostałych trzech wyników została wzięta średnia czasów. Zostały użyte zapytania 3.1 do wyszukania potomków dowolnego węzła (pracownika) oraz potomków węzła, który jest korzeniem oraz zapytanie 3.2 z benchmarku z rozdziału 3.

Dane testowe znajdowały się w tabeli `Emp`.

Zależnie od konfiguracji testu zawierała ona 5, 10, 15 lub 20 poziomów zagłębienia hierarchii oraz 10^3, 10^4, 10^5 lub 10^6 rekordów. Łącznie oznaczało to 16 różnych zbiorów danych.

Zostały założone indeksy w tabeli dodatkowej `Paths` na kolumny: `baseId` oraz `upId`.

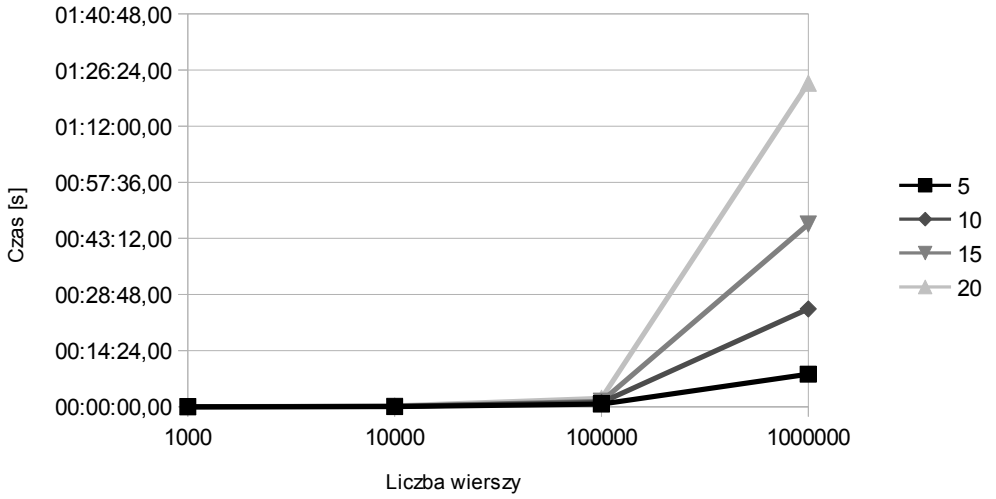
W wyniku testów zaobserwowaliśmy, że metoda ścieżek pełnych jest istotnie szybsza od rozwinięcia wszerz w przypadku (1) dużego poziomu zagłębienia oraz (2) małego poziomu zagłębienia przy dużej liczbie wierszy. W pozostałych przypadkach metoda ścieżek pełnych okazała się porównywalna z rozwinięciem wszerz. Zaletą metody ścieżek pełnych jest zgodność formatu wyniku zapytania z wynikiem zwracanym przez zapytanie rekurencyjne SQL:1999. Wyniki zwracane po rozwinięciu zapytania wszerz są dużo bardziej rozbudowane. Wadą metody ścieżek pełnych jest natomiast konieczność synchronizacji zawartości tabeli dodatkowej po każdej modyfikacji tabeli podstawowej. Jeśli liczba operacji modyfikujących tabelę jest stosunkowo niewielka w porównaniu z liczbą wykonywanych zapytań, metoda ścieżek pełnych jest warta rozważenia. Jeśli operacje modyfikujące dane jednak dominują, metody niekorzystające z danych redundantnych, takie jak testowane w tym punkcie rozwinięcie wszerz, mogą okazać się dużo bardziej skuteczne od metody ścieżek pełnych.

Tworzenie dodatkowej tabeli

W tym teście badaliśmy operację budowy tabeli pomocniczej `Paths` zawierającej dane redundantne niezbędne przy realizacji zapytań rekurencyjnych metodą ścieżek pełnych. Żadne dane pomocnicze nie są potrzebne w przypadku rozwinięcia wszerz. Czas budowy tabeli `Paths` stanowi więc w całości narzut związany z zastosowaniem metody ścieżek pełnych. Wyniki testów przestawiono na rysunku 6.5 oraz w tablicy 6.2. Zawierają one informacje o czasie budowy tabeli `Paths` przy założeniu, że tabela bazowa `Emp` jest już wypełniona danymi. Dla małych rozmiarów tabel `Emp` utworzenie tabeli `Paths` zajmuje około dwóch sekund. Dla najbardziej złożonego przypadku (20 poziomów zagłębienia i 1 000 000 wierszy) czas ten wynosi już ponad godzinę.

Wybieranie potomków danego węzła

W tym teście badaliśmy czas wykonania zapytania o wszystkich potomków zadanego węzła. Przeprowadziliśmy je w dwóch wariantach. W pierwszym z nich wyszukiwaliśmy potomków korzenia, a w drugim potomków węzła wewnętrznego. Wyniki eksperymentów z zapytaniami o potomków korzenia



Rysunek 6.5: Wykres czasu budowy tabeli pomocniczej `Paths`. Poszczególne krzywe odpowiadają różnym poziomom głębokości hierarchii (5, 10, 15 i 20).

Tablica 6.2: Czas budowy tabeli pomocniczej `Paths` (w sekundach)

Liczba rekordów	Poziomy			
	5	10	15	20
1 000	01.62	01.18	01.20	01.22
10 000	03.85	06.34	08.41	15.73
100 000	43.68	01:04.82	01:25.28	02:10.72
1 000 000	08:22.51	25:05.28	46:52.04	01:23:03.00

zebrano w tablicy 6.3 oraz na rysunku 6.6. Z kolei tablica 6.4 i rysunek 6.7 zawierają rezultaty testów pobierania potomków węzła wewnętrznego. W obu tablicach skrót *ŚP* oznacza metodę ścieżek pełnych, a *RW* rozwinięcie wszerz.

W wynikach tych eksperymentów można zaobserwować niespodziewane zjawisko. Dla tabeli zawierającej 100 000 rekordów i małego poziomu zagłębienia (5 oraz 10) metoda ścieżek pełnych jest wolniejsza od rozwinięcia wszerz. Testy zostały powtórzone na sprzęcie o innej konfiguracji oraz na bazie MySQL ze standardowa konfiguracją. Okazało się, że zjawisko to powtórzyło się. W opinii autorów artykułu [5] przyczyną takiego wyniku jest zastosowany w MySQL system zarządzania pamięcią operacyjną. Obliczenia w metodzie ścieżek pełnych są wykonywane na dwóch tabelach. Dla pew-

nych zestawów danych obliczenia przekraczają rozmiar pamięci podręcznej systemu zarządzania bazą danych, co powoduje istotne zmniejszenie efektywności przetwarzania. Dla innych przypadków standardowe ustawienia bazy danych są jednak wystarczające.

W celu weryfikacji wydajności metod zaproponowanych w niniejszym rozdziale, przeprowadziliśmy również eksperymenty z PostgreSQL. Ich wyniki wskazywały jednak na zdecydowaną dominację natywnej implementacji zapytań rekurencyjnych SQL:1999. Uznaliśmy zatem, że badanie wszelkich nienatywnych metod wykonywania zapytań rekurencyjnych w PostgreSQL byłoby sztuczne. Z kolei MySQL jest najważniejszym relacyjnym systemem zarządzania bazą danych, który nie ma zaimplementowanej rekurencji SQL:1999. Szczegółowe eksperymenty wykonaliśmy więc tylko dla MySQL.

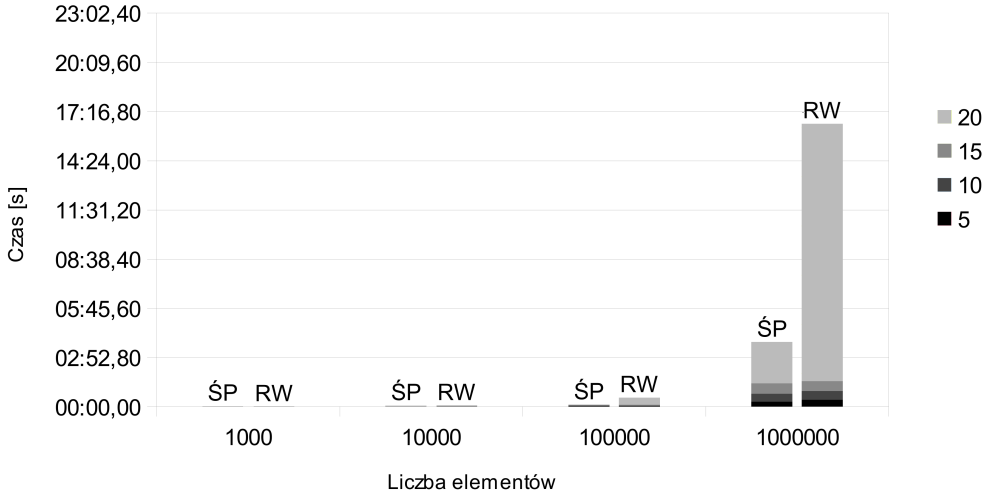
Tablica 6.3: Czasy wykonania zapytania wyszukującego potomków korzenia (w sekundach)

Liczba rekordów	5 poziomów			10 poziomów		
	ŚP	RW	Procentowe przyspieszenie	ŚP	RW	Procentowe przyspieszenie
1 000	00.07	00.14	50.00%	00.08	00.12	66.67%
10 000	00.26	00.28	92.86%	00.30	00.35	85.71%
100 000	01.16	01.81	64.09%	01.79	02.18	82.11%
1 000 000	17.35	23.94	72.47%	28.26	31.16	90.69%

Liczba rekordów	15 poziomów			20 poziomów		
	ŚP	RW	Procentowe przyspieszenie	ŚP	RW	Procentowe przyspieszenie
1 000	00.10	00.18	55.56%	00.12	00.35	34.29%
10 000	01.04	00.42	247.62%	01.20	02.56	46.88%
100 000	02.20	02.55	86.27%	02.91	24.56	11.85%
1 000 000	37.35	35.01	106.68%	02:25.23	15:03.76	16.07%

Znajdowanie korzenia drzewa

W tym teście badaliśmy czasy realizacji zapytania o korzeń drzewa, w którym znajduje się zadany węzeł. Wyniki eksperymentów przedstawiono w tablicy 6.5 oraz na rysunku 6.8. Widać w nich podobną anomalię, jak w teście zapytań o potomków (w tym przypadku dla 10 000 rekordów i 15 poziomów hierarchii). Ma ona analogiczne wyjaśnienie do przedstawionego powyżej. Oprócz tej jednej konfiguracji metoda ścieżek pełnych okazała się być bardziej wydajna niż rozwinięcie wszerz.

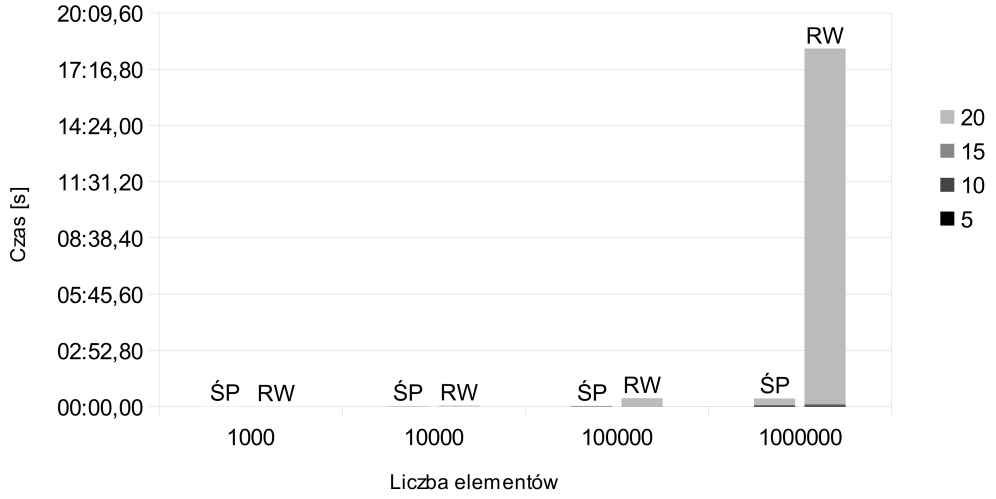


Rysunek 6.6: Wykres czasów wykonania zapytania o potomków korzenia. Poszczególne krzywe odpowiadają różnym poziomom głębokości hierarchii (5, 10, 15 i 20).

Tablica 6.4: Czasy wykonania zapytania wyszukującego potomków węzła wewnętrznego (w sekundach).

Liczba rekordów	5 poziomów			10 poziomów		
	ŚP	RW	Procentowe przyspieszenie	ŚP	RW	Procentowe przyspieszenie
1 000	00.06	00.09	66.67%	00.07	00.07	100.00%
10 000	00.10	00.14	71.43%	00.12	00.15	80.00%
100 000	00.73	00.70	104.29%	00.78	00.69	113.04%
1 000 000	01.09	01.51	72.19%	01.52	02.68	56.72%

Liczba rekordów	15 poziomów			20 poziomów		
	ŚP	RW	Procentowe przyspieszenie	ŚP	RW	Procentowe przyspieszenie
1 000	00.05	00.06	83.33%	00.08	00.31	25.81%
10 000	00.10	00.11	90.91%	00.21	02.48	8.47%
100 000	00.18	00.19	94.74%	00.30	24.63	1.22%
1 000 000	01.72	03.13	54.95%	21.18	18:12.84	1.94%



Rysunek 6.7: Wykres czasów wykonania zapytania o potomków węzła wewnętrznego. Poszczególne krzywe odpowiadają różnym poziomom głębokości hierarchii (5, 10, 15 i 20).

Podobnie jak w eksperymencie opisanym powyżej (wybór potomków danego węzła) przeprowadziliśmy również eksperymenty z PostgreSQL. I tak jak poprzednio, wyniki tych testów wskazały zdecydowaną dominację natywnej implementacji zapytań rekurencyjnych SQL:1999.

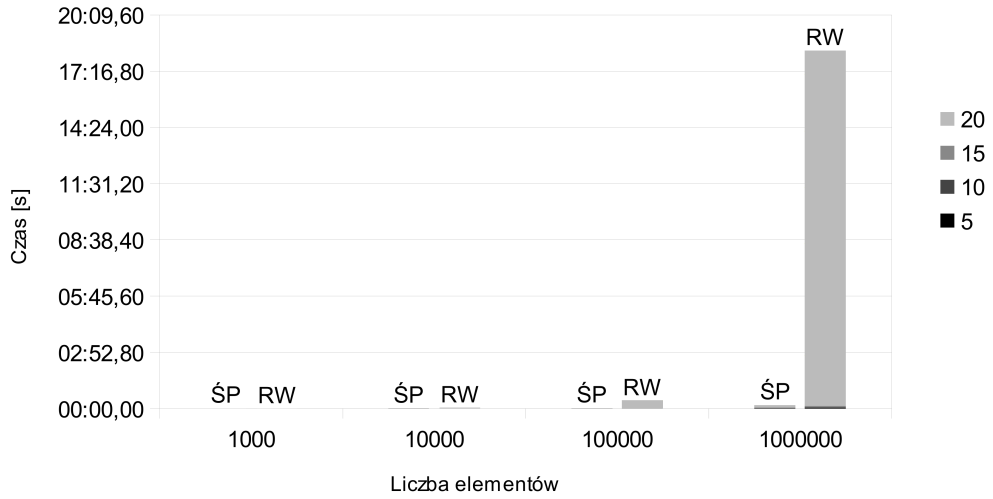
Synchronizacja danych

W tym teście zbadaliśmy narzuty spowodowane koniecznością synchronizacji danych redundantnych po każdej modyfikacji danych podstawowych. Te narzuty dotyczą tylko metody ścieżek pełnych, ponieważ rozwinięcie wszerz nie korzysta z danych wtórnych. Wyniki doświadczeń dowodzą, że narzuty związane z synchronizacją są umiarkowane. Metoda ścieżek pełnych może być zatem z powodzeniem stosowana w aplikacjach. Rezultaty eksperymentów zebrano w tablicy 6.6 i podsumowano na rysunku 6.9. Eksperyment pozwala oszacować wielkość narzutu przy pojedynczej operacji na hierarchii. Proponowane rozwiązanie pomyślane jest jako tania alternatywa (*open-source*) dla mniejszych firm i danych średniej wielkości.

Tablica 6.5: Czasy wykonania zapytania o korzeń drzewa (w sekundach).

Liczba rekordów	5 poziomów			10 poziomów		
	ŚP	RW	Procentowe przyspieszenie	ŚP	RW	Procentowe przyspieszenie
1 000	00.03	00.09	33.33%	00.03	00.07	42.86%
10 000	00.07	00.14	50.00%	00.14	00.15	93.33%
100 000	00.38	00.70	54.29%	00.37	00.69	53.62%
1 000 000	00.55	01.51	36.42%	00.92	02.68	34.33%

Liczba rekordów	15 poziomów			20 poziomów		
	ŚP	RW	Procentowe przyspieszenie	ŚP	RW	Procentowe przyspieszenie
1 000	00.03	00.06	50.00%	00.03	00.31	9.68%
10 000	00.14	00.11	127.27%	00.12	02.48	4.84%
100 000	00.12	00.19	63.16%	00.17	24.63	0.69%
1 000 000	01.05	03.13	33.55%	08.81	18:12.84	0.81%



Rysunek 6.8: Wykres czasów wykonania zapytania o korzeń drzewa. Poszczególne krzywe odpowiadają różnym poziomom głębokości hierarchii (5, 10, 15 i 20).

Tablica 6.6: Czasy synchronizacji tabeli redundantnej `Paths` w reakcji na modyfikację jednego wiersza w tabeli podstawowej `Emp`.

Liczba rekordów	Poziomy			
	5	10	15	20
1 000	00.10	00.14	00.22	00.37
10 000	00.24	00.51	00.97	01.73
100 000	01.55	04.44	08.75	16.24
1 000 000	15.59	46.15	01:29.02	06:24.13

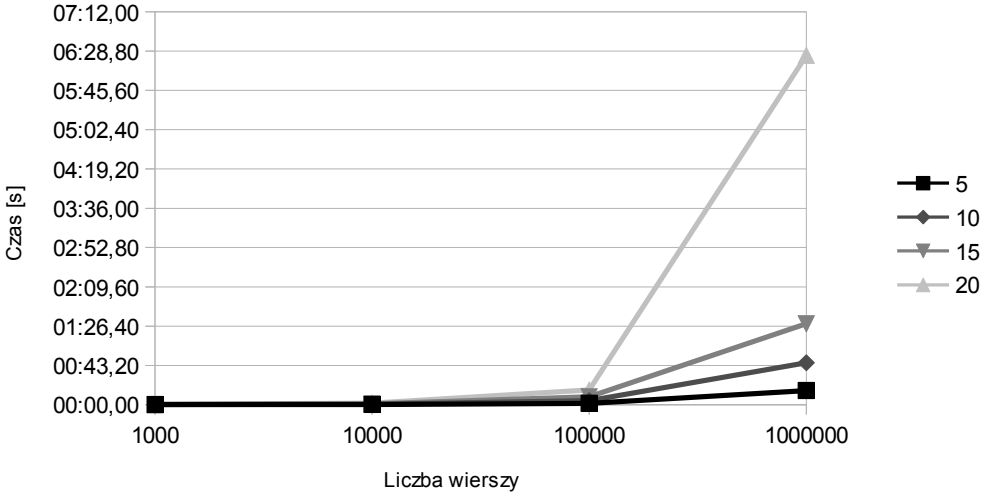
Wpływ względnej częstości zapytań i modyfikacji

Metody korzystające z danych redundantnych są oczywiście tym bardziej efektywne im większy jest stosunek liczby zapytań do liczby modyfikacji. W tym eksperymencie zbadaliśmy łączną wydajność ciągów operacji złożonych z zapytań i modyfikacji dla metody ścieżek pełnych i rozwinięcia wszerz. Każdy z czterech testów składał się ze stu operacji i zawierał odpowiednio 5, 10, 15 oraz 20% zapytań modyfikujących tabelę bazową. Testy przeprowadzono na bazach danych zawierających odpowiednio 100 000 oraz 1 000 000 rekordów w tabeli bazowej oraz 20 poziomów zagłębienia. Wyniki testów przedstawiono w tablicy 6.7. Wynika z nich, że w aplikacjach, w których znacznie przeważają zapytania, wydajność metody ścieżek pełnych znacznie przewyższa wydajność rozwinięcia wszerz. Jeśli modyfikacji jest jednak więcej niż kilka procent, lepiej skorzystać z metody niekorzystającej z danych redundantnych.

Tablica 6.7: Porównanie wydajności metody ścieżek pełnych z rozwinięciem wszerz dla różnych względnych częstości zapytań i modyfikacji.

Liczba rekordów	5% modyfikacji			10% modyfikacji		
	ŚP	RW	Proc. przysp.	ŚP	RW	Proc. przysp.
100 000	00:19.03	00:27.34	70%	00:44.14	00:27.07	163%
1 000 000	01:45.94	04:46.64	37%	03:54.83	04:44.81	82%

Liczba rekordów	15% modyfikacji			20% modyfikacji		
	ŚP	RW	Proc. przysp.	ŚP	RW	Proc. przysp.
100000	01:18.29	00:27.41	286%	01:00.12	00:27.49	219%
1000000	06:45.94	04:43.21	143%	09:22.22	04:43.75	198%



Rysunek 6.9: Wykres czasów synchronizacji tabeli redundantnej `Paths` w reakcji na modyfikację jednego wiersza w tabeli podstawowej `Emp`. Poszczególne krzywe odpowiadają różnym poziomom głębokości hierarchii (5, 10, 15 i 20).

6.3.2 Cztery metody materializujące dane pomocnicze

W eksperymentach opisanych w niniejszym punkcie porównaliśmy wydajność czterech metod wykonywania zapytań rekurencyjnych, które materializują dane redundantne. Testy przeprowadzono z użyciem systemu zarządzania bazą danych MySQL w wersji 5.5 z buforem danych 2 GB na komputerze z procesorem AMD Phenom II 3.4 GHz, pamięcią 8 GB RAM oraz dwoma dyskami Caviar Black 7400 Rpm po 500 GB pojemności każdy.

Każde zapytanie zostało wykonane 5 razy. Czasy najlepszy i najgorszy zostały odrzucone. Z pozostałych trzech wyników została wzięta średnia czasów. Zostały użyte zapytania 3.1 do wyszukania potomków dowolnego węzła (pracownika) oraz potomków węzła, który jest korzeniem oraz zapytanie 3.2 z benchmarku z rozdziału 3.

Dane testowe znajdowały się w tabeli `Emp`. Zależnie od konfiguracji testu zawierała ona 10, 15 lub 20 poziomów zagłębienia hierarchii oraz 10^5 lub 10^6 rekordów. Łącznie oznaczało to 6 różnych zbiorów danych.

Dodatkowo zostały założone indeksy w tabeli `LogPatsh` na kolumnach `eid` oraz `bid`, w tabeli `Emp` na kolumnach `right`, `left` oraz `paths`.

Każdą tabelę z wynikami testów podzieliliśmy na dwie części. W pierw-

szej znajdują się wyniki dla tabeli z 10^5 rekordów, druga zawiera natomiast wyniki dla tabeli z 10^6 rekordów. W poszczególnych kolumnach znajdują się czasy działania operacji dla testowanych metod: metody ścieżek pełnych ($\acute{S}P$), metody ścieżek logarytmicznych ($\acute{S}L$), metody zbiorów zagnieżdżonych (ZZ) oraz metody ścieżek zmaterializowanych ($\acute{S}Z$).

Tablica 6.8: Czas budowy pomocniczych struktur danych.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:47,94	00:00:15,90	00:00:29,41	00:00:12,59
15	00:01:26,20	00:00:17,67	00:00:29,26	00:00:13,23
20	00:02:54,21	00:00:21,82	00:00:29,55	00:00:13,54
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:22:13,21	00:04:53,27	00:34:59,03	00:05:16,97
15	01:00:24,07	00:05:30,94	00:39:47,73	00:05:10,29
20	02:03:50,40	00:06:24,73	00:39:25,69	00:05:31,39

Tablica 6.9: Czasy wykonania zapytania o potomków korzenia.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:01,57	00:00:07,99	00:00:00,51	00:00:00,56
15	00:00:01,91	00:00:08,90	00:00:00,51	00:00:00,60
20	00:00:03,64	00:00:05,27	00:00:00,51	00:00:00,64
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:30,16	00:04:32,97	00:00:10,27	00:00:09,70
15	00:00:40,09	00:07:08,23	00:00:10,96	00:00:10,09
20	00:00:49,72	00:07:44,32	00:00:10,94	00:00:10,08

Przeprowadzone eksperymenty odpowiadają tym opisanym w p. 6.3.1. Po pierwsze zbadaliśmy koszt tworzenia dodatkowych struktur danych niezbędnych do realizacji zapytań każdą z rozważanych metod. Były to dodatkowe tabele `Paths` (ŚP) i `LogPaths` (ŚL) oraz dodatkowe kolumny w tabeli `EMP`: `left` i `right` (ZZ) oraz `paths` (ŚZ). Wynik tego eksperymentu znajduje się w tablicy 6.8. Eksperymenty drugi i trzeci polegały na wykonywaniu zapytań o wszystkich potomków odpowiednio korzenia lub węzła wewnętrznego.

Tablica 6.10: Czasy wykonania zapytania o potomków węzła wewnętrznego.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,01	00:00:00,02	00:00:00,02	00:00:00,02
15	00:00:00,02	00:00:00,02	00:00:00,01	00:00:00,02
20	00:00:00,07	00:00:00,01	00:00:00,01	00:00:00,02
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,27	00:00:00,06	00:00:00,03	00:00:00,48
15	00:00:00,40	00:00:00,12	00:00:00,03	00:00:00,44
20	00:00:00,36	00:00:00,12	00:00:00,03	00:00:00,42

Tablica 6.11: Czasy wykonania zapytania o korzeń drzewa.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,00	00:00:00,01	00:00:00,01	00:00:00,01
15	00:00:00,00	00:00:00,05	00:00:00,01	00:00:00,01
20	00:00:00,01	00:00:00,05	00:00:00,01	00:00:00,01
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,01	00:00:00,01	00:00:00,01	00:00:00,01
15	00:00:00,01	00:00:00,07	00:00:00,01	00:00:00,01
20	00:00:00,01	00:00:00,08	00:00:00,01	00:00:00,01

W tablicy 6.9 przedstawiono wyniki dla korzenia, a w tablicy 6.10 dla węzła wewnętrznego. Eksperyment czwarty dotyczył zapytania o korzeń hierarchii dla zadanego węzła. Jego wyniki znajdują się w tablicy 6.11.

Narzuty operacji modyfikujących dane

Testując cztery metody opisane w niniejszym rozdziale, zdecydowaliśmy się na inny sposób zbadania narzutów spowodowanych koniecznością synchronizacji danych redundantnych. Przeprowadziliśmy oddzielne eksperymenty dla każdej operacji: wstawienia, usunięcia i modyfikacji. Opisywane w niniejszym rozdziale rozwiązanie z użyciem narzędzi ORM jest pomyślane jako tania alternatywa dla mniejszych firm i danych średniej wielkości. Z tego powodu operacje te dotyczą pojedynczego węzła (podobnie jak w 6.3.1). Pierwszy test dotyczył operacji dodawania liścia do hierarchii. Czasy wykonania

Tablica 6.12: Czasy wykonania operacji dodania nowego węzła do hierarchii.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,14	00:00:00,08	00:00:05,22	00:00:00,06
15	00:00:00,20	00:00:00,09	00:00:04,35	00:00:00,06
20	00:00:00,20	00:00:00,08	00:00:16,54	00:00:00,05
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,15	00:00:00,11	00:07:47,67	00:00:00,05
15	00:00:00,17	00:00:00,12	00:08:27,64	00:00:00,05
20	00:00:00,16	00:00:00,12	00:08:16,85	00:00:00,05

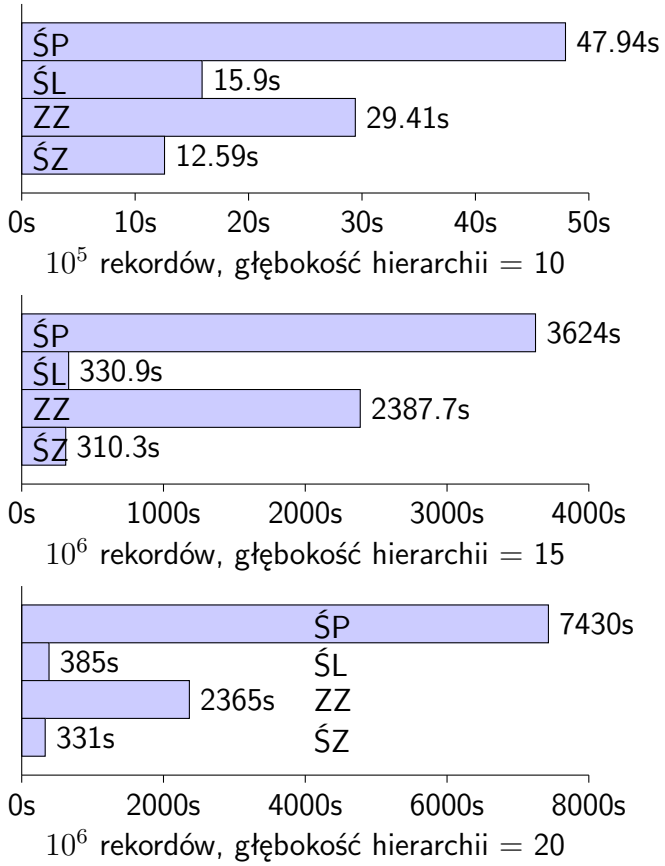
Tablica 6.13: Czasy wykonania operacji usuwania węzła z hierarchii.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,05	00:00:00,08	00:00:05,57	00:00:00,06
15	00:00:00,05	00:00:00,08	00:00:06,67	00:00:00,05
20	00:00:00,04	00:00:00,09	00:00:16,92	00:00:00,04
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:00,04	00:00:00,07	00:09:53,84	00:00:00,05
15	00:00:00,04	00:00:00,08	00:09:37,94	00:00:00,04
20	00:00:00,04	00:00:00,09	00:08:59,66	00:00:00,04

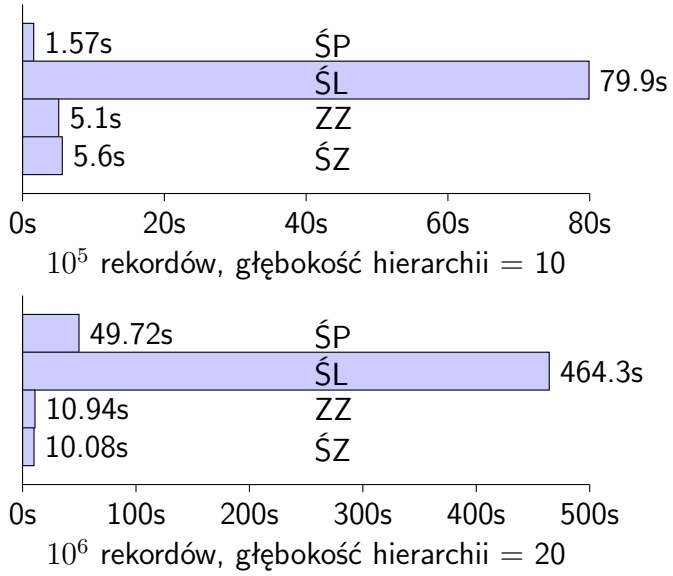
Tablica 6.14: Czasy wykonania operacji zmiany rodzica węzła.

Liczba poziomów	100 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:00:08,53	0:00:26,60	00:00:19,28	00:00:00,61
15	00:00:16,01	00:00:51,14	00:00:08,70	00:00:00,64
20	00:00:52,75	00:01:28,36	00:00:24,15	00:00:01,19
Liczba poziomów	1 000 000 rekordów			
	ŚP	ŚL	ZZ	ŚZ
10	00:04:30,89	00:13:37,25	00:00:22,51	00:00:00,47
15	00:07:49,17	00:35:06,68	00:01:20,40	00:00:00,08
20	00:13:38,99	00:57:39,43	00:00:28,07	00:00:00,92

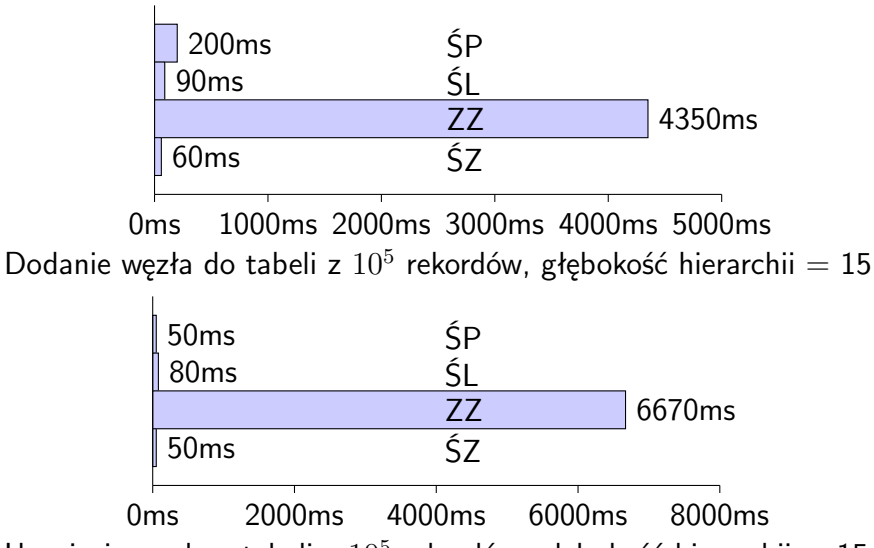
tej operacji przedstawione w tablicy 6.12 obejmują całość wstawienia, tzn. aktualizację danych podstawowych *oraz* danych redundantnych. Drugi test dotyczył operacji usuwania węzła z tabeli bazowej. Usuwane były tylko węzły nie mające węzłów podrzędnych. Usunięcie węzła nie będące liściem wymaga bowiem wcześniejszej aktualizacji danych polegającej na odczepieniu jego węzłów podrzędnych. Tak jak powyżej, czasy wykonania operacji usuwania przedstawione w tablicy 6.13 obejmują całość tej czynności, tzn. aktualizację danych podstawowych *oraz* danych redundantnych. Trzeci test dotyczył operacji zmiany rodzica węzła. Tak jak powyżej, czasy wykonania aktualizacji przedstawione w tablicy 6.14 obejmują całość tej czynności, tzn. aktualizację danych podstawowych *oraz* danych redundantnych.



Rysunek 6.10: Porównanie czasów budowy pomocniczych struktur danych (tabel lub kolumn).



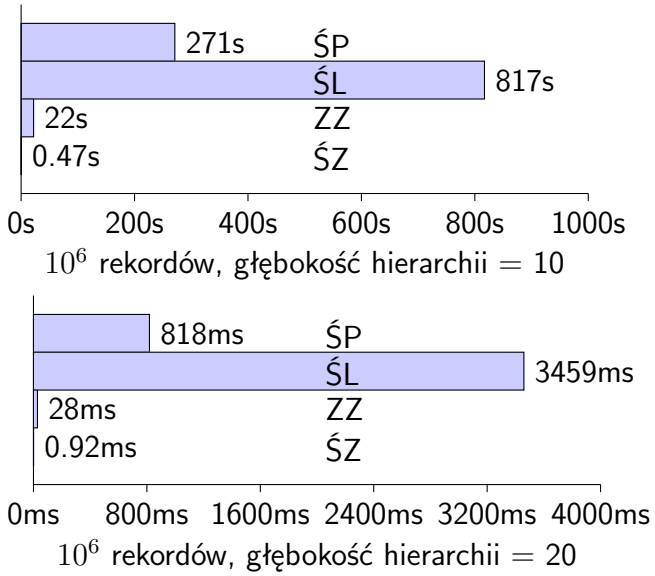
Rysunek 6.11: Porównanie czasów wykonania zapytania o potomków węzła wewnętrznego.



Rysunek 6.12: Porównanie czasów wykonania operacji wstawienia i usuwania węzła.

6.4 Analiza zaproponowanego rozwiązania

Zapytania rekurencyjne polegają w istocie na wyznaczeniu domknięcia przechodniego grafu lub jego podzbioru. Metody obliczania domknięcia prze-



Rysunek 6.13: Porównanie czasów wykonania operacji zmiany rodzica węzła.

chodniego relacji składowanych w bazach danych były już przedmiotem badań innych naukowców. Zaproponowano na przykład rozszerzenie algebry relacyjnej o operację domknięcia przechodniego [18], opracowano algorytmy iteracyjne i nieiteracyjne w celu jak najbardziej efektywnego obliczania domknięcia przechodniego dla grafu [20, 37, 19, 18]. Omawiana w niniejszym rozdziale metoda ścieżek pełnych w istocie materializuje domknięcie przechodnie hierarchii.

Celem istotnej części niniejszej rozprawy doktorskiej jest zbadanie możliwości i zasadności rozszerzania systemów odwzorowań obiektowo-relacyjnych o udogodnienia do wykonywania zapytań rekurencyjnych aplikacji. W wyniku przeprowadzonych badań chcieliśmy też ustalić, które sposoby odwzorowania zapytań rekurencyjnych aplikacji mogą okazać się przydatne w praktyce. Dążyliśmy też do opracowania zaleceń, kiedy poszczególne metody odwzorowania mogą być lepsze od pozostałych.

W niniejszym rozdziale przedstawiono cztery metody odwzorowania zapytań rekurencyjnych, które korzystają ze zmaterializowanych pomocniczych struktur danych. Opis trzech z nich (poza metodą ścieżek logarytmicznych) można znaleźć w [10]. W celu eksperymentalnej weryfikacji przydatności omawianych metod odwzorowania, zaimplementowaliśmy je w postaci prototypowego rozszerzenia najpopularniejszego systemu odwzorowania obiektowo-relacyjnego Hibernate.

Owe cztery metody można podzielić na dwie grupy. Pierwsza grupa zawiera metody, w których tworzona jest dodatkowa tabela do przechowywa-

nia danych redundantnych. Są nimi: metoda ścieżek pełnych oraz metoda ścieżek logarytmicznych. Takie metody są prostsze do wdrożenia, ponieważ nie ingerują w strukturę istniejących tabel. Rozmiar danych redundantnych dla metody ścieżek pełnych to $O(nm)$, gdzie n oznacza rozmiar danych w tabeli podstawowej, a m maksymalną głębokość struktury hierarchicznej. W przypadku metody ścieżek logarytmicznych rozmiar ten wynosi jedynie $O(n \log m)$. Zdecydowaną zaletą metody ścieżek logarytmicznych jest istotnie mniejszy czas potrzebny na utworzenie tabeli z danymi redundantnymi niż w przypadku metody ścieżek pełnych. Gdy weźmiemy pod uwagę wszystkie rozważane metody, czas budowy danych pomocniczych dla metody ścieżek logarytmicznych jest porównywalny tylko z metodą ścieżek zmaterializowanych. Na rysunku 6.10 przedstawiono porównanie czasów budowy redundantnych struktur danych dla poszczególnych metod odwzorowania zapytań rekurencyjnych. Niestety wydajność zapytań w metodzie ścieżek logarytmicznych jest niższa, ponieważ zapytania rekurencyjne aplikacji są odwzorowywane na sumę mnogościową $O(\log m)$ zapytań będących $O(\log m)$-krotnymi samozłączeniami tabeli pomocniczej. Dla głębokości hierarchii równej na przykład 20, należy wykonać 5 zapytań połączonych ze sobą operatorem `union`, z których każde jest 5-krotnym samozłączeniem tabeli `LogPaths`. Na rysunku 6.11 pokazano porównanie czasów wykonania zapytania wybierającego potomków węzła wewnętrznego dla wszystkich opisywanych metod. Metoda ścieżek pełnych może być natomiast bardziej użyteczna w przypadku gromadzenia danych stopniowo. Jak pokazano na rysunku 6.12, zarówno tworzenie nowych węzłów, jak i ich synchronizacja są dość szybkie. Operacja modyfikacji danych jest jednak już dużo bardziej kosztowna, co widać na rysunku 6.13.

Druga grupa metod wiąże się z modyfikacją istniejących tabel podstawowych poprzez dodanie dodatkowych kolumn. Taka ingerencja w schemacie bazy danych może być zbyt trudna do wdrożenia w istniejących i działających aplikacjach. Może być nie do zaakceptowania dla administratora bazy danych działającej w środowisku produkcyjnym. Do tych metod należą: metoda zbiorów zagnieżdżonych oraz metoda ścieżek zmaterializowanych. Pierwsza z nich jest zoptymalizowana pod kątem pobierania potomków dowolnego węzła. Jest ona natomiast zdecydowanie najkosztowniejsza w przypadku wykonania operacji `update`. Operacje, które modyfikują dane są również wolniejsze dla zbiorów zagnieżdżonych niż dla ścieżek zmaterializowanych. Metoda zbiorów zagnieżdżonych jest jednak zdecydowanie najszybsza, gdy główną operacją wykonywaną przez aplikację jest rekonstrukcja poddrzew (znajdowanie wszystkich potomków). Metoda ścieżek zmaterializowanych okazuje się najbardziej efektywną z wszystkich porównywanych metod. Niestety, jest to też jedyna metoda, która przechowuje dane złożone w jednej kolumnie. To

oznacza brak pierwszej postaci normalnej. Nie musi to być jednak przeszkoda w jej stosowaniu.

6.5 Funkcja kosztu dla obciążenia

Eksperymenty opisane w p. 6.3 oraz analiza przeprowadzona w p. 6.4 pozwalają pokusić się o próbę zdefiniowana analitycznej funkcji kosztu dla zadanego obciążenia (ang. *workload*) bazy danych zapytaniami rekurencyjnymi i operacjami modyfikacji danych. Taka funkcja może być stosowana w celu wskazania najlepszej metody materializacji ścieżek w celu wsparcia zapytań rekurencyjnych, gdy stosowany system zarządzania bazą danych ich nie implementuje. Budując taką funkcję należy wziąć pod uwagę zapytania oraz operacje modyfikacji danych. Te ostatnie mogą stanowić istotny ułamek kosztów obciążenia bazy danych, ponieważ obejmują nie tylko modyfikację danych bazowych, ale także odpowiednią pielęgnację zmaterializowanych danych redundantnych.

Eksperymenty opisane w p. 6.3 okazały się niewystarczające ze względu na nieuwzględnienie w nich zmiennej selektywności zapytań. Z tego powodu testy te powtórzyliśmy na użytek badań nad poszukiwaniem empirycznej funkcji kosztu dla operacji. Poniżej znajduje się raport z tych dodatkowych testów oraz badań statystycznych nad ich wynikami.

6.5.1 Parametry eksperymentu

Przygotowaliśmy następujące środowisko eksperymentalne. Skorzystaliśmy z komputera z procesorem Intel Core i5 3570 (3.4/3.8 GHz) oraz 32 GiB RAM (DDR3, 1600 MHz). System i oprogramowanie były składowane na dysku SSD Kingston 120 GiB. Dane utrwalono na dysku 4x Caviar Black 1TB w postaci Logic Volume. Systemem operacyjnym był Debian 7.3. Systemem zarządzania bazą danych był PostgreSQL 9.1. Użyliśmy zwyczajnej wersji PostgreSQL jaka dostarczana jest z wybranym systemem operacyjnym.

Łącznie przetestowaliśmy 12 kopii baz danych. Rozważaliśmy cztery rozmiary tabel, tzn. $10^4, 10^5, 10^6, 10^7$ i trzy głębokości hierarchii, tj. $10, 15, 20$. Jako że wzięliśmy pod uwagę wszystkie możliwe kombinacje tych parametrów, ostatecznie mieliśmy 12 baz danych. Testowaliśmy zapytania o drzewa rozpięte przez wierzchołek z rozmaitymi selektywnościami. Selektywność zapytania miała dwa aspekty. Pierwszym z nich jest liczba zwróconych wierszy, zaś drugim liczba poziomów zwróconego poddrzewa. Liczba zwróconych wierszy wahała się od 3 do $2 \cdot 10^5$,podczas gdy wysokości zwróconych drzew znajdowały się w przedziale $\langle 2, 13 \rangle$. Dla każdego zestawu danych wy-

konaliśmy 1500 zapytań. Zaobserwowaliśmy, że początkowe wykonania dla świeżo uruchomionej bazy danych dają niestabilne wyniki, więc odrzucaliśmy pierwsze 300 wykonań (tj. około 20% całego obciążenia). Założyliśmy, że te początkowe, pomijane w wynikach wykonania, odpowiednio wypełniają (rozgrzewają) inicjalnie pusty (zimny) bufor danych. Późniejsze wykonania zapytań wykazały się odpowiednią stabilnością, aby użyć ich w dalszej analizie.

Każde wykonanie zapytania dotyczyło potencjalnie innego *wiersza początkowego*. Wynikiem zapytania było całe drzewo, którego ten wiersz był korzeniem. Dla każdego wykonania odnotowaliśmy czas działania, liczbę zwróconych wierszy i wysokość zwróconego poddrzewa. Dla porównania z metodami materializacji, przetestowaliśmy także implementację za pomocą natywnych zapytań rekurencyjnych SQL:1999.

6.5.2 Analiza zapytań

Wyniki eksperymentów wg parametrów opisanych w p. 6.5.1 zostały zanalizowane za pomocą narzędzia statystycznego SSPS. Użyliśmy go do budowy modeli regresji kosztu każdej z metod materializacji. Poniżej znajduje się lista zmiennych objaśniających (predyktorów) wraz z ich opisem:

dbSize Wielkość danych, tzn. liczba wierszy w tabeli bazowej.

dbDepth Głębokość danych, tzn. głębokość najgłębszego wiersza w hierarchii składowanej w tabeli bazowej.

qSel Selektywność zapytania, tzn. liczba zwróconych wierszy.

qDepth Głębokość wyniku zapytania, tzn. głębokość najgłębszego wiersza w wyniku zapytania względem wiersza początkowego.

Dwie pierwsze z tych zmiennych zależą wyłącznie od użytej bazy danych. Mają 12 możliwych kombinacji wartości (por. p. 6.5.1). Dwie ostatnie zmienne istotnie różnią się dla poszczególnych wykonań zależnie od wiersza początkowego.

Zmienna objaśniania (oznaczana Q_x, gdzie x to skrót nazwy metody materializacji plus natywny SQL:1999) to czas wykonania zapytania w sekundach.

Ścieżki pełne

Okazało się, że wydajność zapytań w metodzie ścieżek pełnych była istotnie zależna od selektywności zapytania i wielkości bazy danych. Co więcej,

zależy także od iloczynu selektywności i wysokości zwracanego drzewa. Jest to zgodne z naturą tej metody, ponieważ liczba wszystkich redundantnych wierszy jest równa sumie głębokości wszystkich wierzchołków. Metoda regresji pozwoliła wygenerować następującą formułę przy bardzo dobrej wartości współczynnika determinacji R^2 równej 0.998.

$$Q_{SP} = 0.001 + 1.298 \cdot 10^{-6} \cdot qSel + 4.262 \cdot 10^{-11} \cdot dbSize + 3.346 \cdot 10^{-8} \cdot qSel \cdot qDepth$$

Ścieżki logarytmiczne

Empirycznie wyznaczona formuła dla zapytań przy metodzie ścieżek logarytmicznych zależy od logarytmu wysokości zwracanego drzewa, selektywności zapytania i wielkości bazy danych. Jest to w pewnym sensie zgodne z intuicją, ponieważ liczba wierszy redundantnie składowanych dla każdego węzła jest liniowa względem logarytmu jego głębokości w hierarchii. Wartość współczynnika determinacji R^2 wyniosła 0.882. Niestety otrzymana formuła ma bardzo dużą ujemną stałą. To sprawia, że użycie tej formuły przy małych danych jest niemożliwe (wyjdą bowiem ujemne czasy wykonania). Niestety, mimo wielu prób nie udało się znaleźć lepszej formuły metodą regresji.

$$\begin{aligned} Q_{SL} &= -1.863 + 6.877 \cdot 10^{-7} \cdot \ln^2(qDepth) \cdot dbSize \\ &+ 1.980 \cdot 10^{-5} \cdot qSel \cdot \ln(qDepth) \cdot \ln(dbSize) \end{aligned}$$

Ścieżki zmaterializowane

Metodą regresji otrzymaliśmy następującą formułę kosztu dla zapytań przy zastosowaniu metody ścieżek zmaterializowanych. Zależy ona od wielkości bazy danych i głębokości zwróconego poddrzewa. Wartość współczynnika determinacji R^2 była niezwykle wysoka i wyniosła 1.000. Zależność od wielkości bazy danych jest zgodna z intuicją. Nieindeksowalne pole zawierające w każdym wierszu pełną ścieżkę do korzenia (brak pierwszej postaci normalnej) sprawia, że baza danych musi wykonać przegląd pełny tabeli.

$$Q_{SZ} = 0.005 + 1.365 \cdot 10^{-7} \cdot dbSize + \cdot 10^{-9} \cdot dbSize \cdot qDepth$$

Zbiory zagnieżdżone

W przypadku metody zbiorów zagnieżdżonych zapytania o poddrzewo wyznaczone przez wiersz początkowy to zwyczajne zapytanie zakresowe oparte na dodatkowych (redundantnych) kolumnach. Wygenerowaliśmy wiele formuł regresji dla czasów wykonania zapytań tą metodą i żadna z nich nie

wyglądała wystarczająco sensownie i intuicyjnie. Oto jedna z nich mająca współczynnik determinacji R^2 równy 0.700. Niestety, ma ona znaczny ujemny stały składnik:

$$Q_{ZZ} = -0.543 + 0.289 \cdot \frac{sel}{dbSize} + 0.058 \cdot \ln(dbSize) + 4.622 \cdot 10^{-7} \cdot \ln(dbSize) \cdot qSel$$

Teoretycznie czas wykonania tego zapytania powinien zależeć jedynie od liczby zwróconych wierszy, ponieważ na kolumnach selekcji był założony indeks w postaci B^+-drzewa. Niestety formuła regresji oparta tylko na selektywności ma bardzo mały współczynnik determinacji R^2 równy 0.099.

$$Q'_{ZZ} = 0.053 + 4.835 \cdot 10^{-6} \cdot qSel$$

Wygląda na to, że mechanizmy optymalizacyjne PostgreSQL są na tyle złożone, że wykluczają jakąkolwiek prostą formułę empiryczną pojedynczego zapytania opartą na samej selektywności.

Natywna rekurencja SQL:1999

Przeprowadziliśmy także testy i zbudowaliśmy formuły regresji dla natywnej implementacji rekurencji dostępnej w PostgreSQL. Koszt wykonania natywnego zapytania, zgodnie z intuicją, zależy od selektywności zapytania i wielkości bazy danych. Poniższa formuła dla natywnej rekurencji SQL:1999 charakteryzuje się wysokim współczynnikiem determinacji R^2 równym 0.958.

$$Q_{SQL:1999} = qSel \cdot 2.993 \cdot 10^{-6} + dbSize \cdot 1.195 \cdot 10^{-9}$$

6.5.3 Analiza operacji modyfikacji

Analizując zdania modyfikujące dane, przyjęliśmy inne podejście. Ustaliliśmy liczbę wymaganych zdań modyfikujących dane redundantne i liczę wierszy przez nie zmienianych. Następnie określiliśmy eksperymentalnie koszt każdego pojedynczego zdania modyfikującego dane. Okazało się, że koszt operacji INSERT wynosi $0.020 + 1.5 \cdot 10^{-5} \cdot r$, przy czym r to liczba wstawianych wierszy. Z kolei koszt jednego DELETE lub UPDATE to $0.020 + 4.0 \cdot 10^{-4} \cdot r$, przy czym r to liczba usuwanych lub modyfikowanych wierszy.

INSERT and DELETE

Te dwie operacje mogą dotyczyć tylko liści hierarchii ze względu na więzy integralności referencyjnej. W przypadku metody ścieżek zmaterializowa-

nych mamy tylko jedną operację INSERT/DELETE dla każdego zmienianego wiersza (tylko wiersz modyfikowanego liścia musi być zmieniany). Dla zbiorów zagnieżdżonych niestety trzeba przebudować całą strukturę przy każdej zmianie danych. Musimy więc wykonać DELETE wszystkich wierszy i wstawić je na nowo. Można to uczynić w jednej operacji bazodanowej.

W przypadku ścieżek pełnych i ścieżek logarytmicznych musimy wstawić lub odpowiednio usunąć ścieżkę od modyfikowanego wiersza do korzenia. Można to zrobić za pomocą jednej operacji. Przypuśćmy, że głębokość wstawianego lub usuwanego wiersza to d. Wówczas musimy dodać lub usunąć d redundantnych wierszy dla metody ścieżek pełnych oraz odpowiednio $\log d$ redundantnych wierszy dla metody ścieżek logarytmicznych.

UPDATE

Zdanie UPDATE jest interesujące z punktu widzenia naszych rozważań, jeśli powoduje zmianę w hierarchii, tzn. przenosi wierzchołek z jednego miejsca w inne. Oznaczmy starą głębokość tego wierzchołka przez d, zaś nową głębokość tego wierzchołka przez d'. Co więcej, przyjmijmy, że poddrzewo, którego jest korzeniem, ma s wierzchołków. Ścieżki z tego poddrzewa są modyfikowane przez rozważaną operację UPDATE.

Tak jak powyżej, w przypadku metody zbiorów zagnieżdżonych każda modyfikacja oznacza przebudowę całej struktury redundantnej. Dla ścieżek zmaterializowanych musimy zmodyfikować wszystkie wiersze w poddrzewie zmienianego wierzchołka poprzez zmianę przedrostków zapisanych w nich ścieżek. Można to zrobić jednym zdaniem UPDATE zmieniającym s podrzędnych wierszy.

W przypadku ścieżek pełnych musimy usunąć wszystkie ścieżki z poddrzewa do starych przodków modyfikowanego wiersza i wstawić ścieżki do nowych jego przodków. To oznacza dwie operacje: DELETE sd starych wierszy i INSERT sd' nowych wierszy.

Dla ścieżek logarytmicznych niestety cała struktura ścieżek w poddrzewie musi zostać przebudowana. To oznacza dwie operacje: DELETE $s \log d$ starych wierszy i INSERT $s \log d'$ nowych wierszy.

6.5.4 Wnioski

Ponowione eksperymenty dla poszczególnych metod materializacji ścieżek pozwoliły dla niektórych metod zweryfikować eksperymentalnie założenia o ich złożoności i wypracować empiryczne formuły regresji dla czasów wykonania w systemie zarządzania bazą danych PostgreSQL. Nie dla wszystkich metod

otrzymane formuły okazały się zgodne z intuicją. Dokonaliśmy też analizy kosztów realizacji modyfikacji danych dla poszczególnych metod.

Rozdział 7

Podsumowanie

Systemy odwzorowania obiektowo-relacyjnego są odpowiedzią na konieczność współpracy dominujących na rynku relacyjnych baz danych z aplikacjami napisanymi w obiektowych językach programowania. Ta współpraca wiąże się z rozwiązaniem problemu niezgodności modeli danych obu tych narzędzi: modelu relacyjnego i modelu obiektowego. Istniejące narzędzia do odwzorowania obiektowo-relacyjnego mają jedynie podstawowe funkcje do składowania i odczytu obiektów z relacyjnej bazy danych.

Niniejsza rozprawa jest poświęcona badaniu możliwości rozszerzenia systemów odwzorowania obiektowo-relacyjnego o nowe funkcje. Skupiliśmy się na jednym, jednak bardzo ważnym, aspekcie możliwego rozszerzenia.

Rozszerzeniem systemów odwzorowania obiektowo-relacyjnego, któremu poświęcona jest ta rozprawa to obsługa zapytań rekurencyjnych. Zapytania takie dotyczą danych hierarchicznych i grafowych, takich jak struktura zatrudnienia w firmie, kategorie i podkategorie produktów, rozkłady jazdy pociągów czy lotów samolotów. Przedstawiliśmy koncepcję rozszerzenia systemów odwzorowania obiektowo-relacyjnego o obsługę zapytań rekurencyjnych, zarówno wtedy gdy wykorzystywany system zarządzania bazą danych ma zaimplementowane takie zapytania, jak i wtedy, gdy jego dialekt SQL takich zapytań nie obejmuje. Skupiliśmy się na danych hierarchicznych, gdyż są to dane najczęściej występujące w praktycznych zastosowaniach (np. system USOS). Z eksperymentów z natywnymi implementacjami zapytań rekurencyjnych SQL:1999 [29] wynika jednak, że nawet one nie radzą sobie z dowolnymi danymi grafowymi. Przedstawione w niniejszej rozprawie metody można będzie w przyszłości rozszerzyć na dane grafowe, jakkolwiek nie będzie to zadanie banalne.

Przedstawiliśmy i zanalizowaliśmy siedem metod odwzorowania zapytań rekurencyjnych. Cztery z nich korzystają ze zmaterializowanych danych redundantnych, a trzy nie. Przedstawiliśmy prototypową implementa-

cję wszystkich tych siedmiu odwzorowań w Hibernate i przeprowadziliśmy eksperymenty nad ich wydajnością. Okazało się, że dla każdej z metod istnieją warunki, w których jest ona optymalna. Warto więc, by docelowy system odwzorowania obiektowo-relacyjnego wszystkie je oferował. Następnym zadaniem badawczym może być budowa automatycznego doradcy, który na podstawie zapytań rekurencyjnych wysyłanych przez aplikację wybierze optymalny sposób ich odwzorowania.

Rozszerzanie zbadane w niniejszej rozprawie ułatwi pracę głównie programistom aplikacyjnym. Może ono istotnie wpłynąć na wydajność tworzonych aplikacji. Zastosowanie opisanych rozszerzeń o zapytania rekurencyjne zwolni programistów z troski o składnię i semantykę takich zapytań, które mogą się w nieznacznym stopniu różnić u poszczególnych dostawców systemów zarządzania bazami danych. Proponowane rozszerzenia mogą też wpłynąć na czytelność aplikacji oraz zmniejszyć koszty pielęgnacji kodu źródłowego.

Zaproponowana metoda rozszerzenia systemów ORM otwiera drogę do dalszej pracy nad nowymi funkcjonalnościami. Zostały poczynione kolejne kroki, które można odnaleźć w pracach [3], [14].

Dodatek A

Skróty użyte w rozprawie

CTE - Common Table Expresion

CRUD - operacje Create, Read, Upadte, Delete

JDBC - Java DataBase Connectivity

JDO - Java Data Objects

JPA - Java Persitance API

JPC - Java Community Process

JSR - Java Specification Requests

ORM - Narzędzia odwzorowania obiektowo-relacyjnego

RW - Metoda Rozwinięcia Wszerz

SZBD - System Zarządzania Bazą Danych

ŚL - Metoda Ścieżek Logarytmicznych

ŚP - Metoda Ścieżek Pełnych

ŚZ - Metoda Ścieżek Zmaterializowanych

USOS - Uniwersytecki System Obsługi Studiów

ZZ - Metoda Zbiorów Zagnieżdżonych

Dodatek B

Przykładowe dane systemu USOS

W poniższej tabeli przedstawiono liczby rekordów tabel zawierających dane rekurencyjne. Jest to stan na dzień 15 lipca 2014 dla systemu USOS wykorzystywanego produkcyjnie przez Uniwersytet Mikołaja Kopernika w Toruniu.

Nazwa tabeli	Liczba wierszy
dz_grupy	251 635
dz_jednostki_organizacyjne	731
dz_kierunki_studiow	516
dz_log	142 967
dz_parametry	259
dz_prac_zatr	8 608
dz_programy_wymiany	4
dz_pytania_do_studentow	1 312
dz_szkoly	12 864
dz_terminy_grup_sptk	182 470
dz_transakcje	1 327 751
dz_wspolprace	2 010
dz_zmienne_konfig_srednich	4
dz_role_zaleznosci	38
dz_rankingi	2 344
dz_kolejnosc_etapow	4 090

W samej aplikacji USOS jest 50 zapytań rekurencyjnych (stan na 15 lipca 2014). Poniżej pokazano przykładowe zapytania rekurencyjne skopiowane z kodu źródłowego USOS.

Listing B.1: Zapytania rekurencyjne w aplikacji USOS

```
SELECT DISTINCT opis_do_suplementu
  FROM dz_jednostki_organizacyjne
 WHERE opis_do_suplementu IS NOT NULL
CONNECT BY kod = PRIOR jed_org_kod
START WITH kod IN (SELECT jed_org_kod
                     FROM dz_jed_org_programow
                    WHERE prg_kod = :b1 AND administracja = 'T');


SELECT DISTINCT NVL (DECODE (:b1,
                             'Angielska', description,
                             'Niemiecka', opis_nie,
                             'Rosyjska', opis_ros,
                             'Francuska', opis_fra,
                             'Hiszpanska', opis_his,
                             NULL
                            ),
                     opis
                    )
  FROM dz_kierunki_studiow
 WHERE nadrzedny_kod = :b2
   AND LOWER (typ_kierunku_kod) NOT LIKE '%zacja%'
CONNECT BY kod = PRIOR nadrzedny_kod
START WITH kod IN
          (SELECT krstd_kod
             FROM dz_kier_cert_os t_kco, dz_kierunki_studiow k
            WHERE cert_os_id = :b3 AND t_kco.krstd_kod = k.kod);


SELECT DECODE (LEVEL, 1, 1, -1), LEVEL, opis, '', ID
  FROM dz_log
CONNECT BY PRIOR ID = log_id
START WITH ID = :b1;


SELECT rozliczona_w_id, z_rozliczenia_id
  FROM (SELECT rozliczona_w_id, z_rozliczenia_id
          FROM dz_transakcje
         WHERE typ = 'ZETON'
```

```
            AND status <> 'X'
            AND os_id = :b2
            AND zaj_cyk_id = :b1)
CONNECT BY PRIOR rozliczona_w_id = z_rozliczenia_id
START WITH z_rozliczenia_id IS NULL
ORDER BY LEVEL DESC;


SELECT ROWID, gr_nr, nr, gr_zaj_cyk_id, zaj_cyk_id
  FROM dz_grupy
 WHERE (nr, zaj_cyk_id) IN (
          SELECT      nr, zaj_cyk_id
                FROM dz_grupy
          START WITH nr = 1 AND zaj_cyk_id = 150692
          CONNECT BY PRIOR nr = gr_nr
                 AND PRIOR zaj_cyk_id = gr_zaj_cyk_id
          UNION
          SELECT      nr, zaj_cyk_id
                FROM dz_grupy
          START WITH nr = 1 AND zaj_cyk_id = 150692
          CONNECT BY PRIOR gr_nr = nr
                 AND PRIOR gr_zaj_cyk_id = zaj_cyk_id)
   AND (nr != 1 OR zaj_cyk_id != 150692)


SELECT p_rola as nazwa_roli, 1 as poziom
  FROM dual
UNION
SELECT rola_pod as nazwa_roli, level+1 as poziom
  FROM dz_role_zaleznosci
CONNECT BY PRIOR rola_pod = rola_nad
START WITH rola_nad = :b1;


SELECT count(*) into v_tmp
  FROM DZ_SZKOLY
START WITH id = tab_zmian_szk(i)
CONNECT BY PRIOR id = jed_nadrz_id;


SELECT max(1)
```

```
  FROM dz_kolejnosc_etapow
START WITH prg_kod = v_prg_kod
       AND etp_kod = v_etp_kod
CONNECT BY PRIOR prg_kod = prg_kod
       AND PRIOR etp_kod_rel = etp_kod;


SELECT kod
  FROM dz_rankingi
 WHERE czy_zablokowany = 'T'
CONNECT BY PRIOR kod = rank_kod
START WITH kod = :b1;


SELECT count(*) into v_tmp
  FROM DZ_TERMINY_GRUP_SPTK
START WITH id = tab_zmian_tgsp(i).id
CONNECT BY PRIOR id = tgsp_id;
```

Dodatek C

Skrypt tworzący tabele benchmarku

Benchmark omówiony w rozdziale 3 korzysta z tabel zakładanych następującym skryptem:

```
CREATE TABLE EMP (
    empno INTEGER NOT NULL PRIMARY KEY,
    mgr INTEGER,
    ename VARCHAR(100),
    lname VARCHAR(100) DEFAULT 'Wolfeschlegelsteinhausenbergerdorff',
    job VARCHAR(9) DEFAULT 'ENGINEER',
    hiredate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    sal DECIMAL(7,2) DEFAULT 100.23,
    comm DECIMAL(7,2) DEFAULT 1400.00,
    deptno DECIMAL(2,0) DEFAULT 20,
    sex VARCHAR(1) DEFAULT 'F',
    INDEX (empno),
    INDEX (mgr)
);

CREATE TABLE CITIES (
    cid INTEGER NOT NULL PRIMARY KEY,
    city VARCHAR(100),
    INDEX (cid)
);

CREATE TABLE FLIGHTS (
```

```
    departure INTEGER,
    arrival  INTEGER,
    carrier VARCHAR(100),
    fid INTEGER NOT NULL PRIMARY KEY,
    price DECIMAL(7,2),
   INDEX (departure),
   INDEX (arrival),
   FOREIGN KEY (departure)
        REFERENCES CITIES(cid),
   FOREIGN KEY (arrival)
        REFERENCES CITIES(cid)
);

CREATE TABLE TRAINS (
    departure INTEGER,
    arrival  INTEGER,
    railline VARCHAR(100),
    tid INTEGER NOT NULL PRIMARY KEY,
    price DECIMAL(7,2),
   INDEX departure (departure),
   INDEX arrival (arrival),
   FOREIGN KEY (departure)
        REFERENCES CITIES(cid),
   FOREIGN KEY (arrival)
        REFERENCES CITIES(cid)
);
```

Dodatek D

Dodatkowe struktury danych

W niniejszym dodatku przedstawiono procedury wypełniania i pielęgnacji danych redundantnych niezbędnych do realizacji metod wykonywania zapytań rekurencyjnych opisanych w rozdziale 6. Przeznaczeniem tych procedur jest dbanie o spójność danych bazowych i danych redundantnych.

D.1 Metoda ścieżek pełnych

1. Procedura służąca do wypełnienia nowej dodatkowej tabeli przechowującej dane redundantne:

```
CREATE PROCEDURE FillAllPaths()
BEGIN
  DECLARE i INT;
  START TRANSACTION;
  SET i = 0;

  INSERT INTO emp_path
  SELECT empno, empno, 0
    FROM emp;

  MAIN: LOOP
    INSERT INTO emp_path
      SELECT e.empno, p.path_id, p.position + 1
        FROM emp e JOIN emp_path p ON (e.mgr = p.id)
       WHERE p.position = i;

    IF ROW_COUNT() = 0 THEN
```

```
        LEAVE MAIN;
      END IF;

      SET i = i + 1;

    END LOOP MAIN;
    COMMIT;
  END //
```

2. Procedura aktualizująca tabele przechowującą dane redundantne wykonywana po usunięciu z tabeli podstawowej dowolnego węzła:

```
CREATE PROCEDURE DeleteLeaf(IN id_del INT)
BEGIN
  DECLARE ile INT;
  DECLARE cID INT;

  SET cID = id_del;

  SELECT COUNT(*) INTO ile
    FROM emp_path
   WHERE path_id = id_del;

  IF ile = 0 THEN
    DELETE FROM emp_path
     WHERE id = cID;
  END IF;
END //
```

3. Procedura aktualizująca tabelę przechowującą dane redundantne wykonywana po wstawieniu nowego węzła do tabeli podstawowej:

```
CREATE PROCEDURE InsertNewNode(IN id_new INT,
                               IN chief_id_new INT)
BEGIN

  DECLARE no_more_rows BOOLEAN;
  DECLARE cID INT;
  DECLARE v_path_id INT;
  DECLARE v_position INT;
```

```
DECLARE findAncestors
  CURSOR FOR SELECT path_id, position
                FROM emp_path
               WHERE id = cID;
DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET no_more_rows = TRUE;

SET cID = chief_id_new;

INSERT INTO emp_path values (id_new, id_new, 0);

OPEN findAncestors;
main: LOOP
  FETCH findAncestors INTO v_path_id, v_position;
  IF no_more_rows THEN
    CLOSE findAncestors;
    LEAVE main;
  END IF;

  INSERT INTO emp_path VALUES (id_new, v_path_id, v_position + 1);

END LOOP main;

END //
```

4. Procedura aktualizująca tabelę przechowującą dane redundantne wykonywana po modyfikacji dowolnego węzła w tabeli podstawowej:

```
CREATE PROCEDURE UpdateNode(IN id_upd_new INT, IN chief_id_new INT)
BEGIN

  DECLARE no_more_rows BOOLEAN;
  DECLARE no_more_rows_sub BOOLEAN;
  DECLARE cID INT;
  DECLARE cCHFID INT;
  DECLARE v_id INT;
  DECLARE v_path_id INT;
  DECLARE v_position INT;
  DECLARE v_id_c INT;
  DECLARE v_path_id_c INT;
  DECLARE v_position_c INT;
```

```
DECLARE findAncestors
  CURSOR FOR SELECT id, path_id, position
                FROM emp_path
               WHERE id = cCHFID
                 AND position <> 0
              UNION SELECT 0, cCHFID, 0;
DECLARE findAncestorsDesc
  CURSOR FOR SELECT id, path_id
                FROM emp_path
               WHERE id IN (SELECT id
                              FROM emp_path
                             WHERE path_id = cID
                               and position <> 0)
                 AND path_id IN (SELECT path_id
                                   FROM emp_path
                                  WHERE id = cID
                                    and position <> 0)
           AND position <> 0;
DECLARE CONTINUE HANDLER FOR NOT FOUND
  SET no_more_rows = TRUE;

START TRANSACTION;

SET cID = id_upd_new;
SET cCHFID = chief_id_new;

SET no_more_rows = FALSE;

open findAncestorsDesc;
main: LOOP
  FETCH findAncestorsDesc into v_id, v_path_id;
  IF no_more_rows THEN
    CLOSE findAncestorsDesc;
    LEAVE main;
  END IF;

  DELETE FROM emp_path
   WHERE id = v_id
     AND path_id = v_path_id
         AND position <> 0;
```

```
    END LOOP main;

    DELETE FROM emp_path
     WHERE id = cID
       AND position <> 0;

    SET v_id = NULL;
    SET v_path_id = NULL;

    SET no_more_rows = FALSE;

    OPEN findAncestors;
    main: LOOP
      FETCH findAncestors INTO v_id_c, v_path_id_c, v_position_c;
      IF no_more_rows THEN
        CLOSE findAncestors;
        LEAVE main;
      END IF;

      INSERT INTO emp_path
        VALUES (cID, v_path_id_c, v_position_c + 1);

      INSERT INTO emp_path
              SELECT id, v_path_id_c,
                      v_position_c + position + 1
                FROM emp_path
               WHERE path_id = cID and position <> 0;
    END LOOP main;

    COMMIT;

  END //
```

D.2 Metoda ścieżek logarytmicznych

1. Procedura służąca do wypełnienia nowej dodatkowej tabeli przechowującej dane redundantne:

```
CREATE PROCEDURE FillAllPathsPow()
BEGIN
  DECLARE j INT;
  START TRANSACTION;
  SET j = 1;

  INSERT INTO powpath
  SELECT empno, mgr, 1 AS pl
    FROM emp
   WHERE mgr is not null;

  MAIN: LOOP

    INSERT INTO powpath
    SELECT b.e_id, e.b_id, pow(2, j) as pl
      FROM powpath e, powpath b
     WHERE e.e_id = b.b_id
       AND e.pl = pow(2, j-1) and b.pl=pow(2, j-1);

  IF ROW_COUNT() = 0 THEN
      LEAVE MAIN;
  END IF;
  SET j = j + 1;

  END LOOP MAIN;

  COMMIT;

END //
```

2. Procedura aktualizująca tabelę przechowującą dane redundantne wykonywana po usunięciu z tabeli podstawowej dowolnego węzła:

```
CREATE PROCEDURE DeleteLeaf(IN id_del INT)
BEGIN
  DECLARE ile INT;
  DECLARE cID INT;

  SET cID = id_del;

  SELECT COUNT(*) INTO ile
```

```
    FROM powpath
   WHERE path_id = id_del;

  IF ile = 0 THEN
    DELETE FROM powpath
     WHERE id = cID;
  END IF;
END //
```

3. Procedura aktualizująca tabelę przechowującą dane redundantne wykonywana po wstawieniu nowego węzła do tabeli podstawowej:

```
CREATE PROCEDURE InsertPowpath(IN new_id INT, IN new_chef_id INT)
BEGIN
  DECLARE j INT;
  START TRANSACTION;
  SET j = 1;
  INSERT INTO powpath VALUES (new_id, new_chef_id, 1);
  MAIN: LOOP
    INSERT INTO powpath
    SELECT b.e_id, e.b_id, pow(2, j) as pl
      FROM powpath e, powpath b
     WHERE e.e_id = b.b_id
       AND e.pl = pow(2, j-1) and b.pl=pow(2, j-1)
           AND b.e_id = new_id;

    IF ROW_COUNT() = 0 THEN
      LEAVE MAIN;
    END IF;
    SET j = j + 1;
  END LOOP MAIN;
  COMMIT;

END //
```

4. Procedura aktualizująca tabelę przechowującą dane redundantne wykonywana po modyfikacji dowolnego węzła w tabeli podstawowej (w zależności od poziomu zagłębienia):

```
CREATE PROCEDURE UpdatePowpath1(IN upd_id INT, IN new_chief_id INT)
BEGIN
```

```
DECLARE no_more_rows BOOLEAN;
DECLARE no_more_rows_sub BOOLEAN;
DECLARE cID INT;
DECLARE cCHFID INT;
DECLARE v_id INT;
DECLARE v_path_id INT;
DECLARE j INT;

DECLARE findAncestorsDesc
  CURSOR FOR SELECT e_id, b_id
               FROM powpath
              WHERE e_id IN
                (SELECT p.e_id
                   FROM powpath p
                  WHERE p.b_id = cID
                 UNION
                 SELECT p1.e_id
                   FROM powpath p1 JOIN powpath p
                        ON (p1.b_id = p.e_id)
                  WHERE p.b_id = cID
                    AND p1.pl>p.pl)
                AND b_id IN
                  (SELECT p.b_id
                     FROM powpath p
                    WHERE p.e_id = cID
                   UNION
                   SELECT p1.b_id
                     FROM powpath p1 JOIN powpath p
                          ON (p1.e_id = p.b_id)
                    WHERE p.e_id = cID
                      AND p1.pl<p.pl);

DECLARE findDescENDent
  CURSOR FOR SELECT cID
               UNION
               SELECT p.e_id
                 FROM powpath p
                WHERE p.b_id = cID
               UNION
               SELECT p1.e_id
                 FROM powpath p1 JOIN powpath p
```

```
                    ON (p1.b_id = p.e_id)
                WHERE p.b_id = cID
                  AND p1.pl>p.pl;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_rows = TRUE;

START TRANSACTION;

SET cID = upd_id;
SET cCHFID = new_chief_id;

SET no_more_rows = FALSE;

open findAncestorsDesc;
main: LOOP
  FETCH findAncestorsDesc into v_id, v_path_id;
  IF no_more_rows THEN
    CLOSE findAncestorsDesc;
    LEAVE main;
  END IF;

  DELETE FROM powpath
   WHERE e_id = v_id
     AND b_id = v_path_id;

END LOOP main;

DELETE FROM powpath
 WHERE e_id = cID;

SET v_id = NULL;
SET v_path_id = NULL;

INSERT INTO powpath VALUES(cID, cCHFID, 1);
SET no_more_rows = FALSE;
open findDescENDent;
main: LOOP
  FETCH findDescENDent into v_id;
  IF no_more_rows THEN
    CLOSE findDescENDent;
    LEAVE main;
  END IF;
```

```
    IF v_id <> cID THEN
      DELETE FROM powpath
      WHERE e_id = v_id
        AND pl <> 1;
    END IF;

    SET j = 1;

    MAINP: LOOP
      INSERT INTO powpath
      SELECT b.e_id, e.b_id, pow(2, j) as pl
        FROM powpath e, powpath b
       WHERE e.e_id = b.b_id
         AND e.pl = pow(2, j-1) and b.pl=pow(2, j-1)
             AND b.e_id = v_id;

      IF ROW_COUNT() = 0 THEN
        LEAVE MAINP;
      END IF;
      SET j = j + 1;
    END LOOP MAINP;

  END LOOP main;

END //
```

D.3 Metoda zbiorów zagnieżdżonych

1. Procedura służąca do wypełnienia dodatkowych dwóch kolumn w tabeli bazowej przechowującej dane redundantne:

```
CREATE PROCEDURE fillData ()
BEGIN

DECLARE v_empno int;
DECLARE tmp int;
SET tmp = 1;
```

```
    START TRANSACTION;
    SELECT empno into v_empno
      FROM emp
     WHERE mgr is null;

    CALL fillDesc(v_empno, tmp, 1);
    commit;
  END //
  DELIMITER ;

  CREATE PROCEDURE fillDesc(IN p_empno int,
                            INOUT v_tmp int,
                            IN p_commit int)
  BEGIN
    DECLARE no_more_rows BOOLEAN;
    DECLARE v_empno INT;
    DECLARE lft_pom INTEGER;
    DECLARE findAncestors
      CURSOR FOR SELECT empno
                   FROM emp
                  WHERE mgr = p_empno;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET no_more_rows = TRUE;


    SET lft_pom = v_tmp;

    OPEN findAncestors;
    main: LOOP
      FETCH findAncestors into v_empno;
      IF no_more_rows THEN
        CLOSE findAncestors;
        LEAVE main;
      END IF;

      SET v_tmp = v_tmp + 1;
      SET p_commit = p_commit + 1;

      IF p_commit%1000 = 0 THEN
       commit;
       START TRANSACTION;
      END IF;
```

```
      CALL fillDesc(v_empno, v_tmp, p_commit);

  END LOOP main;
  SET v_tmp = v_tmp + 1;
  UPDATE emp
     SET lft = lft_pom, rgh = v_tmp
   WHERE empno = p_empno;

END //
```

2. Procedura wykonywana zamiast usuwania węzła z tabeli podstawowej:

```
create procedure DeleteNodeNS(IN p_empno INT)
BEGIN
  DECLARE v_rgh INT;
  DECLARE v_lft INT;

  SELECT lft, rgh into v_lft, v_rgh
    FROM emp
   WHERE empno = p_empno;

  UPDATE emp
     SET rgh = rgh - 2
   WHERE rgh > v_rgh;

 UPDATE emp
     SET lft = lft - 2
  WHERE lft > v_lft;

  DELETE FROM emp
   WHERE empno = p_empno;

END //
DELIMITER ;
```

3. Procedura aktualizująca dodatkowe dwie kolumny przechowujące dane redundantne wykonywana po wstawieniu nowego węzła do tej tabeli:

```
CREATE PROCEDURE InsertNodeNS(IN p_empno INT, IN p_mgr INT)
```

```
BEGIN
  DECLARE v_lft INT;
  DECLARE v_rgh INT;

  SELECT lft, rgh into v_lft, v_rgh
    FROM emp
   WHERE empno = p_mgr;

  UPDATE emp
     SET rgh = rgh + 2
   WHERE rgh >= v_rgh;

  UPDATE emp
     SET lft = lft + 2
   WHERE lft > v_lft;

  UPDATE emp
     SET lft = v_lft + 1,
         rgh = v_lft + 2
   WHERE empno = p_empno;

END //
```

4. Operacja modyfikacji dowolnego węzła w tabeli podstawowej jest wykonywana przez poniższą procedurę:

```
CREATE PROCEDURE UpdateNodeNS(IN p_empno INT, IN p_mgr INT)
BEGIN

  DECLARE v_empnom, v_empnoe, v_empm, v_empe, v_empno INT;
  DECLARE v_lfte, v_lftm, v_rghe, v_rghm, v_lft, v_rgh INT;
  DECLARE pom INT;
  DECLARE no_more_rows BOOLEAN;
  DECLARE findUpdDesc
    CURSOR FOR SELECT empno, lft, rgh
                 FROM emp
                WHERE mgr = v_empm;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET no_more_rows = TRUE;
```

```
SET v_empnom = p_mgr;
SET v_empnoe = p_empno;

l_mgr: LOOP
    SELECT mgr into v_empm
      FROM emp
     WHERE empno = v_empnom;
    IF v_empm IS NULL THEN
          leave l_mgr;
    END if;
    IF (no_more_rows) THEN
        leave l_mgr;
    END IF;

    l_empno: LOOP
      SELECT mgr into v_empe
        FROM emp
       WHERE empno = v_empnoe;

      IF v_empe IS NULL THEN
        leave l_empno;
      END IF;
      IF (no_more_rows) THEN
        LEAVE l_empno;
      ELSE
        IF v_empm = v_empe THEN
            LEAVE l_mgr;
          ELSE
            SET v_empnoe = v_empe;
        END IF;
      END IF;
    END LOOP;
    SET v_empnom = v_empm;
    SET v_empnoe = p_empno;
END LOOP;

SELECT concat(v_empnoe, ' ', v_empnom);

SELECT lft, rgh into v_lfte, v_rghe
  FROM emp
 WHERE empno = v_empnoe;
```

```
SELECT lft, rgh into v_lftm, v_rghm
  FROM emp
 WHERE empno = v_empnom;

SELECT lft, rgh into v_lft, v_rgh
  FROM emp
 WHERE empno = p_empno;
SET pom = (ceiling((v_rgh - v_lft)/2))*2;

SET no_more_rows = FALSE;

UPDATE emp
    SET mgr = p_mgr
 WHERE empno = p_empno;

OPEN findUpdDesc;

main: LOOP
  fetch findUpdDesc into v_empno, v_lft, v_rgh;
  IF no_more_rows THEN
    CLOSE findUpdDesc;
    LEAVE main;
  END IF;

  IF (v_lfte > v_lftm) THEN
    IF (v_lft >= v_lftm) and (v_lft <= v_lfe) THEN
      IF (v_lft - pom > 0) THEN
        SET pom = v_lft-pom;
        CALL fillDesc(v_empno, pom);
      ELSE
        CALL fillDesc(v_empno, v_lft);
      END IF;
    END IF;
  ELSE
    IF (v_lft >= v_lfte) and (v_lft <= v_lftm) THEN
      IF (v_lft - pom > 0) THEN
        SET pom = v_lft-pom;
        UPDATE emp
           SET lft = pom
         WHERE empno = v_empno;
        CALL fillDesc(v_empno, pom);
```

```
        ELSE
          CALL fillDesc(v_empno, v_lft);
        END IF;
      END IF;
    END IF;

  END LOOP;

END //
DELIMITER ;
```

D.4 Metoda ścieżek zmaterializowanych

1. Procedura służąca do wypełnienia dodatkowej kolumny w tabeli bazowej przechowującej dane redundantne:

```
CREATE PROCEDURE UpdatePathTableRec(IN w INT,
                                    IN p_path VARCHAR(1000))
BEGIN

  DECLARE v_empno INT;
  DECLARE no_more_rows BOOLEAN;
  DECLARE child
    CURSOR FOR SELECT empno
                 FROM emp
                WHERE mgr = w;
  DECLARE CONTINUE HANDLER FOR NOT FOUND
    SET no_more_rows = TRUE;
  DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;
  DECLARE EXIT HANDLER FOR SQLWARNING ROLLBACK;

  OPEN child;

  main: LOOP
   FETCH child INTO v_empno;
   IF no_more_rows THEN
     CLOSE child;
     LEAVE main;
   END IF;
```

```
      UPDATE emp
         SET path_string = CONCAT(p_path, ',',v_empno)
       WHERE empno = v_empno;

      CALL UpdatePathTableRec(v_empno, concat(p_path, ',',v_empno));

    END LOOP main;

END //
DELIMITER ;

DELIMITER //
CREATE PROCEDURE UpdatePathTable()
BEGIN
  UPDATE emp SET path_string = empno WHERE mgr IS NULL;

  MAIN: LOOP
    UPDATE emp current INNER JOIN emp parent
       ON (parent.empno = current.mgr)
       SET current.path_string =
           CONCAT(parent.path_string, ',', current.empno)
     WHERE parent.path_string IS NOT NULL
       AND current.path_string IS NULL;

    IF ROW_COUNT() = 0 THEN
      LEAVE MAIN;
    END IF;
  END LOOP MAIN;

END //
DELIMITER ;
```

2. Operacja usuwania węzła z tabeli podstawowej nie wymaga modyfikacji danych redundantnych. Nie zdefiniowano więc odpowiedniej procedury składowanej.

3. Wstawianie nowego węzła wygląda następująco:

```
INSERT INTO emp(empno, mgr, ename, string_path)
  SELECT :p1, :p2, concat(string_path, ',', :p1)
    FROM emp
```

```
WHERE empno = :p2;
```

4. Modyfikacja dowolnego węzła w tabeli podstawowej jest wykonywana poprzez poniższą procedurę:

```
CREATE PROCEDURE UpdateNode(IN komu INT, IN kogo INT)
BEGIN
  DECLARE currentPath VARCHAR(1000);

  SELECT path_string
    FROM emp
   WHERE empno = komu INTO currentPath;

  UPDATE emp
     SET path_string = NULL
   WHERE path_string LIKE CONCAT(currentPath, ',', '%');

   UPDATE emp
      SET mgr = kogo, path_string = null
    WHERE empno = komu;

   MAIN: LOOP
    UPDATE emp current INNER
      JOIN emp parent ON (parent.empno = current.mgr)
       SET current.path_string =
         CONCAT(parent.path_string, ',', current.empno)
     WHERE parent.path_string IS NOT NULL AND current.path_stri

    IF ROW_COUNT() = 0 THEN
      LEAVE MAIN;
    END IF;
  END LOOP MAIN;

END //
DELIMITER ;
```

Dodatek E

Programy źródłowe

Na załączonej płycie znajduje się dysk maszyny wirtualnej wraz z następującymi programami:

`prium-rcte`

: Są to źródła programu do generowania zapytań rekurencyjnych.

`tester`

: Jest to program do przeprowadzania testów opisanych w niniejszej rozprawie. Plik `README` zawiera instrukcję, w jaki sposób te testy wykonać.

Maszyna wirtualna wymaga programu Virtualbox w systemie 64 bitowym. Maszyna zawiera gotowe środowisko uruchomieniowe wraz z opisem, jak korzystać z wyżej wymienionych programów.

Bibliografia

[1] C. Bauer, G. King. *Hibernate w akcji*. Wydawnictwo Helion, 2007.

[2] H. Böck. Java Persistence API. *The Definitive Guide to NetBeans^{TM} Platform 7*, strony 315–320. Apress, 2011.

[3] A. Boniewicz, M. Gawarkiewicz, P. Wiśniewski. Automatic selection of functional indexes for object relational mapping system. *International Journal of Software Engineering and Its Applications*, 7(4), 2013.

[4] A. Boniewicz, K. Stencel, P. Wisniewski. Unrolling SQL: 1999 recursive queries. T. Kim, J. Ma, W. Fang, Y. Zhang, A. Cuzzocrea, redaktorzy, *Computer Applications for Database, Education, and Ubiquitous Computing - International Conferences, EL, DTA and UNESST 2012, Held as Part of the Future Generation Information Technology Conference, FGIT 2012, Gangneug, Korea, December 16-19, 2012. Proceedings*, wolumen 352 serii *Communications in Computer and Information Science*, strony 345–354. Springer, 2012.

[5] A. Boniewicz, P. Wisniewski, K. Stencel. On materializing paths for faster recursive querying. B. Catania, T. Cerquitelli, S. Chiusano, G. Guerrini, M. Kämpf, A. Kemper, B. Novikov, T. Palpanas, J. Pokorný, A. Vakali, redaktorzy, *New Trends in Databases and Information Systems, 17th East European Conference on Advances in Databases and Information Systems, ADBIS 2013, Genoa, Italy, September 1-4, 2013. Proceedings II*, wolumen 241 serii *Advances in Intelligent Systems and Computing*, strony 105–112. Springer, 2013.

[6] A. Boniewicz, P. Wisniewski, K. Stencel. On redundant data for faster recursive querying via ORM systems. M. Ganzha, L. A. Maciaszek, M. Paprzycki, redaktorzy, *Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, Kraków, Poland, September 8-11, 2013.*, strony 1439–1446, 2013.

[7] M. Burzanska, K. Stencel, P. Suchomska, A. Szumowska, P. Wisniewski. Recursive queries using object relational mapping. T. Kim, Y. Lee, B. H. Kang, D. Slezak, redaktorzy, *Future Generation Information Technology - Second International Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings*, wolumen 6485 serii *Lecture Notes in Computer Science*, strony 42–50. Springer, 2010.

[8] M. Burzanska, K. Stencel, P. Wisniewski. Pushing predicates into recursive SQL common table expressions. J. Grundspenkis, T. Morzy, G. Vossen, redaktorzy, *Advances in Databases and Information Systems, 13th East European Conference, ADBIS 2009, Riga, Latvia, September 7-10, 2009. Proceedings*, wolumen 5739 serii *Lecture Notes in Computer Science*, strony 194–205. Springer, 2009.

[9] M. Burzańska. *New Query Rewriting Methods for Structured and Semi-Structured Databases*. Praca doktorska, Warsaw University, Department of Mathematics, Informatics and Mechanics, Warsaw, Poland, 10 2011.

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

[11] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[12] T. P. P. Council. TPC Benchmarks. Online, 2009.

[13] M. L. Fussell. Foundations of Object-Relational Mapping, 1997.

[14] M. Gawarkiewicz, P. Wisniewski. Partial aggregation using hibernate. T. Kim, H. Adeli, D. Slezak, F. E. Sandnes, X. Song, K. Chung, K. P. Arnett, redaktorzy, *Future Generation Information Technology - Third International Conference, FGIT 2011 in Conjunction with GDC 2011, Jeju Island, Korea, December 8-10, 2011. Proceedings*, wolumen 7105 serii *Lecture Notes in Computer Science*, strony 90–99. Springer, 2011.

[15] A. Ghazal, A. Crolotte, D. Y. Seid. Recursive SQL query optimization with k-iteration lookahead. S. Bressan, J. Küng, R. Wagner, redaktorzy, *Database and Expert Systems Applications, 17th International Conference, DEXA 2006, Kraków, Poland, September 4-8, 2006, Proceedings*, wolumen 4080 serii *Lecture Notes in Computer Science*, strony 348–357. Springer, 2006.

[16] A. Ghazal, D. Y. Seid, A. Crolotte, M. Al-Kateb. Adaptive optimizations of recursive queries in teradata. K. S. Candan, Y. Chen, R. T.

Snodgrass, L. Gravano, A. Fuxman, redaktorzy, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, strony 851–860. ACM, 2012.

[17] J. Gray, redaktor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[18] Y. E. Ioannidis. On the computation of the transitive closure of relational operators. W. W. Chu, G. Gardarin, S. Ohsuga, Y. Kambayashi, redaktorzy, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, strony 403–411. Morgan Kaufmann, 1986.

[19] Y. E. Ioannidis, R. Ramakrishnan. Efficient transitive closure algorithms. F. Bancilhon, D. J. DeWitt, redaktorzy, *Fourteenth International Conference on Very Large Data Bases, August 29 - September 1, 1988, Los Angeles, California, USA, Proceedings.*, strony 382–394. Morgan Kaufmann, 1988.

[20] H. V. Jagadish, R. Agrawal, L. Ness. A study of transitive closure as a recursion mechanism. U. Dayal, I. L. Traiger, redaktorzy, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, strony 331–344. ACM Press, 1987.

[21] M. C. Janina Mincer-Daszkiewicz. USOS - wstęp do dokumentacji wdrożeniowej. Online, 2009.

[22] M. C. Janina Mincer-Daszkiewicz. Ankieta na 10-lecie: Stan wdrożenia USOS na uczelniach. Online, 2010.

[23] M. Keith, M. Schincariol. *Pro EJB 3: Java Persistence API (Pro)*. Apress, Berkely, CA, USA, 2006.

[24] W. Keller. Mapping objects to tables - a pattern language. *Proc. Of European Conference on Pattern Languages of Programming Conference (EuroPLOP)"97*, 1997.

[25] S. Melnik, A. Adya, P. A. Bernstein. Compiling mappings to bridge applications and databases. *ACM Trans. Database Syst.*, 33(4), 2008.

[26] S. Microsystems. Java Specification Requests 317. Online, 2009.

[27] W. Oles, B. Malysiak-Mrozek, D. Mrozek. Porównanie wydajności wybranych narzędzi odwzorowania obiektowo-relacyjnego. *Studia Informatica*, 34(2A):123–144, 2013.

[28] C. Ordonez. Optimizing recursive queries in SQL. F. Özcan, redaktor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, strony 834–839. ACM, 2005.

[29] P. Przymus, A. Boniewicz, M. Burzanska, K. Stencel. Recursive query facilities in relational databases: A survey. Y. Zhang, A. Cuzzocrea, J. Ma, K. Chung, T. Arslan, X. Song, redaktorzy, *Database Theory and Application, Bio-Science and Bio-Technology - International Conferences, DTA and BSBT 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings*, wolumen 118 serii *Communications in Computer and Information Science*, strony 89–99. Springer, 2010.

[30] Solid IT. DB-Engines ranking. Online, 2013.

[31] J. Szedel, M. Kolano, J. Chojnacki. Integracja metodyki obiektowej opartej na zastosowaniu języka uml ze środowiskiem programistycznym borland c++ builder. *Studia Informatica*, 27(4):17–35, 2006.

[32] A. Szumowska. Wsparcie dla zapytań rekurencyjnych w odwzorowaniach obiektowo-relacyjnych. Praca magisterska, Wydział Matematyki i Informatyki, Uniwersytet Mikołaja Kopernika w Toruniu, 2011.

[33] A. Szumowska, A. Boniewicz, M. Burzanska, P. Wisniewski. Hibernate the recursive queries - defining the recursive queries using hibernate ORM. J. Eder, M. Bieliková, A. M. Tjoa, redaktorzy, *ADBIS 2011, Research Communications, Proceedings II of the 15th East-European Conference on Advances in Databases and Information Systems, September 20-23, 2011, Vienna, Austria*, wolumen 789 serii *CEUR Workshop Proceedings*, strony 190–199. CEUR-WS.org, 2011.

[34] A. Szumowska, M. Burzanska, P. Wisniewski, K. Stencel. Efficient implementation of recursive queries in major object relational mapping systems. T. Kim, H. Adeli, D. Slezak, F. E. Sandnes, X. Song, K. Chung, K. P. Arnett, redaktorzy, *Future Generation Information Technology - Third International Conference, FGIT 2011 in Conjunction with GDC 2011, Jeju Island, Korea, December 8-10, 2011. Proceedings*, wolumen

7105 serii *Lecture Notes in Computer Science*, strony 78–89. Springer, 2011.

[35] A. Szumowska, M. Burzanska, P. Wisniewski, K. Stencel. Extending HQL with plain recursive facilities. T. Morzy, T. Härder, R. Wrembel, redaktorzy, *Advances in Databases and Information Systems - 16th East European Conference, ADBIS 2012, Poznań, Poland, September 18-21, 2012. Proceedings II*, wolumen 186 serii *Advances in Intelligent Systems and Computing*, strony 265–272. Springer, 2012.

[36] Tiobe Software. Programming community index. Online, 2013.

[37] P. Valduriez, H. Boral. Evaluation of recursive queries using join indices. *Expert Database Conf.*, strony 271–293, 1986.

[38] T. van den Broek. Object relational mappings. First step to a formal approach. Praca magisterska, Radboud University Nijmegen, Computer Science Department, Holandia, 2007.

[39] P. Wisniewski, M. Burzanska, K. Stencel. The impedance mismatch in light of the unified state model. *Fundam. Inform.*, 120(3-4):359–374, 2012.

[40] M. Wojciechowski, M. Zakrzewicz. TPC Benchmarking. *Materiały VIII konf. PLOUG*, Systemy informatyczne: projektowanie, implementowanie, eksploatowanie, strony 244–259. Stowarzyszenie Polskiej Grupy Użytkowników systemu Oracle, oct 2002.