STUDIA INFORMATICA 2011
Volume 32 Number 2B (97)

Łukasz PIETRZAK, Paweł KASPROWSKI
Politechnika Śląska, Instytut Informatyki

# AUTOMATYZACJA TESTOWANIA WYDAJNOŚCI INTERFEJSÓW BAZODANOWYCH

**Streszczenie**. Testowanie wydajności rozwiązań dostępu do danych w aplikacjach jest poważnym problemem wielu zastosowań. Odpowiedzenie na pytanie „jak będzie się zachowywała konkretna aplikacja w przypadku obciążenia jej większą liczbą jednoczesnych użytkowników?", wymaga przygotowania i przeprowadzenia odpowiednich testów, co często jest czasochłonne. Proponowane w artykule rozwiązanie pozwala zautomatyzować testowanie aplikacji pod obciążeniem. Wykorzystując środowisko rozproszone, umożliwia ono podłączenie do dowolnego interfejsu programowego i przetestowanie jego kluczowych elementów. Umożliwia także porównanie różnych rozwiązań w celu wyboru najlepszego do konkretnego zastosowania.

**Słowa kluczowe**: testowanie, wydajność, interfejsy bazodanowe, ORM, JDBC, Hibernate

# AUTOMATIZATION OF DATABASE INTERFACES PERFORMANCE TESTING

**Summary**. Testing of database applications' performance is a serious problem in many circumstances. The answer to a question how will application behave under normal usage demands preparing specialized tests what is often a time consuming task. The solution presented in the article tries to automate this process. It gives also opportunity to compare different interfaces.

**Keywords**: testing, performance, database interfaces, ORM, JDBC, Hibernate

## 1. Wprowadzenie

Testowanie wydajności rozwiązań dostępu do danych w aplikacjach bazodanowych jest poważnym problemem. Odpowiedzenie na pytanie „jak będzie się zachowywała konkretna

aplikacja w przypadku obciążenia jej większą liczbą jednoczesnych użytkowników?" wymaga przygotowania i przeprowadzenia odpowiednich testów, co często jest pracochłonne. Proponowane w artykule rozwiązanie pozwala zautomatyzować testowanie aplikacji pod obciążeniem.

Celem projektu prezentowanego w artykule jest stworzenie narzędzia pozwalającego na pomiar wydajności dostępu aplikacji do bazy danych. Stworzony został system, który pozwala w stosunkowo krótkim czasie odpowiedzieć na pytanie, czy dane rozwiązanie programowe jest w stanie spełnić stawiane przed nim wymagania wydajnościowe? System działa w środowisku rozproszonym i jego zadaniem jest symulacja grupy użytkowników, którzy będą wysyłać żądania do bazy danych przez warstwę programową, zwaną dalej interfejsem bazodanowym.

W celu sprawdzenia działania systemu użyto go do porównania dwóch popularnych mechanizmów mapujących dane: standardu JDBC [2] i rozwiązania używającego biblioteki Hibernate [1].

### 1.1. Biblioteki typu ORM

Model relacyjny, zaproponowany przez E.F. Codda w latach siedemdziesiątych, od lat cieszy się opinią modelu uniwersalnego, sprawdzającego się w prawie każdym środowisku. Jednak jednym z problemów tego modelu jest jego niedopasowanie do dominującego w dzisiejszych czasach paradygmatu obiektowego, stosowanego w znakomitej większości środowisk programistycznych. Obie idee – algebra relacyjna oraz programowanie obiektowe – wykorzystywane razem posiadają jedną zasadniczą wadę – brak kompatybilności w opisie zależności pomiędzy obiektami. Model relacyjny przechowuje informacje o zależnościach pomiędzy obiektami w postaci pary: klucz główny-klucz obcy. Opis ten nie pozwala na realizację niektórych powiązań występujących w programowaniu obiektowym. Dotyczy to głównie dziedziczenia oraz asocjacji polimorficznych. Sprawa komplikuje się jeszcze bardziej, gdy rozpatrzymy kwestię równości obiektów, a także przechodzenia po grafie obiektów. Powyższe problemy zostały szeroko opisane w literaturze i noszą nazwę „niedopasowania paradygmatów obiektowo-relacyjnych" [1]. Zagadnienie to stało się na tyle istotne, iż zaczęto poszukiwać uniwersalnego rozwiązania problemu niedopasowania. W przeciągu kilku lat badań pojawiło się wiele koncepcji rozwiązania, ale najbardziej obiecujące wydają się dwa pomysły: obiektowe bazy danych oraz mechanizmy ORM (ang. *Object-relational mapping*). Pierwsze z rozwiązań przeżywało największy okres rozwoju w latach dziewięćdziesiątych. Jednak brak elastyczności rynku oraz stabilna pozycja systemów relacyjnych wyparły obiektowe bazy danych na drugi plan. Drugie rozwiązanie, czyli ORM, jest aktualnie najbardziej popularnym rozwiązaniem służącym do automatycznej transformacji danych pomiędzy technologią relacyjną a technologią obiektową.

Podstawowym problemem rozwiązań typu ORM jest wprowadzenie dodatkowej warstwy w komunikacji ze źródłem danych. Można przypuszczać, że warstwa taka spowalnia działanie całości aplikacji. Chcąc to sprawdzić, przeprowadzono testy opisane w artykule.

## 2. Architektura systemu

Celem systemu jest przeprowadzenie badań wydajności interfejsu bazodanowego pewnej aplikacji. Aplikacja ta modeluje pewien problem dziedzinowy – konkretny przypadek biznesowy. Jako przykład do testowania zaproponowano tu model sklepu muzycznego. Dla tego modelu zaprojektowano i porównano warstwy dostępu do danych w dwóch technologiach JDBC i ORM/Hibernate.

Do zadań systemu testującego należą: symulowanie ruchu sieciowego, wykonywanie pomiarów o zróżnicowanym charakterze, gromadzenie wyników uzyskanych podczas testów, monitorowanie stanu pracy aplikacji.

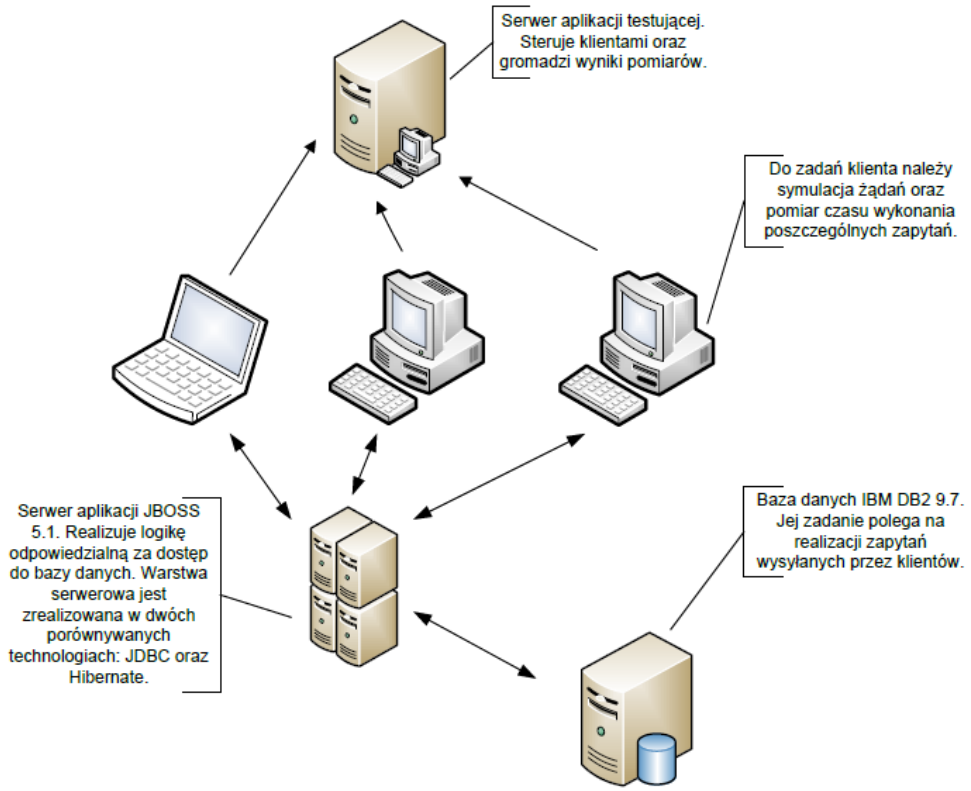
Wyniki symulacji dostarczyły informacji na temat:

- średniej szybkości odpowiedzi zrealizowanego rozwiązania,
- wydajności warstwy dostępu w zależności od konfiguracji systemu oraz liczby klientów,
- występowania błędów podczas interakcji programu oraz bazy danych.

„Sercem" systemu jest moduł gromadzący wyniki pomiarów, zwany dalej serwerem pomiarowym. Rzeczywiste badania wykonują tak zwani klienci. Podstawowym zadaniem klienta jest generowanie żądań do bazy danych oraz pomiar czasu odpowiedzi bazy danych. Uzyskane wyniki są przesyłane przez sieć do serwera pomiarowego, którego zadaniem jest gromadzenie uzyskanych wyników w jednym pliku.

Chcąc umożliwić analizę wydajności w sposób zbliżony do warunków rzeczywistych, całość systemu działa w rzeczywistym środowisku rozproszonym. Dzięki temu pomiar uwzględnia przepustowość sieci i pozwala bez ryzyka spadku wydajności wykonać test nawet dla kilku tysięcy klientów. Gromadzenie danych w jednym miejscu pozwala na późniejszą szybką ich analizę.

Poważnym problemem, z którym należało się zmierzyć, było generowanie symulowanego ruchu w sieci Internet. Problem wynika z heterogeniczności środowiska rozproszonego. Najprostsze rozwiązanie – lawinowe odpytywanie bazy danych z kilku komputerów – jest błędne z merytorycznego punktu widzenia. Użytkownicy nie wykonują operacji w stałych odstępach czasu. Najczęściej użytkownik wykonuje sekwencję operacji, po czym następuje chwilowy brak akcji. Rozwiązanie problemu polegało na zastosowaniu rozkładu Pareto, który dobrze charakteryzuje ruch w sieci internetowej [6].



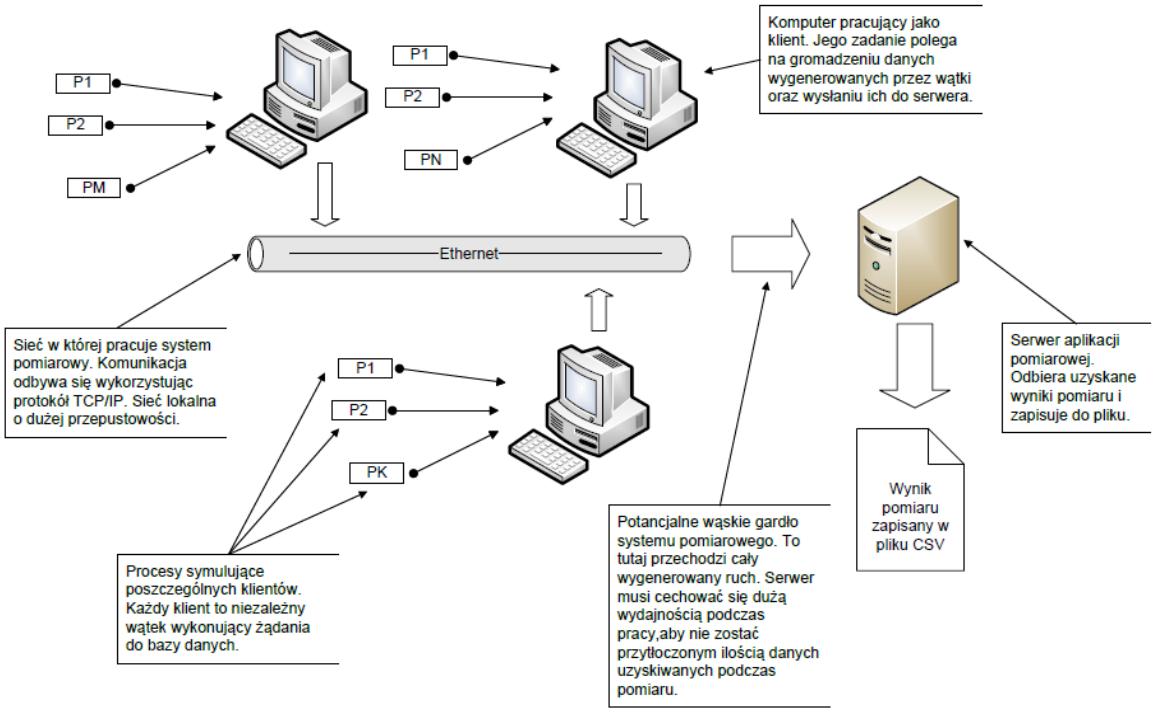

Rys. 1. Schemat przedstawiający ideę realizacji pomiarów. Rysunek przedstawia najważniejsze elementy systemu

Fig. 1. Schema showing the idea of measurements. Figure shows the most important elements of the system

Symulacja dużej liczby klientów wymaga od aplikacji wydajnego mechanizmu przesyłania danych pomiędzy serwerem pomiarowym a klientami. Początkowo mechanizm komunikacji zaimplementowano, wykorzystując bibliotekę wejścia/wyjścia języka Java Java IO (ang. *Java Input/Output*) [3]. Technologia ta nie przyniosła zadowalającego rezultatu, ponieważ wydajność aplikacji znacząco spadała przy podłączeniu większej liczby klientów. Wynika to przede wszystkim z faktu, że biblioteka zmusza programistę do obsługi każdego połączenia z poziomu nowego wątku. Zbyt duża liczba wątków powoduje spadek wydajności aplikacji. Duża liczba równoległych procesów jest kłopotliwa w zarządzaniu oraz zmusza programistę do stosowania różnych struktur kolejkujących podczas przekazywania danych pomiędzy procesami.

Z tego powodu od wersji JDK 1.4 pojawiła się w środowisku Java nowa biblioteka NIO (ang. *New Input/Output*), będąca uzupełnieniem Java IO [4, 5]. Java NIO rozwiązuje problem przez mocniejsze wykorzystanie funkcjonalności dostarczonych przez system operacyjny. Java NIO jest rozwinięciem Java IO, uzupełniającym interfejs wejścia/wyjścia Javy o nową funkcjonalność. Zasadniczą różnicą jest praca w trybie nieblokującym. Pozwala to zrealizować całą komunikację przy użyciu jednego wątku. Multipleksowana, nieblokująca komunika-

cja, oparta na gniazdach, została wykorzystana do utworzenia modułu odpowiedzialnego za komunikację pomiędzy poszczególnymi elementami systemu pomiarowego.




Rys. 2. Model przedstawiający przepływ danych w systemie pomiaro-
Fig. 2. Model presenting data flow in measurement system

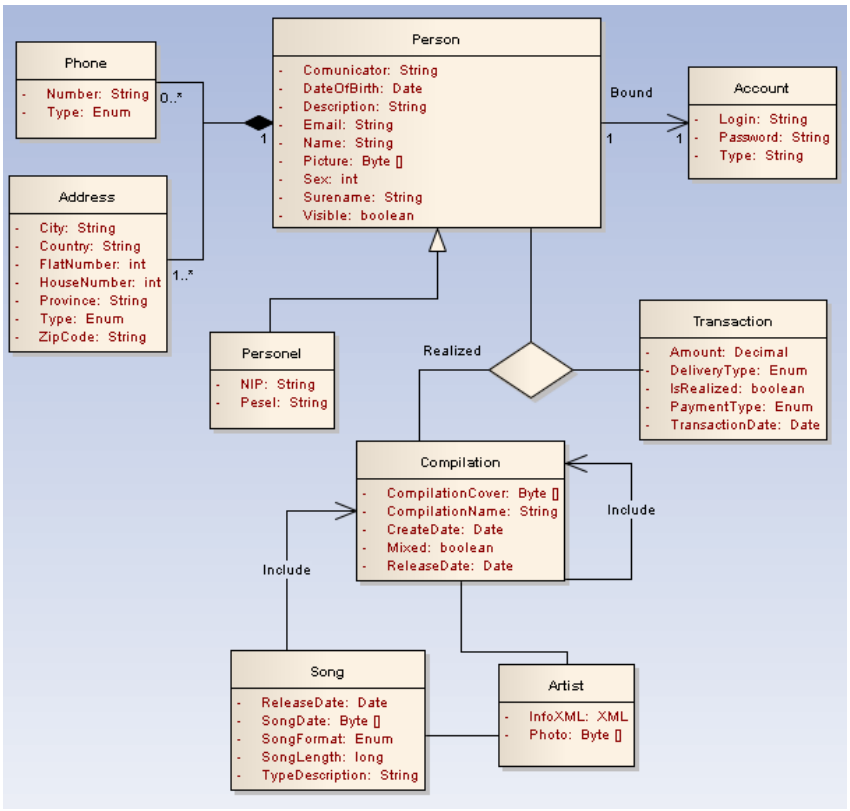
## 3. Uruchamianie testów

Stworzony system służy do testowania interfejsów bazodanowych. Chcąc sprawdzić jego działanie, konieczne było więc najpierw przygotowanie odpowiedniego interfejsu, przygotowanie danych testowych i zdefiniowanie, jakie metody i z jakimi parametrami będziemy badać. Konieczne było także określenie, jakie obciążenia nas interesują.

### 3.1. Model dziedzinowy

Jako model dziedzinowy zastosowano model sklepu internetowego z plikami multimedialnymi. Zaletą tego modelu jest duża różnorodność typów danych (zarówno typy proste, jak i BLOB czy XML) oraz łatwość wypełnienia danymi z Internetu.

Dla podanego modelu dziedzinowego przygotowano dwa niezależne interfejsy: jeden, wykorzystujący klasyczne możliwości standardu JDBC, i drugi, używający ORM. Oba interfejsy umieszczono na serwerze aplikacji JBoss w postaci komponentów EJB.

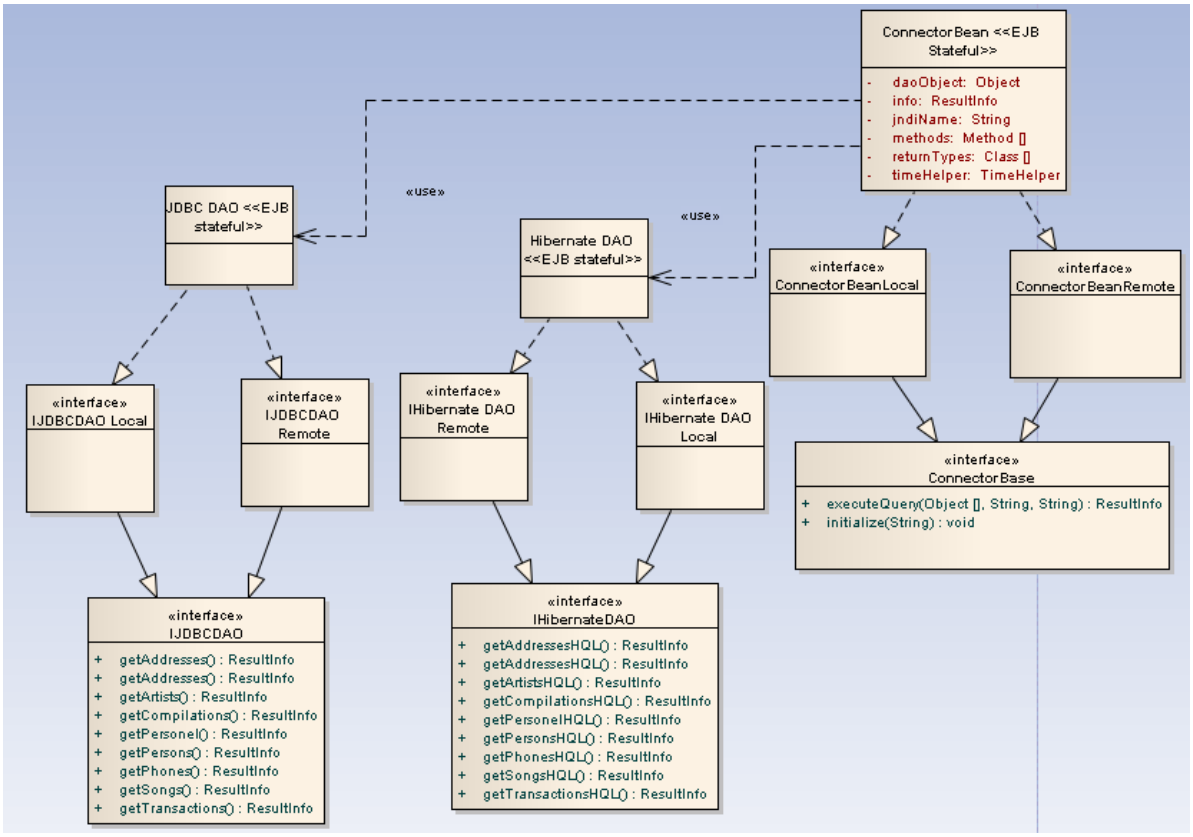



Rys. 3.Diagram klas, reprezentujący model dziedzino-
Fig. 3. Class diagram for domain model

Kolejnym krokiem było przygotowanie planu testów w postaci pliku konfiguracyjnego ładowanego przez klientów. Rodzaje zaproponowanych testów zostaną bardziej szczegółowo przedstawione w kolejnym rozdziale.

Komunikacja systemu testującego, pracującego na niezależnych komputerach z testowanymi interfejsami zainstalowanymi na serwerze aplikacji, odbywała się za pośrednictwem tak zwanego ziarenka pośredniczącego. Ziarenko pośredniczące (klasa ConnectorBean) to klasa umieszczona na serwerze, której zadaniem jest uruchamianie metod w testowanej implementacji warstwy dostępu do danych. Ziarenko wykonuje dwa zadania. Pierwsze z nich to lokalizacja referencji do testowanej implementacji w przestrzeni JNDI. Drugie zadanie polega na wywołaniu dla tej implementacji metod zdefiniowanych w planie testów. Ziarenko wykorzystuje do tego celu mechanizm refleksji języka Java. Schemat reprezentujący zależność pomiędzy ziarnem pośredniczącym a testowaną technologią obrazuje rys. 4.

Pomiar polegał na uruchomieniu odpowiedniej liczby klientów odpytujących serwer aplikacji (a więc odpowiednią implementację za pośrednictwem klasy ConnectorBean) zgodnie z zadanym planem testów. Klienci gromadzili dane na temat czasu wykonywania poszczególnych metod i przesyłali je do serwera pomiarowego. Po zakończeniu eksperymentu wszystkie zmierzone wartości znajdowały się w pliku stworzonym przez serwer pomiarowy i były dostępne do analizy.




Rys. 4. Umiejscowienie ziarenka pośredniczącego (klasy ConnectorBean)
Fig. 4. Placement of transfer bean (ConnectorBean class)

### 3.2. Opis planu definiującego pomiar

Plan jest to zbiór metadanych opisujących pomiar dokonywany przez aplikację. Jest to zwykły plik tekstowy z rozszerzeniem param. W pliku można wyróżnić kilka sekcji: każda sekcja rozpoczyna się oraz kończy znakiem #; pomiędzy znakami # znajduje się nazwa sekcji. Dostępne sekcje to:

- #Paretto# – sekcja opisująca parametry generatora Pareto;
- #ClientStart# – w tym miejscu definiujemy, z jakim interwałem będzie uruchamiany kolejny klient, jako jednostkę przyjmujemy sekundy;
- #JNDI name# – sekcja definiuje nazwę ziarenka EJB wdrożonego na serwerze aplikacji – ziarenko EJB zawiera implementację testowanej technologii;
- #Queries# – sekcja definiuje liczbę iteracji dla każdego klienta oraz zbiór metod, które chcemy przetestować.

Dodatkowo sekcja #Queries pozwala zamieszczać komentarze.

Sekwencja znaków „//” rozpoczyna komentarz. Komentarze są ignorowane podczas wczytywania planu.

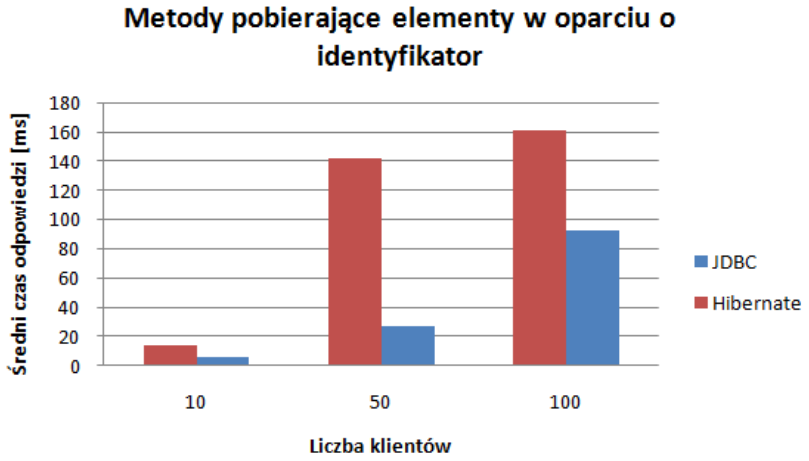
## 4. Pomiary i wyniki

Pomiar warstwy dostępu do danych dotyczył następujących aspektów badanych technologii:

- uzyskiwanie elementów o określonym identyfikatorze,
- pobór wszystkich obiektów przy wykorzystaniu do tego celu różnych strategii sprowadzania obiektów,
- czas wykonywania bardziej złożonych zapytań, obejmujących operacje złączenia dwóch oraz trzech tablic, operacje agregacji oraz grupowania.

Każdy z wykonanych pomiarów był przeprowadzany dla stu, pięćdziesięciu oraz dziesięciu klientów.

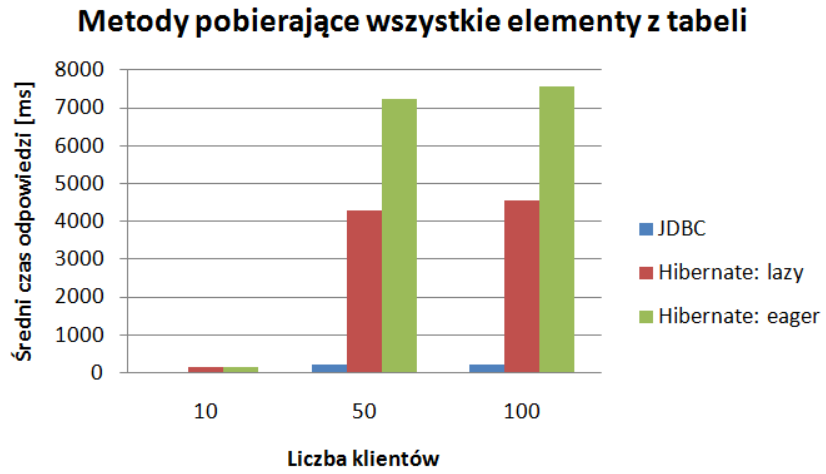
Pierwszy pomiar dotyczył wydajności obu technologii podczas pobierania pojedynczego obiektu, wykorzystując unikalny identyfikator. Pomiar wykazał większą wydajność JDBC, szczególnie dla pięćdziesięciu klientów. W tym wypadku JDBC było około czterokrotnie szybsze od Hibernate'a. Pomiar dla stu klientów wykazał już mniejszą różnicę JDBC w stosunku do Hibernate'a. W tym przypadku Hibernate był średnio dwukrotnie wolniejszy od JDBC. Wyniki pomiaru wskazują, iż wydajność JDBC może znacząco spaść przy wzroście liczby klientów. Potwierdzają to pozostałe wyniki pomiarów.




Rys. 5. Czasy wykonania metod pobierających jeden obiekt o podanym identyfikatorze
Fig. 5. Execution times of methods retrieving one object with known identifier

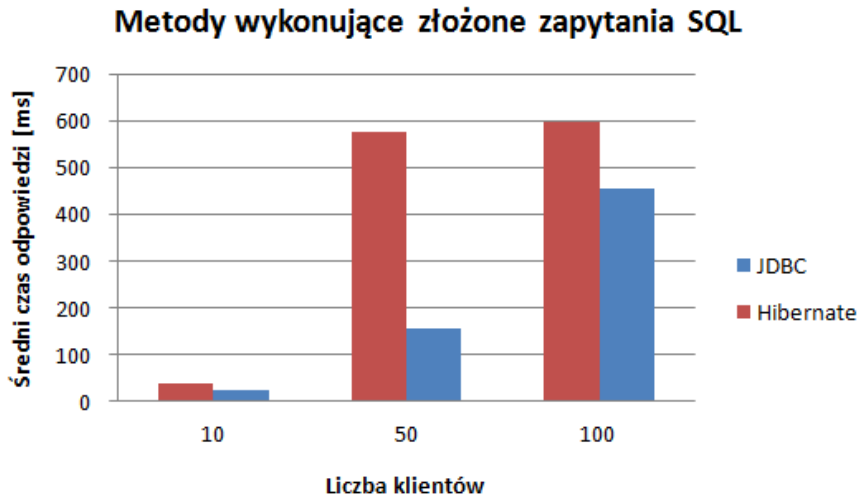
Kolejny wykres przedstawia wydajność testowanych technologii w przypadku pobierania wszystkich elementów z tablicy. Każda tablica zawierała tysiąc rekordów. Pomiar wykazuje znaczącą przewagę JDBC nad frameworkiem Hiberante. Dodatkowo na korzyść JDBC przemawia stosunkowo niewielkie zapotrzebowanie na zasoby, takie jak pamięć operacyjna. Hibernate wymagał zwiększenia sterty maszyny wirtualnej z 512 MB do 800 MB. Praca na standardowych ustawieniach maszyny wirtualnej powodowała zawieszanie serwera aplikacji

podczas pomiarów. Podczas testu wykorzystano dwie strategie sprowadzania obiektów [1]. Sprowadzanie leniwie wykazało się dużo lepszą wydajnością niż sprowadzanie wyprzedzające. Wynika to z faktu, iż podczas sprowadzania wyprzedzającego, Hibernate pobiera z bazy także wszystkie dane powiązane z aktualnymi, aby uniknąć w przyszłości konieczności wysyłania dodatkowych zapytań. Zastosowanie strategii wyprzedzającej spowodowało spadek wydajności oraz wymusiło dalsze powiększanie sterty pamięci maszyny wirtualnej Java.




Rys. 6. Czasy wykonania metod pobierających wszystkie elementy z tabeli
Fig. 6. Execution times of methods retrieving all objects from table

Ostatni pomiar dotyczył bardziej złożonych zapytań SQL. Każde z zapytań zawierało złączenie, warunki ograniczające zbiór wynikowy oraz grupowanie. Każda z metod zwracała niewielkie rezultaty (około kilkunastu wierszy). Również w tym wypadku JDBC okazało się wydajniejsze od Hibernate. Różnica wydajności obu technologii różni się w zależności od liczby klientów. Im większa liczba klientów (powyżej pięćdziesięciu), tym bardziej różnica pomiędzy Hibernatem a JDBC zmniejsza się.




Rys. 7. Czasy wykonania metod wykonujących złożone zapytania SQL
Fig. 7. Execution times of metods executing complex SQL queries

## 5. Wnioski

Uzyskane wyniki badań są zgodne z oczekiwaniami. Zastosowanie frameworku Hibernate wpływa negatywnie na wydajność wykonywanych operacji. Pogorszenie wydajności jest jednak rekompensowane przez łatwiejsze tworzenie kodu dzięki wewnętrznym mechanizmom biblioteki, zarządzającym obiektami, oraz automatycznemu mapowaniu rezultatów zapytań do obiektów języka Java. Na uwagę zasługuje fakt, że dla wszystkich testów w miarę zwiększania obciążenia różnice wydajności JDBC i Hibernate maleją.

Podkreślić jednak należy, że porównanie dwóch interfejsów jest tylko jednym z możliwych zastosowań prezentowanego rozwiązania. Symulowanie obciążenia i proste definiowanie planów testowania dają możliwość sprawdzenia interfejsów bazodanowych konkretnych aplikacji w sposób bardzo zbliżony do późniejszego, rzeczywistego obciążenia. Dzięki użyciu mechanizmów refleksji języka Java, rozwiązanie jest zupełnie niezależne od rodzajów tych interfejsów.

Symulowanie rzeczywistej pracy systemu może przynieść wiele interesujących informacji. Przykładowo, wykonywane podczas przedstawionego eksperymentu pomiary pozwoliły, niejako przy okazji, wskazać metodę, której wydajność może negatywnie wpłynąć na całą aplikację. Analiza danych wskazała metodę getPersonsLoginInfo jako potencjalnie niebezpieczną. Jej średni czas wykonania był około dwukrotnie dłuższy niż czas wykonania pozostałych metod. Co ciekawe, wykorzystanie standardowego analizatora bazy danych wykazało dla zapytania koszt 196 timeronów (timeron – jednostka wykorzystywana przez IBM DB2 do określania kosztu zapytań SQL), czyli wartość podobną do pozostałych zapytań. Metody nie różniły się pod względem: kosztu wykonania zapytania SQL, wielkości zwracanego rezultatu oraz typu zwracanego rezultatu. Pomimo braku różnic pomiędzy analizowanymi metodami, pomiar pozwolił wychwycić metodę, której rzeczywisty średni czas wykonania odbiegał od pozostałych metod. Skrócenie czasu odpowiedzi uzyskano przez zastosowanie indeksu na tablicy Accounts. Czas wykonania metody został zredukowany z 817 ms do 372 ms.

## 6. Podsumowanie

Pomiar wydajności aplikacji przed wprowadzeniem jej do środowiska produkcyjnego jest niesłychanie ważnym elementem każdego wdrożenia. Zaprezentowane rozwiązanie umożliwia wykonanie symulacji pod dużym obciążeniem. Przeprowadzony eksperyment, polegający na analizie dwóch aplikacji, wykazał jego skuteczność.

## BIBLIOGRAFIA

1. Bauer C., King G.: Hibernate w akcji. Helion, Gliwice 2007.
2. Fisher M., Ellis J., Bruce J.: JDBC™ API Tutorial and Reference. Prentice Hall, 2003.
3. Smart J. F.: Java. Praktyczne narzędzia. Helion, Gliwice 2009.
4. Horstmann C., Cornell G.: Java 2. Techniki zaawansowane. Helion, Gliwice 2003.
5. Hitchens R.: Java NIO. O'Reilly, Sebastopol, CA, USA 2002.
6. Hryń G.: Badanie możliwości dostosowania charakterystyki serwera WWW do potrzeb klienta. Rozprawa doktorska, Politechnika Śląska 2005.

Recenzenci: Prof. dr hab. inż. Andrzej Grzywak
Prof. dr hab. inż. Tadeusz Wieczorek

Wpłynęło do Redakcji 31 stycznia 2011 r.

## Abstract

Testing of database applications' performance is a serious problem in many usages. The answer to a question how will application behave under normal usage demands preparing specialized tests what is often a time consuming task. The solution presented in the article tries to automate this process. It gives also opportunity to compare different interfaces. the system is using distributed environment and simulates real load with Pareto distribution.

## Adresy

Łukasz PIETRZAK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska.
Paweł KASPROWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, kasprowski@polsl.pl.