STUDIA INFORMATICA 2014
Volume 35 Number 2 (116)

Piotr NOWAK, Katarzyna HARĘŻLAK
Politechnika Śląska, Instytut Informatyki

ANALIZA WYBRANYCH METOD MAPOWANIA OBIEKTOWO-RELACYJNEGO

Streszczenie. W artykule poruszono zagadnienie integracji relacyjnych systemów baz danych z aplikacjami napisanymi obiektowo. Uwagę skupiono na dwóch popularnych interfejsach mapowania obiektowo-relacyjnego (Entity Framework oraz Hibernate), analizując ich funkcjonalność oraz wydajność. Zbadano również przebieg procesu migrującego bazę danych do innego środowiska, przy (oczekiwanym) zachowaniu identycznej postaci kodu.

Słowa kluczowe: interfejsy mapowania obiektowo-relacyjnego, wydajność

THE ANALYSIS OF THE CHOSEN OBJECT-RELATIONAL MAPPING METHODS

Summary. The problem of relational databases integration with object oriented applications was discussed in the paper. Two popular object-relational mapping interfaces were analyzed in terms of their functionality and performance. In the paper, the database schema migration process with retaining the original form of the application code was studied was studied as well.

Keywords: object-relational mapping interfaces, performance

1. Wstęp

Zagadnienie integracji relacyjnych systemów baz danych z aplikacjami napisanymi obiektowo, przy użyciu mapowania obiektowo-relacyjnego (ORM), jest obecne w świecie informatyki od kilku lat. Technika ORM pozwala programistom użyć semantyki i zasad właściwych obiektowym językom programowania do przeprowadzania operacji na danych, ni-

welując konieczność umieszczania w aplikacji zapytań w tradycyjnej, „surowej" formie. Sposób, w jaki się ten proces odbywa, jest uzależniony od użytego interfejsu ORM.

Fakty te były przesłankami do przeprowadzenia badań popularnych interfejsów mapowania obiektowo-relacyjnego przez analizę ich funkcjonalności, sposobu odczytu i zapisu danych *do* oraz *z* bazy danych, wykorzystania procedur składowanych, a także współpracy z różnymi serwerami relacyjnych baz danych. Sprawdzono także wydajność metod dostępu do danych oferowanych przez analizowane interfejsy. Działania te miały na celu dostarczenie oceny badanych mechanizmów, która może być pomocna w doborze odpowiedniego narzędzia do aktualnie rozwiązywanego problemu.

Podobne badania były już prowadzone m.in. w [4, 6, 11]; niniejsza praca stanowi ich rozwinięcie i uzupełnienie.

2. Architektura środowiska pracy i scenariusze testowe

Prace badawcze były prowadzone z wykorzystaniem dwóch popularnych narzędzi ORM – Entity Framework oraz Hibernate – związanych z określonymi platformami programistycznymi – odpowiednio .NET Framework oraz Java. Oba analizowane interfejsy połączono z serwerami SQL Server, MySQL oraz PostgreSQL za pomocą wybranego dostawcy (ang. *provider*). Na potrzeby testów zaprojektowano i stworzono bazy danych dla systemu ułatwiającego zarządzanie hotelem, które zostały wypełnione danymi na podstawie wskazówek z [7]. Środowisko pracy przedstawiono w tabeli 1.

Tabela 1

Technologie używane w badaniach

	Entity Framework	**Hibernate**
Platforma	.NET 4.5.50709	Java SE v7u40
Wersja interfejsu ORM	5.0.0.0	4.2.3 Final
Dostawca dla MySQL	Connector/Net 6.7.4	Connector/J 5.1.26
Dostawca dla SQLServer	System.Data.SqlClient (wbudowane w .NET)	Microsoft JDBC Driver for SQL Server, 4.0
Dostawca dla PostgreSQL	Npgsql 2.0.12.0	JDBC4 Postgresql Driver, Version 9.2-1003

Tematyka mapowania obiektowo-relacyjnego jest bardzo rozległa i niełatwo jest opisać wybrane rozwiązania w sposób zwięzły, dlatego do celów analizy metod ORM określono listę fundamentalnych zagadnień związanych z tą dziedziną, które następnie zostały rozpatrzone w badaniach. Każdy z wybranych interfejsów został poddany analizie w kontekście realizacji podstawowych operacji na bazie danych, na zmianie jej schematu oraz na efektywności działania.

Ponadto w badaniach znalazł się tzw. scenariusz migracyjny. Zakładał on utworzenie projektu przy użyciu jednego narzędzia ORM i jednej określonej bazy danych, a następnie symulację migracji środowiska z tej bazy danych na inną, przy (oczekiwanym) zachowaniu identycznej postaci kodu testowego.

2.1. Operacje odczytu

Analizując sposoby realizacji operacji odczytu, można stwierdzić, że w przypadku obu badanych interfejsów dostępne są dwie dominujące grupy metod. W jednej znajdują się rozwiązania cechujące się większą obiektowością i są to dla EF – LINQ to Entities (L2E) oraz Query by Criteria (QBC) dla interfejsu Hibernate. Druga z grup obejmuje rozwiązania, które wykorzystują własne implementacje języka SQL – odpowiednio Entity SQL oraz HQL.

Wszystkie wymienione metody pozwalają wykonywać złożone zapytania, jednakże czasami pociąga to za sobą określone problemy. Dla przykładu rozwiązanie L2E korzysta z silnego typowania, co znacznie ułatwia refaktoryzację kodu, może natomiast utrudnić tworzenie dynamicznych zapytań. Z kolei jego odpowiednik interfejsu Hibernate – QBC – operujący na łańcuchach znaków, lepiej nadaje się do tworzenia dynamicznych rozwiązań, jednakże nawet najdrobniejsze pomyłki w tekście będą skutkować wyjątkami pojawiającymi się dopiero w trakcie działania programu.

Wyniki zapytań są zwracane przez oba opisane interfejsy w postaci zmaterializowanej, tj. instancji encji. Nieco inaczej wygląda to przy selekcji kilku pojedynczych wartości. W przypadku LINQ to Entities wyniki są zwracane w postaci typów anonimowych, co zwiększa poziom abstrakcji, znacząco ułatwiając implementację kodu. W przypadku zapytań kryterialnych oraz języków ESQL i HQL wyniki są zawarte w kolekcjach, a odwołanie się do nich nieco obniża poziom obiektowości rozwiązania.

Przeprowadzone testy pokazały również wpływ sposobu tłumaczenia zapytań na język zrozumiały dla konkretnej bazy danych. Negatywnym tego skutkiem, dla przykładu, była próba wykonania zapytania zawierającego klauzulę *GROUP BY*, w której zabrakło części elementów frazy *SELECT*. Zapytanie tego typu w środowisku MySQL zostało wykonane bez żadnych problemów, co wynika ze składni implementacji języka SQL w tym systemie [13]. Próba migracji środowiska testowego do współpracy z systemem Microsoft SQL Server skutkowała błędem. Z kolei do zalet tego drugiego serwera można zaliczyć automatyczną parametryzację przetłumaczonych zapytań, zapobiegającą atakom typu SQL Injection.

Przykładowe zapytania do bazy danych zaprezentowano w tabeli 2.

Tabela 2

Przykładowe zapytania w języku SQL

Selekcja, projekcja, sortowanie
SPS2: selekcja kilku kolumn – osoby urodzone po dacie, malejąco
	`SELECT Imie, Nazwisko, DataUrodzenia`
	`FROM TOsoba`
	`WHERE DataUrodzenia > '1980-01-01'`
	`ORDER BY DataUrodzenia`

Złączenia, podzapytania
ZP3: imiona i nazwiska gości, którzy dokonali jakiejkolwiek płatności na kwotę większą niż 100000
	`SELECT DISTINCT TGosc.IdGosc, TOsoba.Imie, TOsoba.Nazwisko`
	`FROM TOsoba`
	`JOIN TGosc ON TOsoba.IdOsoba = TGosc.IdGosc`
	`JOIN TRezerwacja ON TGosc.IdGosc = TRezerwacja.IdGosc`
	`WHERE TRezerwacja.IdRezerwacja IN`
	`(SELECT IdRezerwacja`
	`FROM TPlatnosc`
	`WHERE Kwota > 100000)`

Grupowanie, funkcje agregujące
GFA3: suma przychodów dla gości z poszczególnych miast
	`SELECT TGosc.Miasto, SUM(TPlatnosc.Kwota)`
	`FROM TPlatnosc`
	`LEFT JOIN TRezerwacja ON TPlatnosc.IdRezerwacja = TRezerwacja.IdRezerwacja`
	`RIGHT JOIN TGosc ON TRezerwacja.IdGosc = TGosc.IdGosc`
	`WHERE TGosc.Miasto IS NOT NULL`
	`GROUP BY TGosc.Miasto`

2.2. Lazy Loading

Podczas badań zwrócono uwagę na zagadnienie *lazy loading*, czyli opóźnionego ładowania informacji powiązanych z daną encją. W metodach udostępnianych przez interfejsy ORM zapytania mogą zwracać nie tylko wskazane kolumny, lecz także całe encje i encje powiązane z nimi. Moment, w którym te niewskazane jawnie dane są ładowane z bazy do aplikacji obiektowej, kontroluje mechanizm opóźnionego ładowania.

Ciekawym spostrzeżeniem jest nieco odmienne podejście do tego zagadnienia w obu analizowanych interfejsach. W Entity Framework wyłączenie opóźnionego ładowania powoduje, że programista sam musi zadbać o dołączenie do zapytania powiązanych danych. W Hibernate natomiast oznacza to, że przy zapytaniu o konkretną z encji od razu nastąpi pobranie danych encji z nią powiązanych. Wynika to z możliwości konfiguracyjnych dotyczących omawianej właściwości. EF umożliwia konfigurowanie opóźnionego ładowania jedynie na poziomie kontekstu, choć zaletą jest możliwość programistycznego włączania i wyłączania tej funkcji. Hibernate jest bardziej elastyczny w tej pierwszej kwestii, gdyż pozwala na zdefiniowanie *lazy loading* również na poziomie pojedynczych tabel (klas), a nawet kolumn (właściwości), co jest szczególnie przydatne, w przypadku gdy zawierają one dużą ilość danych (np. kolumny typu BLOB). Niestety jednak *lazy loading* w Hibernate jest definiowane w plikach z opisami mapowania, przez co nie da się w prosty sposób zmienić jego ustawień w trakcie wykonywania programu.

2.3. Zapis, współbieżność dostępu do danych

W podobny sposób, jak to opisano w poprzednim podrozdziale, zostały przetestowane operacje zmieniające dane w bazie – *update*, *insert* oraz *delete*, ze szczególnym uwzględnieniem śledzenia zmian (ang. *change tracking*) w encjach (obiektach) oraz związanego z nim problemu współbieżnego dostępu do danych.

Polecenia z grupy dodawania rekordów, oprócz prostego dodania wpisu do tabeli, miały za zadanie sprawdzić działanie zarówno automatycznej generacji wartości klucza głównego, jak i więzów kluczy głównego i obcego.

W dalszej kolejności zostały wykonane zapytania typu *update* i *delete*. Poza typowym uaktualnieniem wiersza tabeli zostały one użyte do analizy współbieżnego dostępu do rekordu, przy wykorzystaniu przygotowanych do tego celu kolumn typu *rowversion* (znanych też pod nazwą *timestamp*).

Entity Framework do utrwalenia modyfikacji dotyczących danych korzysta z metody kontekstu – podstawowej klasy odpowiedzialnej za interakcje z obiektami. Kontekst jest odpowiedzialny za wysłanie do bazy danych wszystkich zmian dokonanych w obszarze jego aktywności. Za śledzenie zmian odpowiadają wpisy *ObjectStateEntry*. Wspomniana metoda kontekstu domyślnie otacza wysyłane zapytania transakcją.

Hibernate, w zależności od wybranego menadżera transakcji, udostępnia kilka strategii zapisu danych. W projektach użyto czystego JDBC (ang. *plain JDBC*), co sprawiło, że sesja była ograniczona do transakcji i kończyła się wraz z zatwierdzeniem (ang. *commit*) lub cofnięciem (ang. *rollback*) zmian. Śledzenie zmian jest tutaj mniej przejrzyste niż w EF, co może być zarówno zaletą, jak i wadą, gdyż z jednej strony wydajność zastosowanego rozwiązania w Hibernate jest dużo lepsza niż w EF (co zostanie pokazane w rozdziale 3), z drugiej jednak istnieją sytuacje, w których przydatny bywa dostęp do oryginalnej oraz zmienionej wartości.

Jeśli chodzi o stosowanie strategii *optimistic concurrency*, oba interfejsy obsługują typy danych *rowversion* (*timestamp*) i radzą sobie z podstawowymi problemami współbieżnej aktualizacji.

W badaniach zauważono problemy wynikające z działania systemów zarządzania bazami danych. Nie wszystkie one są jednakowe i nawet małe różnice mogą nieco utrudnić migrację: PostgreSQL ma własną kolumnę przechowywania wersji rekordu [14], a MySQL nie dysponuje typem danych (G)UID [13]. Przy zmianie wykorzystywanego serwera bazy danych (migracji) pociąga to za sobą konieczność wykonania dodatkowych operacji. W pierwszym przypadku można dokonać mapowania wspomnianej kolumny na właściwość używaną do kontroli wersji lub dodać kolejną, oprócz już istniejącej. Drugi problem można w prosty

(jednak konieczny) sposób rozwiązać, wykorzystując w bazie danych kolumnę typu *varchar* i mapując go, po stronie aplikacji, na właściwość typu *string*.

2.4. Procedury składowane i funkcje użytkownika

W bazach danych często wykorzystywane są procedury składowane (ang. *stored procedures*) i funkcje zdefiniowane przez użytkownika (ang. *user-defined functions*), których wywołanie powoduje wykonanie, bezpośrednio po stronie serwera, wcześniej zdefiniowanych operacji na danych. Wynikiem tych działań mogą być zwracane zbiory rekordów.

Możliwość korzystania z takich obiektów z poziomu aplikacji jest pożądaną opcją w interfejsach typu ORM, którą sprawdzono na przykładzie sześciu procedur składowanych. Analizowano, w jaki sposób badane interfejsy obsługują wywołanie procedur i funkcji oraz jak zarządzają wynikami ich działania.

W pliku modelu Entity Framework zawarte są nagłówki procedur – tj. metadane, takie jak nazwa, rodzaje argumentów oraz zwracanych wartości. Na ich podstawie interfejs EF generuje metody, które je wywołują, oraz klasy rezultatów powiązane z ich wynikami. Rolą dostawcy jest przetłumaczenie wywołania na polecenie zrozumiałe dla serwera.

Hibernate definiuje procedury jako „nazwane zapytania", których sposób wywołania, w zależności od systemu bazodanowego, programista musi zdefiniować w mapowaniu tuż obok pozostałych metadanych, stąd bowiem pobierana jest operacja wysyłana do bazy danych. Używanie podczas wywołania nazw procedur i parametrów w postaci łańcuchów znaków jest jednak mniej wygodne niż w postaci obiektów.

Funkcje zarówno w EF, jak i Hibernate są traktowane jako elementy dedykowanych języków zapytań ESQL i HQL. Entity Framework, podobnie jak w przypadkach innych obiektów, metadane dotyczące funkcji przechowuje w definicji modelu, interfejsowi Hibernate wystarczy zaś rejestracja samych ich nazw w momencie konfiguracji.

Migracja omawianych elementów pomiędzy systemami Microsoft SQL Server i MySQL w środowisku Entity Framework nie wymagała żadnej ingerencji w kod klas – wywołane procedury i funkcje zwróciły poprawne wyniki. Niestety nie udało się wykonać testów procedur składowanych oraz funkcji dla systemu PostgreSQL – dostawca Npgsql w użytej wersji nie obsługuje procedur składowanych. Przenoszenie kodu związanego z narzędziem Hibernate z projektu łączącego się z MySQL do powiązanego z systemem Microsoft SQL Server nie odbyło się bez drobnych korekt. Wymagane były poprawki w sposobach wywołania procedur – zamiast polecenia *call* ten drugi system używa *exec*.

3. Analiza wydajności

Testom wydajności posłużyły metody mierzące czas wykonania zapytań odczytu oraz zapisu dużej liczby rekordów *z* i *do* baz danych. We wszystkich opisywanych przypadkach badania zostały zrealizowane we współpracy z systemem SQL Server i obejmowały logicznie takie same zapytania, dostosowane do mechanizmów udostępnianych przez każdy z opisywanych interfejsów.

Testy wydajności odczytu dotyczyły dwóch zbiorów danych. Pierwszy z nich obejmował 100 tysięcy, natomiast drugi 500 tysięcy rekordów. Zapytania podzielono dodatkowo na dwie grupy, w których jedna korzysta z właściwości śledzenia zmian. Testowanie aktualizacji rekordów polegało na utworzeniu i zapisie 2000, 5000 oraz 10 000 obiektów do przykładowej encji. Podobnie zaprojektowane testy przeprowadzono w [2].

3.1. EF – porównanie wydajności odczytu

W testach interfejsu Entity Framer zapytania zrealizowano trzema metodami – LINQ to Entities, Entity SQL Query oraz Entity Client. Ostatnia z metod, choć poddaje zapytania najmniejszej liczbie transformacji, wymaga jednak ręcznej materializacji otrzymanych danych do postaci obiektów określonego w zapytaniu typu encji.

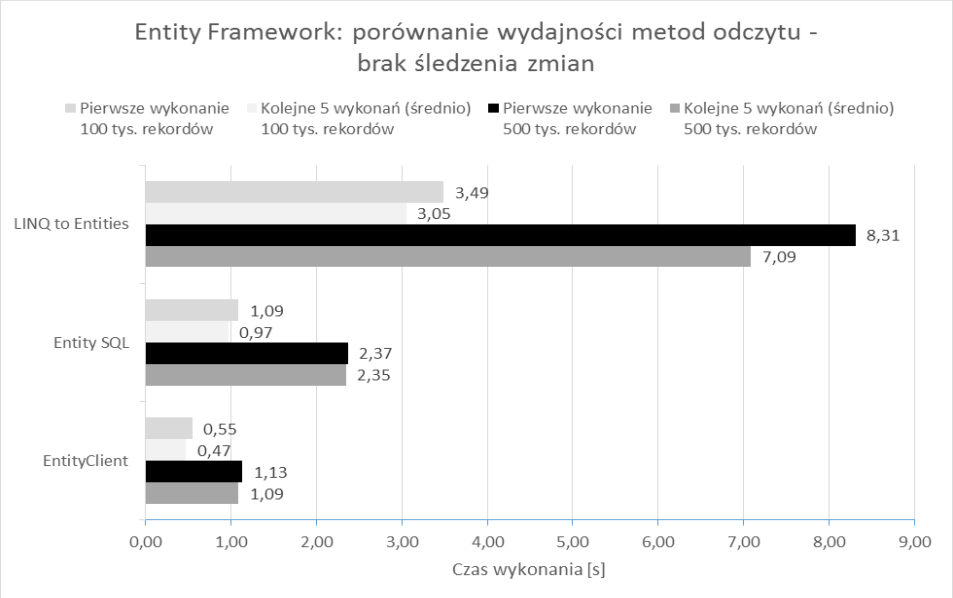
W Entity Framework włączone śledzenie zmian oznacza, że kontekst nadzoruje zmiany w pobranych obiektach. W momencie zapisu obiektów do bazy danych na podstawie stanu *ObjectStateEntry* podejmowana jest decyzja, jakie polecenie języka SQL należy wygenerować. W przeciwnym przypadku programista musi samodzielnie dołączyć obiekt do kontekstu i ustawić odpowiedni dla stanu obiektu. Wyłączenie śledzenie zmian w LINQ to Entities jest realizowane wywoływaną dla zapytania metodą *AsNoTracking()*. Odpowiednikiem w ESQL jest ustawienie właściwości zapytania *MergeOption* na *NoTracking*.

Na rys. 1 zaprezentowano wyniki badań przeprowadzonych z wyłączoną opcją śledzenia zmian. Zmierzono czasy pierwszego przebiegu oraz uśrednione czasy pięciu kolejnych przebiegów pętli odczytującej rekordy z bazy danych. Uzyskane rezultaty potwierdzają fakt, że większa liczba transformacji na drodze od wywołania zapytania do uzyskania – w postaci obiektowej – wyniku pociąga za sobą dłuższy czas jego realizacji. Przykładem może być najbardziej wysokopoziomowe rozwiązanie LINQ to Entities, dla którego uzyskano kilkukrotnie gorsze wyniki niż dla pozostałych metod. Entity SQL przechodzi jedną transformację mniej, co widać w znacznym przyroście wydajności zapytań. Entity Client znajduje się „najbliżej" źródła danych, w związku z czym wydajność tej metody jest największa.

Poza wybraną metodą zapytań oraz śledzeniem zmian w Entity Framework wpływ na wydajność zapytań ma jeszcze kilka opcji [11]:

- *Query Plan Caching* – determinuje przechowywanie raz wykonanego zapytania w specjalnej pamięci podręcznej, do której następuje odwołanie w momencie kolejnego uruchomienia zapytania,
- *Pre-generated Views* – umożliwia ładowanie modelu już podczas kompilacji,
- *Compiled Queries* – umożliwia traktowanie zmiennych z zapytaniami jako statycznych (*static*) i tylko do odczytu (*read only*), przez co są one kompilowane razem z aplikacją,
- *Automatic Compiled Queries* – jest to domyślnie włączony odpowiednik *Query Plan Caching* dla zapytań L2E,
- *Lazy Loading* – opisywane wcześniej opóźnione ładowanie, które niewykorzystywane prawidłowo może wpłynąć negatywnie na wydajność,
- projekt modelu konceptualnego – duże różnice między modelami konceptualnym i składowania powodują opóźnienia w transformacjach zapytań.

Kilka powyższych zagadnień znajduje swoje odzwierciedlenie na wykresie przedstawionym na rys. 1. Różnice między pierwszym i kolejnym wykonaniem zapytań tego samego typu wynikają z niezastosowania strategii *Pre-generated Views*, przez co model był przy pierwszym zapytaniu ładowany do pamięci. W tym samym miejscu widoczny jest wpływ automatycznie kompilowanych zapytań dla wariantu L2E – użycie opcji *CompiledQuery* w miejscu zastosowanego automatycznego rozwiązania mogłoby przyspieszyć wykonywanie zapytań.



Rys. 1. EF: porównanie wydajności metod odczytu – brak śledzenia zmian
Fig. 1. EF: select query efficiency comparison with change tracking disabled

Włączenie śledzonych zapytań pokazuje, jak duży wpływ mają one na wydajność zarządzania obiektami encji (tabela 3). Otóż przy 100 tysiącach rekordów pierwsze zapytanie L2E zajęło nieco mniej niż 50 sekund. Przy 500 tysiącach rekordów wykonanie przerwano po

kilkunastu minutach ze względu na podejrzenie, że długość jego trwania w zależności od liczby danych wzrasta wykładniczo. Dodatkowy test na 10 tysiącach rekordów potwierdził te przypuszczenia. Śledzenie zmian sprawia, że podczas materializacji pobranych rekordów do obiektów encji Entity Framework sprawdza, czy podobny obiekt przypadkiem nie istnieje w pamięci podręcznej kontekstu i jeśli tak jest, to go dodaje i zaczyna śledzić. Powoduje to tym większy narzut czasowy, im więcej obiektów jest śledzonych. Kolejne zapytania wykonywały się znacznie krócej, gdyż nie wymagały aktualizacji zawartości kontekstu.

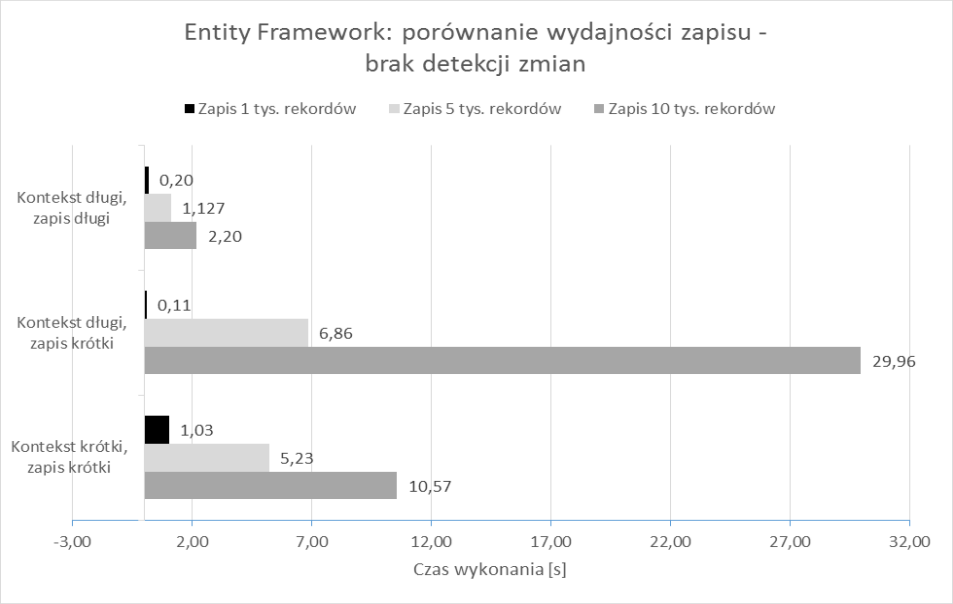
Tabela 3

Entity Framework: porównanie wydajności metod odczytu – śledzenie zmian

Metoda	Wykonanie	10 tysięcy rekordów	100 tysięcy rekordów	500 tysięcy rekordów
LINQ to Entities	Pierwsze	1,36 s	47,72 s	[Przerwano]
	Kolejne	0,03 s	0,29 s	[Przerwano]
Entity SQL	Pierwsze	1,08 s	49,3 s	[Przerwano]
	Kolejne	0,02 s	0,19 s	[Przerwano]

3.2. EF – porównanie wydajności zapisu

Badanie wydajności modyfikacji danych zostało przeprowadzone w kilku konfiguracjach. Pierwszym parametrem był wybór metody zapisu. Można do tego podejść na dwa sposoby – wykonywać metodę *SaveChanges()* po każdorazowym utworzeniu obiektu („zapis krótki") lub wywołać ją raz po utworzeniu wszystkich („zapis długi"). Drugim ustawieniem, które można konfigurować, jest długość trwania kontekstu. Tu również można wyróżnić kilka metod; najbardziej skrajne to – utworzenie jednego „długiego" kontekstu i utrzymywanie go przez cały czas trwania pętli zapisującej oraz tworzenie „krótkiego" kontekstu na rzecz dodania pojedynczego elementu. Krótki kontekst jest charakterystyczny dla aplikacji internetowych i usług, długi zaś dla aplikacji typu desktop. Trzeci wyróżnik to automatyczne wykrywanie zmian, ustawiane właściwością *AutoDetectChangesEnabled* dostępną na poziomie kontekstu.

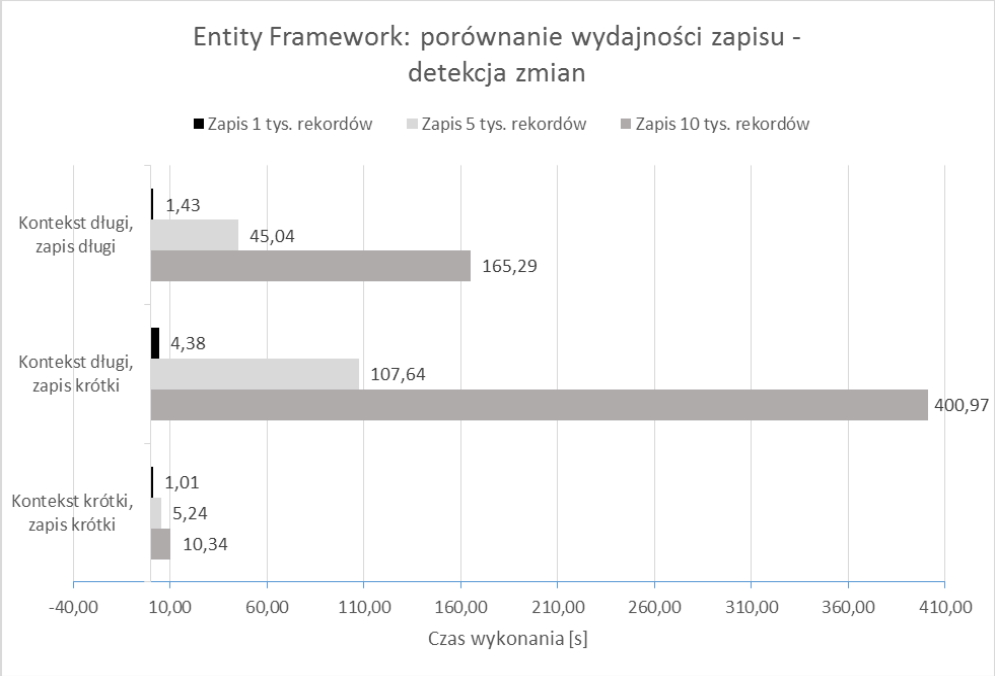


Rys. 2. EF: porównanie wydajności zapisu – brak detekcji zmian
Fig. 2. EF: update statements efficiency comparison without change detection

Wykres na rys. 2 przedstawia wyniki badań przy wyłączonej automatycznej detekcji zmian. Najszybciej wykonały się grupowe zapisy w ramach długiego kontekstu. Z kolei każdorazowe wywoływanie metody *SaveChanges()* w ramach takiego kontekstu (zapis krótki) poskutkowało dla dużej liczby rekordów dużo gorszymi wynikami czasowymi. Można wnioskować, że jest to spowodowane narzutem czasowym operacji wykonywanych przez metodę *SaveChanges()*. Pośrodku znalazło się ustawienie, w którym kontekst tworzy się w celu zapisu jednego rekordu – widać tutaj narzut akcji wielokrotnego tworzenia kontekstu.

Sytuacja zmienia się po włączeniu automatycznej detekcji zmian – wykres na rys. 3. Przy długim kontekście czas wykonywania operacji znacznie się wydłużył (nawet kilkadziesiąt razy). Automatyczne wykrywanie zmian powoduje wywołanie metody *DetectChanges()* w momencie wykonywania się metody *SaveChanges()*. Iteruje ona po wszystkich śledzonych przez kontekst obiektach i m.in. porównuje ich obecne wartości z tymi przechowywanymi w migawce *ObjectStateEntry*. Zatem przy *n* śledzonych obiektach podczas dodawania kolejnego skanowane są wszystkie poprzednio dodane. Klasa złożoności takiego zachowania to $O(n^2)$, co wyjaśnia tak duży przyrost czasu względem badania przy wyłączonej detekcji zmian.

Przy ustawieniu krótkiego kontekstu śledzenie zmian nie wpływa w sposób zauważalny na wydajność. Jest to spowodowane tym, że kontekst jednocześnie śledzi zaledwie jeden obiekt, po czym znika z pamięci. W jego miejsce tworzony jest nowy obiekt kontekstu, który podobnie zapisuje do bazy danych tylko jeden rekord. W rezultacie czasochłonne „skanowanie" wspomniane powyżej w tym przypadku nie występuje.



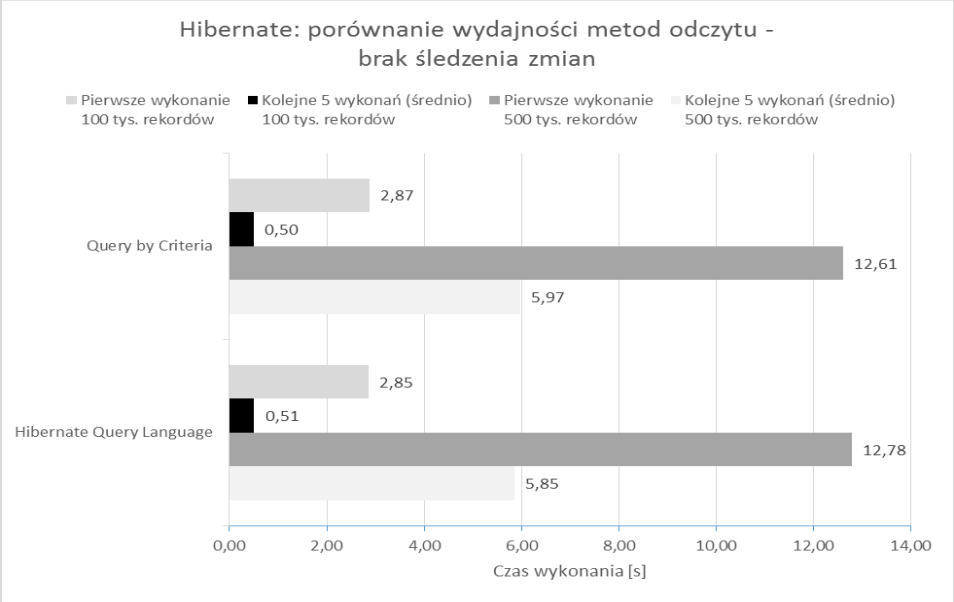
Rys. 3. EF: porównanie wydajności zapisu – detekcja zmian
Fig. 3. EF: update statements efficiency comparison with change detection

3.3. Hibernate – porównanie wydajności odczytu

W badaniach wydajności odczytu drugiego z interfejsów użyto zapytań kryterialnych oraz języka HQL, które dostarczają jako rezultat zmaterializowane obiekty odpytywanych encji.

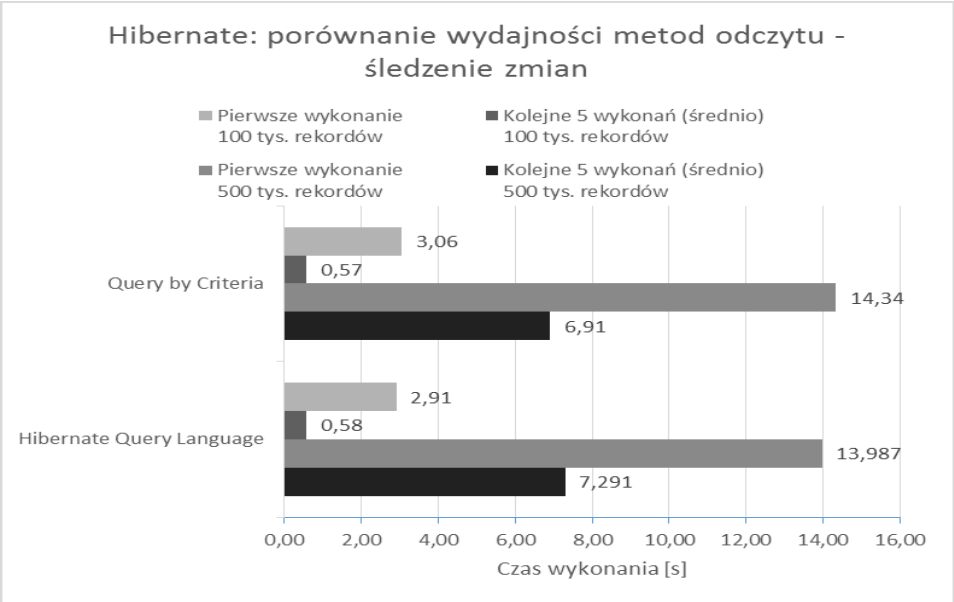
Śledzenie zmian w Hibernate jest zwane *automatic dirty checking*. Stan brudnych obiektów nie jest widoczny w aplikacji, wie o nim sesja i to powinno wystarczyć do poprawnego zaprojektowania zapytań. Parametr ustawiający tę właściwość jest dostępny na kilku poziomach – sesji, zapytania oraz obiektu. Włączenie jej w pierwszym przypadku sprawia, że śledzeniu podlegają wszystkie obiekty encji powstałe w ramach sesji. W przypadku zapytania (zarówno Criteria, jak i HQL) dotyczy to wszystkich zwróconych przez nie obiektów. Ostatnia opcja pozwala zdefiniować śledzenie na poziomie pojedynczej instancji.

Wykres prezentowany na rys. 4 przedstawia rezultaty pomiarów przy wyłączonym śledzeniu zmian. Widać na nim, że obie metody zapytań mają podobną wydajność. Teoretycznie zapytania kryterialne powinny skutkować nieco większą wydajnością ze względu na brak konieczności ich analizy składniowej (ang. *parsing*), która musi nastąpić w HQL w celu wydobycia drzewa składniowego z obiektu typu *string* [3].



Rys. 4. Hibernate: porównanie wydajności metod odczytu – brak śledzenia zmian
Fig. 4. Hibernate: update statements efficiency comparison without change detection

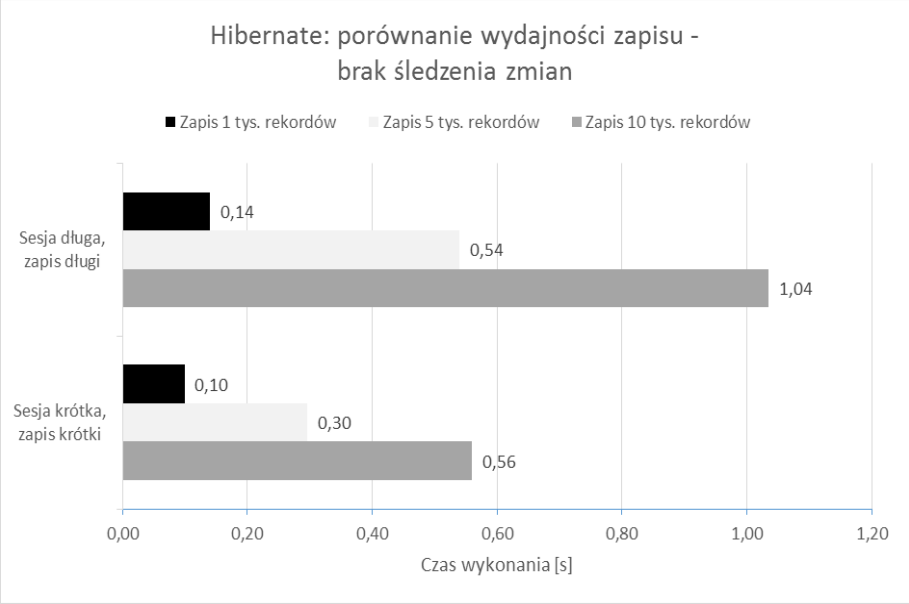
Względnie duży rozrzut wyników nastąpił między pierwszym i kolejnym wykonaniem tego samego zapytania. Jest to spowodowane m.in. koniecznością załadowania modelu z plików XML i stworzeniem tym samym fabryki sesji – zaobserwowano, że w stworzonym modelu trwa to około 2 sekundy. Wyniki przy włączonym śledzeniu zmian przedstawia wykres na rys. 5. Można zauważyć tylko nieznaczny spadek wydajności w odniesieniu do poprzedniego wykresu. Jest to dowód na to, że Hibernate w ramach sesji bardzo dobrze radzi sobie z zarządzaniem trwałymi obiektami.



Rys. 5. Hibernate: porównanie wydajności metod odczytu – śledzenie zmian
Fig. 5. Hibernate: update statements efficiency comparison with change detection

3.4. Hibernate – porównanie wydajności zapisu

Badanie wydajności modyfikacji zależało od kilku czynników. Pierwszy z nich to wybór metody zapisu rekordów – grupowo („zapis długi") lub indywidualnie. Kolejny to utrzymywanie jednej sesji przez cały czas trwania testu („sesja długa") lub tworzenie za każdym razem nowej. Jak można przypuszczać, ostatnim wyznacznikiem było śledzenie zmian, które zostało tym razem ustawione na poziomie sesji.

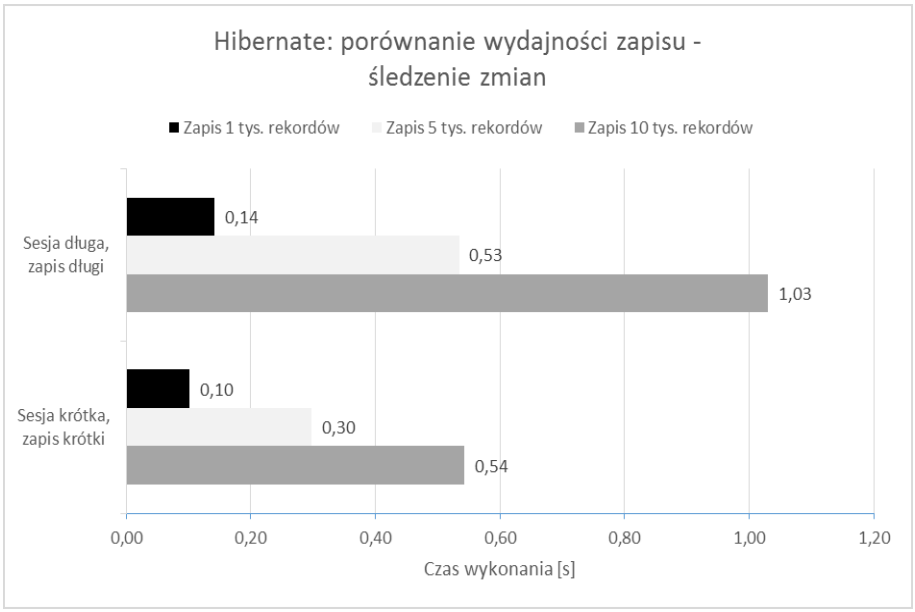


Rys. 6. Hibernate: porównanie wydajności metod zapisu – śledzenie zmian
Fig. 6. Hibernate: update statements efficiency comparison with change detection

W środowisku Hibernate zapytania DML są odwlekane w wykonaniu tak długo, jak to tylko jest możliwe – najczęściej wykonują się podczas zatwierdzania zmian transakcji („commit") [3]. Jest to spowodowane faktem, że Hibernate zwykle zakłada blokady w bazie danych do momentu zakończenia transakcji, więc pożądane jest, by trwały one jak najkrócej.

W zastosowanym podejściu sesja wiąże się z pojedynczą transakcją. W związku z tym ustawienia długości sesji oraz zapisu mają sens jedynie dla par: sesja krótka – zapis krótki; sesja długa – zapis długi. Niewymieniona tu sesja długa z krótkim zapisem byłaby możliwa dzięki wykonaniu metody *flush()* (dla sesji), która wysyła do bazy danych kwerendy DML, jednakże całość i tak byłaby zapisana dopiero po ostatnim „wypłukaniu" (bo wtedy też skończyłaby się transakcja).

Wykres przedstawiony na rys. 5 prezentuje wyniki czasowe zapisu przy wyłączonym śledzeniu zmian. Lepszym podejściem niż zwlekanie okazuje się wykonywanie małych, częstych zapisów. Pozytywnym zaskoczeniem jest znikomy wpływ śledzenia zmian na operację zapisu dużej liczby rekordów, co jest widoczne na rys. 7.



Rys. 7. Hibernate: porównanie wydajności zapisu – śledzenie zmian
Fig. 7. Hibernate: update statements sfficiency comparison with change tracking

3.5. Ocena wydajności badanych interfejsów

Przeprowadzone badania pokazały wpływ kilku elementów na wydajność komunikacji ze źródłem danych. Dla interfejsu EF podczas odczytu wraz ze spadkiem obiektowości metody zapytania zwiększała się jego wydajność, natomiast włączenie śledzenia zmian przy większej liczbie rekordów kilkusetkrotnie wydłużało czasy wykonania zapytań.

Hibernate cechuje się dużo lepszą wydajnością podczas zapisu. Śledzenie zmian jest tutaj rozwiązane w sposób, który nie dodaje dużego narzutu, co jest ogromną zaletą. Ponadto śledzenie zmian można wyłączyć na poziomie sesji, podczas gdy EF pozwala to zrobić na poziomie zapytania typu select. Przy zapisie w EF można tylko manipulować właściwością *AutoDetectChangesEnabled*, która włączona, wiąże się z ogromnym narzutem czasowym.

To, czego nie widać na pierwszy rzut oka, to dobra wydajność dla zapytań, które zarządzają małymi liczbami rekordów – obiektów (rzędu kilkuset), co sprawia, że oba interfejsy bez wątpienia nadają się do większości zastosowań biznesowych.

4. Podsumowanie

Celem badań zaprezentowanych w artykule była analiza porównawcza dwóch popularnych interfejsów mapowania obiektowo-relacyjnego: Entity Framework oraz Hibernate. Analiza ta dotyczyła zarówno udostępnianych mechanizmów, jak i sposobu ich realizacji. Ponadto przeprowadzono szereg testów dotyczących wydajności operacji realizowanych na zbio-

rach danych o różnej liczności, z zastosowaniem różnych konfiguracji wykorzystywanego środowiska testowego.

Uzyskane wyniki nie pozwoliły na wyróżnienie jednego z interfejsów. Porównując narzędzia ORM przy użyciu jednakowych scenariuszy testowych, pokazano ich mocniejsze i słabsze strony.

Możliwości konfiguracyjne Hibernate – zarówno na poziomie ogólnoprojektowym, jak i pojedynczych klas – są dużo większe niż Entity Framework. Jest to w głównej mierze wynikiem tego, że Hibernate jest narzędziem kilka lat starszym od EF, a co za tym idzie – lepiej rozwiniętym. Przez mnogość ustawień interfejs ten daje programiście większą kontrolę nad akcjami odczytywania i utrwalania danych, choć odbywa się to kosztem żmudnej konfiguracji.

W zaprezentowanych interfejsach za trwałość obiektów encji odpowiadają określone mechanizmy – kontekst w Entity Framework oraz sesja (wraz z fabryką) w Hibernate. Ich celem jest zarządzanie tymi obiektami w trakcie działania programu, w tym synchronizacja danych między aplikacją obiektową i serwerem przez śledzenie zmian. Są one punktami wyjściowymi wszelkich interakcji z bazą danych, gdyż jednym z ich obowiązków jest także zarządzanie połączeniami.

Opisywane narzędzia udostępniają, niezależne od źródła danych, dedykowane języki zapytań, które można podzielić na dwa rodzaje. Jeden z nich umożliwia budowanie zapytań przez obiektowe API (LINQ to Entities, Query by Criteria), w drugim są one składniowo zbliżone do języka SQL (Entity SQL, Hibernate Query Language). Każdy interfejs na swój sposób rozwiązuje problemy współbieżnego dostępu do danych, inaczej też radzi sobie z zagadnieniem *lazy loading*. Udostępniane są również metody umożliwiające kontakt z bazą danych za pomocą zapytań natywnych.

Obecne w bazach danych procedury składowane mogą być uruchamiane z poziomu aplikacji zarówno w Entity Framework, jak i Hibernate, jednak to w tym pierwszym ich obsługa wydaje się lepsza. Oba rozwiązania integrują wywoływanie funkcji użytkownika z językami ESQL oraz HQL.

Jedną z istotnych zalet interfejsów ORM jest możliwość dopasowania tworzonej aplikacji w większym stopniu do wymogów biznesowych niż do istniejącej struktury bazy danych. Badania objęły operacje zmiany nazw i struktury encji, dziedziczenia, podziałów pionowych i poziomych oraz mapowania warunkowego. Zaplanowane modyfikacje udało się odzwierciedlić w opisywanych narzędziach, choć nie obeszło się bez pewnych problemów, wynikających głównie z niedopracowania niektórych rozwiązań. Dziedziczenie w Entity Framework nie współpracuje dobrze z podziałem poziomym, w związku z czym trzeba było wybierać między jednym a drugim. Również mapowanie warunkowe jest w tym rozwiązaniu ograniczone do pojedynczego warunku. Środowisko Hibernate z kolei okazało się mieć zbyt re-

strykcyjne wymagania co do procedur przesłaniających kwerendy DML, wymagając odpowiedniej kolejności parametrów w definicji procedury po stronie bazy danych.

W kontekście wydajności, przy uwzględnieniu charakterystycznego dla ORM mechanizmu śledzenia zmian, Entity Framework wypada słabo przy obsłudze dużej ilości danych. Znacznie lepiej radzi sobie z tym interfejs Hibernate. Niemniej w przypadku większości biznesowych zastosowań, w których chlebem powszednim jest częsta wymiana mniejszych porcji informacji, oba interfejsy wykazują się dobrą wydajnością i nie generują wielkiego narzutu.

Trudno jednoznacznie ocenić wygodę programowania przy użyciu obu opisanych narzędzi. Wiele zależy od środowiska pracy, a te, z racji innych platform programistycznych, bardzo się różnią. Na uznanie zasługuje integracja Entity Framework z Visual Studio. Po stronie Hibernate zaletą może być używanie klas POJO i standardowych metod Java. EF wykorzystuje znane w środowisku .NET wyrażenia lambda oraz język LINQ, które należy przyswoić przed rozpoczęciem właściwych prac. W środowisku EF dużo błędów jest wykrywanych na etapie kompilacji, podczas gdy Hibernate częściej zgłasza wyjątki w momencie wykonywania się programu.

W artykule przedstawiono jedynie pewien niewielki obszar badań dotyczący naistotniejszych możliwości dzisiejszych narzędzi ORM, których potencjał konfiguracyjny i funkcjonalny jest ogromny. Zaprojektowane testy wydają się uniwersalne i nadają się do nauki oraz testowania innych rozwiązań mapowania obiektowo-relacyjnego. Utworzone scenariusze mogą być rozwijane przez dodawanie testów innych funkcjonalności. Można do tego zaliczyć np. analizę mechanizmów pamięci podręcznej dotyczących obiektów trwałych, manipulacje transakcjami, obsługę dużych plików binarnych (również w kontekście wydajności) czy integrację z innymi elementami platform programistycznych (np. Spring w Java czy ASP.NET w .NET).

BIBLIOGRAFIA

1. Garcia-Molina H., Ullman J. D., Widom J.: Database Systems: The Complete Book, 2nd Edition. Prentice Hall, 2008.
2. Lerman J.: Programming Entity Framework, 2nd Edition. O'Reilly Media, 2010.
3. Bauer C., King G.: Java Persistence with Hibernate. Manning, 2007.
4. Błoch P., Wojciechowski M.: Analiza porównawcza technologii odwzorowania obiektowo-relacyjnego dla aplikacji Java. XIII Konferencja PLOUG: Systemy informatyczne. Projektowanie, implementowanie, eksploatowanie, Zakopane 2007.

5. Rosiek Z.: Mapowanie obiektowo-relacyjne (ORM) – czy tylko dobra idea? Zeszyty Naukowe Warszawskiej Wyższej Szkoły Informatyki, Zeszyt 4/2010.
6. Oleś W., Małysiak-Mrozek B., Mrozek D.: Porównanie wydajności wybranych narzędzi odwzorowania obiektowo-relacyjnego. Studia Informatica, Vol. 34, No. 2A, Gliwice 2013, s. 123÷144.
7. Smilgin R., Piaskowy A.: Dane testowe. Teoria i praktyka. Helion, Gliwice 2011.
8. Roebuck K.: Object-relational mapping (ORM). Tebbo, 2011.
9. Demystifying Entity Framework Strategies: Model Creation Workflow: http://msdn.microsoft.com/en-us/magazine/hh148150.aspx, 2013.
10. Literals (Entity SQL): http://msdn.microsoft.com/en-us/library/bb399176.aspx.
11. Performance Considerations for Entity Framework 5: http://msdn.microsoft.com/en-us/data/hh949853.aspx, 2013.
12. Hibernate Reference Documentation: http://www.hibernate.org/docs, 2013.
13. Hibernate Types – http://docs.jboss.org/hibernate/orm/4.2/devguide/en-US/html/ch08.html, 2013.
14. MySQL 5.7 Reference Manual: http://dev.mysql.com/doc/refman/5.7/en/index.html, 2013.
15. PostgreSQL 9.2 Documentation: http://www.postgresql.org/docs/9.2/static/reference.html, 2013.

Wpłynęło do Redakcji 29 stycznia 2014 r.

Abstract

The problem of relational databases integration with object oriented applications was discussed in the paper. Two popular object-relational mapping interfaces – Entity Framework, and Hibernate – were analyzed in terms of their functionality and efficiency. In the paper, the database schema migration process with retaining the original form of the application code was studied was studied as well. Both examined interfaces were used in conjunction with SQL Server, MySQL and PostgreSQL servers (Table 1). Two types of query methods were used: object-oriented and storage-independent query language similar to SQL. They were, in the first case, LINQ to Entities (L2E) and Query by Criteria (QBC) for EF and Hibernate respectively, and Entity SQL and HQL in the second one.

The interfaces comparison concerned data manipulation statements, lazy loading mechanisms and introducing object-oriented elements to the database model. Performance tests

were conducted against two sets of records consisting of 100 000 and 500 000 records managed by the SQL Server system. The defined queries were divided into two groups – using changes tracking and read-only records. In case of data modification methods, few strategies were used – short and long sessions and small and big amount of data. The obtained results are presented in figures 1-7 and table 2. Their discussion can be found in chapters 3.5 and 4. The performed tests showed that several elements have influence on performance of database and application cooperation and it is difficult to indicate one appropriate solution. The important finding is that efficiency of managing small amount of data in both tools is satisfying, so they can be used in most business application.

Adresy

Katarzyna HARĘŻLAK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, katarzyna.harezlak@polsl.pl.
Piotr NOWAK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, piotrn22@gmail.com.