STUDIA INFORMATICA 2013
Volume 34 Number 2A (111)

Witold OLEŚ, Bożena MAŁYSIAK-MROZEK, Dariusz MROZEK
Politechnika Śląska, Instytut Informatyki

PORÓWNANIE WYDAJNOŚCI WYBRANYCH NARZĘDZI ODWZOROWANIA OBIEKTOWO-RELACYJNEGO

Streszczenie. W artykule przedstawiono wyniki badań, w których przeprowadzono analizę porównawczą pod względem czasów wykonania oraz generacji kodu języka *SQL* narzędzi odwzorowań obiektowo-relacyjnych na podstawie platformy *.NET*. Wyniki analizy i przeprowadzonych eksperymentów zostały zilustrowane wykresami i odpowiednio zinterpretowane. W artykule opisano także projekt systemu pozwalającego na przeprowadzenie takich badań oraz strukturę bazy danych zawierającej niezbędne dane testowe.

Słowa kluczowe: baza danych, interfejsy obiektowo-relacyjne, platforma *.NET*

PERFORMANCE COMPARISON OF SOME OBJECT-RELATIONAL MAPPING TOOLS

Summary. In this paper we present the results of research, in which we conducted a comparative analysis taking into account execution time and generation of *SQL* code of the object-relational mapping tools based on the *.NET* platform.

The results of the analysis and the experiments are illustrated by the charts and properly interpreted. In this paper we also describe the design of the system supporting the conduct of such research and the structure of the database containing the necessary test data.

Keywords: database, object-relational interfaces, *.NET* platform

1. Wprowadzenie

Na przestrzeni ostatnich lat proces tworzenia aplikacji biznesowych uległ znacznej ewolucji. Zmieniło się podejście do procesu wytwarzania oprogramowania, wprowadzono modularną budowę aplikacji oraz warstwy, w których można wyraźnie wyodrębnić każdy jej element.

Większość systemów znajduje się na serwerach przetwarzających dużą liczbę żądań. Żądania te muszą spełniać kryteria dotyczące czasów odpowiedzi, a co za tym idzie dostęp do danych powinien być zrealizowany w jak najbardziej wydajny sposób. Najczęściej zachodzi również potrzeba adaptacji sposobu dostępu do danych w systemach zarządzania bazami danych do obiektowego języka programowania. Odwzorowanie relacyjnej bazy na system obiektowy nie jest zagadnieniem trywialnym. Istnieje wiele aspektów, które muszą zostać uwzględnione podczas przechodzenia z jednego modelu na drugi. Narzędzia mapujące mają za zadanie odwzorowanie modelu relacyjnego na obiektowy oraz implementację podstawowych operacji przy zachowaniu wszystkich niezbędnych informacji dotyczących powiązań i typów.

Z tych powodów w latach dziewięćdziesiątych ubiegłego stulecia zostało wprowadzone pojęcie niezgodności impedancji, które w skrócie traktowało o różnicach języka zapytań *SQL* w stosunku do proceduralnych języków programowania, w tym obiektowych języków programowania [1]. Niezgodności te dotyczyły różnic pomiędzy takimi cechami, jak np. składnia, typy danych, interpretacja wartości zerowych, dziedziczenie [2].

Niestety pojęcie niezgodności impedancji relacyjnych baz danych nie jest jedynym problemem, który należy mieć na uwadze, mając do czynienia z mechanizmami odwzorowania obiektowo-relacyjnego. Wspomniane mechanizmy muszą spełniać także wymagania dotyczące transakcji i ich poziomów izolacji, uwzględnienia sytuacji powodujących wyjątki, jak np. naruszenie więzów integralności, zarządzanie stanem, serializacja, deserializacja obiektów oraz wydajne przechowywanie danych w pamięci operacyjnej czy fizycznej komputera.

Komercyjni dostawcy rozwiązań oraz dystrybutorzy darmowego oprogramowania podchodzą do tego zagadnienia, korzystając z różnorodnych metod, mniej lub bardziej wydajnych.

Dla autorów artykułu najistotniejszym aspektem było zbadanie wydajności pod względem czasu odpowiedzi oraz sposobu generacji kodu języka *SQL* formułowanych zapytań przez wybrane narzędzia odwzorowujące działające na platformie *.NET*.

2. Zalety i wady stosowania technologii *ORM*

Myśląc o technologii *ORM* (ang. *object-relational mapping*) jak o pewnego rodzaju pomoście już nie tylko pomiędzy kolejnymi warstwami aplikacji, lecz także między aplikacją a bazą danych, można dojść do wniosku, że skoro istnieje interfejs, to powinna nastąpić znaczna redukcja kodu. Jak najbardziej można się zgodzić z tak postawionym sformułowaniem. Zastosowanie technologii *ORM*, gwarantuje implementację podstawowych operacji, takich jak: dodawanie, odczyt, modyfikacja, usuwanie nazywanymi operacjami typu *CRUD* (ang. *create*, *read*, *update*, *delete*). Bez wątpienia fakt ten wpływa na przyspieszenie

procedury tworzenia kodu przez programistę, zwalniając go tym samym z obowiązku utworzenia procedur składowanych odpowiedzialnych za wspomniane operacje. W tym przypadku zastosowanie gotowego rozwiązania nie wyklucza możliwości użycia niestandardowej procedury dla konkretnych warunków, przy czym istnieje także możliwość implementacji odpowiedniej funkcji z poziomu samej aplikacji. Wszystkie te cechy gwarantują elastyczność i dostosowanie rozwiązania do ustalonych wymagań.

Podczas procesu projektowania, a później tworzenia relacyjnej bazy danych można kierować się różnego rodzaju przesłankami, które mają wpływ na końcową postać schematu bazy danych. Czasem będzie to minimalizacja czasu dostępu do danych, innym razem oszczędność pamięci fizycznej lub w skrajnych przypadkach będą to błędy na poziomie projektowym. Jednak bez względu na przyczynę taki model rzadko w pełni odzwierciedla logikę, jaką będzie musiała realizować aplikacja. W takich przypadkach zastosowanie technologii *ORM* pozwoli utworzyć taki model biznesowy, jaki w rzeczywistości jest wymagany, co także usprawni dalszy proces rozwoju systemu oraz traktowania go jako zgodnego z założeniami. Ta cecha jest szczególnie istotna także z punktu widzenia procesu wytwarzania oprogramowania przy wzięciu pod uwagę faktu, że w pracy nad projektem zaangażowani są zarówno projektanci, mający na celu utworzenie jak najbardziej wydajnej struktury bazy danych, jak i programiści, którzy skupiają się na samej logice systemu.

Do niedawna najbardziej popularnymi sposobami dostępu do danych z poziomu programu aplikacji były zanurzenie poleceń języka *SQL* w kodzie samej aplikacji lub zapisanie tych poleceń w postaci procedur składowanych po stronie serwera bazy danych i ich wywołanie z programu aplikacji. Jednakże w przypadku zmian w strukturach tabel w bazie danych należało uaktualnić wymagane procedury związane z daną tabelą lub kod aplikacji. Natomiast jeżeli dany obiekt został wcześniej odwzorowany, wówczas w przypadku zmian w strukturze tabeli należy tylko uaktualnić odpowiadający mu model, bardzo często za pomocą wygodnego i szybkiego kreatora. Rozwiązanie takie gwarantuje oszczędność czasu z zachowaniem spójności z istniejącym schematem bazy danych.

Można wyobrazić sobie sytuację, w której aplikacja jest przeznaczona dla określonego systemu zarządzania bazą danych. Wówczas dane mogą być pozyskiwane na podstawie procedur składowanych, jeśli logikę przetwarzania danych implementujemy po stronie serwera. Jeżeli w tej sytuacji zajdzie potrzeba zmiany dostawcy bazy danych (o ile przeniesienie tabel i rekordów nie jest operacją aż tak kłopotliwą), to przepisanie procedur jest kosztowne i uciążliwe. Jeżeli logika byłaby realizowana w kodzie programu przy użyciu systemu *ORM*, zmiana dostawcy danych praktycznie byłaby operacją niewymagającą żadnych wielkoskalowych modyfikacji. Należałoby w tym wypadku zmodyfikować plik konfiguracyjny, dostosowując go do nowych wymagań.

Podsumowując, używając interfejsów obiektowo-relacyjnych w aplikacji można:

- zmniejszyć objętość kodu, jaki musi zostać napisany ręcznie, przez implementację podstawowych operacji na bazie danych typu: utwórz, odczytaj, aktualizuj oraz usuń (przykłady zamieszczono w podrozdziale 5.2);
- uprościć kod aplikacji przez hermetyzację warstwy dostępu do danych od pozostałych modułów. Dzięki temu w systemie może powstać jasny podział odpowiedzialności dla każdej z warstw;
- zmniejszyć czas oraz koszty dzięki wykorzystaniu udostępnionych interfejsów i infrastruktury przez poszczególne mechanizmy odwzorowania obiektowo-relacyjnego (przykłady zamieszczono w podrozdziale 5.1);
- skupić się na wymaganiach funkcjonalnych systemu, wówczas zespół projektowy nie musi definiować całkowicie od początku warstwy dostępu do danych, lecz może skupić się na logice i założeniach systemu, nad którym aktualnie pracuje;
- wprowadzić modyfikacje dotyczące struktury bazy bez konieczności zmian w procedurach składowanych, przy założeniu że cała odpowiedzialność za realizację logiki spoczywa po stronie systemu. Wystarczy wówczas przedefiniować odpowiednie metody zamiast zmian kodu procedur składowanych oraz kodu w aplikacji klienta obsługującego ich wywołania;
- uniezależnić się od konkretnego dostawcy bazy danych, dzięki mechanizmowi wykrywania silnika bazy danych, który należy obsłużyć, i generowania dla niego odpowiedniego kodu języka SQL (przykłady zamieszczono w podrozdziale 5.2). Należy także pamiętać, iż rozpatrujemy system, w którym cała odpowiedzialność za realizację logiki biznesowej spoczywa po stronie kodu aplikacji.

Jednak każde podejście do rozwiązania problemu niesie ze sobą nie tylko pewne korzyści lecz także negatywne skutki. Oprócz faktu, że aby używać mechanizmów odwzorowujących należy utworzyć odpowiednie pliki odwzorowujące oraz zadbać o ich utrzymanie, trzeba mieć także świadomość, że można stracić w niektórych sytuacjach na szybkości działania danego programu. Wspomniany spadek wydajności jest spowodowany wprowadzeniem dodatkowej warstwy, za jaką można uznać *ORM*. Warto także wiedzieć, że zapytania, jakie są generowane do bazy, nie zawsze zbudowane są w najlepszy z możliwych sposobów. W tej sytuacji można stracić na prędkości na rzecz ogólności, bo przecież kwerendy są budowane zgodnie z pewnymi algorytmami, które mimo że w większości wypadków działają dobrze – czasem mogą zadziałać w mało oczekiwany sposób [2].

3. Narzędzia *ORM* poddane analizie

Obecnie na rynku istnieje sporo narzędzi typu *ORM* (ang. *object-relational mapping*) mających za zadanie uprościć translację modelu bazy danych na model obiektowy oraz implementować mechanizmy związane z podstawowymi operacjami wykonywanymi w bazie danych.

Autorzy pracy do przeprowadzenia analizy porównawczej pod względem czasów odpowiedzi na wykonywane operacje w bazie danych oraz generacji treści formułowanych zapytań *SQL* wybrali następujące narzędzia: *ADO.NET*, *LINQ to SQL*, *Entity Framework*, *NHibernate* i *Telerik Open Access ORM*. Przy wyborze każdego z narzędzi brano pod uwagę przede wszystkim dużą popularność wśród firm zajmujących się tworzeniem oprogramowania (na tej podstawie powstała duża liczba systemów i portali) oraz istnienie dość obszernej dokumentacji na temat każdego z nich.

ADO.NET jest natywną implementacją dostępu do danych wbudowaną w platformę *.NET*, która wykorzystuje kontenery zwane *DataSet*. Nie może być traktowana jako pełnowartościowe narzędzie *ORM*, choć należy pamiętać, że było to rozwiązanie wykorzystywane w wielu systemach jeszcze w czasie ostatniej dekady. Jednak w dobie szybko postępującej ewolucji oprogramowania i w czasie dynamicznej zmiany rynku jest ono wypierane przez nowsze, łatwiej skalowalne i wydajniejsze rozwiązania [7].

LINQ to SQL [2, 1, 6] i *Entity Framework* [8] także są narzędziami wbudowanymi w platformę *.NET*, jednak w tym przypadku udostępniona funkcjonalność pozwala traktować je jako pełnowartościowe mechanizmy typu *ORM*. Choć oba narzędzia odwzorowujące pozwalają na szybką implementację dostępu do danych, różnią się nieco od siebie. *Entity Framework* wspiera prawie wszystkie systemy zarządzania bazami danych, natomiast *LINQ to SQL* jest zalecany tylko dla baz danych firmy Microsoft i nie wspiera odwzorowania typu wiele-do-wielu. *Entity Framework* jest przeznaczony dla większych systemów, najczęściej wielowarstwowych, umożliwia przy tym korzystanie z różnych podejść w tworzeniu logiki systemu.

NHibernate jest produktem dostępnym na warunkach wolnej licencji (ang. *open source*), a jego pierwowzorem była biblioteka *Hibernate*, przeznaczona do pracy z platformą Java. To narzędzie dostarcza bardzo rozbudowanej infrastruktury pozwalającej na dostrojenie konfiguracji do określonych warunków; pozwala to na obsługę większości popularnych systemów zarządzania bazami danych [9, 10].

Telerik Open Access ORM [11] jest jednym z nowych rozwiązań dostępnych na rynku spełniającym wymogi, jakie są stawiane przed nowoczesnymi narzędziami *ORM*. Jest produktem komercyjnym o funkcjonalności bardzo zbliżonej do *Entity Framework* oraz *NHibernate*. Podobnie jak *Entity Framework* oraz *LINQ to SQL* ma kreator służący do

określenia odwzorowania modelu bazy danych na model obiektowy, przy czym nie ogranicza się tylko do tego sposobu, udostępniając możliwość wykorzystania podejść typu *najpierw kod* (ang. *code first*) oraz *najpierw model* (ang. *model first*).

4. Projekt systemu

Dla przeprowadzenia testów wydajnościowych wybranych narzędzi *ORM* oraz analizy generowanego kodu języka *SQL* formułowanych zapytań został zaprojektowany i zaimplementowany odpowiedni system. Utworzono także biblioteki, które udostępniają funkcjonalności pozwalające na przeprowadzenie badań oraz zapis wyników do odpowiednich plików. Za pomocą zaimplementowanego systemu możliwe jest badanie czasów odpowiedzi w zależności od warunków początkowych, jakimi może być np. liczba rekordów czy liczba iteracji w ramach procesu [3, 4, 5].

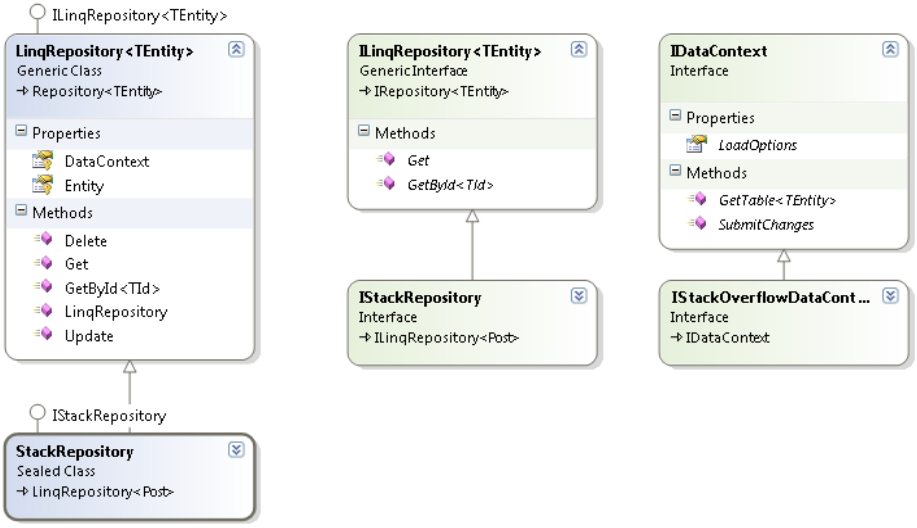
System jest zorganizowany w podsystemy, gdzie każdy z nich ma interfejs umożliwiający wykonanie podstawowych operacji na bazie danych dla konkretnego narzędzia *ORM*. Operacjami tymi mogą być projekcja, dodawanie, usuwanie, modyfikacja oraz pewne rozszerzenia generujące niestandardowe zapytania.

4.1. Architektura systemu

Strukturalnie projekt systemu składa się z bibliotek, a każda z nich jest odpowiedzialna za obsługę jednego z testowanych narzędzi *ORM*. Dodatkowo projekt zawiera podsystem dostarczający implementację metod wspólnych dla całej aplikacji. Do przeprowadzania testów została wykorzystana klasa *Test* z podsystemu o tej samej nazwie, a do importu danych z plików *XML* – biblioteka *DataImport*.

Dostęp do narzędzi *LINQ to SQL*, *Entity Framework* i *NHibernate* został zbudowany na podstawie wzorca repozytorium, który dostarcza infrastrukturę pozwalającą na implementację podstawowych operacji wykonywanych na rzecz bazy danych. Klasa *LinqRepository<TEntity>* jest klasą generyczną dziedziczącą po klasie *Repository<TEntity>* i implementującą interfejs *ILinqRepository<TEntity>*. Takie podejście umożliwia utworzenie wyspecjalizowanego repozytorium realizującego założenia logiki. W tym projekcie powstało repozytorium o nazwie *StackRepository*, które zostało utworzone na podstawie konkretnego typu encji. Dzięki takiemu modelowi programista jest w stanie w szybki sposób utworzyć szkielet aplikacji odpowiedzialny za dostęp do danych. Dostawcy rozwiązań dostarczają pewne zachowania pozwalające na śledzenie zmian i zarządzanie zasobami w ramach sesji. W tym celu *LINQ to SQL* korzysta z typu *System.Data.Linq.DataContext*, *Entity Framework* z *Sys-*

tem.Data.Objects.ObjectContext, natomiast *NHibernate* z obiektu typu *ISession*. Aby w ramach sesji wyizolować metody, które mają zostać udostępnione klientowi, klasa repozytorium zawiera obiekt interfejsu udostępniającego wskazane operacje. W tym przypadku można zauważyć, że są to metody *GetTable<TEntity>* i *SumbitChanges* oraz właściwość *LoadOptions*, która pozwala odpowiednio dostroić jednostkę. Taki scenariusz dobrze komponuje się z wygenerowanymi klasami, które dla każdego nowo powstałego schematu mają obowiązek dziedziczyć po swoich typach kontekstu. Pozwala to uniezależnić przedstawiony na rys. 1 typ *LinqRepository<TEntity>* od konkretnej implementacji jednostki kontekstu. Przedstawiona na rys. 1 klasa główna repozytorium udostępnia operacje typu usuwanie, modyfikacje, wyszukiwanie rekordu na podstawie *id* czy też zwracanie interfejsu *IQueryable<T>*, w celu umożliwienia budowania zapytań klientowi. Taki model jest otwarty na rozszerzenia, w których możliwe jest zastosowanie metod w zależności od dostarczonego typu [7].



Rys. 1. Fragment schematu klas z biblioteki *ORM.LinqToSQL*
Fig. 1. The fragment of class scheme from *ORM.LinqToSQL* library

4.2. Schemat bazy danych

Testy porównawcze czasów wykonania zapytań, a także generacji kodu przeprowadzono na produkcyjnej bazie danych popularnego serwisu programistycznego o nazwie *StackOverflow*, który udostępnia dane w postaci pliku o strukturze dokumentu *XML*. Udostępnione dane przed operacją importu do bazy danych mają rozmiar rzędu kilku gigabajtów, zatem po wykonaniu tej operacji poszczególne tabele mogą zawierać dużą liczbę wierszy, dochodzącą do kilku milionów.

Schemat bazy danych (rys. 2) jest powiązany z aplikacją mającą za zadanie udostępniać mechanizmy pozwalające na wymianę wiadomości pomiędzy użytkownikami w sieci Internet. W tym celu do ich gromadzenia została utworzona tabela *Posts*, zawierająca kolumny opisujące konkretną wiadomość. Kluczem głównym jest pole identyfikatora *Id* – tak

samo jak w przypadku pozostałych tabel. *Posts* jest powiązana relacjami typu jeden-do-wielu z obiektami *Comments* oraz *Votes*. *Comments* ma za zadanie przechowywać dane dotyczące komentarzy użytkownika na temat wiadomości, *Votes* zaś przechowuje liczbę głosów konkretnego typu dla wiadomości. Tabela *Users* odpowiada za przechowywanie danych na temat użytkowników systemu i jest powiązana relacjami typu jeden-do-wielu z opisywaną wcześniej tabelą *Posts* oraz relacją typu jeden-do-jednego z tabelą *Badges*, która dostarcza informacji na temat rangi użytkownika w systemie. Z uwagi na fakt, iż taki schemat bazy danych w przejrzysty sposób opisuje zadania logiki systemu, projekt aplikacji został utworzony według podejścia *najpierw baza*.

Rys. 2. Schemat bazy danych
Fig. 2. The database schema

5. Przeprowadzone badania i wnioski

Przeprowadzane eksperymenty zostały podzielone na dwie części: pierwsza dotyczyła badania czasów wykonania dla podstawowych operacji na danych, druga zaś – porównania kodów języka *SQL* wygenerowanych przez poszczególne narzędzia *ORM*. Program testujący został zaimplementowany w języku *C#*. Testy przeprowadzono w trybie *Release* z włączoną optymalizacją kompilatora. Maszyna, na której przeprowadzono badania, była wyposażona w procesor *Intel Core i7* z pamięcią operacyjną 4 GB oraz system operacyjny *Microsoft Windows 7*.

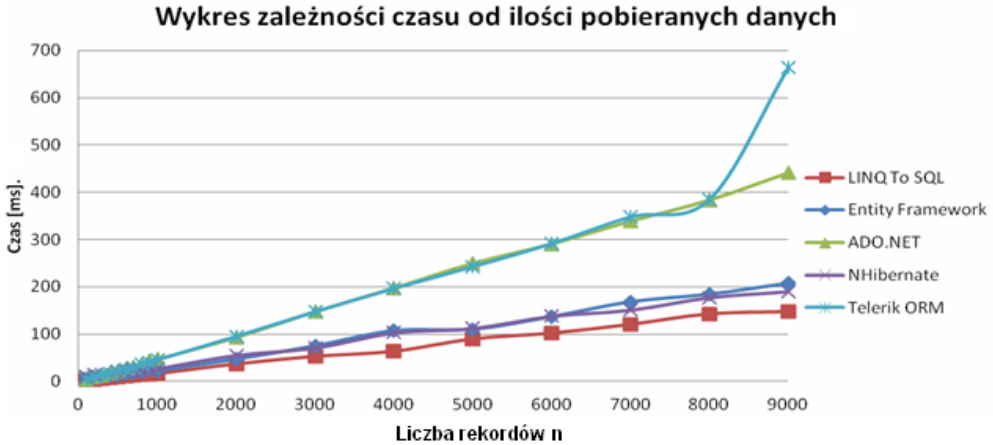
5.1. Eksperymenty dotyczące wydajności przetwarzania danych

Pierwsza grupa eksperymentów dotyczyła porównania czasów wykonania dla podstawowych operacji, takich jak: wstawianie, modyfikacja oraz usuwanie dla różnych liczby wierszy

i różnej liczby iteracji. Do pomiaru czasów wykonania została wykorzystana klasa *StopWatch* znajdująca się w przestrzeni nazw *System.Diagnostics*. Pozwala ona na pomiary czasów wykonania kodu z dokładnością co do milisekundy i liczby taktów procesora.

5.1.1. Pobieranie danych

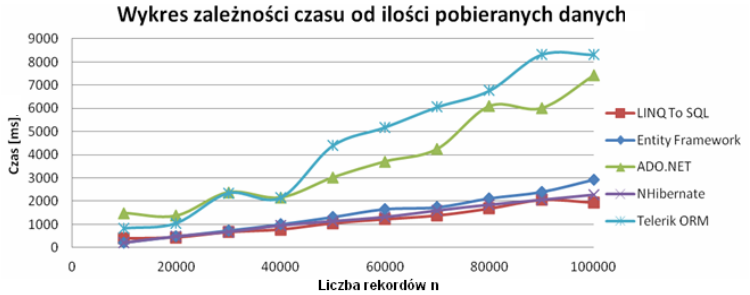
Pomiary dotyczyły czasu mierzonego od momentu wysłania zapytania do bazy danych do momentu materializacji wyników w pamięci operacyjnej komputera. W przypadku *ADO.NET* materializacja odbywała się do typu *DataTable*, w pozostałych przypadkach zaś do typu *List<T>*. Wymienione typy są natywnymi typami dla testowanych narzędzi odwzorowujących, co pozwala uniknąć niedokładności pomiarów w przypadku ewentualnych konwersji. Przedstawione wyniki są uśrednioną wartością z kilkunastu cykli pomiarowych. Pobieranie danych odbywało się w ramach jednej sesji dla każdego programu odwzorowującego i dotyczyło tabeli *Posts*. Wszystkie narzędzia zostały skonfigurowane w trybie opóźnionego ładowania (ang. *lazy loading*), co zagwarantowało, iż dane z tabel referencyjnych nie zostały pobrane do pamięci operacyjnej. Na rys. 3 przedstawiono wykres zależności czasu dla małej ilości danych z zakresu 10 – 9000 rekordów. Jak można zauważyć, dla liczby rekordów nieprzekraczającej wartości 1000 czasy dla każdego narzędzia nie przekraczały wartości 50 ms. Powyżej tej wartości można zauważyć, że przyrost czasów ma charakter liniowy i zależy od ilości danych do przetworzenia. W tych warunkach najlepsze osiągi uzyskują *LINQ to SQL*, *Entity Framework* oraz *NHibernate*, gdzie czasy dla największej wartości argumentu *n* są na poziomie 200 ms. Z pewnością *ADO.NET* mógłby osiągnąć lepszy czas, gdyby zastosowano manualny odczyt za pomocą mechanizmów odczytywania danych „tylko w przód” (ang. *DataReader*) zamiast tabeli danych (ang. *DataTable*).



Rys. 3. Zależność czasu od ilości pobieranych danych dla zakresu 10-9 000 rekordów
Fig. 3. Dependency of the time on the amount of selected data for the range of 10-9 000 records

W przypadku pobierania większej ilości danych wyniki są zgodne z wykresem przedstawionym na rys. 4. Najlepsze wyniki zostały uzyskane także przez *LINQ to SQL*, *Entity Framework* oraz *NHibernate*, natomiast gorsze czasy zostały uzyskane przez *ADO.NET* oraz

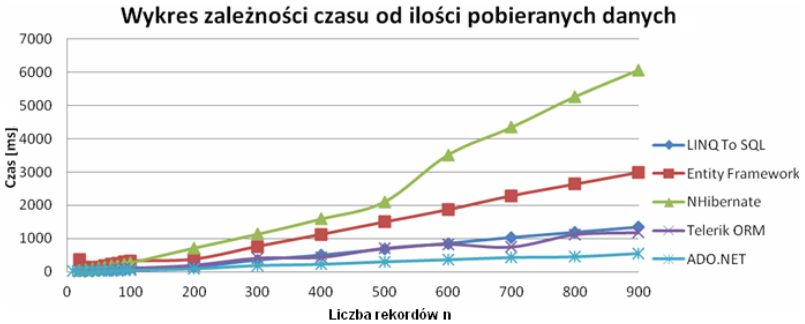
Telerik Open Access ORM. Dla wartości n = 100000 *ADO.NET* potrzebowało powyżej 7 s na zapis wyników do swojej struktury, *Telerik ORM* z kolei potrzebował niecałych 9 s.



Rys. 4. Zależność czasu od ilości pobieranych danych dla zakresu 10 000-100 000 rekordów
Fig. 4. Dependency of the time on the amount of selected data for the range of 10 000-100 000 records

Na wykresach czasowych *ADO.NET* oraz *Telerik ORM* można zauważyć, że nie są one funkcją stale rosnącą. Na te niedoskonałości pomiarów mogą wpływać np. system zarządzania zasobami platformy *.NET* czy też inne procesy systemu operacyjnego. Chociaż *Entity Framework* ma jedną warstwę więcej w wewnętrznej implementacji w porównaniu z *LINQ to SQL*, to jego wyniki są na podobnym poziomie. W tym teście bardzo dobrze wypada także *NHibernate*, którego klasy modelu nie zostały wygenerowane w sposób automatyczny jak w przypadku narzędzi firmy *Microsoft* czy *Telerik*.

Drugą metodą pomiarów było pobieranie pojedynczych wartości na podstawie identyfikatora tabeli. Tabela nie miała włączonej opcji, automatycznego generowania klucza głównego, dlatego charakter zapytań był zróżnicowany z uwagi na fakt, iż nie wszystkie zapytania zwracały rekord. Testy zostały przeprowadzone dla wartości z zakresów 10 – 100 rekordów z krokiem równym 10, 100 – 900 rekordów z krokiem równym 100 (rys. 5), oraz 1 000 – 10 000 rekordów z krokiem równym 1 000 (rys.6).

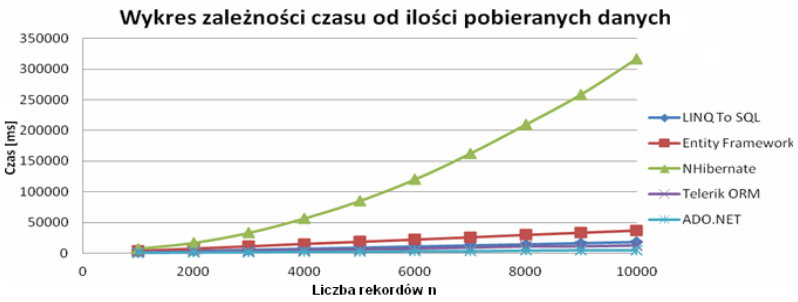


Rys. 5. Zależność czasu od ilości pobieranych danych dla zakresu 10-900 rekordów
Fig. 5. Dependency of the time on the amount of selected data for the range of 10-900 records

W wykresie przedstawionym na rys. 6 *NHibernate* uzyskuje najgorsze czasy. Jest to spowodowane faktem, iż wykonuje on bardzo dużo operacji poza kulisami, jak np. utrzymywanie stanu sesji. Możliwa jest zmiana tego zachowania, lecz na potrzeby testowe brane są pod uwagę najbardziej typowe ustawienia narzędzi. Z kolei dużo lepiej przed-stawiają się osiągi dostawców *ADO.NET* i *Telerik ORM*. Powodem takiego zachowania w przypadku *ADO.NET*

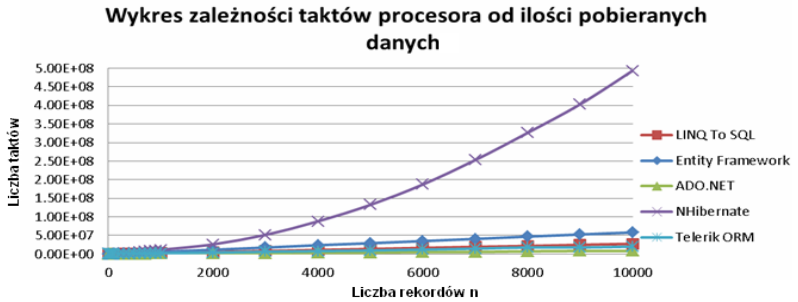
jest materializacja danych do bardziej optymalnej struktury niż w poprzednim przypadku. Ciekawym rozwiązaniem jest realizacja tego zapytania przez *Telerik ORM*, który w pierwszym etapie przygotowuje procedurę, a następnie wysyła tylko wymagane parametry. Bez wątpienia takie podejście skraca czas wysłania strumienia do bazy.

Wykres przedstawiony na rys. 6 obrazuje czasy odpowiedzi dla zakresu danych 1 000 – 10 000 rekordów. W tym przypadku czas dla *NHibernate* jest nieakceptowalny w przypadku typowych aplikacji klienckich, dlatego dla takich zapytań pozostaje dostrojenie konfiguracji w inny sposób niż domyślny. Wykresy w tym eksperymencie dla pozostałych narzędzi mają charakter liniowy, co jest zgodne z przewidywaniami.



Rys. 6. Zależność czasu od ilości pobieranych danych dla zakresu 1 000-10 000 rekordów
Fig. 6. Dependency of the time on the amount of selected data for the range of 1 000-10 000 records

Ostatnim pomiarem w ramach operacji pobierania rekordów jest wykres zależności taktów procesora od liczby pobieranych rekordów (rys. 7). Przedstawiony wykres odzwierciedla wartości czasowe i pokazuje, że najwięcej operacji musiał wykonać *NHibernate*, który dla wartości liczby rekordów n = 10 000 potrzebował aż około 50 milionów taktów. Pozostałe programy nie przekraczały wartości 10 milionów. W tym teście *NHibernate* wypada około 5 razy gorzej na tle innych aplikacji.

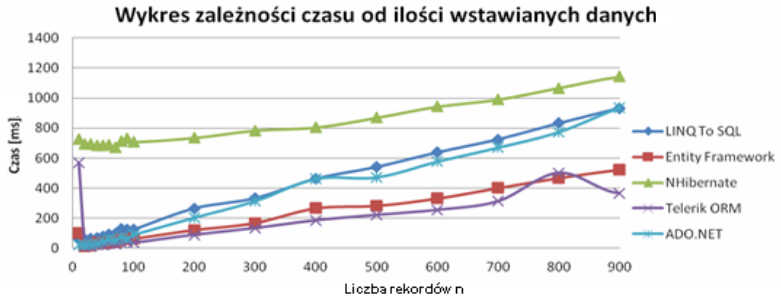


Rys. 7. Zależność liczby taktów procesora od ilości pobieranych danych dla zakresu 10-10 000 rekordów
Fig. 7. Dependency of number CPU cycles on the amount of selected data for the range of 10-10 000 records

5.1.2. Wstawianie danych

Dla procesu wstawiania danych zostały wykonane testy porównawcze dla trzech zakresów danych. Pierwszy dotyczył zakresu 10 – 100 z krokiem równym 10, drugi zakresu 100 – 900

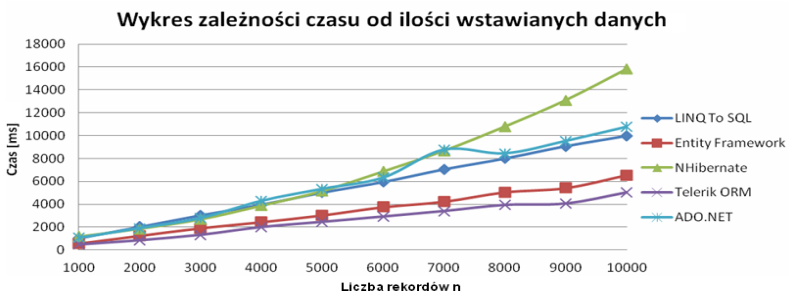
z krokiem równym 100, trzeci zaś zakresu 1 000 – 10 000 z krokiem równym 1 000. Eksperyment przeprowadzono kilkukrotnie, przedstawione wyniki są wartościami średnimi z wszystkich pomiarów. Wstawianie odbywało się według zasady: zakres na jedną transakcję. Taka forma wstawiania pozwoliła na przyspieszenie działania niektórych systemów *ORM*. Na rys. 8 przedstawiono wykres czasu w zależności od ilości wstawianych danych dla zakresu 10 – 900 rekordów. W tym przypadku najlepsze rezultaty uzyskał *Telerik ORM*, gdzie czas nie przekraczał wartości 500 ms, na podobnym poziomie wyniki kształtowały się w przypadku narzędzia *Entity Framework*, a *LINQ to SQL* i *ADO.NET* uzyskały nieco gorsze rezultaty, które dla wartości wstawiania n = 900 rekordów nie przekroczyły 1 s.



Rys. 8. Zależność czasu od ilości wstawianych danych dla zakresu 10-900 rekordów
Fig. 8. Dependency of the time on the amount of inserted data for the range of 10-900 records

W przedstawionym teście najgorsze czasy uzyskał *NHibernate*, gdzie dla wartości n = 900 rekordów czas wynosił około 1,2 s. W tym przypadku, jak zostało wcześniej napisane, wstawianie rekordów odbywało się w jednym zapytaniu na konkretną serię danych, co pozwoliło na poprawę czasu dla programu *NHibernate*, który bez tej modyfikacji uzyskiwał czasy kilkukrotnie gorsze. Podczas tego testu można by się spodziewać, że zdefiniowana procedura dla dostawcy *ADO.NET* będzie szybciej wykonywać operację wstawiania. Jednak nie dzieje się tak z powodu wygenerowania automatycznie dodatkowego kodu, który po wstawianiu dokonuje projekcji, a następnie przypisuje wstawiany rekord do odpowiedniego kontenera. Takie zachowanie może nawet do dwóch razy pogorszyć wydajność. Zatem jeśli programiście zależy na szybkości działania, powinien zastanowić się nad zastosowaniem procedury składowanej. Warto także zauważyć, że *Telerik* na pierwsze pobieranie potrzebuje więcej czasu w stosunku do następnych operacji (rys. 8). Sytuacja ta jest spowodowana inicjalizacją wymaganych struktur, a także kompilowaniem zapytań.

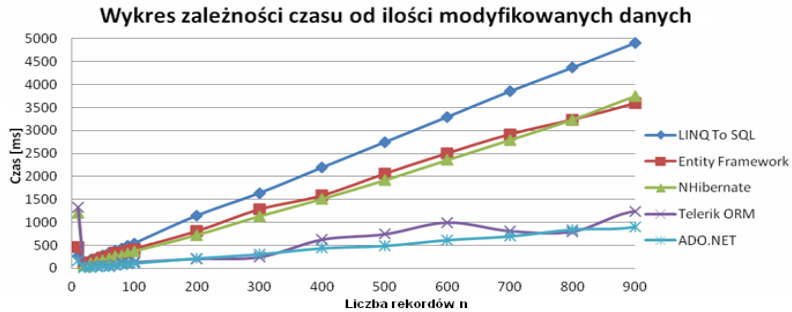
Podobnie jest też w przypadku pozostałych narzędzi, jednak różnica ta nie jest w tym wypadku tak znacząca. Na rys. 9 przedstawiono wyniki dla tej samej operacji dla zakresu 1 000 – 10 000 rekordów. Analizując te wartości, można zauważyć podobny trend jak dla mniejszej ilości wstawianych danych. Dla wartości n = 10 000 rekordów najlepszy czas wciąż uzyskuje *Telerik ORM*, *NHibernate* zaś jest około trzech razy wolniejszy.



Rys. 9. Zależność czasu od ilości wstawianych danych dla zakresu 1 000-10 000 rekordów
Fig. 9. Dependency of the time on the amount of inserted data for the range of 1 000-10 000 records

5.1.3. Modyfikacja danych

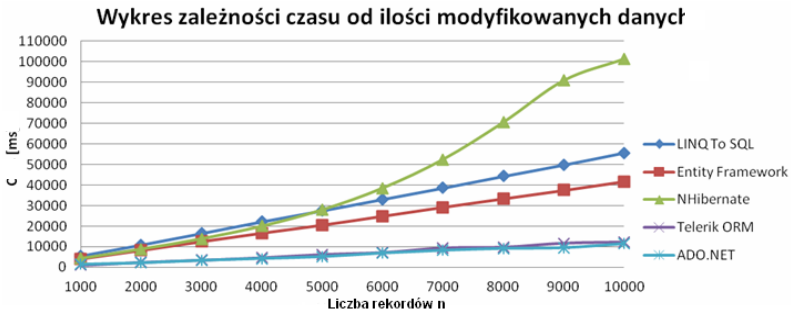
Dla operacji modyfikacji danych testy porównawcze były przeprowadzane kilkukrotnie, dlatego przedstawione wyniki są wartościami średnimi z wykonanych pomiarów. Operacja modyfikowania danych w przypadku niektórych narzędzi *ORM* wymaga pobrania określonego rekordu. Zmiana tego zachowania jest możliwa, jednak w tym przypadku należałoby zdefiniować kod języka *SQL* manualnie. Najbardziej interesujące z punktu widzenia prowadzonych badań są wyniki uzyskane bez ingerencji w generowany kod.



Rys. 10. Zależność czasu od ilości modyfikowanych danych dla zakresu 10-900 rekordów
Fig. 10. Dependency of the time on the amount of modified data for the range of 10-900 records

Na rys. 10 przedstawiono zależność czasu od liczby rekordów dla operacji modyfikowania danych. Także tutaj można zaobserwować, że program *Telerik ORM* potrzebuje nieco więcej czasu podczas formułowania pierwszego zapytania. Po tej operacji pozostałe wyniki są na podobnym poziomie jak w przypadku *ADO.NET*, gdzie operacja sprowadzała się do ustalenia połączenia i odesłania zapytania do bazy danych, bez obowiązku utrzymywania stanu sesji i wykonywania dodatkowych operacji. Wymienione narzędzia odwzorowujące dla wartości n = 900 rekordów potrzebowały odpowiednio 752 ms oraz 869 ms. Dla przedstawionej wartości argumentów czasy dla *NHibernate* i *Entity Framework* były niemal identyczne, *LINQ to SQL* zaś był w tym przypadku najwolniejszy dla liczby rekordów n = 900 potrzebował aż 5 s, aby zmodyfikować dane. Na takie zachowanie miała wpływ treść formułowanego kodu *SQL*, gdzie LINQ zawsze generował zapytanie, w którym do serwera były wysyłane wszystkie wartości kolumn i nie miało znaczenia, czy były one modyfikowane.

Na rys. 11 można zaobserwować wyniki dla tej samej operacji, lecz dla zakresu danych 1 000 – 10 000 rekordów. W tym przypadku wyniki są nieco inne niż te oczekiwane – przy wzięciu pod uwagę rys. 11. Od wartości n = 5 000 *NHibernate* zaczyna wykonywać zapytania wolniej od pozostałych narzędzi, natomiast *LINQ to SQL* i *Entity Framework* utrzymują swój charakter liniowego przyrostu. Bardzo dobre osiągi uzyskują pozostałe systemy, np. *Telerik ORM*, który mimo posiadania większej liczby warstw w swojej implementacji utrzymuje się na podobnym poziomie jak rozwiązanie dostawcy *ADO.NET*.

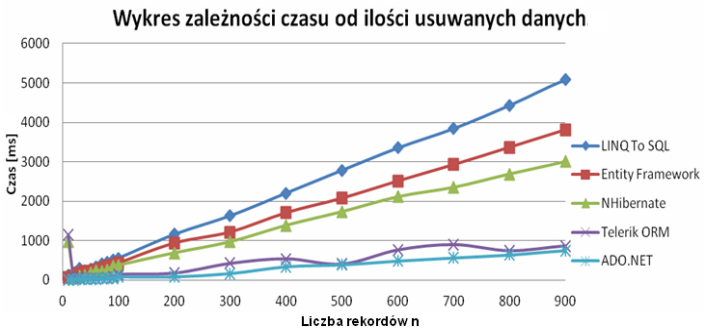


Rys. 11. Zależność czasu od ilości modyfikowanych danych dla zakresu 1 000-10 000 rekordów
Fig. 11. Dependency of the time on the amount of modified data for the range of 1 000-10 000 records

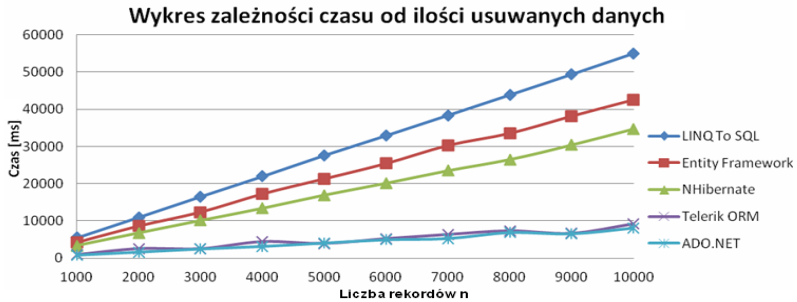
5.1.4. Usuwanie danych

Operacja usuwania danych, choć jest coraz rzadziej spotykana w dobie stale powiększającej się pojemności dyskowej, również została poddana testom. Testy dla tej operacji zostały przeprowadzone dla trzech zakresów, podobnie jak podczas operacji modyfikacji i wstawiania. Dla operacji usuwania danych z bazy danych dla zakresu 10 – 900 rekordów najlepsze osiągi oprócz natywnego dostawcy *ADO.NET* zostały uzyskane przez *Telerik ORM*, gdzie poza dłużej trwającym pierwszym zapytaniem, wynoszącym nieco ponad 1 s, uzyskiwał on najlepsze czasy odpowiedzi. *NHibernate* dużo lepiej wypada w tym teście niż we wcześniejszych (rys. 12). Na takie zachowanie wpłynęło zastosowanie bezstanowej sesji, która bardzo dobrze sprawdza się w operacjach o masowym charakterze. *LINQ to SQL* na usunięcie n = 200 rekordów potrzebował około 1,1 s, podobnie jak *Entity Framework*, gdzie wymagany czas był na poziomie 937 ms. Dla ostatniej testowanej wartości argumentu *n* w ramach tego zakresu, wynoszącej 900, *LINQ to SQL* potrzebował około 5 s, *Entity Framework* około 4 s, zaś *NHibernate* około 3 s. Także w tym przypadku wykresy mają charakter zbliżony do funkcji o przyroście liniowym, zależnym od ilości danych *n*.

Analizując wykres przedstawiony na rys. 13, można wyciągnąć wniosek, że operacja usuwania jest najbardziej złożona i czasochłonna – zajmuje wielokrotnie więcej czasu niż pozostałe operacje przedstawione wcześniej w tym punkcie. Dla największej wartości z przedstawionego zakresu, n = 10 000 rekordów, wymagane czasy dochodzą do wartości 55 s. Dlatego w tym przypadku, gdy architektura wymaga stosowania masowych operacji usuwania, warto rozważyć kwestię zastosowania procedur składowanych.



Rys. 12. Zależność czasu od ilości usuwanych danych dla zakresu 10-900 rekordów
Fig. 12. Dependency of the time on the amount of deleted data for the range of 10-900 records



Rys. 13. Zależność czasu od ilości usuwanych danych dla zakresu 1 000-10 000 rekordów
Fig. 13. Dependency of the time on the amount of deleted data for the range of 1 000-10 000 records

5.2. Porównanie złożoności zapytań

Druga grupa eksperymentów polegała na porównaniu wygenerowanego kodu języka *SQL* przez poszczególne narzędzia odwzorowujące dla różnych typów zapytań.

```
public void Update()
{
        var i1 = this.linq.Get().Where(x => x.Id == 4).ToList().First();
        i1.User.DisplayName = "Test display name 1";
        this.linq.Commit();

        var i2 = this.entity.Get().Where(x => x.Id == 4).ToList().First();
        i2.User.DisplayName = "Test display name 2";
        this.entity.Commit();

        var i3 = this.hibernate.Get().Where(x => x.Id == 4).ToList().First();
        i3.User.DisplayName = "Test display name 3";
        this.hibernate.BeginTransaction();
        this.hibernate.CommitTransaction();

        var i4 = this.telerik.Posts.Where(x => x.Id == 4).ToList().First();
        i4.User.DisplayName = "Test display name 4";
        this.telerik.SaveChanges();
}
```

Rys. 14. Wywołanie dla operacji modyfikowania danych
Fig. 14. A call of function for data modification

Pierwszym zapytaniem, dla którego nastąpiło porównanie wygenerowanego kodu *SQL* przez poszczególne narzędzia odwzorowujące, dotyczyło modyfikowania danych. W pierwszej fazie następowało znalezienie odpowiedniego wiersza z tabeli *Posts*. Następnie po uzyskaniu wskazanego wiersza jedna z kolumn tabeli referencyjnej *Users* była modyfikowana odpowiednią wartością. Po odpowiedniej modyfikacji następowało potwierdzenie transakcji.

Modyfikowaną kolumną była kolumna określająca wyświetlaną nazwę użytkownika (*Display Name*). Kod dla opisanej sytuacji przedstawiono na rys. 14.

LINQ to SQL	`Exec sp_executeSQL N'UPDATE [dbo].[Users] SET [DisplayName] = @p12 WHERE ([Id] = @p0) AND ([Reputation] = @p1) AND ([CreationDate] = @p2) AND ([DisplayName] = @p3) AND ([LastAccessDate] = @p4) AND ([WebsiteUrl] = @p5) AND ([Location] = @p6) AND ([Age] = @p7) AND ([AboutMe] = @p8) AND ([Views] = @p9) AND ([UpVotes] = @p10) AND ([DownVotes] = @p11)',N'@p0 int,@p1 int,@p2 datetime,@p3 nvarchar(4000),@p4 datetime,@p5 nvarchar(4000),@p6 nvarchar(4000),@p7 int,@p8 nvarchar(4000),@p9 int,@p10 int,@p11 int,@p12 nvarchar(4000)',@p0=100,@p1=3077,@p2='2008-08-01 20:41:59',@p3=N'Test display name 4',@p4='2012-02-27 19:44:59',@p5=N'http://www.ehaskins.net',@p6=N'Elkhorn, WI',@p7=20,@p8=N'Empty',@p9=321,@p10=591,@p11=12,@p12=N'Test display name 1'`
Entity Framework	`Exec sp_executeSQL N'update [dbo].[Users] set [DisplayName] = @0 where ([Id] = @1) ',N'@0 nvarchar(40),@1 int',@0=N'Test display name 2',@1=100`
NHibernate	`Exec sp_executeSQL N'UPDATE Users SET Reputation = @p0, CreationDate = @p1, DisplayName = @p2, LastAccessDate = @p3, WebsiteUrl = @p4, Location = @p5, Age = @p6, AboutMe = @p7, Views = @p8, UpVotes = @p9, DownVotes = @p10 WHERE Id = @p11',N'@p0 int,@p1 datetime,@p2 nvarchar(4000),@p3 datetime,@p4 nvarchar(4000),@p5 nvarchar(4000),@p6 int,@p7 nvarchar(4000),@p8 int,@p9 int,@p10 int,@p11 int',@p0=3077,@p1='2008-08-01 20:41:59',@p2=N'Test display name 3',@p3='2012-02-27 19:44:59',@p4=N'http://www.ehaskins.net',@p5=N'Elkhorn, WI',@p6=20,@p7=N'Empty',@p8=321,@p9=591,@p10=12,@p11=100`
Telerik *ORM*	`declare @p1 int set @p1=1 exec sp_prepexec @p1 output,N'@p0 nvarchar(40),@p1 int,@p2 nvarchar(40)',N'UPDATE [Users] SET [DisplayName]=@p0 WHERE [Id] = @p1 AND [DisplayName]=@p2',@p0=N'Test display name 4',@p1=100,@p2=N'Test display name 3' select @p1`

Rys. 15. Wygenerowany kod *SQL* dla operacji modyfikowania danych
Fig. 15. A generated *SQL* code for data modification

Na rys. 15 pokazano kod wygenerowany przez poszczególne systemy odwzorowujące. W tym porównaniu *LINQ to SQL* oraz *NHibernate* w swoim zapytaniu w warunku *WHERE* przesyłają wszystkie kolumny, jakie są zdefiniowane w tabeli. Z pewnością nie jest to najbardziej optymalna forma zapytania, w tym przypadku dochodzi do niepotrzebnego wysyłania strumienia do serwera oraz niepotrzebnego obciążania bazy danych. Natomiast *Entity Framework* oraz *Telerik ORM* wygenerowały w tym przypadku dobrej jakości kod. *Entity Framework* w swoim zapytaniu odnajduje wskazany rekord na podstawie jego identyfikatora, a następnie modyfikuje wybraną kolumnę, *Telerik ORM* zaś dodatkowo dokonuje sprawdzenia jej poprzedniej wartości – takie zachowanie jest związane z mechanizmami rozwiązywania konfliktów w razie próby jednoczesnego dostępu przez różne procesy do tego samego wiersza.

W systemach relacyjnych baz danych bardzo często wykonywanymi operacjami są złączenia tabel; kolejny test dotyczył tych operacji. Po operacji złączenia dwóch tabel (*Posts* i *Users*) na podstawie odpowiednich kluczy, określony został identyfikator użytkownika. Po tej operacji dane zostały zgrupowane według kolumny *Tags*, a na końcu nastąpiła agregacja przy użyciu operatora zliczającego wystąpienia *COUNT*. Przykład kodu realizującego opisany

scenariusz został przedstawiony na rys. 16. W tej sytuacji identyfikatorem, według którego następuje wyszukiwanie użytkownika, jest kolumna *Id* o wartości 100.

```
public void JoinGroupBy()
{
        var i1 = (from p in this.linq.Context.GetTable<Linq.Post>()
                  join u in this.linq.Context.GetTable<Linq.User>()
                     on p.OwnerUserId equals u.Id
                  where u.Id == 100
                  group p by p.Tags into g
                  select g.Count()).ToList();

        var i2 = (from p in this.entity.Context.CreateObjectSet<Entity.Post>()
                  join u in this.entity.Context.CreateObjectSet<Entity.User>()
                    on p.OwnerUserId equals u.Id
                  where u.Id == 100
                  group p by p.Tags into g
                  select g.Count()).ToList();

        var i3 = (from p in this.hibernate.Session.Query<NHibernate.Post>()
                  join u in this.hibernate.Session.Query<NHibernate.User>()
                     on p.OwnerUserId equals u.Id
                  where u.Id == 100
                  group p by p.Tags into g
                  select g.Count()).ToList();

        var i4 = (from p in this.telerik.Posts
                  join u in this.telerik.Users on p.OwnerUserId equals u.Id
                  where u.Id == 100
                  group p by p.Tags into g
                  select g.Count()).ToList();
}
```

Rys. 16. Wywołanie dla operacji złączenia, grupowania i agregacji
Fig. 16. A call of function for join, grouping and aggregation

Na rys. 17 przedstawiano wygenerowane kody dla wcześniejszego wywołania (rys. 16). Dla programisty definiującego przedstawione zapytanie jako procedurę składowaną najbardziej oczywistym rozwiązaniem tak postawionego problemu jest wykonanie operacji złączenia, następnie grupowania i agregacji odpowiedniej kolumny. Zapytania wygenerowane przez *LINQ to SQL* oraz *NHibernate* realizują przedstawiony problem w bardzo intuicyjny sposób. Z kolei *Entity Framework* realizuje omawiane zapytanie nieco inaczej. Kod *SQL* zawiera dwa operatory *SELECT*, najbardziej wewnętrzne zapytanie pobiera wymagane dane, tworząc odpowiednie aliasy, zewnętrzny operator zaś wykorzystuje je, zwracając dane. Algorytm, realizowany przez narzędzie *Telerik ORM*, tworzy zapytanie łączące wymagane tabele na podstawie operatora *JOIN*, natomiast w dalszej fazie nie następuje użycie operatora typu *GROUP BY*, lecz pobranie wartości identyfikatora tabeli *Posts* oraz *Users*. Następnie na podstawie zapamiętanych wartości *Id* wykonywane jest pobieranie wartości z tabeli *Posts*, a zliczanie wystąpień w grupach realizowane jest w kodzie aplikacji. Taki sposób tworzenia kodu *SQL* może być bardzo użyteczny w przypadku, gdy oprócz wartości obliczonej na podstawie operatora *COUNT* wymagane są także pewne inne pola powiązane z daną grupą.

Ostatnim typem zapytania poddanym analizie jest połączenie operatora złączenia typu JOIN z warunkiem filtrującym określającym zakres identyfikatorów (w tym przypadku od

wartości 100 do 200) oraz przedstawienie wyników w postaci anonimowego typu zawierającego liczbę wystąpień w ramach grupy i nazwę kolumny *Tags* (rys. 18).

LINQ to SQL	`exec sp_execute`*`SQL`* `N'SELECT COUNT(*) AS [value]`
	`FROM [dbo].[Posts] AS [t0]`
	`INNER JOIN [dbo].[Users] AS [t1] ON [t0].[OwnerUserId] = ([t1].[Id])`
	`WHERE [t1].[Id] = @p0`
	`GROUP BY [t0].[Tags]',N'@p0 int',@p0=100`

Entity Framework	`SELECT`
	`[GroupBy1].[A1] AS [C1]`
	`FROM ( SELECT`
	`[Extent1].[Tags] AS [K1],`
	`COUNT(1) AS [A1]`
	`FROM [dbo].[Posts] AS [Extent1]`
	`INNER JOIN [dbo].[Users] AS [Extent2] ON [Extent1].[OwnerUserId] = [Extent2].[Id]`
	`WHERE 100 = [Extent2].[Id]`
	`GROUP BY [Extent1].[Tags]`
	`) AS [GroupBy1]`

NHibernate	`exec sp_execute`*`SQL`* `N'select cast(count(*) as INT) as col_0_0_ from Posts post0_, Users user1_ where user1_.Id=post0_.OwnerUserId and user1_.Id=@p0 group by post0_.Tags',N'@p0 int',@p0=100`
Telerik ORM	`SELECT a.[Id] AS COL1, b.[Id] AS COL2 FROM [Posts] a JOIN [Users] AS b ON (a.[OwnerUserId] = b.[Id]) WHERE b.[Id] = 100` `ORDER BY COL1` `declare @p1 int` `set @p1=1` `exec sp_prepexec @p1 output,N'@p0 int',N'SELECT [AcceptedAnswerId] AS COL1, [AnswerCount] AS COL2, [Body] AS COL3, [ClosedDate] AS COL4, [CommentCount] AS COL5, [CreationDate] AS COL6, [FavoriteCount] AS COL7, [LastActivityDate] AS COL8, [LastEditDate] AS COL9, [LastEditorUserId] AS COL10, [OwnerDisplayName] AS COL11, [OwnerUserId] AS COL12, [ParentId] AS COL13, [PostTypeId] AS COL14, [Score] AS COL15, [Tags] AS COL16, [Title] AS COL17, [OwnerUserId] AS COL18, [ViewCount] AS COL19 FROM [Posts] WHERE [Id] = @p0',@p0=237` `select @p1` `exec sp_execute 1,@p0=2942` `exec sp_execute 1,@p0=4458` `exec sp_execute 1,@p0=48872`

Rys. 17. Wygenerowany kod *SQL* dla operacji złączenia, grupowania i agregacji
Fig. 17. A generated *SQL* code for join, grouping and aggregation

W przedstawionej sytuacji (rys. 19) *LINQ to SQL* najpierw wykonuje operacje grupowania, tworząc odpowiedni zbiór danych, następnie dokonuje się ponownie operacja złączenia, aby pozyskać nazwę pola *Tags.* Można zauważyć, że dodawany jest także szereg warunków sprawdzających np. to, czy wartość pola nie jest różna od wartości *NULL*. Dopiero po wykonaniu wskazanych operacji następuje właściwa projekcja wyników. Jeśli chodzi o *Entity Framework*, zapytanie składa się z kilku zagnieżdżonych zapytań. Najbardziej wewnętrzne pozyskuje dane z określonego zakresu w wyniku operacji *JOIN,* tworząc odpowiedni obiekt nazwany aliasem *Distinct1*. Następnie podobna operacja jest ponownie przeprowadzana. Na tym etapie następuje pobranie wartości z drugiego zapytania oraz trzecie już z kolei złączenie. Dopiero po czterokrotnym wygenerowaniu podzapytań możliwa jest projekcja wyniku. Taki kod jest bardzo trudny w analizowaniu oraz mało czytelny i mało optymalny. *NHibernate* generuje krótki kod, który jest znacznie czytelniejszy od poprzedniej wersji. Można zauważyć, iż w tym wypadku nie istnieją niepotrzebne złączenia. *Telerik ORM* generuje kod *SQL* podobny jak w poprzednim przypadku, jednak można postawić pytanie, czy w przedstawionej sytuacji wszystkie odwołania do tabeli *Users* są konieczne (jest ich aż 290).

```
Public void JoinSelect()
{
        var i1 = (from p in this.linq.Context.GetTable<Linq.Post>()
                  join u in this.linq.Context.GetTable<Linq.User>()
                     on p.OwnerUserId equals u.Id
                  where u.Id >= 100 && u.Id <= 200
                  group p by p.Tags into g
                  select new
                  {
                       Count = g.Count(),
                       Tags = g.First().Tags
                  }).ToList();

        var i2 = (from p in this.entity.Context.CreateObjectSet<Entity.Post>()
                  join u in this.entity.Context.CreateObjectSet<Entity.User>()
                     on p.OwnerUserId equals u.Id
                  where u.Id >= 100 && u.Id <= 200
                  group p by p.Tags into g
                  select new
                  {
                       Count = g.Count(),
                       Tags = g.FirstOrDefault().Tags
                  }).ToList();

        var i3 = (from p in this.hibernate.Session.Query<NHibernate.Post>()
                  join u in this.hibernate.Session.Query<NHibernate.User>()
                     on p.OwnerUserId equals u.Id
                  where u.Id >= 100 && u.Id <= 200
                  group p by p.Tags into g
                  select new
                  {
                       Count = g.Count(),
                       Tags = g.First().Tags
                  }).ToList();

        var i4 = (from p in this.telerik.Posts
                  join u in this.telerik.Users on p.OwnerUserId equals u.Id
                  where u.Id >= 100 && u.Id <= 200
                  group p by p.Tags into g
                  select new
                  {
                       Count = g.Count(),
                       Tags = g.First().Tags
                  }).ToList();
}
```

Rys. 18. Wywołanie dla operacji złączenia, grupowania i projekcji
Fig. 18. A call of function for join, grouping and projection

Gdy weźmie się pod uwagę przedstawione w tym podrozdziale wyniki, programista, generując niestandardowe, zapytania powinien mieć na uwadze to, że kod utworzony przez poszczególne mechanizmy *ORM* nie zawsze jest zgodny z oczekiwaniami, czasem nawet jest bardzo zaskakujący. W takich przypadkach należy sprawdzić kształt wygenerowanego zapytania i dokonać stosownych modyfikacji.

LINQ to SQL	

```
exec sp_executeSQL N'SELECT [t2].[value] AS [Count], (
    SELECT [t5].[Tags]
    FROM (
        SELECT TOP (1) [t3].[Tags]
        FROM [dbo].[Posts] AS [t3]
        INNER JOIN [dbo].[Users] AS [t4] ON [t3].[OwnerUserId] = ([t4].[Id])
        WHERE ((([t2].[Tags] IS NULL) AND ([t3].[Tags] IS NULL)) OR
(([t2].[Tags] IS NOT NULL) AND ([t3].[Tags] IS NOT NULL) AND ([t2].[Tags] =
[t3].[Tags]))) AND ([t4].[Id] >= @p0) AND ([t4].[Id] <= @p1)
        ) AS [t5]
    ) AS [Tags]
FROM (
    SELECT COUNT(*) AS [value], [t0].[Tags]
    FROM [dbo].[Posts] AS [t0]
    INNER JOIN [dbo].[Users] AS [t1] ON [t0].[OwnerUserId] = ([t1].[Id])
    WHERE ([t1].[Id] >= @p0) AND ([t1].[Id] <= @p1)
    GROUP BY [t0].[Tags]
    ) AS [t2]',N'@p0 int,@p1 int',@p0=100,@p1=200
```

Entity Framework	SELECT 1 AS [C1], [Project4].[C2] AS [C2], [Project4].[C1] AS [C3] FROM (SELECT [Project3].[C1] AS [C1], (SELECT COUNT(1) AS [A1] FROM [dbo].[Posts] AS [Extent5] INNER JOIN [dbo].[Users] AS [Extent6] ON [Extent5].[OwnerUserId] = [Extent6].[Id] WHERE ([Extent6].[Id] >= 100) AND ([Extent6].[Id] <= 200) AND (([Project3].[Tags] = [Extent5].[Tags]) OR (([Project3].[Tags] IS NULL) AND ([Extent5].[Tags] IS NULL)))) AS [C2] FROM (SELECT [Distinct1].[Tags] AS [Tags], (SELECT TOP (1) [Extent3].[Tags] AS [Tags] FROM [dbo].[Posts] AS [Extent3] INNER JOIN [dbo].[Users] AS [Extent4] ON [Extent3].[OwnerUserId] = [Extent4].[Id] WHERE ([Extent4].[Id] >= 100) AND ([Extent4].[Id] <= 200) AND (([Distinct1].[Tags] = [Extent3].[Tags]) OR (([Distinct1].[Tags] IS NULL) AND ([Extent3].[Tags] IS NULL)))) AS [C1] FROM (SELECT DISTINCT [Extent1].[Tags] AS [Tags] FROM [dbo].[Posts] AS [Extent1] INNER JOIN [dbo].[Users] AS [Extent2] ON [Extent1].[OwnerUserId] = [Extent2].[Id] WHERE ([Extent2].[Id] >= 100) AND ([Extent2].[Id] <= 200)) AS [Distinct1]) AS [Project3]) AS [Project4]
NHibernate	exec sp_executeSQL N'select cast(count(*) as INT) as col_0_0_, post0_.Tags as col_1_0_ from Posts post0_, Users user1_ where user1_.Id=post0_.OwnerUserId and user1_.Id>=@p0 and user1_.Id<=@p1 group by post0_.Tags',N'@p0 int,@p1 int',@p0=100,@p1=200
Telerik *ORM*	declare @p1 int set @p1=1 exec sp_prepexec @p1 output,N'@p0 int',N'SELECT [AcceptedAnswerId] AS COL1, [AnswerCount] AS COL2, [Body] AS COL3, [ClosedDate] AS COL4, [CommentCount] AS COL5, [CreationDate] AS COL6, [FavoriteCount] AS COL7, [LastActivityDate] AS COL8, [LastEditDate] AS COL9, [LastEditorUserId] AS COL10, [OwnerDisplayName] AS COL11, [OwnerUserId] AS COL12, [ParentId] AS COL13, [PostTypeId] AS COL14, [Score] AS COL15, [Tags] AS COL16, [Title] AS COL17, [OwnerUserId] AS COL18, [ViewCount] AS COL19 FROM [Posts] WHERE [Id] = @p0 ',@p0=237 select @p1 exec sp_execute 1,@p0=264 exec sp_execute 1,@p0=265 exec sp_execute 1,@p0=289 . . .exec sp_execute 1,@p0=98376

Rys. 19. Wygenerowany kod *SQL* dla operacji złączenia, grupowania i projekcji
Fig. 19. A generated *SQL* code for join, grouping and projection

6. Podsumowanie

Na podstawie przeprowadzonych eksperymentów, dotyczących zbadania czasów odpowiedzi dla różnych mechanizmów odwzorowania obiektowo-relacyjnego oraz analizy wygenerowanego kodu na przykładzie różnego typu zapytań, nasuwa się kilka wniosków. Po pierwsze, stając przed wyborem odpowiedniego systemu odwzorowującego, trzeba zadać sobie pytanie, jakie kryteria powinien spełniać system, gdyż to właśnie będzie determinować wybór odpowiedniego narzędzia. Następnie ważnym aspektem jest kwestia typu przetwarza-

nia danych. Czy system będzie wykonywać duże transakcje, czy też pojedyncze zapytania? Wszystkie wymienione aspekty wymagają dużego doświadczenia ze strony projektanta. Według analizy wyników uzyskanych w ramach przeprowadzonych badań najlepszym narzędziem odwzorowującym jest *Telerik Open Access ORM*, który dla większości operacji uzyskał czasy porównywalne z *ADO.NET* lub lepsze. *ADO.NET* nie oferuje takiej elastyczności i skalowalności. Co więcej, konfiguracja oraz kształt końcowy kodu *SQL* generowanego przez *Telerik* wydają się bardzo dobrze przemyślane. Choć *ADO.NET* nie dostarcza takich metod jak inne interfejsy, to zapewne znajdzie on zastosowanie w systemach, w których programista chce mieć całkowitą kontrolę nad generowanym kodem wysyłanym do bazy danych. Bardzo korzystnym faktem jest implementacja wzorca repozytorium przez narzędzie *Telerik*, co pozwala na skrócenie czasu podczas pracy nad systemem. Z kolei *NHibernate* jest narzędziem, które dostarcza infrastrukturę do bardzo dokładnej konfiguracji. Jeśli zostanie stwierdzone, że jakaś część nie spełnia założonych wymagań i narzutów czasowych, można zmienić szereg ustawień, jak to miało miejsce podczas testowania operacji usuwania. Pozostałe narzędzia, takie jak: *LINQ to SQL* czy *Entity Framework*, także nie odbiegają od poprzednich. Bez wątpienia *LINQ to SQL* bardzo dobrze będzie nadawał się do małych i średnich projektów ze względu na nieskomplikowany, prosty sposób konfiguracji. Wracając raz jeszcze do zagadnienia generacji kodu przez poszczególne programy, można zauważyć, iż czasem kod *SQL* jest bardzo enigmatyczny i długi, nawet w przypadku bardzo prostej funkcjonalności. Ten aspekt należy także wziąć pod uwagę, ponieważ wąskim gardłem, niektórych aplikacji jest właśnie problem komunikacji z serwerem bazy danych. W tym przypadku liczba przesyłanych bajtów w strumieniu powinna być jak najmniejsza.

BIBLIOGRAFIA

1. Microsoft. Microsoft, 2012, http://msdn.microsoft.com.
2. Johnson G.: Accessing Data with Microsoft .NET Framework 4. Microsoft Press, Washington 2011.
3. Jałowiecki P. A. O.: Język programowania wysokiego poziomu C++. Programowanie obiektowe. SGGW, Warszawa 2006.
4. Esposito D.: Programming Microsoft ASP.NET MVC. Microsoft Press, Washington 2010.
5. Seemann M.: Dependency Injection in .NET. Manning Publications Co., NY 2011.
6. Evjen B., Hanselman S., Rader D.: Zaawansowane programowanie ASP.NET 4 z wykorzystaniem C# i VB. Helion, Gliwice 2010.
7. Troelsen A.: Język C# 2010 i platforma .NET4. PWN, Warszawa 2011.

8. Lerman J.: Programming Entity Framework, 2nd edition. O'Reilly, 2010.
9. Dentler J.: NHibernate 3.0 Cookbook. Packt Publishing Ltd., Birmingham 2010.
10. NHibernate. 2012, http://nhforge.org.
11. Telerik. 2012, http://www.telerik.com.

Wpłynęło do Redakcji 16 stycznia 2013 r.

Abstract

In this paper we present the results of ongoing research, in which we conducted a comparative analysis taking into account execution time and generation of *SQL* code of the object-relational mapping tools based on the *.NET* platform.

The results of the analysis and the experiments are illustrated by the charts and properly interpreted. In this paper we also describe the design of the system supporting the conduct of such research and the structure of the database containing the necessary test data.

This article is divided into three parts. In the first part, we describe the basic features of the object-relational mapping mechanisms and present the chosen ones used to platform a comparative analysis.

In the second part, we show the design of the system and its architecture. We can find here details of the project structure, architectural solutions and applied design patterns. In this place we present the database structure too.

The third part concerns ongoing research, it includes descriptions of the tests with the corresponding interpretations and conclusions. Each of the tests are illustrated by the appropriate chart. The final part of the article contains a short summary.

Adresy

Witold OLEŚ: TA Group Sp. z o.o., ul. Mariacka 4, 40-014 Katowice, witek_oles@wp.pl.
Bożena MAŁYSIAK-MROZEK: Politechnika Śląska, Instytut Informatyki,
ul. Akademicka 16, 44-100 Gliwice, Polska, bozena.malysiak@polsl.pl.
Dariusz MROZEK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, dariusz.mrozek@polsl.pl.