Marcin Ostrowski, Michał Karbowańczyk
Politechnika Łódzka Instytut Informatyki
ul. Wólczańska 215, 90-924 Łódź
email: michal.karbowanczyk@p.lodz.pl

REDUKOWANIE CZASU REALIZACJI ŻĄDAŃ PRZEZ WIELOWARSTWOWĄ APLIKACJĘ SIECIOWĄ

Streszczenie – W artykule rozważono możliwe techniki optymalizacji wielowarstwowej aplikacji sieciowej pod kątem redukcji czasu realizacji żądania. Badana aplikacja została poddana modyfikacjom zarówno w zakresie algorytmów, jak i techniki operowania w bazie danych oraz wykorzystania mechanizmów wspomagających, takich jak przetwarzanie równoległe i buforowanie. Otrzymane wyniki stanowią wskazówkę odnośnie celowości stosowania określonych technik dla rozważanego typu aplikacji. Artykuł jest streszczeniem pracy dyplomowej magisterskiej [1].

Słowa kluczowe: aplikacje sieciowe, aplikacje wielowarstwowe, optymalizacja

1 Wprowadzenie

Problem redukowania czasu realizacji żądań przez działające już aplikacje jest tematem dosyć trudnym i złożonym. Niestety, trudno znaleźć jednoznaczne rozwiązania w literaturze, które opisują działania jakie należy podjąć, aby zwiększyć wydajność aplikacji. W Internecie można znaleźć ogólne wskazówki pozwalające na redukcję czasu realizacji żądań. Jednakże nie zawierają one solidnych i rzetelnych opracowań, które by przedstawiały wyższość jednego rozwiązania nad drugim. Istnieją jednak proste, ogólne zasady, których należy przestrzegać podczas projektowania systemów informatycznych [2]. Znając takie zasady, z łatwością można zastosować je w istniejących systemach informatycznych w celu zwiększenia wydajności, co pozwoli na redukcję czasu realizacji żądań.

Właściwe miejsce przetwarzania danych

Operacja, której zadaniem jest przetwarzanie dużej ilości danych powinna się znajdować jak najbliżej bazy danych. Funkcje agregujące powinny być wykonywane w bazie danych, natomiast operacje przetwarzające pojedyncze rekordy powinny być wykonywane w aplikacji.

Wczytywanie tylko koniecznych danych

Bardzo często stosowane jest wczytywanie z bazy danych, które nie są w danym momencie potrzebne. Jest to rozwiązanie proste i ukierunkowane na kolejne (jeszcze nie znane) funkcjonalności systemu. Jednakże takie rozwiązanie prowadzi do obniżenia wydajności, w szczególności jeśli nowe funkcjonalności systemu, które mogłyby wykorzystać dodatkowe dane, nigdy nie powstaną.

Unikanie zbędnej generalizacji

Problem stosowania nadmiernej generalizacji powstaje w sytuacji, gdy programiści próbują przewidywać kolejne etapy rozwoju systemu. Powoduje to powstawanie wielu interfejsów, klas abstrakcyjnych, które mogą powodować wczytywanie niepotrzebnych danych.

Rozwijanie i testowanie aplikacji na prawdziwych danych

Praca na prawdziwych danych pozwala na lokalizację miejsc w aplikacji, które negatywnie wpływają na czas realizacji żądań. Już na etapie rozwoju oprogramowania można zaobserwować obniżoną wydajność wynikającą z przetwarzania dużej ilości danych.

2 Wybrane techniki redukcji czasu realizacji żądań

Równoległe przetwarzanie

Celem stosowania równoległego przetwarzania (paralelizmu) jest redukcja czasu wykonywania danego zadania poprzez podzielenie go na mniejsze zadania, które będą wykonywane równolegle [3][4]. W literaturze można znaleźć przykłady, które potwierdzają skuteczność stosowania paralelizmu w kontekście zmniejszania czasu przetwarzania danych. W przykładzie opisanym w [5] program wykonywał zadanie, które polegało na inkrementacji wielu liczników od 0 do 10 000 000. Przetwarzanie równoległe okazało się być znacznie wydajniejsze. Dla niewielkiej liczby liczników różnica jest niewielka, jednakże wraz ze wzrostem liczby liczników różnica staje się znacznie większa. Przetwarzanie równoległe przynosi największe korzyści podczas wykonywania wielu operacji na dużych zbiorach danych.

Stosowanie paralelizmu nie zawsze musi przynieść korzyści w postaci szybszego przetwarzania danych. Na podstawie specyfiki działania przetwarzania równoległego można sformułować warunki, które powinny być spełnione podczas korzystania z równoległego przetwarzania [5][6][7][8][9].

Przetwarzanie dużych ilości danych

Dla niewielkiej ilości danych w większości przypadków nie opłaca się korzystać z paralelizmu, ponieważ dane i tak zostaną bardzo szybko przetworzone.

Przetwarzanie pojedynczego elementu trwa pewien czas

Oznacza to, że obliczenia i operacje, które muszą zostać wykonane dla danego elementu są czasochłonne.

Proces przetwarzania elementów może być wykonywany równolegle

Nie każdy algorytm w danej formie może być przetwarzany równolegle. Operacje, które mają zostać wykonane muszą być powtarzalne oraz niezależne od siebie.

Proces nie jest uruchamiany w środowisku wielowątkowym (np. kontener aplikacji sieciowych)

Środowisko wielowątkowe z definicji zapewnia, że otrzymane żądania będą wykonywane równolegle. Dodatkowe zrównoleglenie wewnątrz kodu realizującego żądanie nie wpłynie pozytywnie na czas przetwarzania pojedynczego żądania. Może nawet wydłużyć ten czas.

Stosowanie paralelizmu jest bardzo istotne. Obecny trend architektur CPU zmierza w kierunku zwiększania liczby rdzeni. Ma to wpływ na architekturę oprogramowania. Oprogramowanie należy przygotowywać w taki sposób, aby wykorzystywać zalety nowych architektur CPU. Nie powinno się już polegać na zwiększeniu prędkości zegarów w procesorach. Oczywiście, nie każdy problem może zostać zrównoleglony, natomiast wykorzystywanie paralelizmu wszędzie tam, gdzie jest to wskazane ma bardzo duży wpływ na redukcję czasu realizacji żądań [5][10]. Projektując aplikację należy zatem zastanowić się, które operacje nie mogą być zrównoleglone, a następnie warto zastosować paralelizm tam, gdzie jest to możliwe.

Ograniczanie połączeń z bazą danych

Jednym z głównych komponentów odpowiedzialnych za szybkość działania aplikacji wielowarstwowych jest styk aplikacji z warstwą składowania, którą zazwyczaj jest baza danych. Baza danych może znajdować się na tym samym urządzeniu, na którym znajduje się aplikacja, bądź też na innej maszynie, do której dostęp odbywa się poprzez sieć. W przypadku, gdy dostęp do bazy danych odbywa się poprzez sieć istotne jest ograniczenie liczby zapytań, ponieważ wywołania sieciowe są bardzo kosztowne. Ponadto, szybkość bazy danych zależna jest od urządzenia blokowego, na którym dane są składowane. Po osiągnięciu limitu przepustowości urządzenia nie jest

już możliwe zwiększenie przepustowości aplikacji. Rozwiązaniem fizycznych ograniczeń bazy danych jest stosowanie buforowania (ang. cache), łączenia zapytań, procedur wbudowanych oraz minimalizacji zapytań [11].

Buforowanie

Proces ten polega na pobraniu danych ze zdalnej bazy danych podczas pierwszego odwołania oraz pozostawieniu ich w przestrzeni składowania o wyższej wydajności, do wykorzystania przez kolejne odwołania do tych danych. W ten sposób zostaje ograniczona konieczność łączenia z bazą danych i przesyłania danych.

Wśród technik buforowania danych składowanych w zdalnych bazach danych należy wymienić:

- utrzymywanie kopii danych w lokalnej bazie danych, umieszczonej w sieci lokalnej wraz z serwerem aplikacji;
- buforowanie danych w pamięci bazy danych, co pozwala na redukcję odczytów z fizycznej przestrzeni składowania danych;
- buforowanie danych w pamięci aplikacji, co pozwala na redukcję odwołań do bazy danych.

W niniejszej pracy główna uwaga zostanie poświęcona buforowaniu danych w aplikacji. Z punktu widzenia wydajności, kierowanie zapytań do aplikacji zamiast do bazy danych znacznie skraca czas realizacji żądań. Korzystając z takiego rozwiązania należy zwrócić uwagę na konieczność odświeżania danych. Kolejnym aspektem jest ograniczenie przestrzeni pamięci, w której dane będą przechowywane [12]. Istotnym kryterium stosowalności buforowania jest powtarzalność zapytań kierowanych do źródeł danych. W wielu sytuacjach wynik końcowy może wymagać wielu skomplikowanych obliczeń z różnych źródeł danych. Jeśli dane wejściowe do obliczeń nie różnią się, a niewielkie różnice wyników końcowych nie mają znaczenia dla użytkownika końcowego, wtedy zastosowanie buforowania będzie rozsądną decyzją [13]. Stosowanie buforowania po stronie aplikacji ma tak znaczący wpływ na redukcję czasu realizacji żądań również dlatego, że dzięki niemu pomijane jest wykonywanie wielu operacji, które musiały mieć miejsce w przypadku realizacji zapytania do bazy danych. Do takich operacji należą m. in.: generowanie SQL z pierwotnego formatu zapytania ORM, wysłanie zapytania sieciowego, parsowanie SQL, konwersja danych z bazy danych na obiekty [14].

Łączenie zapytań

Łączenie zapytań polega na stosowaniu pojedynczego złożonego zapytania, zamiast kilku prostych. Jedną z możliwości jest stosowanie funkcji JOIN dla zapytań w języku SQL. W wyniku łączenia zapytań

oczywistym jest, że zmniejszy się liczba odwołań do bazy danych, co powinno przyczynić się do wzrostu wydajności. Niemniej jednak należy zwracać szczególną uwagę na to, czy stworzone zapytanie łączone wciąż jest optymalne.

Procedury wbudowane

Kolejnym sposobem na ograniczenie zapytań do bazy danych jest zastosowanie procedur wbudowanych. Procedury wbudowane umożliwiają wykonanie wielu zapytań do bazy danych w ramach pojedynczego odwołania do bazy danych z aplikacji, które będzie miało na celu uruchomienie wybranej procedury. Stosowanie procedur wbudowanych posiada wiele zalet [2][15]:

- enkapsulacja struktury danych – ukrycie struktur relacyjnej bazy danych z punktu widzenia aplikacji. W przypadku, gdy struktury relacyjnej bazy danych zostaną zmodyfikowane, wymagane są jedynie zmiany w procedurach wbudowanych;
- wydajność – funkcjonalność biznesowa, która wymagałaby wielu zapytań do bazy danych byłaby związana z wysłaniem każdego zapytania, a następnie z przetworzeniem odpowiedzi każdego z nich. Umieszczenie zapytań w pojedynczej procedurze ogranicza liczbę zapytań aplikacji do bazy danych. Ponadto ponowne wykonanie procedury wbudowanej jest realizowane w oparciu o plan zapytania umieszczony w buforze procedury, dzięki czemu nie ma konieczności ponownej analizy realizowanych zapytań;
- zmniejszenie transferu pomiędzy aplikacją, a serwerem bazy danych – procedury umożliwiają wykonywanie zapytań, które mogą być zależne od wyników poprzednich zapytań. W przypadku wykonywania każdego zapytania osobno, wynik musiałby być przesyłany do aplikacji. W przypadku wykonywania procedury, wynik danego zapytania jest przekazywany do kolejnego zapytania pozostając wciąż po stronie bazy danych. W efekcie, tylko wynik końcowy zostanie przesłany do aplikacji, co znacznie ograniczy transfer pomiędzy aplikacją, a bazą danych. Efekt tego działania jest szczególnie widoczny w przypadku iteracyjnego wykonywania wielu zapytań.

Stosowanie procedur wbudowanych jest jednak obarczone znacznym kosztem. Część funkcjonalności aplikacji jest wówczas realizowana w odrębnym środowisku, w innym języku programowania, co utrudnia proces wytwarzania i utrzymywania aplikacji.

Minimalizacja zapytań

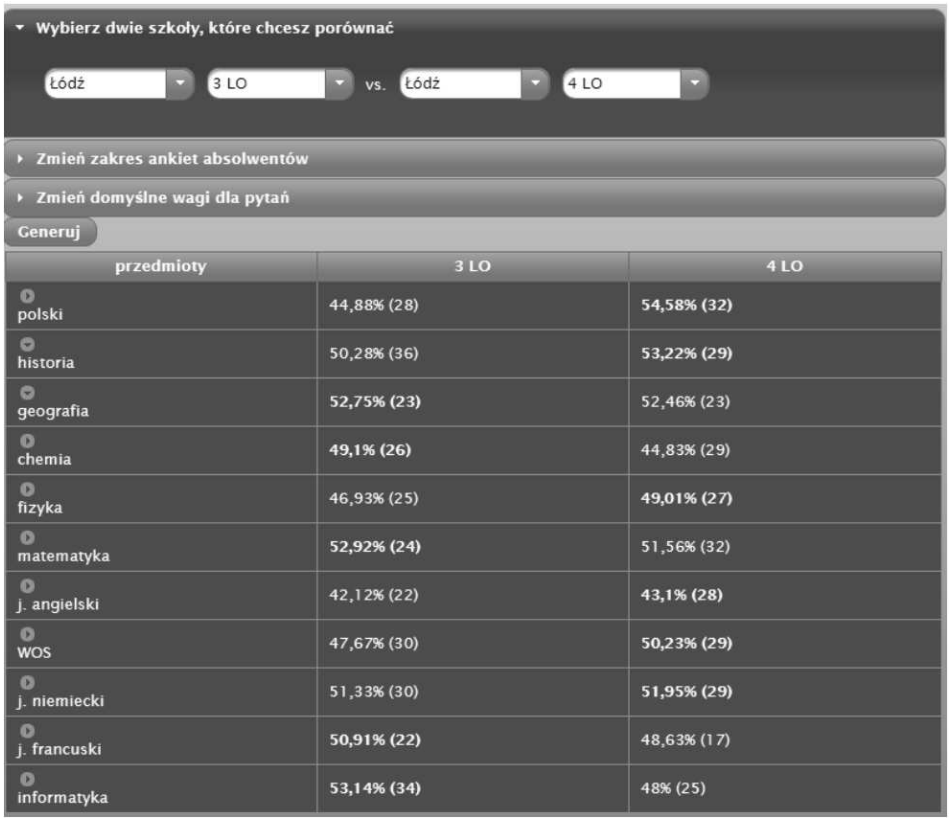
Należy zwrócić uwagę, czy stosowanie każdego zapytania jest konieczne. Wykonanie nowego, kolejnego zapytania w celu pozyskania wymaganych danych jest prostsze, ale może okazać się nadmiarowe, jeśli takie dane zostały już wczytane wcześniej (np. w ramach poprzedniego zapytania). Rezygnacja z danych, które są łatwo dostępne, może wiązać się z koniecznością zastosowania innego algorytmu, którego zadaniem będzie wykonanie wymaganych obliczeń. Nawet, jeśli nowy algorytm będzie bardziej skomplikowany może okazać się dużo bardziej opłacalny w stosunku do kolejnego zapytania zwracającego dużą ilość danych.

Tworząc oprogramowanie ściśle zgodne z zasadami programowania zorientowanego obiektowo okazuje się, że wywoływana jest nadmierna liczba zapytań do bazy danych. Istotne jest znalezienie kompromisu korzystając z zasad programowania zorientowanego obiektowo, w celu ograniczenia liczby zapytań do bazy danych. Taki kompromis może zostać uzyskany poprzez zastąpienie wielu pojedynczych zapytań pojedynczą operacją logiczną, bądź przeniesieniem tych operacji do procedury wbudowanej. W literaturze można znaleźć przypadki znaczącej poprawy wydajności aplikacji dzięki redukcji zapytań. Dla przykładu zostanie przedstawiony przypadek, w którym czas realizacji żądania użytkownika wynosił 180 sekund [16]. Czas ten wynikał z wielu zapytań do bazy danych. Zauważono również, że wyrażenie INSERT wprowadzające pojedynczy wiersz do bazy danych, wykonywane jest 348 razy. W celu ograniczenia liczby tych wyrażeń, w omawianym przypadku zastosowano funkcjonalność umożliwiającą wstawianie wielu wierszy za pomocą jednego wywołania. W wyniku czego, liczba wywołań odpowiedzialnych za wprowadzanie wierszy do bazy danych została ograniczona do 4. Ponadto wyeliminowano wiele zbędnych zapytań do bazy danych. Dzięki zastosowaniu powyższych optymalizacji osiągnięto zmniejszenie czasu realizacji żądania o 54%.

3 Badana aplikacja

Badaniom poddana została aplikacja „Porównywarka Liceów Ogólnokształcących”, stanowiąca część praktyczną pracy dyplomowej inżynierskiej [17]. Bazując na opiniach absolwentów, aplikacja wskazuje różnice pomiędzy uzyskanymi ocenami przedmiotów. Pozwala to na wskazanie szkoły, w której przedmioty są lepiej prowadzone. Opinie absolwentów zbierane są za pomocą ankiet, których postać jest konfigurowana podczas uruchamiania systemu. Porównanie szkół jest możliwe dzięki odpowiedniej interpretacji udzielonych odpowiedzi na pytania z ankiety. Polega ona na obliczeniu oceny danego przedmiotu w

danej szkole na podstawie udzielonych odpowiedzi na pytania z ankiety. Przykładowy wynik działania aplikacji został przedstawiony na Rys. 1.

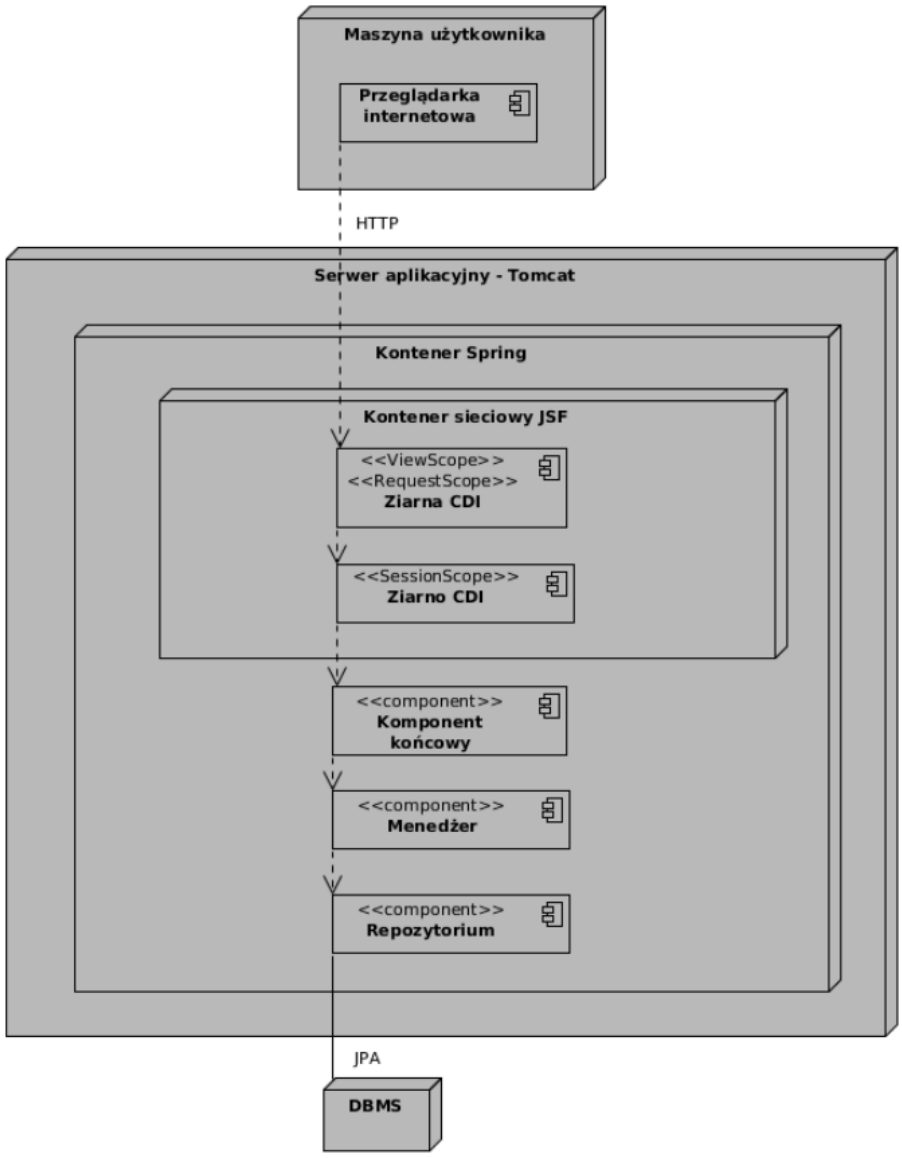


przedmioty	3 LO	4 LO
polski	44,88% (28)	**54,58% (32)**
historia	50,28% (36)	**53,22% (29)**
geografia	**52,75% (23)**	52,46% (23)
chemia	**49,1% (26)**	44,83% (29)
fizyka	46,93% (25)	**49,01% (27)**
matematyka	**52,92% (24)**	51,56% (32)
j. angielski	42,12% (22)	**43,1% (28)**
WOS	47,67% (30)	**50,23% (29)**
j. niemiecki	51,33% (30)	**51,95% (29)**
j. francuski	**50,91% (22)**	48,63% (17)
informatyka	**53,14% (34)**	48% (25)

Rys. 1. Przykładowy wynik działania aplikacji

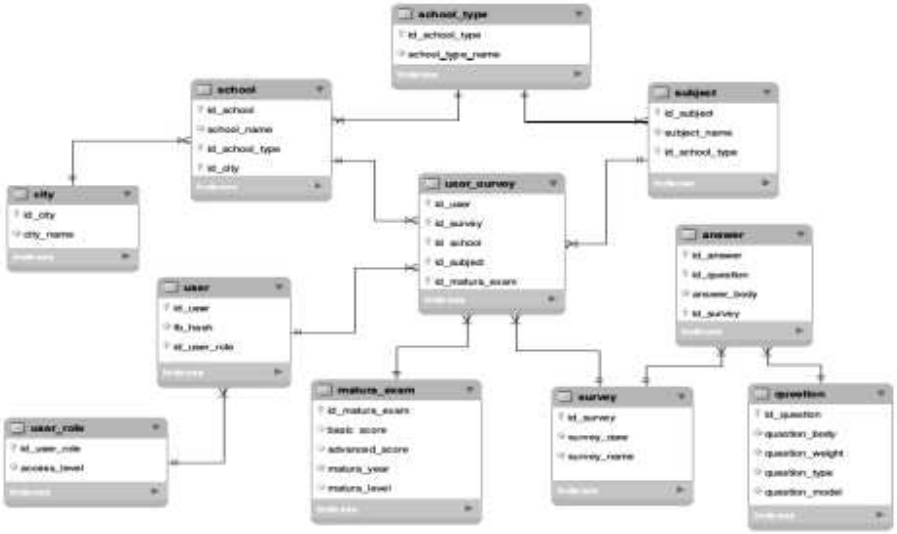
Architektura

Aplikacja została zrealizowana w klasycznej architekturze trójwarstwowej. Składa się z warstwy widoku, warstwy logiki biznesowej oraz warstwy danych. Warstwa widoku została stworzona w technologii Java Server Faces (JSF). Kontener sieciowy JSF kontroluje zawartość generowanych stron HTML oraz zarządza przepływem pomiędzy nimi za pomocą ziaren pełniących funkcję kontrolera. Ziarno CDI (SessionScope) odpowiada za przechowywanie potrzebnych danych w sesji HTTP.

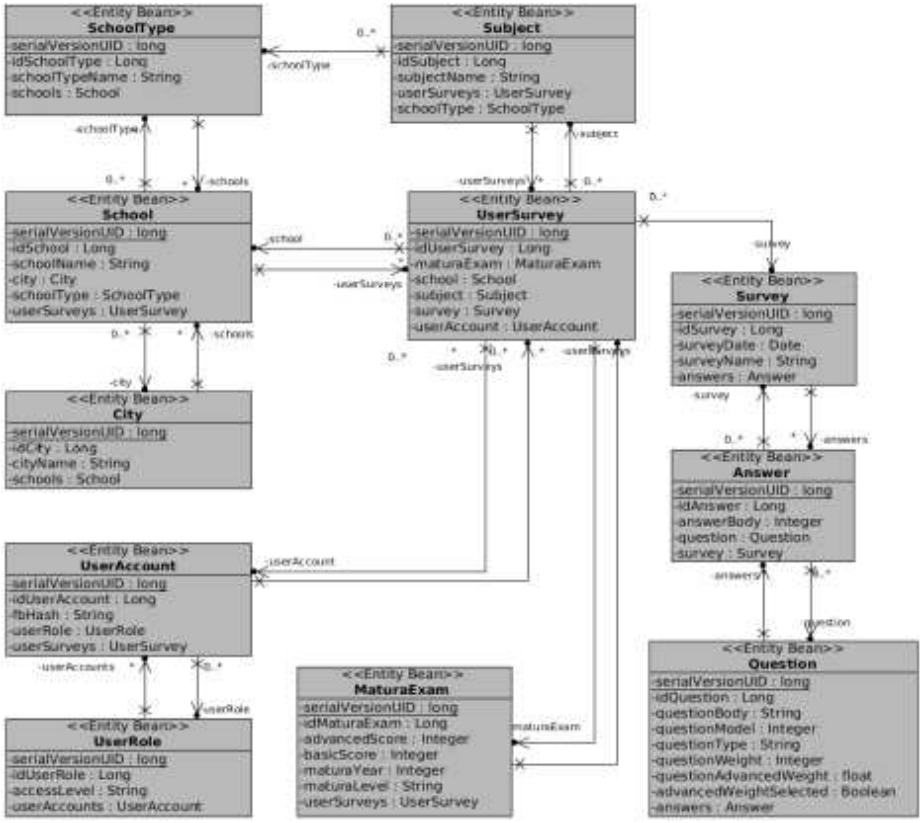


Rys. 2. Architektura aplikacji

Jako warstwa składowania danych wykorzystana została relacyjna baza danych PostgreSQL, zaś dostęp do danych został w aplikacji zrealizowany poprzez użycie mapowania obiektowo-relacyjnego (biblioteka JPA/Hibernate). Struktura bazy relacyjnej została zaprezentowana na Rys. 3, zaś odpowiadający jej model klas encyjnych – na Rys. 4.



Rys. 3. Struktura bazy danych aplikacji

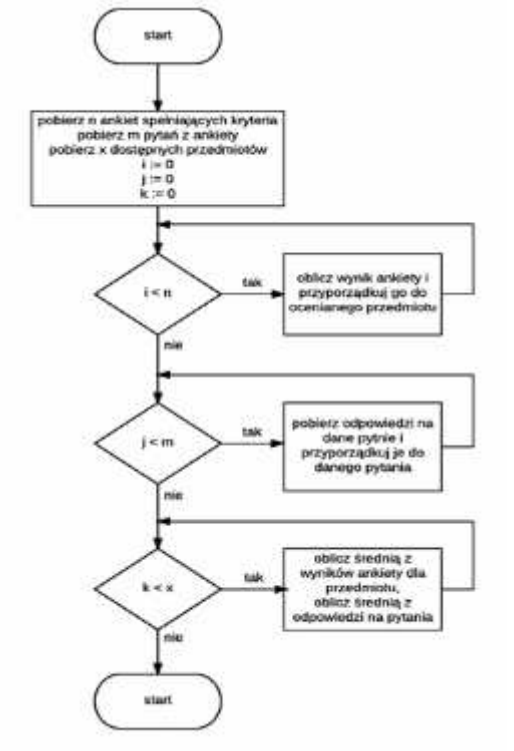


Rys. 4. Model klas encyjnych aplikacji

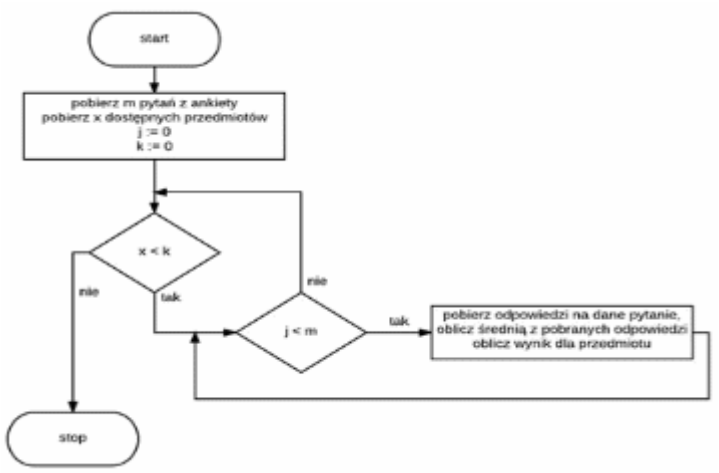
4 Zastosowane techniki redukcji czasu realizacji żądań

Poprawa algorytmów i architektury aplikacji

Przed zastosowaniem konkretnych metod, mających na celu skrócenie czasu realizacji żądań przeanalizowano obecne rozwiązanie i upewniono się, czy zastosowana implementacja zapewnia możliwie jak najkrótszy czas realizacji żądania. Poprawiając algorytmy i architekturę aplikacji przede wszystkim zwrócono szczególną uwagę na ograniczeniu wczytywania danych z bazy danych. Poprawa polegała na całkowitej rezygnacji z wykonywania niektórych zapytań, które okazały się być nadmiarowe. Zrezygnowano także z wykonywania kolejnych zapytań do bazy danych kosztem modyfikacji algorytmu, który w wyniku stał się bardziej skomplikowany, lecz pozwalał na wykonanie obliczeń na wcześniej wczytanych danych, aby uzyskać ten sam efekt. Schematy blokowe algorytmów przed i po modyfikacji przedstawione zostały odpowiednio na Rys. 5 i Rys. 6.



Rys. 5. Schemat blokowy algorytmu porównującego szkoły – przed optymalizacją



Rys. 6. Schemat blokowy algorytmu porównującego szkoły - po optymalizacji

Zastosowanie przetwarzania równoległego

Próba zastosowania przetwarzania równoległego w omawianej aplikacji jest o tyle interesująca, że aplikacja ta spełnia zarówno przesłanki przemawiające za takim krokiem (przetwarzanie dużej ilości danych, tu – odpowiedzi na pytania z ankiet), jak i te przemawiające przeciw (jako uruchamiana w kontenerze sieciowym jest już poddana przetwarzaniu równoległemu). Przetwarzanie równoległe zostało więc wprowadzone w celu umożliwienia porównania go z innymi rozwiązaniami, ze świadomością, iż wynik może być niekorzystny. Do realizacji tej techniki zastosowano równoległe strumienie danych wprowadzone w języku Java 8 (Collection.parallelStream).

Zastosowanie procedur wbudowanych

Zgodnie z przedstawioną wcześniej charakterystyką procedur wbudowanych wybrano metodę obliczającą średnią ocenę przedmiotu jako tę, która powinna zostać poddana przekształceniu w procedurę wbudowaną. Metoda wykonuje stosunkowo prostą operację arytmetyczną jaka jest obliczenie średniej na dużych ilościach danych wybieranych za pomocą kwerendy, jako wynik zwracając pojedynczą wartość. Zastosowanie procedury wbudowanej powinno więc przynieść znaczną redukcję ilości danych przesyłanych między bazą a aplikacją. Modyfikację wprowadzono w dwóch wariantach celem ich porównania. W pierwszym wariancie stworzono procedurę wbudowaną obliczająca wynik dla danej szkoły i przedmiotu, która musiała być wywoływana dwukrotnie w celu porównania oceny danego przedmiotu w obu szkołach. W drugim wariancie procedura obliczała w jednym wywołaniu odpowiednie wartości dla obu szkół. Jako dodatkowy wariant do

porównania zaproponowane zostało wykorzystanie wbudowanej funkcji agregującej bezpośrednio w zapytaniu formułowanym przez aplikację zamiast umieszczania jej w procedurze wbudowanej.

Zastosowanie buforowania

Kolejnym rozwiązaniem mającym na celu redukcję czasu realizacji żądań jest buforowanie danych w pamięci aplikacji. W przypadku aplikacji wykorzystanej na potrzeby niniejszej pracy rozwiązanie to wydaje się być kontrowersyjne ze względu na model biznesowy prowadzący do niewielkiego prawdopodobieństwa korzystania z tych samych danych w kolejnych żądaniach. Aplikacja umożliwia porównywanie każdej szkoły z każdą inną w Polsce. W efekcie liczba kombinacji może być ogromna. Wyniki dla każdej kombinacji musiałyby być przechowywane w pamięci aplikacji, która jest ograniczona. Mimo to postanowiono porównać jaki wpływ ma zastosowanie buforowania danych w aplikacji na czas realizacji żądań. W celu zastosowania buforowania skorzystano z biblioteki SimpleCacheManager, przy czym buforowaniu poddane zostały jedynie metody znajdujące się w warstwie danych.

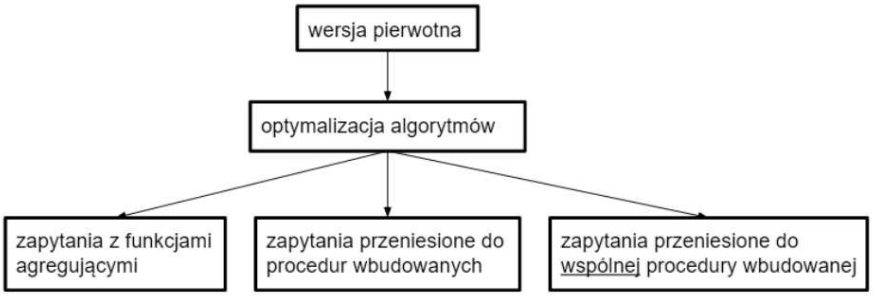
5 Przeprowadzane testy

Badane wersje aplikacji

Celem badania było ustalenie, która z zastosowanych technik odniosła najbardziej znaczący, pozytywny skutek dla wydajnosci aplikacji. Przygotowano zatem szereg wersji aplikacji różniących się istotnie zmianami w kodzie źródłowym, według następującego schematu:

- pierwotna wersja aplikacji, oznaczana *branch_v_0_init*;
- wersja aplikacji ze zmodyfikowanym algorytmem, oznaczana *branch_v_1_alg*;
- podwersje aplikacji ze zmodyfikowanym algorytmem oraz wprowadzonymi zmianami sposobu odwołania do bazy danych:
  - wykorzystująca funkcję agregującą w kwerendzie wykonywanej przez aplikację, oznaczana *branch_v_2_query*;
  - wykorzystująca procedurę składowaną, wykonywaną dwukrotnie w celu porównania wyników dwóch szkół, oznaczana *branch_v_2_proc1S;*
  - wykorzystująca procedurę składowaną, obliczająca w jednym wywołaniu wyniki dwóch szkół, oznaczana *branch_v_2_proc2S*.

Ponadto dla każdej z tak uzyskanych wersji stosowano modyfikacje związane z wykorzystaniem dodatkowych mechanizmów zwiększania wydajności: przetwarzania równoległego (dodatkowy przyrostek P) oraz buforowania (dodatkowy przyrostek C). Dla przykładu wersja aplikacji ze zmodyfikowanym algorytmem, ale stosująca tylko standardowe kwerendy, za to z włączonym buforowaniem, nosi oznaczenie *branch_v_1_alg_C*.



Rys. 7. Drzewo wersji aplikacji

Scenariusze testów

Wszystkie warianty aplikacji zostały poddane identycznym testom z wykorzystaniem aplikacji Apache JMeter. Każdy z testów polegał na zgłoszeniu do aplikacji 60 żądań porównania wyników dla dwóch szkół. Kolejne żądania mogły dotyczyć tej samej pary szkół (być powtarzalne) lub losowo wybranej pary szkół. Ponadto żądania mogły być zgłaszane w jednym wątku (sekwencyjnie) lub w 4 równoległych wątkach, odpowiadających równoczesnemu korzystaniu z aplikacji przez 4 użytkowników. Dobór liczby wątków nie jest tu przypadkowy – testy były przeprowadzane z użyciem procesora zdolnego do jednoczesnej obsługi takiej właśnie liczby wątków. Podsumowując, dla każdej z 15 wersji aplikacji przeprowadzane były 4 testy o różnych scenariuszach – kombinacjach wspomnianych ustawień. Jako wynik testu w danym scenariuszu traktowana była średnia wartość czasu realizacji pojedynczego żądania. Ponadto wszystkie testy przeprowadzane były na identycznym zbiorze danych wejściowych obejmujących 50 szkół, dla każdej z nich w bazie znajdowało się 300 zestawów odpowiedzi dla identycznej ankiety obejmującej 7 pytań.

6 Wyniki i wnioski

Wyniki testów dla wszystkich wersji aplikacji zostały zestawione w Tabeli 1.

Tabela. 1. Wyniki przeprowadzonych testów [ms]

	Żądania powtarzalne		**Żądania losowe**	
	1 wątek	**4 wątki**	**1 wątek**	**4 wątki**
branch_v_0_init	16477	19530	18997	21673
branch_v_1_alg	9941	12639	12538	14598
branch_v_2_query	10035	12499	12842	14899
branch_v_2_proc1S	9991	12469	12645	14862
branch_v_2_proc2S	10049	12490	12636	15073
branch_v_0_init_P	5260	19201	5746	21824
branch_v_1_alg_P	3257	13995	3873	15962
branch_v_2_query_P	3221	13690	3899	16671
branch_v_2_proc1S_P	3256	13779	3904	15753
branch_v_2_proc2S_P	3156	14423	3893	15554
branch_v_0_init_C	5929	6474	11477	12715
branch_v_1_alg_C	232	282	4663	5657
branch_v_2_query_C	224	273	5041	5994
branch_v_2_proc1S_C	231	276	4899	6400
branch_v_2_proc2S_C	223	275	8707	15474

Wnioskiem, który nasuwa się jako pierwszy, jest zasadnicze znaczenie przeprowadzonej jako pierwszy krok optymalizacji algorytmu, która prowadziła przede wszystkim do zmniejszenia odwołań do źródeł danych. Zmiana ta powodowała zawsze redukcję czasu realizacji żądań o 30-40% niezależnie od scenariusza testowego oraz ewentualnych dodatkowych mechanizmów.

Jeżeli chodzi o zastosowanie buforowania, to można zauważyć, że przynosi ono zawsze korzystny efekt dla pierwotnej wersji aplikacji. Wynika to z faktu, że aplikacja przed optymalizacją algorytmu wykonywała zbędne wielokrotne pobieranie tych samych danych, co sprzyjało skuteczności buforowania. Natomiast po zoptymalizowaniu algorytmu korzystny wpływ buforowania jest widoczny tylko w przypadku, gdy wykonywane są powtarzalne żądania, zaś w przypadku żądań losowych narzut wymagany do obsługi buforowania, przy braku jego realnego wykorzystania, wręcz prowadzi do zwiększenia czasu realizacji żądania. Dodatkowo można zauważyć, że przeniesienie części obliczeń na bazę danych, czy to w formie funkcji użytej w kwerendzie czy w formie procedur składowanych, nie wpływa na czas realizacji

żądań. Oba te spostrzeżenia pośrednio dowodzą, że przeprowadzona w pierwszym kroku optymalizacja algorytmu była skuteczna.

Ciekawej obserwacji można też dokonać porównując wyniki przy zastosowaniu wewnątrz aplikacji przetwarzania współbieżnego. W przypadku, gdy żądania były prowadzone sekwencyjnie, zastosowanie przetwarzania współbieżnego prowadziło do redukcji czasu realizacji żądań o ponad 50%. Potwierdza to, że sam algorytm jest podatny na paralelizację. Jednak w przypadku aplikacji działających w kontenerach sieciowych, takich jak badana aplikacja, należy pamiętać o tym, że sam kontener wprowadza już przetwarzanie wielowątkowe żądań, i bardzo prawdopodobnym jest, że liczba jednocześnie przetwarzanych żądań przekroczy liczbę wątków obsługiwanych jednocześnie przez procesor. Jeżeli dodatkowo spowodujemy multiplikację wątków poprzez wprowadzenie przetwarzania współbieżnego wewnątrz danego żądania, to nie tylko nie osiągniemy korzyści, lecz wręcz możemy doprowadzić do pogorszenia wydajności aplikacji, które jest widoczne wszędzie w przypadku, gdy scenariusz testowy uwzględniał wielowątkowe zgłaszanie żądań.

Bibliografia

[1] Ostrowski M., Redukowanie czasu realizacji żądań przez wielowarstwową aplikację sieciową, Praca dyplomowa magisterska pod kierunkiem dra inż. Michała Karbowańczyka, 2017

[2] Plattner H., Zeier A., In-Memory Data Management: Technology and Applications. Springer Science and Business Media, 2012

[3] Barish G., Building Scalable and High-performance Java Web Applications Using J2EE Technology. Addison-Wesley, 2002

[4] Holub A., Taming Java Threads. Apress, 2000

[5] Lea D., Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley, 2000

[6] Should i always use a parallel stream when possible? http://stackoverflow.com/questions/20375176/shouldi-always-use-a-parallel-stream-when-possible, [dost. 2017-04-06]

[7] What's wrong in java 8, part iii: Streams and parallel streams. https://dzone.com/articles/whatswrong-java-8-part-iii, [dost. 2017-04-06]

[8] Shirazi J., Java Performance Tuning. O'Reilly Media, Inc., 2003

[9] Müller M., Java Lambdas and Parallel Streams. Apress, 2016

[10] Warburton R., Java 8 Lambdas: Pragmatic Functional Programming. O'Reilly Media, 2014

[11] Monson-Haefel R., Burke B., Enterprise JavaBeans 3.0. O'Reilly Media, 2006

[12] Albertoni F. i in., WebSphere Application Server V8.5 Concepts, Planning, and Design Guide. IBM Redbooks, 2013
[13] Shivakumar S. K., Architecting High Performing, Scalable and Available Enterprise Web Applications. Morgan Kaufmann, 2014
[14] Why use your application-level cache if database already provides caching? http://stackoverflow.com/questions/2963819/why-use-your-application-level-cache-if-databasealready-provides-caching, [dost. 2017-03-30]
[15] England K., Stanley N., The SQL Server 7.0 Handbook: A Guide to Microsoft Database Computing. Digital Press, 1999
[16] Millsap C., Holt J., Optimizing Oracle Performance: A Practitioner's Guide to Optimizing Response Time. O'Reilly Media, 2003
[17] Ostrowski M., Porównywarka liceów ogólnokształcących bazująca na ocenach przedmiotów wystawionych przez absolwentów w technologii Spring. Praca dyplomowa inżynierska pod kierunkiem dra inż. Mateusza Smolińskiego, 2016

REDUCTION OF REQUEST PROCESSING TIME BY MULTITIER NETWORK APPLICATION

Summary – In this article possible techniques for optimizing a multitier network application, in terms of reducing the time of request execution, are considered. The considered application has been subjected to modifications both in terms of algorithms and techniques for operating in the database and the use of support mechanisms, such as parallel processing and buffering. The obtained results provide an indication of the desirability of using specific techniques for considered type of application. The article is a digest of Master thesis [1]

Keywords: network applications, multitier applications, optimization