AUTOMATYKA • 2005 • Tom 9 • Zeszyt 3

Konrad Grzanek[*], Rafał Grzybowski[*, **]

Metody przechowywania danych w systemie rozpoznawania wzorców projektowych w oprogramowaniu

1. Wstęp

W dobie gwałtownego rozwoju branży IT i masowego powstawania systemów informatycznych wzrastają potrzeby związane z oceną jakości oprogramowania. Podstawowym problemem jest niemożliwość utrzymania jakości systemów informatycznych na odpowiednio wysokim poziomie wraz z ich rozwojem i rozbudową. Problem ten jest szeroko opisywany, m.in. w [5, 6]. Symptomatyczne jest to, że jakość oprogramowania nabiera znaczenia właśnie na etapie rozwoju systemów, a niekoniecznie implementacji. W dobie dobrze zaprojektowanych języków programowania o precyzyjnej składni i semantyce najważniejszym elementem procesu tworzenia oprogramowania jest projekt. Złe praktyki towarzyszące procesowi projektowania oprogramowania przenoszą się na system jako całość i mają swoje zasadnicze negatywne znaczenie w trakcie jego rozbudowy.

Dla poznania struktury powstającego oprogramowania, na tyle dogłębnego, by możliwe było zidentyfikowanie miejsc wystąpień nieprawidłowości projektowych lub implementacyjnych, konieczne jest stworzenie automatów zdolnych do wykonania gruntownej analizy kodu źródłowego stanowiącego to oprogramowanie. Główną motywacją jest tutaj złożoność systemów informatycznych, która powoduje, że przeglądy kodu dokonywane w zespołach projektowych przez ich członków w celach wyszukania nieprawidłowości są kosztowne i nie gwarantują sukcesu.

Równoległym zagadnieniem jest kwestia identyfikacji wzorców projektowych w oprogramowaniu. Koncepcja wzorców projektowych (*design patterns*) sięga swoimi korzeniami działalności słynnego tzw. GoF (*Gang of Four*) [3]. Opisy i katalogi wzorców projektowych, jak również sposoby ich graficznego przedstawiania są obecnie dostępne dla niemalże każdej dziedziny i technologii związanej z oprogramowaniem. W szczególności możemy tu przytoczyć wzorce projektowe dla platform Java [8] i J2EE [5]. Stosowanie określonych wzorców projektowych w projektach programistycznych jest uznawane w społecznościach twórców oprogramowania za przesłankę do powstawania wysokiej jakości produktów, za

* Zakład Metod Przetwarzania Informacji, Wyższa Szkoła Humanistyczno-Ekonomiczna, Łódź
** Instytut Informatyki, Politechnika Łódzka

wyróżnik dobrego stylu programisty i kompetencji przedsiębiorstwa, które takich programistów zatrudnia. Z kolei brak powtarzalności działań programistycznych objawiający się w postaci odrzucenia tworzenia modeli opartych na wypracowanych wzorcach lub – co gorsza – pojawianie się charakterystycznych, powtarzalnych podobnie jak wzorce błędów, zwanych antywzorcami [6], prowadzi do wielu niekorzystnych zjawisk występujących w przedsiębiorstwie. Do zjawisk tych zaliczamy przede wszystkim niską efektywność programistów tworzących oprogramowanie, kłopoty w trakcie rozbudowy systemów i trudne do zidentyfikowania i naprawy problemy z samym oprogramowaniem, pociągające wzrost kosztów, problemy z terminowością itd.

Autorzy niniejszej pracy działają na polu oceny jakości oprogramowania i wyszukiwania plagiatów od 2003 roku. W wyniku prac nad systemem wykrywania plagiatów w oprogramowaniu obiektowym [2], opartym o metodę opisaną w [1], powstała idea wykorzystania danych pochodzących z analizy statycznej kodu źródłowego do wykrywania wystąpień wzorców projektowych.

W dalszej części artykułu zostaną przedstawione zagadnienia związane z implementacją warstwy zarządzającej tymi danymi.

2. Opis problemu

Analiza statyczna oprogramowania polega na wykonywaniu czynności analitycznych na kodzie źródłowym, bez kompilacji i uruchamiania systemu. W wyniku tej analizy gromadzone są szczegółowe dane o tym, jak zbudowane są poszczególne jednostki kompilacji lub interpretacji analizowanego programu. Wyodrębnieniu podlegają wszystkie konstrukcje składniowe występujące w każdej jednostce kompilacji. Analizator statyczny działa więc na zasadzie zbliżonej do kompilatora języka programowania, w którym zaimplementowany jest badany system.

Największym problemem analizy statycznej kodu źródłowego w zastosowaniach związanych z wykrywaniem wzorców jest ogromna ilość danych powstających w jej trakcie. Istotność tych danych może być zróżnicowana z punktu widzenia działania algorytmu wyszukującego i dodatkowo różnice te objawiają się dla różnych strategii wyszukiwania, odpowiadających różnym kryteriom poszukiwań. Dlatego nie możemy założyć odrzucenia części z tych danych na etapie ich gromadzenia. Wszystkie one powinny być gromadzone dla potrzeb późniejszego wyszukiwania wzorców.

Rozmiar danych możemy oszacować, biorąc pod uwagę kilka czynników. Po pierwsze tworzone obecnie systemy są budowane w oparciu o gotowe, dostępne do wykorzystania:

– ramy (*frameworks*), takie jak J2EE, Microsoft.NET, Struts, Springframework;
– biblioteki programistów;
– gotowe serwery aplikacji i związane z nimi komponenty wielokrotnego użycia.

Analizując dany system, musimy więc brać pod uwagę również zależności od innych fragmentów oprogramowania, które staje się automatycznie kolejnym podmiotem analizy.

Ponadto stwierdzić należy, że duże systemy informatyczne składają się z wielu setek lub nawet tysięcy jednostek kompilacji. Każda jednostka kompilacji może się z kolei składać z kilkuset lub (w przypadku niektórych języków programowania) elementów semantycznych, które będą odgrywały rolę na etapie analizy.

Badania przeprowadzone przez nas na kodzie źródłowym platformy Java 2 SDK w wersji 1.5.0 potwierdziły, że w trakcie analizy kodów źródłowych biblioteki standardowej należącej do tej platformy, powstaje ok. 3,9 miliona elementów.

Ta ilość danych, które muszą zostać zgromadzone, powoduje, że prowadzone są badania nad metodami, które korzystają w ograniczonym stopniu z analizy statycznej. Przykładem może tu być chociażby system FUJABA i jego moduł wyszukiwania wzorców na bazie analizy czasu wykonania, czyli w trakcie działania programu [4].

Analiza statyczna ma jednak duże zalety w stosunku do analizy czasu wykonania, co powoduje, że pozostajemy przy koncepcji jej wykorzystania, gdyż:

- pozwala zachować niezależność od narzędzi analitycznych czasu wykonania, tj. debuggerów, profilerów i specyficznych danych produkowanych przez maszyny wirtualne;
- pozwala przeanalizować wszystkie bez wyjątku elementy systemu, w tym również te, które nie przejmują sterowania w trakcie działania systemu; jest to ogromna zaleta w porównaniu z analizą *runtime*.

Zagadnienie związane z wyborem odpowiedniej metody zarządzania danymi i szybkiego ich wyszukiwania jest więc zagadnieniem kluczowym z punktu widzenia odniesienia końcowego sukcesu.

3. Kryteria wyboru właściwego sposobu przechowywania, zarządzania i wyszukiwania danych

Poszukując właściwej metody zarządzania danymi pochodzącymi z naszych analiz przyjęliśmy niżej wymienione kryteria.

- Zdolność do efektywnego wstawiania i wybierania wielu dziesiątków lub setek milionów rekordów.
- Niezawodność działania w warunkach dużego obciążenia systemu.
- Dobra skalowalność i stabilność działania wraz z przyrostem danych.
- Zdolność do pracy na ogólnie dostępnych platformach sprzętowych, czyli w środowiskach stacji roboczych i na komputerach domowych.
- Łatwe w użyciu API umożliwiające dostęp do danych
- W miarę możliwości niezależność od systemu operacyjnego. Jest to kryterium związane z systemem wyszukiwania wzorców jako finalnym produktem dla menadżerów i projektantów systemów.

Ponieważ dane, którymi zarządzamy, mają charakter analityczny, nie ma konieczności stosowania transakcyjności. Z naszego punktu widzenia jest ona nawet niewskazana, jako element niepotrzebnie obciążający.

4. Analizy znanych metod zarządzania danymi

W celu dokonania wyboru jak najefektywniejszego modułu zarządzania danymi, poddaliśmy analizie niżej wymienione komponenty:

- Berkeley DB JE 1.7 – baza danych oparta o koncepcję B-drzewa, wersja na platformę Java [10] [11]. Testowana na maszynie wirtualnej dla systemu Linux oraz dla Windows – test w systemie Windows XP (BDB JE WinXP).
- Berkeley DB 4.3 C++ Btree – baza danych oparta o koncepcję B-drzewa, wersja kompilowana do kodu rodzimego, dla języka C++.
- Berkeley DB 4.3 C++ hash – baza danych oparta na funkcji mieszającej, wersja kompilowana do kodu rodzimego, dla języka C++.
- MySQL 4.1 – relacyjna baza danych z dostępem z poziomu platformy Java poprzez JDBC.

Testy wydajnościowe zostały przeprowadzone na następującej platformie sprzętowej:

- procesor: Athlon 1.3 GHz,
- pamięć RAM: 785 MB,
- dysk twardy: Seagate Barracuda ST317221A (17.2 GB, 5400 obr./min).

Systemem operacyjnym, pod którego kontrolą działało oprogramowanie testowe, był Linux, jądro 2.6.8, glibc 2.3.3 oraz dla testu BDB JE WinXP – Windows XP Professional.

Wykorzystaliśmy ponadto:

- kompilator GNU C/C++ 3.4.1 (BDB C++ hash i BTree),
- Java 2 SDK 1.4.2_04 dla systemu Linux (BDB JE),
- Java 2 SDK 1.4.2._08 dla systemu Windows (BDB JE WinXP).

Biblioteka Berkeley DB 4.3 została skompilowana z domyślnymi opcjami optymalizacji (–O2).

Kody programów testowych stworzone w języku C++ zostały skompilowane opcjami konfiguracyjnymi:

- O2,
- Wall,
- Pedantic,
- Fomit,
- Frame,
- Pointer.

Maszyna wirtualna Javy dla programów testowych była uruchamiania z następującymi opcjami [7]:

- Java,
- Server,
- Xmx256m,
- Xms256m.

Staraliśmy się, aby procedura testowa zaimplementowana w poszczególnych programach odpowiadała realnym warunkom działania naszego systemu. W bazie gromadzone były rekordy składające się z klucza będącego liczbą całkowitą podwójnej precyzji (*long*) oraz z literału znakowego (*string*) długości 37.

Test składał się z następujących etapów:

1. Wstawianie 4 mln rekordów do pustej początkowo bazy. W trakcie przebiegu gromadzone były informacje o czasach wstawiania kolejnych 10 tys. rekordów (tab. 1–5).
2. Zatrzymanie systemu, uruchomienie i wstawienie kolejnych 2 mln rekordów (z zakresu 4–6 mln). W trakcie przebiegu gromadzone były informacje o czasach wstawiania kolejnych 10 tys. rekordów.
3. Zatrzymanie systemu, uruchomienie i wstawienie kolejnych 2 mln rekordów (z zakresu 6–8 mln). W trakcie przebiegu gromadzone były informacje o czasach wstawiania kolejnych 10 tys. rekordów.
4. Przetworzenie 100 tys. zapytań o dane przy użyciu kluczy generowanych losowo, o których wiedzieliśmy, że istnieją w bazie. W trakcie tego testu były gromadzone informacje o czasach przetworzenia kolejnego tysiąca zapytań.
5. Przetworzenie 100 tys. zapytań o dane przy użyciu kluczy generowanych losowo, o których wiedzieliśmy, że nie istnieją w bazie. W trakcie tego testu były gromadzone informacje o czasach przetworzenia kolejnego tysiąca zapytań.

Etapy 1–3 odpowiadają symulowanym importom danych pochodzących z analizy statycznej kolejnych trzech rozległych projektów. Poniżej prezentujemy wyniki testu (tab. 1–5).

Tabela 1

Czasy wstawiania grup 10 tys. rekordów w trakcie przetwarzania pierwszych 4 mln rekordów – wyniki w ms

	BDB JE	BDB Btree C++	BDB hash C++	Java + MySQL	BDB JE WinXP
ŚREDNIA	1071,01	596,4	2811,21	1986,36	472,30
MEDIANA	812	618	901	1797	375
MAX	7079	1336	35133	4011	2875

Tabela 2

Czasy wstawiania grup 10 tys. rekordów w trakcie przetwarzania od 4 do 6 mln rekordów – wyniki w ms

	BDB JE	BDB Btree C++	BDB hash C++	Java + MySQL	BDB JE WinXP
ŚREDNIA	2139,68	1827,53	11841,55	1974,11	472,19
MEDIANA	1855	649	1073,5	1825	382,5
MAX	6998	17262	55748	4393	3860

Tabela 3
Czasy wstawiania grup 10 tys. rekordów w trakcie przetwarzania od 6 do 8 mln rekordów – wyniki w ms

	BDB JE	BDB Btree C++	BDB hash C++	Java + MySQL	BDB JE WinXP
ŚREDNIA	2173,26	4098,7	Brak danych	1972,6	477,34
MEDIANA	1846,5	654	Brak danych	1823,5	391
MAX	6528	48907	Brak danych	4322	2844

Tabela 4
Czasy wyszukiwania grup 1000 rekordów w trakcie przetwarzania 100 tys. zapytań po istniejących losowo wybranych kluczach – wyniki w ms

	BDB JE	BDB Btree C++	BDB hash C++	Java + MySQL	BDB JE WinXP
ŚREDNIA	13556,02	1920,1	819,74	260,3	1417,96
MEDIANA	13335,5	1614	835	209	781
MAX	23209	6838	1509	829	10109

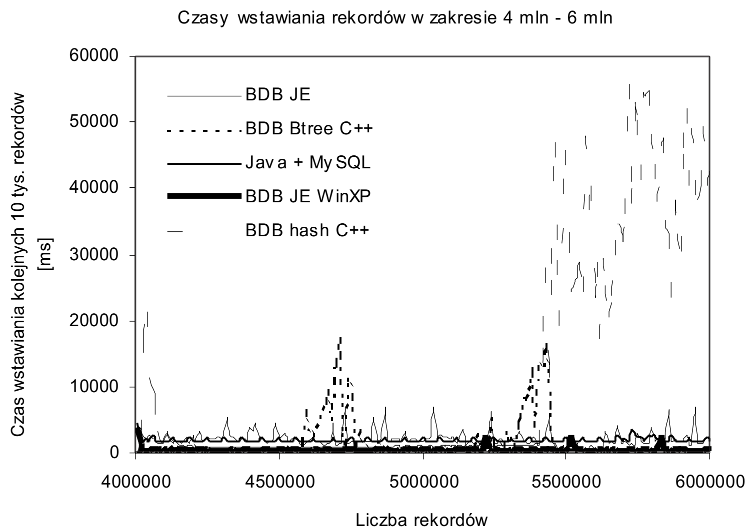
Tabela 5
Czasy wyszukiwania grup 1000 rekordów w trakcie przetwarzania 100 tys. zapytań po nieistniejących losowo wybranych kluczach – wyniki w ms

	BDB JE	BDB Btree C++	BDB hash C++	Java + MySQL	BDB JE WinXP
ŚREDNIA	112,35	70,68	1008,82	215,68	46,24
MEDIANA	89	41	961	171	16
MAX	655	448	1915	806	375

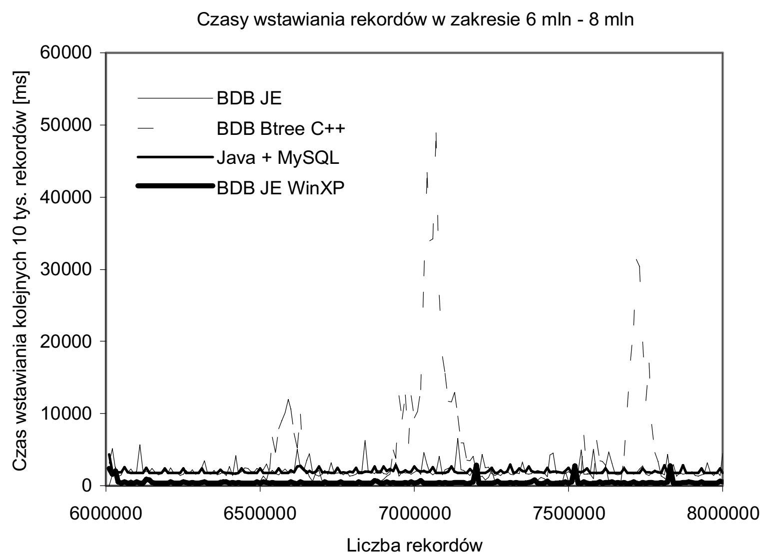
Z punktu widzenia stabilności procesu wstawiania rekordów (rys. 1 i 2) najlepiej prezentują się: Berkeley DB JE w środowisku Linux i Win32 oraz baza MySQL. One też uzyskują najlepsze czasy wykonania. Znamienny jest fakt, że rozwiązanie kompilowane (C++) przegrywa z maszyną wirtualną. Wersja BDB 4.3 z indeksem opartym o funkcję mieszającą w ogóle nie przetworzyła ostatnich 2 mln wstawień.

W teście wyboru rekordów (rys. 3) wygrywa bezapelacyjnie baza MySQL. BDB JE 1.7 w systemie Windows XP jest prawie czterokrotnie wolniejsze, ale niestety tak doskonały wynik bazy MySQL ma miejsce wyłącznie na w systemie Linux. Jak pokazały badania przeprowadzone niezależnie od przytoczonych tutaj przez jednego z autorów, w środowi-

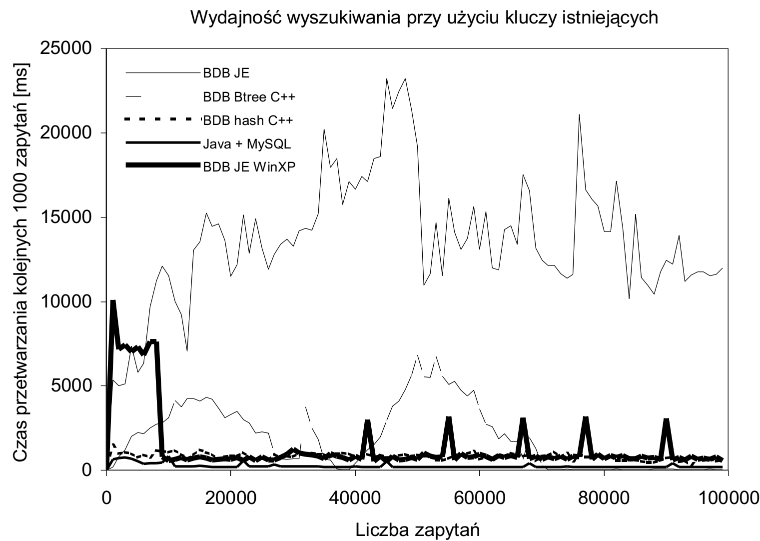
sku Win32 MySQL działał zatrważająco powoli – proces wstawiania rekordów trwał kilkadziesiąt razy wolniej, niż to ma miejsce w wersji dla systemu Linux. Niestety zachowanie to jest potwierdzane przez programistów i administratorów tego systemu.



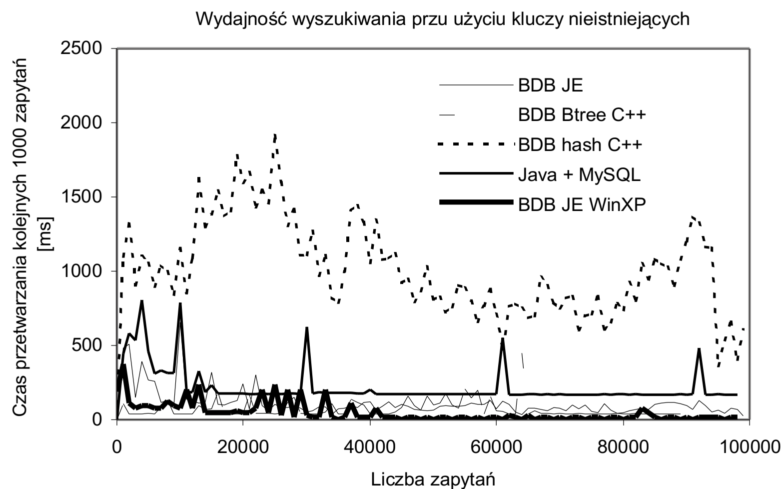
Rys. 1. Przebieg wstawiania rekordów w zakresie 4–6 mln



Rys. 2. Przebieg wstawiania rekordów w zakresie 6–8 mln



Rys. 3. Przebieg wyszukiwania rekordów z wykorzystaniem kluczy istniejących



Rys 4. Przebieg wyszukiwania rekordów po nieistniejących kluczach

W przypadku wyszukiwania po kluczach nieistniejących (rys. 4) zdecydowanym liderem jest baza Berkeley DB JE w systemie Windows. Pozostałe rozwiązania wypadają znacznie gorzej. Weryfikacja istnienia rekordów w bazie jest niezmiernie ważna z punktu widzenia procesów importu danych do systemu. W trakcie tego importu rozwiązywane są zależności pomiędzy różnymi elementami analizowanego systemu, w tym analizowane są

odwołania do tych elementów, które albo w ogóle nie istnieją w składnicy danych, albo zostaną osiągnięte dopiero w kolejnych fazach procesu importowego.

W całym przebiegu testów znamienne jest to, że rozwiązanie kompilowane do kodu rodzimego w znaczący sposób przegrywa z bazą, która jest uruchamiana na maszynie wirtualnej. Wyjaśnienie tego faktu nie jest łatwe, zjawisko może mieć bowiem różne źródła. Nasuwającym się powodem może być np. znaczne pofragmentowanie pamięci w przypadku procesu uruchamianego natywnie. Problemy tego typu występują szczególnie tam, gdzie mamy do czynienia z wieloma faktami alokacji i zwalniania pamięci [9], co może mieć miejsce przy przetwarzaniu wielkiej ilości rekordów. Kolejnym potencjalnym powodem może być różnica implementacji bazy Berkeley DB dla platformy Java i jej kompilowanego protoplasty [10].

Osobnym tematem jest znaczna przewaga wydajności i stabilności bazy Berkeley DB JE uruchamianej na maszynie wirtualnej dla systemu Windows wobec tego samego systemu uruchamianego w środowisku Linux. Uzasadnieniem może być fakt, że JVM dla systemu Windows jest najbardziej zoptymalizowaną wersją maszyny wirtualnej.

5. Podsumowanie

Na podstawie przeprowadzonych badań stwierdzamy, że najkorzystniejszą platformą, z punktu widzenia postawionych wcześniej kryteriów, jest baza Berkeley DB JE 1.7 uruchamiania na maszynie wirtualnej Javy dla środowiska Windows. Dalsze prace prowadzone będą właśnie w oparciu o ten moduł bazy danych. Oprócz korzyści jakościowych zyskujemy pełną przenośność między platformami systemowymi oraz możliwość osadzenia bazy danych w procesie realizującym algorytm.

Literatura

[1] Chidamber S.R., Kemerer C.F.: *A Metrics Suite for Object Oriented Design*. M.I.T. Sloan School of Management, December 1993

[2] Grzybowski R., Grzanek K.: *Zastosowanie miar oprogramowania obiektowego do wykrywania plagiatów i do oceny jakości kodu źródłowego przy wykorzystaniu miar WMC i NOM*. Environmental Mechanics Methods of Computer Science, Simulations, Lwiv 2004

[3] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns*. ISBN 0201633612

[4] Wendehals L.: *Improving Design Pattern Instance Recognition by Dynamic Analysis*. FINITE project by German Research Foundation, SCHA 745/2-1

[5] Alur D., Crupi J., Malks D.: *Core J2EE Patterns: Best Practices and Design Strategies. 2nd ed*. HELION, 2004

[6] Dudney B., Asbury S., Krozak J.K., Wittkopf K.: *J2EE AntiPatterns*. Wiley Publishing Inc., 2003

[7] *Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine*. Sun Microsystems, 2003

[8] *Java[tm] Language Specification*. Sun Microsystems, 2003

[9] Wilson P.R., Johnstone M.S., Neely M., Betes D.: *Dynamic Storage Allocation: A Survey and Critical Review*. Proc. Int'l Workshop on Memory Management, Kinross Scotland, Springer Verlag LNCS, 1995

[10] *Getting Started with Berkeley DB Java Editio*n. Sleepycat Software, 2004, http://www.sleepycat.com
[11] *Berkeley DB Java Edition Collections Tutorial*. Sleepycat Software, 2004, http://www.sleepycat.com
[12] *Getting Started with Berkeley DB for C++*. Sleepycat Software, 2004, http://www.sleepycat.com
[13] *Getting Started with Berkeley DB XML for C++*. Sleepycat Software,2004, http://www.sleepycat.com