Różne sposoby zapewnienia trwałości obiektom.

Paweł Chodarcewicz

Przedstawienie problemu

Chcielibyśmy mieć mechanizm, który pozwoliłby programowi obiekowemu:

- zachować stan programu między kolejnymi jego uruchomieniami (np. *zapisanie stanu gry*);
- zapisywać dane programu w sposób umożliwiający ich późniejsze użycie (np. *program z listą zakupów*).

Podstawowe wymagania

Trwałość obiektów (danych) – możliwość trwałego zapisywania obiektów (danych) w celu późniejszego ich odczytania (nawet po ponownym uruchomieniu aplikacji).

Niezawodność – gwarancja, że jesteśmy w stanie odtworzyć obiekty (dane), które wcześniej zapisaliśmy.

Plan prezentacji

- HSQL + JDBC;
- Hibernate / JDO;
- serializacja;
- Prevlayer;
- Db4o;
- porównanie wydajności poszczególnych technologii.

O czym nie będzie

- Przedstawione zostaną jedynie mechanizmy dostępne dla Javy (jako przedstawiciela programowania obiektowego).
- Nie będzie szczegółowego omówienia API poszczególnych produktów…
- … ani dogłębnej analizy przedstawionych fragmentów kodu.

Kategorie oceny poszczególnych technologii

- Jak się ma model obiektowy programu do modelu w jakim przechowywane są dane.
- Mechanizm zadawania zapytań oraz pobierania obiektów.
- Transakcje.
- Skalowalność, szybkość działania.
- Przenośność i elastyczność.
- Łatwość korzystania i konfiguracji.

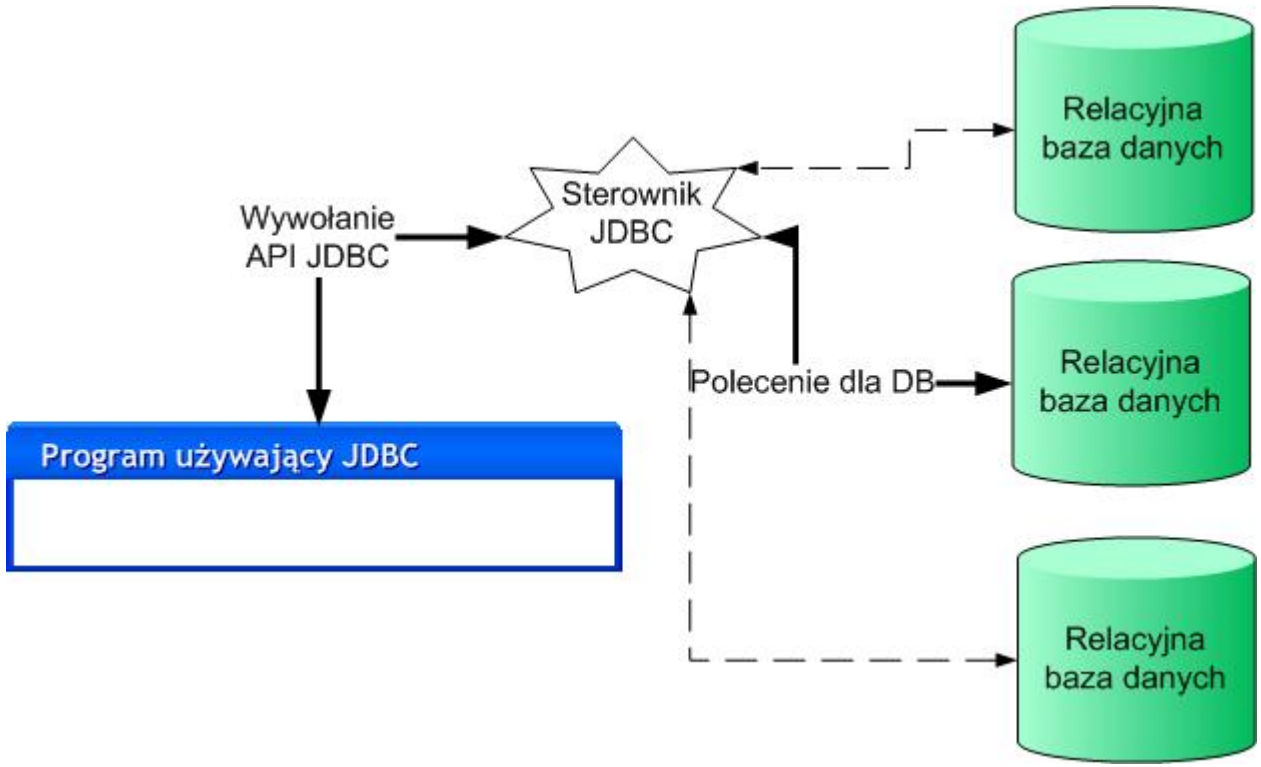
Rozwiązanie klasyczne

Użycie bazy danych

(np. HSQL, Postrges, Oracle)

- Połączenie się z bazą z poziomu programu w Javie za pomocą JDBC.
- używanie języka SQL (dalekiego od obiektowości) do zapisywania/uaktualniania/odczytywania obiektów.
- przekonwertowanie modelu relacyjnego na model obiektowy.

Schemat komunikacji z bazą danych przy użyciu JDBC



Zalety użycia relacyjnej bazy danych wraz z JDBC

- Szybkość działania (brak dodatkowych warstw pośrednich).
- Rozwiązanie sprawdzone – używane od wielu lat.
- Ogromna siła wyrazu języka zapytań SQL.
- Rozwiązanie niezależne od konkretnej bazy danych.

Wady użycia relacyjnej bazy danych wraz z JDBC

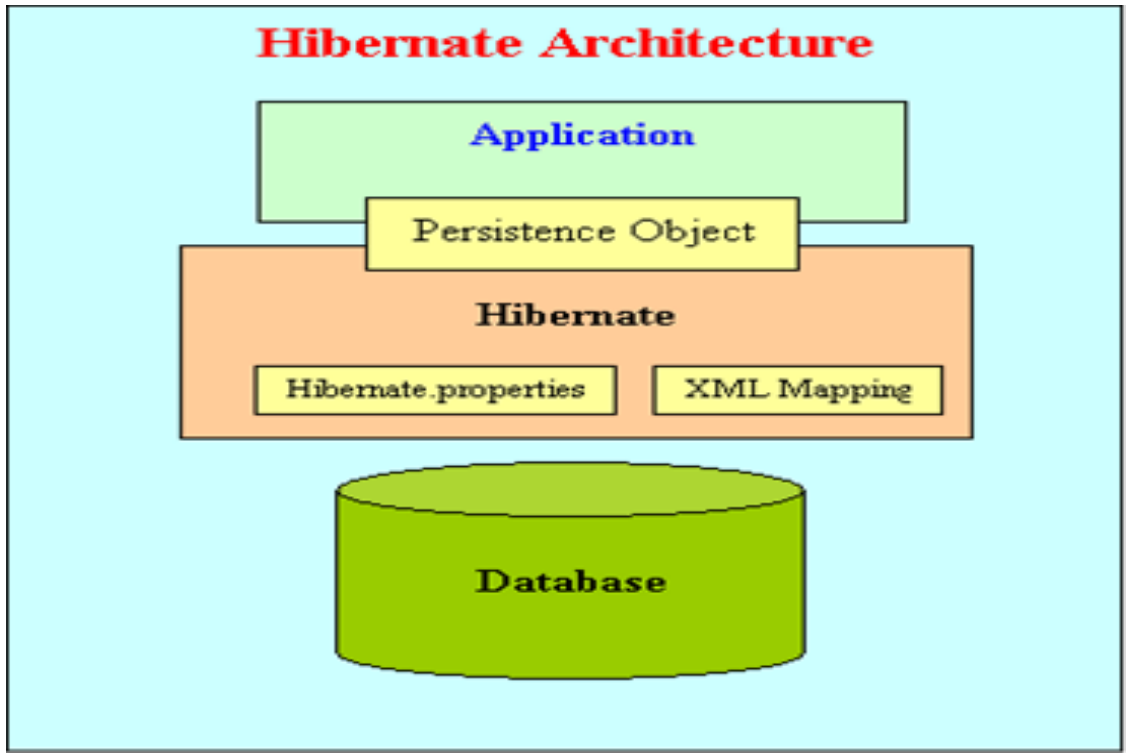
- Konieczność radzenia sobie z podwójnym modelem danych: obiektowym i relacyjnym („ręczna” konwersja z jednego do drugiego).
- Powstaje mnóstwo kodu związanego z JDBC i konwersją modeli danych.
- Model relacyjny nie zachowuje kapsułkowania, polimorfizmu, dziedziczenia.
- Trudności w utrzymywaniu projektu (zmiana modelu obiektowego pociąga konieczność zmian w modelu relacyjnym).

Automatyczne mapowanie obiektów do relacji (na przykładzie Hibernate)

Pomysł: ***Używamy programu (biblioteki), którego zadaniem jest mapowanie obiektów do relacji.***

Pisząc kod naszego programu odnosimy wrażenie, że obiekty przechowywane są w formie natywnej, a nie relacyjnej.

Schemat działania programów mapujących obiekty do relacji na przykładzie Hibernate



Rysunek zaczerpnięty z http://hibernate.org/

Przykładowy kod programu wykorzystującego Hibernate – klasa do utrwalenia

```
public class Cat {
    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat(){};
    public Cat(String name, char sex, float weight) {
                this.name = name; this.sex = sex; this.weight = weight;
    }

    public String getId() { return id;}
    private void setId(String id) { this.id = id; }

      // pozostałe gettery i settery
      ……..
}
```

Przykładowy kod programu wykorzystującego Hibernate – wstawienie obiektu

```
Session session = HibernateUtil.currentSession();
Transaction tx = session.beginTransaction();

Cat princess = new Cat("Princess”, 'F‘, 7.4f);

session.save(princess);

tx.commit();
HibernateUtil.closeSession();
```

Przykładowy kod programu wykorzystującego Hibernate - zapytanie

```
Transaction tx = session.beginTransaction();

//zapytanie o wszystkie kotki (koty płci żeńskiej)
Query query = session.createQuery("from Cat as c where c.sex =
    :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Sposoby zadawania zapytań w Hibernate

- Za pomocą HQL (Hibernate Query Language) – język zaprezentowany w przykładzie – zbliżony do SQL.
- Za pomocą Hibernate Criteria Query – język opary w mniejszym stopniu o napisy, a w większym o obiekty.
- Za pomocą Query By Example – wyszukiwane są obiekty „podobne” do zadanego.
- Za pomocą standardowego SQL.

Przykładowy kod programu wykorzystującego Hibernate – plik odwzorowania

```
<hibernate-mapping>
    <class name="myClasses.Cat" table="CAT">
     <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
     </id>
     <property name="name">
            <column name="NAME" length="16" not-null="true"/>
     </property>
     <property name="sex"/>
     <property name="weight"/>
    </class>
</hibernate-mapping>
```

Czym muszą się charakteryzować obiekty utrwalane przez Hibernate?

- Muszą posiadać konstruktor bezparametrowy.
- Wszystkie pola, które mają zostać utrwalone muszą mieć zdefiniowane akcesory.
- Muszą posiadać pole identyfikatora (jeśli nie posiadają nie można korzystać z części funkcjonalności Hibernate).

UWAGA: zarówno wymagany konstruktor jak i akcesory nie muszą być publiczne!!!

W jaki sposób Hibernate dostaje się do danych przeznaczonych do utrwalenia?

Hibernate nie nakłada na klasy przeznaczone do utrwalenia obowiązku implementowania żadnego interfejsu, ani dziedziczenia z jakiejś klasy bazowej!

Hibernate zapewnia sobie dostęp do danych przeznaczonych do utrwalenia za pomocą mechanizmu refleksji (działającego w czasie wykonywania programu).

Zalety Hibernate

- Przezroczysta trwałość - użytkownik w swoim programie widzi jedynie świat obiektów.
- Niezależność od używanej bazy danych.
- Nie trzeba implementować żadnego interfejsu, czy dziedziczyć z jakiejś klasy.
- Nie ma prekompilacji ani postkompilacji.
- Potężny, przejrzysty, umożliwiający pełne wykorzystanie obiektowości język zapytań (np. pozwala na zapytania polimorficzne).

Wady Hibernate

- Konieczność pisania pliku mapowania.
- Zauważalny spadek wydajności.
- Klasy utrwalane muszą mieć akcesory do utrwalanych atrybutów.

JDO (Java Data Objects)

- Promowane przez SUN.
- Celuje w ten sam fragment rynku, co Hibernate.
- Rozwiązanie tylko dla Javy, ale za to przemyślane i dopasowane do języka.
- SUN wyspecyfikował sam interfejs – implementacja inne firmy.

JDO – zasada działania

Wymagana jest **postkompilacja** – po skompilowaniu program trzeba poddać procesowi „wzbogacenia” – modyfikuje ono klasy, które chcemy, aby były trwałe.

Dzięki temu nie musimy tworzyć akcesorów do atrybutów.

Hibernate, JDO – co wybrać?

- JDO szybsze, bo nie używa refleksji w czasie wykonywania.
- Hibernate ma większą społeczność – błędy naprawiane są szybciej.
- Oba projekty mają dobrze zorganizowane „zaplecze” (dokumentacja, wtyczki do Eclipsa…).
- Funkcjonalność podobno ta sama.
- Twórca Hibernate pracuje wraz ze specjalistami SUN’a nad JDO 2.0.

Najprostsze podejście nie korzystające z relacyjnej bazy danych – Serializacja

Serializacja obiektu – zapisanie obiektu w postaci np. strumienia bajtów. Oczywiście z tak otrzymanego strumienia można z powrotem stworzyć obiekt.

Trwałość możemy osiągnąć poprzez zapisanie zserializowanej formy obiektu do pliku. A następnie odczytanie w momencie, gdy obiekt ponownie będzie nam potrzebny.

Serializacja w Javie

```
public class Calendar implements java.io.Serializable{
    String username;
    CyclicList events;
}
Calendar myCalendar = new Calendar ();
// Write to disk with ObjectOutputStream
FileOutputStream f_out = new FileOutputStream("myobject.data");
ObjectOutputStream obj_out = new ObjectOutputStream (f_out);
obj_out.writeObject (myCalendar );

// Read from disk using ObjectInputStream
FileInputStream f_in = new FileInputStream("myobject.data");
ObjectInputStream obj_in = new ObjectInputStream (f_in);
Calendar myCalendar = (Calendar )obj_in.readObject();
```

Serializacja - zalety

- Bardzo prosty mechanizm – łatwy w użyciu.
- Szybkość działania.
- Jedyne, czego wymaga się od programisty przy deklaracji trwałych klas, to dodanie informacji, że klasa implementuje interfejs **Serializable.**
- Całkowita niezależność od zewnętrznych bibliotek, programów.
- Podejście czysto obiektowe.

Serializacja – wady

- Brak systemu zapytań, przeszukiwanie możliwe jedynie zwykłymi pętlami.
- Pliki z serializowanymi obiektami są bardziej narażone na awarie niż bazy danych (brak mechanizmu tworzenia kopii).
- Aby uniknąć utraty danych po awarii (która może zdarzyć się w nieoczekiwanym momencie) należałoby po każdej modyfikacji obiektu ponownie go serializować.

Prevayler – serializacja z pełnym bezpieczeństwem

Pomysł: trzymamy wszystkie obiekty biznesowe w pamięci RAM. Jednocześnie dbamy o ich bezpieczeństwo na wypadek awarii.

Snapshot – zrzut wszystkich utrwalanych obiektów z pamięci na dysk.

„Plik Komend” – zapis kolejnych poleceń modyfikujących obiekty trwałe.

Prevayler – przejaw geniuszu, czy narzędzie dla krótkowzrocznych?

- Ceny pamięci RAM są stosunkowo niskie – więc można mieć jej dużo.
- Przydatne tylko w sytuacjach, gdy mamy pewność, że zbiór trwałych obiektów zawsze będzie odpowiednio mały (aby mieścił się w pamięci).
- No ale co z bezpieczeństwem?

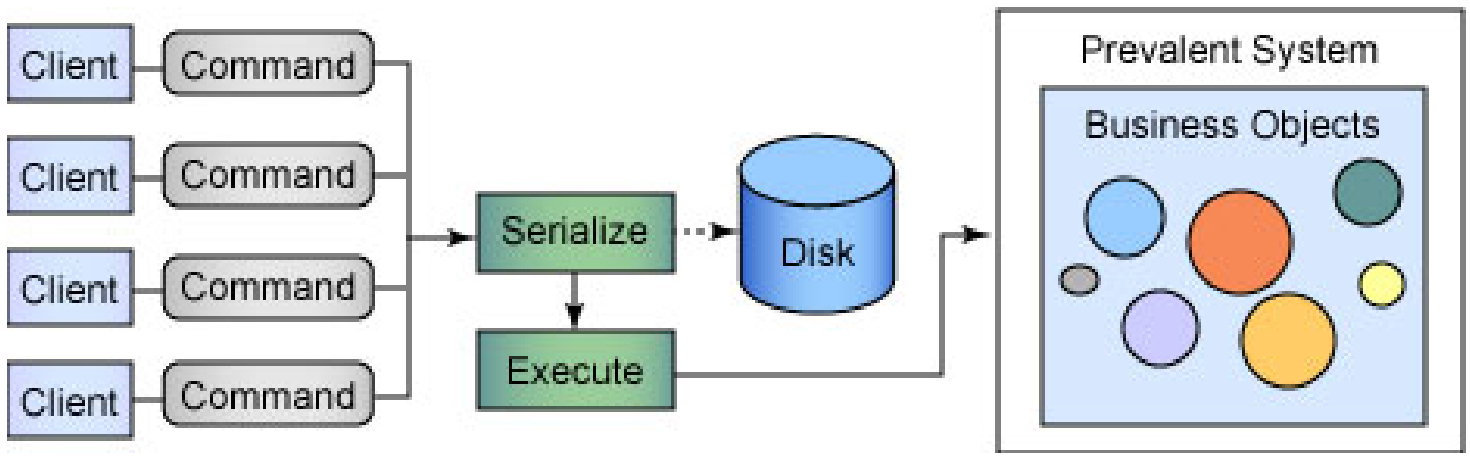
Prevayler – jak to działa?

- Każda komenda wydawana obiektowi trwałemu jest zapisywana w „pliku komend” a następnie aplikowana do tego obiektu.
- Co pewien czas (np. raz na godzinę) robiony jest automatycznie *snapshot* systemu – czyli zapisywany jest obecny stan wszystkich obiektów trwałych do pliku. „Plik Komend” jest wtedy czyszczony.

Można nakazać zrobienie *Snapshotu* na życzenie - w dowolnym momencie.

System nie musi być zatrzymywany!

Prevayler – ale jak to działa?



Prevayler – a co jeśli zabraknie prądu?

Prevayler potrafi odtworzyć stan obiektów sprzed awarii:

- wgrywany jest stan obiektów zapisany w ostatnim *snapshocie*;
- aplikowane są kolejno koemndy z „*pliku komend*”.

Prevayler – wymagania

- Obiekty biznesowe i operacje na nich muszą być **serializowalne**.
- Obiekty biznesowe muszą być **deterministyczne** (dla takich samych parametrów i stanu obiektu, metody obiektu muszą zmieniać go w dokładnie ten sam sposób) – problem z metodami zależnymi od np. czasu.

Prevayler – zalety

- Niezwykła szybkość działania (autorzy chwalą się, że Prevayler jest 3000 razy szybszy niż MySQL z JDBC !!!!!! [test składał się z kilkuset tysięcy małych zapytań]).
- Prosty w użytkowaniu.
- Mamy pełną wolność jeśli chodzi o dostęp do obiektów.

Prevayler – wady

- Brak wbudowanego narzędzie do zadawania zapytań (można jednak użyć np. JXPath).
- Ograniczona do wielkości RAM objętość gromadzonych obiektów.
- Obiekty muszą być deterministyczne.

JXPath – przykład

Kod Javy:

```
Address address = null;
Collection locations = vendor.getLocations();
Iterator it = locations.iterator();
while (it.hasNext()){
    Location location = (Location)it.next();
    String zipCode = location.getAddress().getZipCode();
    if (zipCode.equals("90210")){
      address = location.getAddress();
      break;
 }}
```

Równoważne wyrażenie JXPath:

```
Address address = (Address)JXPathContext.newContext(vendor).
        getValue("locations[address/zipCode='90210']/address");
```

Db4o – obiektowa baza danych – rozwiązanie niemal idealne

- Obiekty przechowywane są w bazie danych w formie natywnej – nie ma żadnych konwersji.
- Db4o przeznaczone jest wyłącznie dla Javy i .NET.
- Zmiana w klasie pociąga **automatyczną** zmianę w strukturze bazy.
- Nic nie trzeba zmieniać w klasie, aby można było jej instancje przechowywać w bazie.
- Przemyślany i wygodny system zapytań (QBE i SODA [Simple Object Database Access]).

Db4o - przykład

```
public class Player {
    protected String name;
    protected int squadNumber;
    protected float battingAverage;
    protected Team team;
      // metody dostępowe
}

Query q = db.query();
q.constrain(Player.class);
q.descend("battingAverage").constrain(new Float(0.3f)).greater();
ObjectSet result = q.execute();
while(result.hasNext()) {
    System.out.println(result.next());
}
```

Jest równoważne: „*SELECT * FROM players WHERE battingAverage > 0.3*”

Db4o – jakieś wady?

- W ogromnych systemach sprawdza się średnio (straty wydajności).
- Dostępne wyłącznie w Javie i .Net.

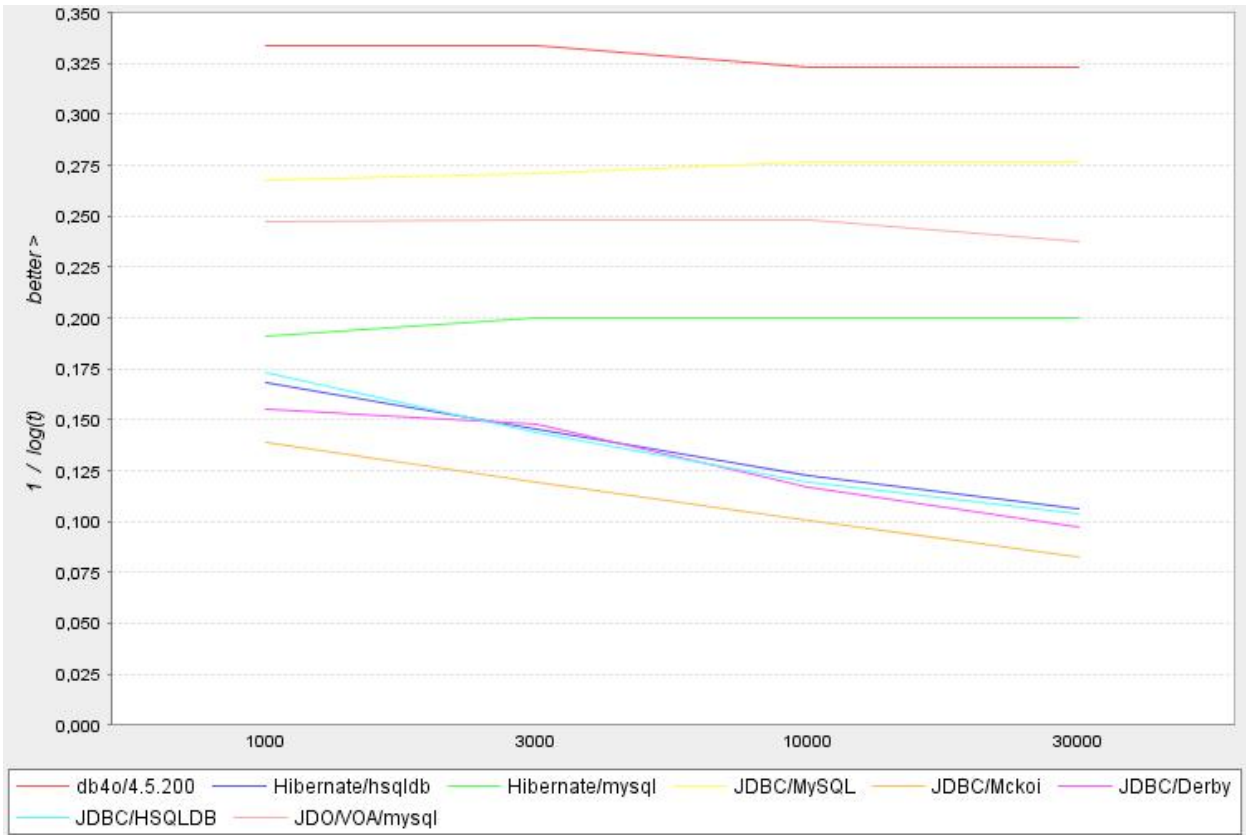
Porównanie wydajności

Przedstawione testy wydajności zaczerpnięte są z serwisu http://www.polepos.org.

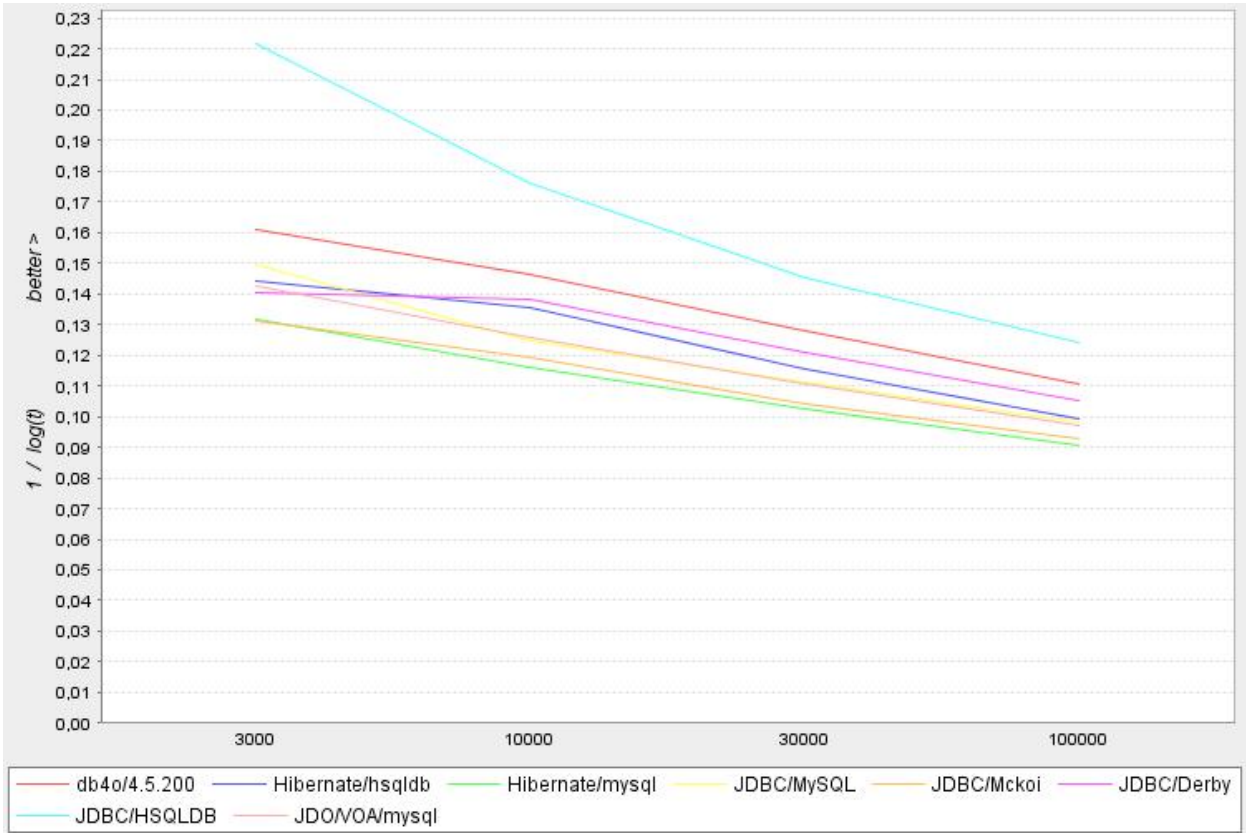
Porównywano:

- db4o/4.5
- Hibernate/hsqldb
- Hibernate/mysql
- JDBC/MySQL
- JDBC/Mckoi
- JDBC/Derby
- JDBC/HSQLDB
- JDO/VOA/mysql

Test zapytań dla hierarchii o głębokości dziedziczenia 5



Test zapisów dla hierarchii płaskiej



Bibliografia

- Technologia JDBC http://java.sun.com/products/jdbc/.
- Baza relacyjna HSQL http://hsqldb.org/.
- Mapper o/r Hibernate http://www.hibernate.org/.
- Mapper o/r JDO http://java.sun.com/products/jdo/.
- Prevayler http://www.prevayler.org/.
- DB4o http://www.db4o.com/.
- Porównanie szybkości baz http://www.polepos.org/.