Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Adam Michalik
Nr albumu: 209482

Sterownik JDBC do semistrukturalnej bazy danych LoXiM

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla, prof. UW
Instytut Informatyki

Czerwiec 2009

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiono implementację sterownika JDBC (Java Database Connectivity) dla bazy danych LoXiM rozwijanej na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego pod kierunkiem dra hab. Krzysztofa Stencla, prof. UW. Standard JDBC jest uznanym sposobem dostępu do relacyjnych baz danych w języku Java. Praca ma na celu udostępnienie tego interfejsu dla semistrukturalnej bazy danych, jaką jest LoXiM oraz omówienie charakterystycznych rozwiązań wymuszonych potrzebą dostosowania interfejsu do nierelacyjnego modelu danych.

Słowa kluczowe

SBQL, SBA, bazy danych, LoXiM, JDBC, sterownik JDBC, Java

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2 DATABASE MANAGEMENT
H.2.m Miscellaneous

Tytuł pracy w języku angielskim

A JDBC driver for LoXiM – a semistructural database

Spis treści

Wprowadzenie . . . 5
Przegląd pracy . . . 5

1. Cel pracy . . . 7

2. JDBC . . . 9
2.1. Architektury sterowników JDBC . . . 9
2.1.1. Typ 1 - mostek JDBC-ODBC . . . 10
2.1.2. Typ 2 - sterownik z kodem macierzystym . . . 11
2.1.3. Typ 3 - sterownik z serwerem pośredniczącym . . . 11
2.1.4. Typ 4 - sterownik ze specyficznym protokołem . . . 12
2.2. Wersje JDBC . . . 13
2.3. Przegląd klas i interfejsów JDBC . . . 14
2.3.1. Pakiet `java.sql` . . . 15
2.3.2. Pakiet `javax.sql` . . . 16

3. Implementacja JDBC w LoXiMie . . . 17
3.1. Wybór architektury sterownika . . . 17
3.2. Wybór wersji specyfikacji JDBC . . . 17
3.3. Protokół sieciowy dla LoXiMa . . . 17
3.4. Specyfika semistrukturalnego charakteru LoXiMa w kontekście JDBC . . . 18
3.4.1. Różnica modeli danych . . . 18
3.4.2. Tworzenie zbioru wyników `ResultSet` . . . 20
3.4.3. Przykłady . . . 21
3.5. Przyjęte ograniczenia implementacji . . . 22
3.6. Szczegóły implementacji przetwarzania wyników . . . 25
3.6.1. Wykonywanie zapytania . . . 25
3.6.2. Dostęp do wyników . . . 26
3.6.3. Typy danych . . . 26
3.7. Przykłady użycia . . . 27
3.7.1. Nawiązywanie połączenia . . . 27
3.7.2. Wykonywanie zapytania . . . 28
3.7.3. Przetwarzanie wyników . . . 28

4. Podsumowanie . . . 29
4.1. Kierunki rozwoju . . . 29

A. Instrukcja kompilacji . . . 31

B. Konsolowy klient JDBC 33

C. Zawartość płyty CD 35

Bibliografia 37

Wprowadzenie

LoXiM ([LoXiM] - strona domowa projektu) jest semistrukturalną bazą danych rozwijaną na licencji GPL na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego pod kierunkiem dr. hab. Krzysztofa Stencla, prof. UW. Baza ta realizuje paradygmat podejścia stosowego (SBA - *Stack-Based Approach*) wraz ze specyficznym językiem zapytań - SBQL (*Stack-Based Query Language*), zaproponowanymi przez prof. dr. hab. inż. Kazimierza Subietę [SBA]. Praca przedstawia próbę udostępnienia nowopowstałego interfejsu sieciowego LoXiMa za pośrednictwem standardu Java Database Connectivity - JDBC. Omówiona zostanie sama specyfikacja w zakresie potrzebnym do implementacji podstawowych funkcjonalności sterownika, sposób jej adaptacji do architektury stosowej reprezentowanej przez LoXiM, sposób wykorzystania protokołu sieciowego oraz najciekawsze szczegóły implementacyjne. Zaproponowane zostaną kierunki rozwoju sterownika zgodnie z istniejącymi funkcjonalnościami bazy danych oraz kierunki rozwoju samego LoXiMa zgodnie z dostępnymi możliwościami JDBC.

Przegląd pracy

Praca składa się z czterech rozdziałów. W rozdziale 1 przedstawiono motywację implementacji sterownika JDBC do systemu LoXiM. Zalety JDBC, przegląd typów sterowników oraz wersje specyfikacji wraz z opisem przewidzianych nią interfejsów opisano w rozdziale 2. Następnie, w rozdziale 3, przedstawiono rozwój sterownika do LoXiMa - począwszy od protokołu sieciowego, który stał się podstawą implementacji, przez dokonanie wyborów architektury sterownika, przyjęte ograniczenia, po kluczowe decyzje projektowe wynikające z potrzeby dostosowania relacyjnej architektury JDBC do obiektowego modelu SBA. Zaprezentowano również najciekawsze rozwiązania programistyczne oraz instrukcje użycia. W rozdziale 4 zaproponowano możliwe kierunki rozwoju sterownika i LoXiMa.

Rozdział 1

Cel pracy

Do roku 2008 LoXiM nie posiadał interfejsu pozwalającego na dostęp programistyczny. Istniał jedynie generyczny klient konsolowy, jednak protokół sieciowy komunikacji z serwerem był na tyle niejednolity i nieudokumentowany, że dostęp za pomocą tego klienta był jedynym sposobem komunikacji z LoXiMem.

Po zaimplementowaniu nowego protokołu sieciowego przez P. Tabora (zob. [LoXiM Protocol]), możliwy stał się ustandaryzowany dostęp do serwera LoXiM oraz budowanie bardziej wyrafinowanych narzędzi do komunikacji z bazą danych.

Jednym z uznanych sposobów dostępu do bazy danych w sposób programistyczny jest wykorzystanie sterownika JDBC dostarczonego najczęściej przez producenta bazy. Implementacja takowego w przypadku LoXiMa przynosi następujące korzyści:

- udowodnienie, że LoXiM jest platformą dojrzałą, zgodną z obowiązującymi standardami programistycznymi,
- udostępnienie interfejsu programistycznego, abstrahującego od specyfiki LoXiMa i bezpośredniej komunikacji sieciowej,
- stworzenie uniwersalnego sterownika nie tylko do pierwotnego LoXiMa (napisanego w C++), ale również do jego wariantów tworzonych w innych językach, jak np. JLoXiM (zob. [JLoXiM]) pisany w Javie czy LoXiM#, implementowany w C#.
- umożliwienie przezroczystego korzystania z LoXiMa jako bazy danych w aplikacjach klienckich (nie jest wymagane pisanie aplikacji pod bazę LoXiM, a jedynie zgodnie ze specyfikacją JDBC),
- uniezależnienie aplikacji klienckich od zmian w LoXiMie (interfejs pozostaje niezmienny, a w przypadku rozwoju LoXiMa jedynie sterownik wymaga zmian),
- wyznaczenie kierunków rozwoju LoXiMa zgodnych z możliwościami JDBC,
- oraz inne zalety samego JDBC opisane szczegółowo w rozdziale 2.

Rozdział 2

JDBC

JDBC - Java Database Connectivity - jest standardem dostępu do baz danych w języku Java (zob. [Sun-JDBC-Overview]). W jego skład wchodzą cztery moduły:

- JDBC API - zbiór interfejsów JDBC, których implementacja tworzy sterownik JDBC.
- JDBC Driver Manager - menadżer sterowników, ładujący klasy sterowników i wywołujący właściwy sterownik zależnie od bazy danych, do której nawiązywane jest połączenie.
- JDBC Test Suite - zbiór testów pozwalający na stwierdzenie, czy sterownik spełnia wymagania specyfikacji
- mostek JDBC-ODBC - zob. 2.1

Aplikacja kliencka pisana jest jest z wykorzystaniem JDBC API, a następnie, zależnie od podanego adresu URL wskazującego na serwer bazy danych, jeden z dostępnych podczas działania klienta sterowników jest wybierany do obsługi komunikacji z bazą. Dzięki takiemu podejściu:

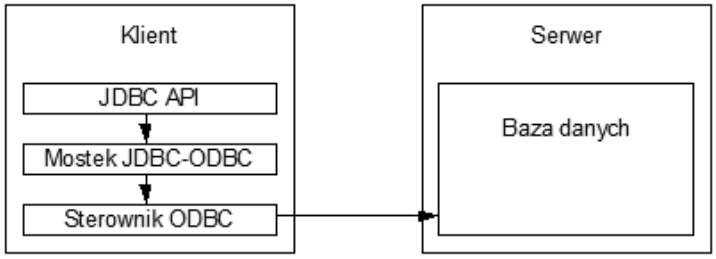
- program kliencki jest pisany niezależnie od wykorzystywanej bazy danych;
- różne bazy danych mogą być wykorzystywane przez klienta i mogą być wybierane podczas działania programu;
- zmiana wykorzystywanej bazy danych bądź sterownika nie wymaga rekompilacji klienta.

Co ważne w kontekście niniejszej pracy, JDBC jest przeznaczone dla relacyjnych baz danych.

2.1. Architektury sterowników JDBC

Sterownik JDBC może reprezentować jeden z czterech schematów implementacji. Typy 1 i 2 najczęściej służą jako pomost między Javą a istniejącymi wcześniej rozwiązaniami dostępu do baz danych w innych językach, typy 3 i 4 są implementacjami zrealizowanymi w pełni w języku Java i powinny być preferowane ze względu na lepszą wydajność. Wybór konkretnego rozwiązania zależy od sytuacji, wymagań i oczekiwań wobec zachowania klienta.

2.1.1. Typ 1 - mostek JDBC-ODBC



W przypadku niektórych baz danych dostawca może zapewniać jedynie sterownik ODBC. ODBC jest standardem dostępu do baz danych niezależnym od języka, w którym się go wykorzystuje, zależnym natomiast od platformy, gdyż napisany jest w kodzie macierzystym (zob. [ODBC]). Typ 1. sterownika JDBC pozwala na wykorzystanie ODBC w Javie - tworzy mostek tłumaczący wywołania JDBC na ODBC i rezultaty ODBC na JDBC. J2SE 1.4 i wersje późniejsze dostarczają referencyjną implementację mostka JDBC-ODBC (`sun.jdbc.odbc.JdbcOdbcDriver`), pozwalającą na połączenie się poprzez dowolny zainstalowany na komputerze klienta sterownik ODBC. Jednakże mostek ten napisany jest z użyciem kodu macierzystego i przeznaczony jedynie do użytku eksperymentalnego, kiedy nie istnieje żaden inny sterownik poza ODBC. Dostawcy systemów zarządzania bazami danych, jeśli sami nie udostępniają sterownika JDBC, a tworzą sterownik ODBC, implementują mostek do JDBC, najczęściej o wyższym poziomie integracji i lepszej wydajności niż generyczne rozwiązanie dostarczone wraz z JDK.

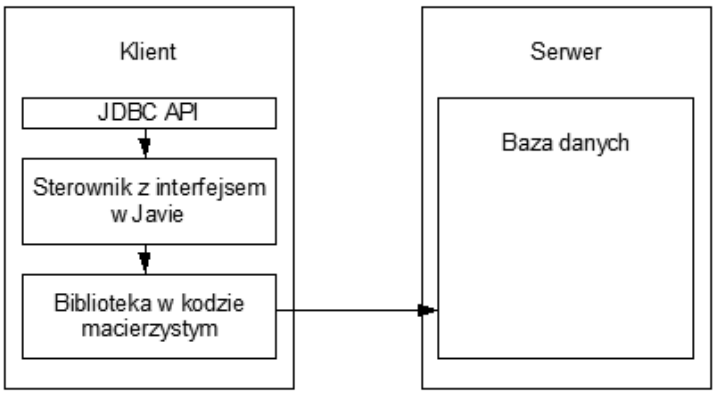
Zalety

- jeśli istnieje sterownik ODBC, mostek umożliwia łatwe użycie go w Javie.

Wady

- narzut czasowy związany z tłumaczeniem zapytań i wyników między JDBC i ODBC;
- narzut czasowy związany z wywołaniem kodu macierzystego (w którym napisany jest sterownik ODBC);
- sterownik ODBC musi być wcześniej zainstalowany na maszynie klienta (uniemożliwia to m. in. wykorzystanie tego typu sterownika w apletach);
- aplikacja nie jest przenośna między systemami operacyjnymi (sterownik ODBC jest napisany w kodzie macierzystym, choć najczęściej istnieją wersje dla różnych systemów operacyjnych).

2.1.2. Typ 2 - sterownik z kodem macierzystym



Dostawca systemu zarządzania bazami danych może udostępnić sterownik JDBC oparty na kodzie macierzystym. W języku Java implementowane są wówczas interfejsy specyfikacji JDBC, natomiast właściwe operacje na bazie danych przeprowadzane są z wykorzystaniem bibliotek skompilowanych na platformę klienta, w języku innym niż Java. Takie rozwiązanie może być przyjęte przez dostawcę, jeśli istnieją już biblioteki umożliwiające dostęp do bazy danych, a sterownik JDBC jest wtedy tylko ich "opakowaniem" standaryzującym interfejs i udostępniającym je na platformie Java.

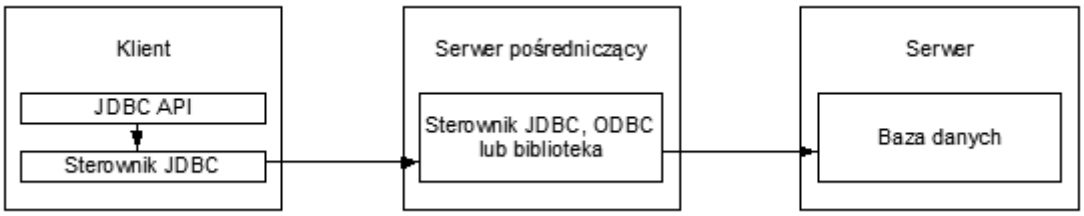
Zalety

- możliwość wykorzystania istniejących bibliotek w kodzie macierzystym;
- lepsza wydajność niż typ 1., gdyż nie ma potrzeby przeprowadzania dwustronnego tłumaczenia JDBC $\leftrightarrow$ ODBC.

Wady

- narzut czasowy związany z wywołaniem kodu macierzystego;
- biblioteki macierzyste muszą być wcześniej zainstalowane na maszynie klienta (uniemożliwia to m. in. wykorzystanie tego typu sterownika w apletach);
- ze względu na biblioteki macierzyste aplikacja nie jest przenośna między systemami operacyjnymi.

2.1.3. Typ 3 - sterownik z serwerem pośredniczącym



Kolejnym podejściem do realizacji sterownika JDBC jest zastosowanie serwera pośredniczącego między klientem a bazą danych, zgodnie z ideą aplikacji trójwarstwowych. Klient wywołuje

funkcje JDBC na serwerze, ten zaś przekazuje zapytania do baz danych wykorzystując specyficzne biblioteki dostępu do baz danych, najczęściej sterowniki JDBC typu 1 lub 2 albo inne biblioteki w kodzie macierzystym.

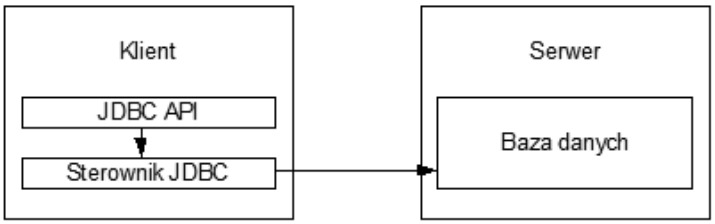
Zalety

- serwer jest warstwą abstrakcyjną, pozwalającą klientowi korzystać z bazy danych w sposób przezroczysty. Klient nie musi posiadać żadnego oprogramowania (sterowników, bibliotek) specyficznych dla wykorzystywanej bazy danych, gdyż to serwer zapewnia komunikację z konkretną bazą. W ten sposób jeden sterownik może stanowić interfejs do wielu baz danych różnych producentów;
- sterownik jest lekki i przenośny, napisany w pełni w Javie;
- serwer może zapewniać typowe usługi usprawniające operacje bazodanowe: pula połączeń i pamięć podręczną (połączeń, wyników zapytań itp.), równoważenie obciążenia oraz funkcje administracyjne jak np. logowanie i nadzór;
- może być używany w apletach, gdyż nie wymaga instalacji specyficznego oprogramowania na komputerze klienta.

Wady

- niezbędna jest wcześniejsza instalacja bibliotek macierzystych na serwerze pośredniczącym, który może wymagać również specyficznej konfiguracji do ich wykorzystania;
- dodatkowa warstwa pośrednicząca może stać się wąskim gardłem przy dużym obciążeniu. Zazwyczaj wada ta jest jednak równoważona przez opisane wyżej usługi zwiększające wydajność.

2.1.4. Typ 4 - sterownik ze specyficznym protokołem



Gdy dostęp do systemu zarządzania bazami danych jest możliwy z wykorzystaniem ustandaryzowanego, specyficznego protokołu sieciowego, a tak jest w przypadku znakomitej większości systemów, dostawca zapewnia najczęściej właśnie ten typ sterownika. Pozwala on na bezpośrednie połączenie z bazą danych, przekształcając wywołania JDBC API na komunikaty protokołu sieciowego właściwego dla bazy danych.

Zalety

- bezpośrednie połączenie z bazą danych (najczęściej przez gniazdo sieciowe), bez potrzeby tłumaczenia standardów jak w przypadku typu 1., wywołań kodu macierzystego, jak w typach 1. i 2. czy zastosowania warstwy pośredniczącej, jak w typie 3., pozwala na uzyskanie najlepszej wydajności;

- sterownik jest napisany w całości w Javie, co zapewnia przenośność między platformami i pozwala na wykorzystanie w apletach.

Wady

- każdy system zarządzania bazami danych wymaga zarejestrowania odrębnego sterownika, ze względu na odmienny protokół sieciowy;
- względem typu 3. brak usług zapewnianych przez serwer aplikacji: obsługa połączeń w puli czy pamięci podręcznej rezultatów.

2.2. Wersje JDBC

JDBC jest częścią JavaSE od wersji JDK 1.1. Od wersji 3.0 JDBC jest rozwijane jako część Java Community Process - JSR 54 ([JSR 54]) określa specyfikację JDBC 3.0, JSR 114 ([JSR 114]) wprowadza dodatek Rowset, zaś JSR 221 ([JSR 221]) jest specyfikacją JDBC 4.0. Poniższa tabela przedstawia historię wersji JDBC i wersje JDK, do których zostały dołączone. Szczegóły specyfikacji i wersji - zob. [JDBC Spec].

Wersja JDBC	Wersja JDK	JSR	Rok wydania JDK
4.0	6	221	2006
3.0	1.4	54	2002
2.1	1.2		1998
1.2	1.1		1997

Na potrzeby niniejszej pracy warte uwagi są jedynie specyfikacje 3.0 i 4.0, będące w najpowszechniejszym użytku. Wersje przed 3.0 można traktować jako archiwalne i niewspierane.

Najważniejsze cechy wprowadzane przez poszczególne wersje JDBC to:

Wersja 1.2

- modyfikowalne zbiory wyników (ResultSet).

Wersja 2.0 Optional Package API

- użycie interfejsu DataSource do uzyskiwania połączeń;
- wykorzystanie JNDI (*Java Naming and Directory Interface*) do uzyskiwania połączeń;
- umożliwienie utrzymywania połączeń w puli;
- umożliwienie stosowania transakcji rozproszonych;
- wprowadzenie interfejsu `RowSet` jako rozszerzenie `ResultSet`.

Wersja 2.1

- swobodny dostęp do zbioru wyników (`ResultSet`) - przechodzenie do przodu i tyłu oraz do określonego wiersza wyników;
- wykonywanie zapytań pojedyńczo lub jako wsad (*batch*);

- wsparcie dla typów danych wprowadzonych przez SQL:1999 - Blob, Clob, Array, Ref i Structured Type;
- wsparcie dla stref czasowych w typach danych Date, Time i Timestamp.

Wersja 3.0

- możliwość ponownego wykorzystania przygotowanych zapytań (`PreparedStatement`) w pulach połączeń;
- dodatkowa konfiguracja pul połączeń;
- dodanie punktów przywracania (`Savepoint`) - podczas transakcji można utworzyć taki punkt, a następnie cofnąć zmiany tylko do tego punktu zamiast wycofywania całej transakcji;
- możliwość odczytu wartości z kolumn, których dane są generowane automatycznie;
- dodanie typu danych `Boolean`;
- przekazywanie parametrów do funkcji i procedur (`CallableStatement`);
- dodanie interfejsu `DatabaseMetaData` pozwalającego na uzyskanie różnych informacji o statusie i konfiguracji bazy danych.

Wersja 4.0

- sterownik nie musi być ładowany explicite (`Class.forName("klasa.Sterownika")`), wystarczy, że w paczce JAR istnieje plik-deskryptor `/META-INF/services/java.sql.Driver` zawierający wpis o sterowniku, a zostanie on zarejestrowany automatycznie;
- poprawiona obsługa wyjątków - iterowanie po wyjątkach `SQLException` w głąb łańcucha przyczyn, specyficzne podklasy `SQLException`;
- wsparcie dla typu danych `RowId`;
- rozpakowywanie interfejsów JDBC do interfejsów dostawcy lub obiektów, do których wywołania są delegowane (`Wrapper`);
- dodatkowa konfiguracja pulowania połączeń i zapytań;
- wsparcie typu danych XML (`SQLXML`), wprowadzonego przez SQL:2003.

2.3. Przegląd klas i interfejsów JDBC

JDBC składa się z dwóch podzbiorów: podstawowych klas i interfejsów, umieszczonych w pakiecie `java.sql` oraz dodatkowych, wprowadzonych przez JDBC 2.0 Optional Package ([JDBC 2.0 Opt]), umieszczonych w pakiecie `javax.sql`. W obecnej chwili (JDBC 4.0, JDK 6) oba pakiety są dołączone do JDK. Poniżej przedstwiono przegląd najważniejszych, z punktu widzenia niniejszej pracy i minimalnej kompatybilności z JDBC, typów JDBC 4.0. Szczegółowy opis API jest częścią dokumentacji Javadoc do JDK 6 ([JDK 6 API]).

Klasy i typy wyliczeniowe przedstawiono czcionką pochyłą, a interfejsy czcionką zwykłą. Typy wprowadzone w JDBC 4.0 oznaczono gwiazdką.

2.3.1. Pakiet `java.sql`

Typy główne

- `java.sql.CallableStatement` - interfejs pozwalający na wywołanie procedur SQL;
- `java.sql.Connection` - połączenie (sesja) z bazą danych. Umożliwia obsługę fizycznego połączenia sieciowego, przeprowadzanie operacji na transakcjach (zatwierdzenie i wycofanie transakcji), tworzenie zapytań i procedur oraz niektórych typów danych (`Array`, `Blob`, `Clob`, `SQLXML`);
- `java.sql.DatabaseMetaData` - dostarcza różnorodnych informacji (możliwości, konfiguracja) na temat bazy danych;
- `java.sql.Driver` - podstawowy interfejs sterownika. Umożliwia nawiązywanie połączenia (tworzenie `java.sql.Connection`) z bazą danych;
- *`java.sql.DriverManager`* - usługa rejestrująca dostępne sterowniki i dopasowująca właściwy sterownik do wywoływanego adresu URL bazy danych;
- `java.sql.ParameterMetaData` - interfejs opisujący parametry dostarczone do zapytania przygotowanego (`java.sql.PreparedStatement`);
- `java.sql.PreparedStatement` - przygotowane (prekompilowane) zapytanie;
- `java.sql.ResultSet` - zbiór wyników zapytania;
- `java.sql.ResultSetMetaData` - interfejs opisujący metadane zbioru wyników zapytania (`java.sql.ResultSet`) jak np. typy i właściwości kolumn;
- `java.sql.Savepoint` - punkt przywracania. Zamiast wycofywać transakcje w całości, można ją cofnąć tylko do założonego wcześniej punktu;
- `java.sql.Statement` - interfejs pozwalający na wykonywanie zapytań i uzyskiwanie wyników;
- *`java.sql.Types`* - zbiór stałych-identyfikatorów typów SQL;
- `*java.sql.Wrapper` - interfejs pozwalający dostać się do interfejsu dostawcy sterownika lub do instancji, do której delegowane są żądania, jeśli klasa implementująca jest typu proxy. Rozszerzany m. in. przez `java.sql.CallableStatement`, `java.sql.Connection`, `java.sql.DatabaseMetaData`, `javax.sql.DataSource`, `java.sql.ParameterMetaData`, `java.sql.PreparedStatement`, `java.sql.ResultSet`, `java.sql.ResultSetMetaData`, `javax.sql.RowSet`, `java.sql.Statement`.

Wyjątki

- *`java.sql.SQLException`* - wyjątek rzucany z powodu błędu bazy danych lub innego;
- *`*java.sql.SQLFeatureNotSupportedException`* - wyjątek rzucany przy wywołaniu metod, które są uznane za opcjonalne i sterownik ich nie implementuje.

Typy danych

- `java.sql.Array` - odpowiednik typu SQL `ARRAY`;
- `java.sql.Blob` - odpowiednik typu SQL `BLOB`;
- `java.sql.Clob` - odpowiednik typu SQL `CLOB`;
- *`java.sql.Date`* - odpowiednik typu SQL `DATE`;
- `java.sql.Ref` - odpowiednik typu SQL `REF`, który jest referencją do struktury SQL;
- `*java.sql.RowId` - odpowiednik typu SQL `ROWID`;
- `*java.sql.SQLXML` - odpowiednik typu SQL `XML`;
- `java.sql.Struct` - odpowiednik strukturalnego typu SQL;
- *`java.sql.Time`* - odpowiednik typu SQL `TIME`;
- *`java.sql.Timestamp`* - odpowiednik typu SQL `TIMESTAMP`.

2.3.2. Pakiet `javax.sql`

- `*javax.sql.CommonDataSource` - interfejs definiujący metody wspólne dla `javax.sql.DataSource`, `javax.sql.XADataSource` i `javax.sql.ConnectionPoolDataSource`;
- `javax.sql.ConnectionPoolDataSource` - fabryka pulowalnych połączeń (`javax.sql.PooledConnection`);
- `javax.sql.DataSource` - fabryka połączeń do źródła danych (bazy danych) reprezentowanego przez ten obiekt;
- `javax.sql.PooledConnection` - połączenie, które może być wykorzystywane przez usługę pulowania połączeń;
- `javax.sql.RowSet` - rozszerzenie `java.sql.ResultSet` dodające wspacie dla modelu JavaBeans;
- `javax.sql.XAConnection` - połączenie obsługujące transakcje rozproszone;
- `javax.sql.XADataSource` - fabryka połączeń obsługujących transakcje rozproszone (`javax.sql.XAConnection`).

Rozdział 3

Implementacja JDBC w LoXiMie

3.1. Wybór architektury sterownika

Twórcy projektu LoXiM stawiają sobie za cel zapewnienie jego przenośności między platformami systemowymi. Od kiedy system udostępniono w serwisie Wolnego Oprogramowania SourceForge [LoXiM], trwają prace nad zapewnieniem kompatybilności z najpopularniejszymi systemami operacyjnymi: Linux, Mac OS X, Windows. LoXiM działa niezależnie od architektury procesora: 32- bądź 64-bitowej, posługuje się również specyficznym protokołem sieciowym, omówionym w punkcie 3.3. Te przesłanki jednoznacznie implikowały wybór typu 4. sterownika JDBC (zob. 2.1.4). Taka decyzja pozwoli na zapewnienie maksymalnej uniwersalności sterownika - platforma Java zapewnia przenośność między systemami operacyjnymi i architekturami procesorów, a ponieważ protokół sieciowy jest generowany automatycznie i pozwala na współdziałanie klienta i serwera napisanych w różnych językach, sterownik JDBC będzie umożliwiał łączenie się z dowolnym wariantem LoXiMa. Obecnie oprócz jego podstawowej implementacji w C++ zgodny pod względem protokołu jest również projekt JLoXiM - LoXiM implementowany w Javie (zob. [JLoXiM]).

3.2. Wybór wersji specyfikacji JDBC

W części 2.2 przedstawiono rozwój wersji Java SDK wraz ze specyfikacjami JDBC zawartymi w poszczególnych wydaniach SDK. W chwili obecnej (wiosna 2009) JDK 6.0 jest wersją aktualną, zaś JDK 5.0 znajduje się w końcowym okresie wsparcia - *End of Life (EOL)*. Na dzień 30 października 2009 przewidziano zaprzestanie wspierania tej wersji i przeniesienie jej do etapu *End of Service Life (EOSL)* (zob. [JDK EOL]). Stąd decyzja o implementacji sterownika do LoXiMa zgodnie z najnowszą wersją JDK - 6.0, a co za tym idzie, z wykorzystaniem JDBC 4.0. Jakkolwiek wiele funkcji wprowadzonych w tym wydaniu pozostało niezaimplementowanych (por. 2.2, 3.5), to pozostaje możliwość łatwego dodania nowych możliwości. Za kilka lat, gdy LoXiM osiągnie pełną dojrzałość, sterownik zgodny z najnowszą specyfikacją będzie znaczącym atutem.

3.3. Protokół sieciowy dla LoXiMa

Podstawą implementacji sterownika JDBC dla LoXiMa jest opracowany przez Piotra Tabora w 2008 roku protokół sieciowy ([LoXiM Protocol]). Koncepcja protokołu oparta jest na stukturach - paczkach - niosących komunikaty i dane. Implementacja protokołu - paczek oraz struktur odpowiedzialnych za ich serializację i deserializację w sieci - została stworzona w

językach C++ i Java. Pozwala to na przezroczystą komunikację między sterownikiem JDBC a serwerem LoXiM napisanym w C++. Ten sam protokół używany jest również w projekcie JLoXiM.

Protokół sieciowy pozwala abstrahować od niskopoziomowych operacji komunikowania się przez sieć Internet. Zapewnia serializację paczek danych do postaci ciągu bajtów oraz ich powrotną deserializację. Implementacja sterownika JDBC wykorzystuje protokół według następującego schematu:

1. otworzenie gniazda sieciowego do serwera bazy danych;
2. przekazanie otwartego gniazda do instancji klasy `PackageIO`;
3. w obiekcie `PackageIO` - otworzenie strumieni do serializacji (`PackageOutputStream`) i deserializacji (`PackageInputStream`) paczek na strumieniach przekazanego gniazda do - odpowiednio - pisania i czytania;
4. wykorzystanie obiektu `PackageIO` do wysyłania paczek do serwera i odczytywania rezultatów;
5. zamknięcie obiektu `PackageIO`, co powoduje zamknięcie gniazda, a tym samym i strumieni do przesyłania paczek.

Specyficzną cechą protokołu jest zdefiniowanie usługi podtrzymywania połączenia (*keep-alive*) przez asynchroniczne, cykliczne wysyłanie przez serwer paczki A-SC-PING i oczekiwanie na odpowiedź klienta paczką A-SC-PONG. W przypadku gdy klient nie odpowie, połączenie jest zrywane. Takie podejście umożliwia serwerowi wykrywanie niepoprawnie zakończonych połączeń i zamykanie gniazd sieciowych oraz wątków obsługi klienta, pozwalając tym samym redukować obciążenie i odzyskiwać niewykorzystywane zasoby. Podczas implementacji sterownika JDBC zastosowano następujące rozwiązanie:

1. po utworzeniu obiektu `PackageIO` tworzony jest oddzielny wątek T odczytujący paczki ze strumienia `PackageInputStream`. Odczyt jest blokujący;
2. jeśli ze strumienia `PackageInputStream` zostanie odczytana paczka A-SC-PING, natychmiast zostaje wysłana paczka A-SC-PONG;
3. jeśli zostanie odczytana inna paczka, zostaje ona dodana na koniec kolejki odebranych paczek;
4. obiekt `PackageIO` nie odczytuje paczek bezpośrednio ze strumienia paczek, ale z blokującej kolejki wypełnianej przez wątek T. W ten sposób paczki A-SC-PING są automatycznie odfiltrowywane na najniższym możliwym poziomie i nie są udostępniane na zewnątrz klasom sterownika.

3.4. Specyfika semistrukturalnego charakteru LoXiMa w kontekście JDBC

3.4.1. Różnica modeli danych

Specyfikacja JDBC przewidziana jest jako model dostępu do relacyjnych baz danych. LoXiM jest bazą semistrukturalną, obiektową i w wielu przypadkach semantyka interfejsu JDBC musiała być interpretowana tak, by udostępnić możliwości LoXiMa za pomocą zdefiniowanych w

specyfikacji funkcji sterownika. Podstawowe różnice dotyczą klas implementujących interfejsy `java.sql.ResultSet` i `java.sql.ResultSetMetaData`, ponieważ to one reprezentują model danych.

Koncepcja relacyjna definiuje `ResultSet` następująco:

- `ResultSet` jest reprezentacją tabeli wyników relacyjnej bazy danych;
- `ResultSet` zawiera uporządkowaną listę danych reprezentowanych przez wiersze;
- każdy wiersz, a tym samym każda dana, ma tę samą strukturę - te same kolumny;
- każda kolumna ma określony typ;
- kolumny są uporządkowane;
- kolumny mają swoje unikalne nazwy;
- typy, nazwy i porządek kolumn są stałe dla danego obiektu `ResultSet`.

SBA nie zapewnia tak statycznej struktury danych wynikowych. W przypadku LoXiMa rezultatem zapytania jest zawsze pojedynczy obiekt - w szczególnym przypadku jest to obiekt typu `Void`, reprezentujący brak wyniku. Najbliższym koncepcji relacyjnej jest rezultat w postaci zbioru obiektów (typ `Bag`, `Sequence` lub `Struct`), jednakże i w tym przypadku zbiór wynikowy może zawierać obiekty o zupełnie różnej strukturze. Stąd niektóre funkcje JDBC oparte na danych statycznych są niemożliwe do realizacji, inne zaś zostały adaptowane do zwracania wyników w sposób dynamiczny. W szczególności niemożliwa jest implementacja `ResultSetMetaData` - interfejs ten pozwala na poznanie różnorakich metainformacji o kolumnach zbioru wyników, np. ich liczbę (`getColumnCount()`) oraz, podając jako argument numer kolumny, m. in.:

- nazwę kolumny (`getColumnName(int column)`, `getColumnLabel(int column)`);
- klasę obiektów języka Java, której odpowiada zawartość kolumny (`getColumnClassName(int column)`);
- numeryczną dokładność danych z kolumny (`getPrecision(int column)`);
- określenie czy wartość kolumny może być nullem (`isNullable(int column)`);
- nazwę tabeli, do której kolumna należy (`getTableName(int column)`);
- określenie czy wartość kolumny jest automatycznie zwiększana (`isAutoIncrement(int column)`);
- określenie czy dozwolony jest zapis danych w kolumnie (`isWritable(int column)`, `isDefinitelyWritable(int column)`) czy może tylko ich odczyt (`isReadOnly(int column)`).

Przyjmując podejście stosowe i abstrahując od definicji słowa "kolumna", nie można określić takich właściwości globalnie, na całym zbiorze wyników, a tego wymaga specyfikacja `ResultSetMetaData`.

3.4.2. Tworzenie zbioru wyników `ResultSet`

Wynikiem zapytania SBQL jest jeden z typów obiektów:

- atomowy;
- identyfikator (referencja);
- wiązanie (*binder*);
- struktura (*struct*);
- kolekcja powyższych - multizbiór (*bag*) lub sekwencja (*sequence*).

Zgodnie z [SBA-Results], interpretacja powyższych typów przy odwzorowaniu na model relacyjny jest następująca:

1. obiekt niebędący kolekcją traktowany jest jako kolekcja jednoelementowa;
2. elementy kolekcji odpowiadają kolejnym wierszom tabeli;
3. pojedynczy element kolekcji, który nie jest strukturą, traktowany jest jak struktura jednoelementowa;
4. kolejne elementy struktury odpowiadają kolejnym kolumnom tabeli;
5. jeśli elementem struktury jest wiązanie, to nazwa kolumny odpowiada nazwie wiązania, w przeciwnym przypadku kolumna nie ma nazwy.

W przypadku SBA model danych jest znacznie bogatszy niż relacyjny, gdyż:

- kolejne elementy kolekcji wynikowej nie muszą być homogeniczne - mogą być zupełnie różnymi typami obiektów lub mieć odmienną strukturę;
- dozwolony jest dowolny poziom zagnieżdżenia obiektów;
- w jednej strukturze może występować kilka wiązań z tą samą nazwą, podczas gdy w tabeli relacyjnej nie może być kilku kolumn o tej samej nazwie;
- struktura może składać się z obiektów innych niż wiązania, a zatem w odwzorowaniu na wiersz mogą istnieć kolumny bez nazw.

Z uwagi na ostatni punkt, zmieniona musiała zostać semantyka metod `get`<*typ danych*>`(String columnName)` interfejsu `ResultSet`, pozwalających na otrzymanie wartości z kolumny o nazwie *columnName*. Specyfikacja wymaga, aby w przypadku dostępu do nieistniejącej kolumny, rzucany był wyjątek `SQLException`, jednak w modelu obiektowym odwołanie do obiektu o nieistniejącej nazwie jest poprawne i zwraca pusty rezultat. Stąd metody odwołujące się do nieistniejących kolumn w sterowniku JDBC dla LoXiMa zwracają *null*. Podobnie w przypadku gdy istnieje wiele wiązań z tą samą nazwą - wtedy metoda `getObject(String columnName)` zwraca kolekcję wszystkich obiektów o tej nazwie (pozostałe metody `get`<*typ danych*>`(String columnName)` rzucają w tym przypadku wyjątek). Szczegóły implementacji przekształcania wyników zwracanych przez LoXiM na model relacyjny w sterowniku JDBC przedstawiono w części 3.6.2.

3.4.3. Przykłady

Poniżej przedstawiono kilka przykładów przekształcenia wyników wysyłanych przez LoXiM do tabeli relacyjnej reprezentowanej przez obiekt `ResultSet`. Znak `X` oznacza brak wartości w danej kolumnie. W przypadku gdy wszystkie elementy w danej kolumnie są wiązaniami z nazwą n, zaznaczono to w nagłówku kolumny, jeśli zaś struktura kolumny jest niejednorodna, to wiązania opisano wewnątrz poszczególnych komórek jako *nazwa* → *obiekt*.

<table>
<tr><th>Rezultat LoXiM</th><th>ResultSet</th></tr>
<tr><td>1</td><td><table><tr><td>1</td></tr></table></td></tr>
<tr><td>imię→Adam</td><td><table><tr><th>imię</th></tr><tr><td>Adam</td></tr></table></td></tr>
<tr><td>bag{
struct{i_1, i_2},
struct{i_3, i_4},
struct{i_5, i_6}
}</td><td><table><tr><td>i_1</td><td>i_2</td></tr><tr><td>i_3</td><td>i_4</td></tr><tr><td>i_5</td><td>i_6</td></tr></table></td></tr>
<tr><td>bag{
struct{
imię → Adam,
nazwisko → Michalik
},
struct{
imię → Jan,
nazwisko → Kowalski
},
struct{
imię → Józef,
nazwisko → Nowak
}
}</td><td><table><tr><th>imię</th><th>nazwisko</th></tr><tr><td>Adam</td><td>Michalik</td></tr><tr><td>Jan</td><td>Kowalski</td></tr><tr><td>Józef</td><td>Nowak</td></tr></table></td></tr>
<tr><td>bag{
1,
struct{
imię → Jan,
imię → Józef,
nazwisko → Kowalski
},
struct{
nazwisko → Nowak
imię → Ryszard,
},
‘napis’,
samochód → fiat,
}</td><td><table><tr><td>1</td><td>X</td><td>X</td></tr><tr><td>imię → Jan</td><td>imię → Józef</td><td>nazwisko → Kowalski</td></tr><tr><td>nazwisko → Nowak</td><td>imię → Ryszard</td><td>X</td></tr><tr><td>napis</td><td>X</td><td>X</td></tr><tr><td>samochód → fiat</td><td>X</td><td>X</td></tr></table></td></tr>
</table>

3.5. Przyjęte ograniczenia implementacji

Specyfikacja JDBC [JSR 221] stwierdza, że część ze zdefiniowanych interfejsów i metod jest obowiązkowa w każdej implementacji sterownika, część natomiast jest opcjonalna, zgodnie z możliwościami udostępnianymi przez bazę danych. Niezaimplementowane metody zobowiązane są do rzucania wyjątku `SQLFeatureNotSupportedException`. Ze względu na zakres pracy zrealizowany został jedynie podstawowy zbiór funkcji pozwalający na posługiwanie się sterownikiem w dostępie do LoXiMa. Poniżej przedstawiono wymagania specyfikacji i stopień ich realizacji w sterowniku dla LoXiMa.

Wymagania spełnione:

- wsparcie dla transakcji; zatwierdzanie automatyczne i nieautomatyczne;
- automatyczna rejestracja sterownika;
- obsługa `ResultSet` w trybie `TYPE_FORWARD_ONLY`;
- obsługa współbieżności `ResultSet` w trybie `CONCUR_READ_ONLY`;
- implementacja interfejsu `java.sql.Driver`;
- implementacja interfejsu `java.sql.Wrapper`;
- implementacja interfejsu `javax.sql.DataSource`;
- implementacja interfejsu `CallableStatement` z wyjątkiem metod opcjonalnych o ile `DatabaseMetaData.supportsStoredProcedures()` zwraca *true* - LoXiM nie obsługuje procedur, a `DatabaseMetaData.supportsStoredProcedures()` zwraca $false$;
- implementacja interfejsu `ResultSet` z wyjątkiem opcjonalnych metod:
  - metody `updateXXX`,
  - `absolute`,
  - `afterLast`,
  - `beforeFirst`,
  - `cancelRowUpdates`,
  - `deleteRow`,
  - `first`,
  - `getArray`,
  - `getBlob`,
  - `getClob`,
  - `getNClob`,
  - `getNCharacterStream`,
  - `getNString`,
  - `getRef`,
  - `getRowId`,

- getSQLXML,
- getURL,
- getBigDecimal(int i,int scale),
- getBigDecimal(String colName,int scale),
- getCursorName,
- getObject(int i, Map<String,Class<?>> map),
- getObject(String colName, Map<String,Class<?>> map),
- getRow,
- getUnicodeStream,
- insertRow,
- isAfterLast,
- isBeforeFirst,
- isFirst,
- isLast,
- last,
- moveToCurrentRow,
- moveToInsertRow,
- previous,
- refreshRow,
- relative,
- rowDeleted,
- rowInserted,
- rowUpdated,
- updateRow

Zaimplementowano wszystkie wymagane metody. Z metod opcjonalnych zaimplementowano:

- getURL,
- getBigDecimal(int i,int scale),
- getRow,
- getUnicodeStream,
- isAfterLast,
- isBeforeFirst,
- isFirst,
- isLast,
- rowDeleted,
- rowInserted,
- rowUpdated.

Pozostałe metody opcjonalne nie zostały zaimplementowane ze względu na fakt, że jedynym obsługiwanym sposobem przeglądania obiektu `ResultSet` jest `TYPE_FORWARD_ONLY`, zaś jedynym typem obsługiwanej współbieżności jest `CONCUR_READ_ONLY` oraz ze względu na brak obsługi niektórych opcjonalnych typów danych przez LoXiM.

Wymagania spełnione częściowo:

- implementacja interfejsu `java.sql.DatabaseMetaData` - interfejs ten definiuje ok. 200 metod, które nie są kluczowe dla niniejszej pracy. Dostarczają one rozmaitych informacji o konfiguracji i możliwościach bazy danych;
- implementacja interfejsu `Connection` z wyjątkiem opcjonalnych metod:
  - `createArrayOf`,
  - `createBlob`,
  - `createClob`,
  - `createNClob`,
  - `createSQLXML`,
  - `createStruct`,
  - `getTypeMap`,
  - `setTypeMap`,
  - `prepareStatement(String sql, Statement.RETURN_GENERATED_KEYS)`,
  - `prepareStatement(String sql, int[] columnIndexes)`,
  - `prepareStatement(String sql, String[] columnNames)`,
  - `setSavePoint`,
  - `rollback(java.sql.SavePoint savepoint)`,
  - `releaseSavePoint`

  Zaimplementowano wszystkie najważniejsze metody. Pominięto metody opcjonalne oraz te dotyczące informacji o kliencie oraz poziomie izolacji transakcji, jako że LoXiM nie obsługuje takich informacji;
- implementacja interfejsu `Statement` z wyjątkiem opcjonalnych metod:
  - `cancel`,
  - `execute(String sql, Statement.RETURN_GENERATED_KEYS)`,
  - `execute(String sql, int[] columnIndexes)`,
  - `execute(String sql, String[] columnNames)`,
  - `executeUpdate(String sql, Statement.RETURN_GENERATED_KEYS)`,
  - `executeUpdate(String sql, int[] columnIndexes)`,
  - `executeUpdate(String sql, String[] columnNames)`,
  - `getGeneratedKeys`,
  - `getMoreResults(Statement.KEEP_CURRENT_RESULT)`,
  - `getMoreResults(Statement.CLOSE_ALL_RESULTS)`,
  - `setCursorName`

Pominięto metody opcjonalne oraz ograniczające wielkość przesyłanego zbioru wyników (`setMaxRows`) i wielkość przesyłanych wartości (`setMaxFieldSize`), ponieważ LoXiM nie obsługuje takich parametrów, tylko przesyła zawsze całość zbioru wyników.

Wymagania niespełnione:

- wsparcie dla SQL 92 oraz dodatkowo polecenie `DROP TABLE` - LoXiM posługuje się językiem SBQL, niezgodnym z SQL;
- sterownik musi udostępniać wszystkie, również niestandardowe, funkcjonalności bazy danych - ze względu na zakres pracy nie wszystkie funkcje LoXiMa są udostępnione przez sterownik JDBC;
- wsparcie dla aktualizacji wsadowych (*batch updates*) - zgodnie z definicją `Statement.executeBatch()`, wywołanie aktualizacji wsadowej rzuca `BatchUpdateException` w przypadku gdy któryś z elementów wsadu próbuje zwrócić wynik (`ResultSet`). W LoXiMie wynik w postaci obiektu zwraca każde zapytanie, bez rozróżnienia na aktualizacje i zapytania, zatem próby wywołania aktualizacji wsadowych zawsze kończyłyby się niepowodzeniem. Stąd wszystkie metody do obsługi aktualizacji wsadowych rzucają `SQLFeatureNotSupportedException`;
- implementacja interfejsu `java.sql.ParameterMetaData` - LoXiM nie obsługuje przygotowanych, parametryzowanych zapytań;
- implementacja interfejsu `java.sql.ResultSetMetaData` - implementacja interfejsu `ResultSetMetaData` nie ma sensu w kontekście nierelacyjnej bazy danych. Szerzej opisano to w części 3.4.1;
- implementacja interfejsu `PreparedStatement` z wyjątkiem wskazanych metod opcjonalnych - LoXiM nie obsługuje przygotowanych, parametryzowanych zapytań;

3.6. Szczegóły implementacji przetwarzania wyników

3.6.1. Wykonywanie zapytania

Wykonanie zapytania składa się z następujących etapów na poziomie klienta sterownika:

1. utworzenie obiektu `Statement` z obiektu `Connection`;
2. wywołanie zapytania jedną z metod: `Statement.execute(String)`, `Statement.executeQuery(String)`, `Statement.executeUpdate(String)`;
3. odebranie zbioru wyników zapytania - jako wynik metody `Statement.executeQuery(String)` lub przez wywołanie `Statement.getResultSet()`.

Wewnętrznie wykonanie zapytania zrealizowane jest następująco:

1. obiekt zapytania `LoXiMStatementImpl` wywołuje na implementacji interfejsu `Connection` - klasie `LoXiMConnectionImpl` metodę `execute(LoXiMStatement, String)` realizującą komunikację z serwerem LoXiM;
2. rozpoczynana jest transakcja, o ile nie jest rozpoczęta;
3. wysyłany jest do serwera pakiet `Q-C-STATEMENT` z flagą `EXECUTE` oraz treścią zapytania;

4. odczytywany jest pakiet `V-SC-SENDVALUES` co oznacza początek przesyłania wyników;
5. sterowanie przekazywane jest do klasy `ResultReader` która odczytuje kolejne pakiety `V-SC-SENDVALUE`;
6. klasa `ResultReader` konwertuje wartość każdego z pakietów `V-SC-SENDVALUE` na obiekt języka Java (zob. 3.6.3);
7. rezultaty zostają przekształcone jako kolekcja (zob. 3.4.2) na listę wyników zapytania i zwrócone do `LoXiMConnectionImpl.execute(LoXiMStatement, String)`;
8. lista zostaje opakowana w obiekt `ExecutionResult`, który zostaje zwrócony do wywołującego obiektu `LoXimStatemantImpl`;
9. wewnątrz obiektu `LoXimStatemantImpl` możliwe jest uzyskanie z otrzymanego `ExecutionResult` informacji o dokonanych modyfikacjach (`getUpdateCount()`) oraz zbioru wyników zapytania (`getResultSet()`).

3.6.2. Dostęp do wyników

Dostępowi do bieżącego obiektu wskazywanego kursorem `ResultSet` służy zbiór metod `get`<*typ danych*>, przyjmujących jako argument nazwę bądź numer kolumny. Zgodnie z opisem w części 3.4.2, wyszukiwanie po nazwie działa według następującego algorytmu.

Obiekty proste uznajemy za struktury jednoelementowe, zatem bieżący obiekt jest strukturą wielo- lub jednoelementową. Niech n będzie nazwą kolumny. Wtedy poszukiwane jest wiązanie (`Binding`) o nazwie n:

1. jeśli takie wiązanie nie istnieje, zwracany jest *null*;
2. jeśli istnieje dokładnie jedno takie wiązanie, zwracana jest jego wartość, rzutowana na odpowiedni typ danych oznaczony w sygnaturze metody. Jeśli rzutowanie jest niemożliwe, rzucany jest wyjątek `SQLDataException`;
3. jeśli istnieje więcej niż jedno wiązanie o tej nazwie, to zwracana jest lista ich wartości w przypadku wywołania metody `getObject(String columnName)` lub rzucany jest wyjątek w przypadku wywołania innej metody `get`<*typ danych*>`(String columnName)`;

W przypadku użycia metod przyjmujących za argument numer kolumny, odwołanie do nieistniejącego elementu skutkuje, zgodnie ze specyfikacją, rzuceniem wyjątku `SQLDataException`.

3.6.3. Typy danych

Odwzorowanie pakietów danych na obiekty języka Java przeprowadzane jest dwojako. Paczki niosące dane "proste" znajdują odpowiedniki w klasach należących do standardu języka. Obiekty "złożone" - właściwe dla SBA - przekształcane są na specyficzne klasy będące częścią implementacji sterownika.

Paczka	**Klasa Java**
Dane proste	
`UINT8`	`java.lang.Short`
`UINT16`	`java.lang.Integer`

UINT32	java.lang.Long
UINT64	java.math.BigInteger
SINT8	java.lang.Byte
SINT16	java.lang.Short
SINT32	java.lang.Integer
SINT64	java.lang.Long
BOOL	java.lang.Boolean
DATE	java.util.Calendar
TIME	java.util.Calendar
DATETIME	java.util.Calendar
TIMEZ	java.util.Calendar
DATETIMEZ	java.util.Calendar
DOUBLE	java.lang.Double
BOB	java.lang.String
VARCHAR	java.lang.String
Dane złożone	
VOID	pl.edu.mimuw.loxim.data.VoidImpl
LINK	pl.edu.mimuw.loxim.data.LinkImpl
BINDING	pl.edu.mimuw.loxim.data.BindingImpl
REF	pl.edu.mimuw.loxim.data.RefImpl
EXTERNALREF	pl.edu.mimuw.loxim.data.ExtReferenceImpl
BAG	pl.edu.mimuw.loxim.data.BagImpl
STRUCT	pl.edu.mimuw.loxim.data.StructImpl
SEQUENCE	pl.edu.mimuw.loxim.data.SequenceImpl

3.7. Przykłady użycia

3.7.1. Nawiązywanie połączenia

Połączenie z bazą danych LoXiM można nawiązać na trzy sposoby:

- Ładując sterownik do zarządcy sterowników `java.sql.DriverManager` a następnie wykorzystując go do uzyskania połączenia z bazą danych na podstawie adresu JDBC URL.

```
Class.forName("pl.edu.mimuw.loxim.jdbc.LoXiMDriverImpl");
    // Opcjonalne w Java 6
Connection con = DriverManager.
    getDriver("jdbc:loxim:students.mimuw.edu.pl:2000/db", "root", "");
```

- Tworząc instancję sterownika i nawiązując połączenie bezpośrednio za jej pomocą.

```
Driver driver = new LoXiMDriverImpl();
Properties info = new Properties();
info.setProperty("user", "root");
info.setProperty("password", "");
Connection con = driver.
    connect("jdbc:loxim:students.mimuw.edu.pl:2000/db", info);
```

- Wykorzystując interfejs `javax.sql.DataSource`

```
DataSource ds = new LoXiMDataSourceImpl(
    "jdbc:loxim:students.mimuw.edu.pl:2000/db");
Connection con = ds.getConnection("root", "");
```

W każdym wypadku adres dostępu do bazy danych jest postaci `jdbc:loxim:<adres hosta>:<port>/<nazwa bazy danych>`, gdzie `<nazwa bazy danych>` obecnie nie jest używana i może być dowolna.

3.7.2. Wykonywanie zapytania

W celu wykonania zapytania należy uzyskać obiekt *stmt* typu `Statement` z połączenia `Connection` za pomocą wywołania `Connection.createStatement()`. Następnie postąpić można dwojako:

- `ResultSet result = stmt.executeQuery(sbql);`
- ```
  stmt.execute(sbql);
  ResultSet result = stmt.getResultSet();
  ```

3.7.3. Przetwarzanie wyników

Po otrzymaniu obiektu *result* typu `ResultSet` możliwe jest iterowanie po jego elementach, które są strukturami (w zdegenerowanym przypadku - obiektami atomowymi) i dostęp do poszczególnych elementów tych struktur. Najprościej przetwarzać jest rezultaty następująco:

```
while(result.next()) {
    Object o = result.getObject(i); // i jest numerem kolumny
    doSomething(o);
}
```
```

Rozdział 4

Podsumowanie

Praca nad implementacją sterownika dla systemu LoXiM pozwoliła osiągnąć następujące cele:

- zapewnienie popularnego, wysokopoziomowego, przenośnego i ustandaryzowanego interfejsu dostępu do bazy danych;
- udowodnienie, że model stosowy, wykorzystany w LoXiMie, może być odwzorowany na model relacyjny, a dodatkowo dostarcza większej elastyczności i więcej możliwości;
- utworzenie jednorodnego interfejsu klienckiego dla rodziny baz danych LoXiM (LoXiM podstawowy, JLoXiM, LoXiM#);
- przetestowanie i wprowadzenie poprawek w bibliotece protokołu sieciowego;
- przetestowanie i poprawienie implementacji protokołu sieciowego na serwerze LoXiM;

4.1. Kierunki rozwoju

Jak przedstawiono w rozdziale 3.5, niniejsza implementacja sterownika JDBC dla systemu LoXiM jest wersją spełniającą podstawowe wymagania i udostępniającą jedynie najważniejsze możliwości interfejsu. Większość brakujących funkcji wymaga oprogramowania odpowiednich rozwiązań po stronie serwera, część jedynie rozwinięcia kodu sterownika. Poniżej przedstawiono subiektywną listę sugestii rozwoju obu projektów - LoXiMa i sterownika - mając na uwadze maksymalizację wykorzystania potencjału interfejsu JDBC 4.0.

Rozwój sterownika JDBC

- obsługa połączeń w puli (`ConnectionPoolDataSource`);
- rozwinięcie możliwości poznania danych konfiguracyjnych bazy (`DataBaseMetaData`).

Rozwój LoXiMa zgodnie z możliwościami JDBC

- prekompilowane zapytania (`PreparedStatement`);
- procedury (`CallableStatement`);
- rozproszone transakcje (`XADataSource`, `XAConnection`);
- automatyczne zatwierdzanie transakcji po stronie serwera;

- interfejs pozwalający poznać dane konfiguracyjne bazy (`DataBaseMetaData`).

Dodatek A

Instrukcja kompilacji

Do kompilacji sterownika niezbędne jest narzędzie Maven 2 dostępne pod adresem `http://maven.apache.org`. Maven pobiera wszystkie zależności projektu z Internetu, dzięki czemu żadna konfiguracja nie jest potrzebna. Niezbędne jest jedynie ręczne zainstalowanie bibliotek protokołu sieciowego dla LoXiMa, dostępnych z repozytorium SVN [LoXiM]. Kompilacja i uruchomienie sterownika wymaga JDK 6.

Kroki kompilacji:

1. zainstalować JDK6;
2. ustawić zmienną środowiskową `JAVA_HOME` na miejsce instalacji JDK;
3. dodać `$JAVA_HOME/bin` do zmiennej środowiskowej `PATH`;
4. zainstalować Maven 2 zgodnie z instrukcjami na stronie domowej projektu;
5. zainstalować `java_protolib` - bibliotekę obsługi pakietów protokołu sieciowego LoXiM;
6. zaistalować `loxim_protocol` - bibliotekę pakietów protokołu sieciowego LoXiM;
7. w katalogu głównym projektu sterownika JDBC (zawierającym plik `pom.xml`) wydać polecenie `mvn clean install`, co spowoduje zbudowanie pliku sterownika `driver-1.0.jar` w katalogu `target`.

Biblioteki protokołu sieciowego LoXiMa kompiluje się wydając polecenie `mvn clean install` w katalogach zawierających źródła tych projektów.

Dodatek B

Konsolowy klient JDBC

Część projektu sterownika JDBC dla LoXiMa stanowi testowy klient pozwalający komunikować się z konsoli z wybraną bazą danych. Należy zbudować archiwum JAR sterownika zgodnie z opisem z dodatku A oraz archiwum JAR dla klas testowych, wywołując w katalogu głównym projektu polecenie `mvn jar:test-jar`, co spowoduje zbudowanie pliku `driver-1.0-tests.jar` w katalogu `target`. Program kliencki można wywołać wydając komendę `java -cp <classpath> pl.edu.mimuw.loxim.client.Client <db url> <user> <password>` gdzie kolejne parametry to:

- `<classpath>` - classpath, lista archiwów JAR będących zależnościami projektu. Wymienione są one w pliku `pom.xml`. Prócz tego należy dodać plik sterownika i testów: `driver-1.0.jar` i `driver-1.0-tests.jar`;
- `<db url>` - URL dostępu do bazy danych w formacie zgodnym ze specyfikacją JDBC;
- `<user>` - nazwa użytkownika bazy danych;
- `<password>` - hasło dostępu do bazy danych.

Po uruchomieniu możliwe jest wykonywanie zapytań SBQL. Klient wczytuje ze standardowego wejścia pojedynczą linię zapytania i wyświetla na standardowym wyjściu tekstową reprezentację rezultatu otrzymanego z serwera. Działanie kończy polecenie `quit`.

Dodatek C

Zawartość płyty CD

Do pracy została załączona płyta CD z następującą zawartością:

- `/doc/` - dokumentacja sterownika
  - `/doc/Adam Michalik - Sterownik JDBC do semistrukturalnej bazy danych LoXiM.pdf` - niniejsza praca magisterska
  - `/doc/Adam Michalik - Sterownik JDBC do semistrukturalnej bazy danych LoXiM.tex` - plik źródłowy pracy magisterskiej
  - `/doc/*.png` - ilustracje użyte w pracy
  - `/doc/pracamgr.cls` - szablon formatowania pracy dostarczony przez Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego
- `/src/` - pliki źródłowe sterownika oraz testów
  - `/src/pom.xml` - plik konfiguracyjny Maven 2
  - `/src/src/main/` - pliki źródłowe sterownika
  - `/src/src/test/` - pliki źródłowe testów
- `/bin/` - skompilowany sterownik oraz zależności
  - `/bin/driver-1.0.jar` - plik sterownika
  - `/bin/driver-1.0-tests.jar` - plik testów do sterownika (zawiera konsolowy program kliencki)
  - `/bin/lib/*.jar` - zależności

Bibliografia

[SBA] Kazimierz Subieta, *Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL)*, http://www.sbql.pl (2006).

[SBA-Results] Kazimierz Subieta, *SBA: Environment Stack in the AS0 Store Model*, http://www.sbql.pl/Topics/Environment%20Stack%20in%20SBA%20in%20AS0.html#ResultsOfQueries (2008).

[LoXiM Protocol] Piotr Tabor, *Projekt i implementacja protokołu sieciowego dla semistrukturalnej bazy danych*, https://apd.uw.edu.pl (2008).

[LoXiM] LoXiM - strona główna projektu, http://loxim.sourceforge.net

[JLoXiM] JLoXiM - strona główna projektu, http://jloxim.mimuw.edu.pl

[Sun-JDBC-Overview] Sun Developer Network, *JDBC Overview*, http://java.sun.com/products/jdbc/overview.html (2008).

[ODBC] Microsoft Technical Support, *ODBC - Open Database Connectivity Overview*, http://support.microsoft.com/kb/110093 (2007).

[JSR 54] Java Community Process, *JDBC 3.0 Specification*, http://jcp.org/en/jsr/detail?id=54 (2002).

[JSR 114] Java Community Process, *JDBC Rowset Implementations*, http://jcp.org/en/jsr/detail?id=114 (2004).

[JSR 221] Java Community Process, *JDBC 4.0 API Specification*, http://jcp.org/en/jsr/detail?id=221 (2006).

[JDBC Spec] Sun Developer Network, *JDBC specifications*, http://java.sun.com/products/jdbc/download.html (2009).

[JDBC 2.0 Opt] Sun Microsystems Inc., *JDBC 2.0 Standard Extension API*, http://java.sun.com/products/jdbc/jdbc20.stdext.pdf (1998).

[JDK 6 API] Sun Microsystems Inc., *Java Platform, Standard Edition 6 API Specification*, http://java.sun.com/javase/6/docs/api/ (2008).

[JDK EOL] Sun Microsystems Inc., *Java SE & Java SE for Business Support Road Map*, http://java.sun.com/products/archive/eol.policy.html (2008)