SQLJ w Oracle9i – rozszerzenia standardu

Marek Wojciechowski, Maciej Zakrzewicz
Politechnika Poznańska, Instytut Informatyki
ul. Piotrowo 3a, 60-965 Poznań
e-mail: {marek,mzakrz}@cs.put.poznan.pl

Abstrakt. Standard SQLJ dotyczy zagnieżdżania instrukcji SQL w programach w języku Java. SQLJ stanowi alternatywę dla bezpośredniego korzystania z interfejsu JDBC, pozwalając na pisanie bardziej zwartych i czytelnych programów. Ponadto, SQLJ zapewnia kontrolę poprawności odwołań do obiektów bazy danych i weryfikację składni instrukcji SQL na etapie kompilacji programu. Zgodnie ze standardem SQLJ instrukcje SQL zagnieżdżane w programie mają charakter statyczny, co może niekiedy stanowić istotne ograniczenie. Niniejszy artykuł poświęcony jest rozszerzeniom funkcjonalności SQLJ dostępnym w Oracle9i, ze szczególnym uwzględnieniem wsparcia dla dynamicznych instrukcji SQL.

1. Wprowadzenie

Standard SQLJ dotyczy integracji technologii baz danych z językiem Java. Jest on wynikiem współpracy firm Oracle, Sybase, IBM, Tandem (obecnie część Compaq), JavaSoft i Informix. Standard SQLJ składa się z trzech części. Część 0 definiuje sposób zagnieżdżania instrukcji SQL w programach języka Java (*SQLJ Part 0: SQLJ Embedded SQL*). Część 1 dotyczy wykorzystywania statycznych metod Java jako składowanych procedur i funkcji SQL (*SQLJ Part 1: SQLJ Stored Procedures and UDFs*). Część 2 dotyczy używania czystych klas Java jako abstrakcyjnych typów danych (ang. *abstract data types*) SQL (*SQLJ Part 2: SQLJ Data Types*). Ostatnia z wymienionych części standardu stanowi alternatywę dla abstrakcyjnych typów danych standardu SQL3. Najwcześniej, bo już w 1998 roku, zatwierdzona jako standard ANSI została Część 0 [4]. Ponadto, obecnie jest ona jedyną z części standardu SQLJ, która jest również standardem ISO [5] (od 2000 roku). Części 1 i 2 są jedynie standardami ANSI.

Mianem Oracle SQLJ określana jest implementacja Części 0 standardu SQLJ dostępna w produktach firmy Oracle. Należy jednak podkreślić, że Oracle wspiera również pozostałe części standardu SQLJ (ładowanie klas Java do bazy danych i wykorzystywanie statycznych metod jako procedur i funkcji składowanych oraz używanie klas Java jako obiektowych typów SQL). W Oracle9i Oracle SQLJ w pełni wspiera standard ISO, dodatkowo oferując wiele interesujących rozszerzeń. Rozszerzenia te dotyczą zarówno funkcjonalności SQLJ, jak i sposobu generacji kodu wynikowego.

Niniejszy artykuł przedstawia rozszerzenia dostępne w Oracle9i na tle standardu SQLJ. Struktura artykułu jest następująca. Rozdział 2 prezentuje podstawową składnię SQLJ i sposoby realizacji dostępu do bazy danych w tym standardzie. Rozdział 3 opisuje przebieg translacji programów SQLJ i środowisko uruchomieniowe. Rozdział 4 zawiera listę i krótki opis rozszerzeń standardu SQLJ dostępnych w Oracle9i. Rozdział 5 poświęcony jest w całości wsparciu dla dynamicznego SQL w Oracle9i SQLJ, co stanowi najbardziej istotne rozszerzenie w stosunku do standardu. Rozdział 6 zawiera podsumowanie.

2. Dostęp do baz danych z programów Java - składnia SQLJ

Powszechnie używanym standardem pozwalającym na dostęp do baz danych z aplikacji języka Java jest standard JDBC zdefiniowany przez Sun Microsystems [2]. JDBC jest interfejsem programisty (API) dostępnym w postaci klas języka Java. Producenci systemów baz danych mogą dostarczać swoje implementacje standardu (oraz jego rozszerzenia) w postaci własnych sterowników JDBC. W standardzie JDBC treść instrukcji SQL jest przekazywana jako parametr

tekstowy odpowiednim metodom obiektu reprezentującego instrukcję SQL lub połączenie z bazą danych w programie. Składnia SQL nie jest weryfikowana na etapie kompilacji programu i dlatego ewentualne błędy w instrukcjach SQL są wykrywane dopiero w trakcie działania programu. Standard SQLJ stanowi odpowiedź na tę istotną wadę JDBC. SQLJ pozwala na zagnieżdżanie instrukcji SQL w kodzie Java w sposób umożliwiający weryfikację ich poprawności na etapie kompilacji programu. Takie rozwiązanie w istotny sposób zwiększa wydajność programisty. W przeciwieństwie do programów Java realizujących dostęp do baz danych w standardzie JDBC, programy SQLJ muszą być poddane prekompilacji, gdyż oprócz kodu w języku Java zawierają specyficzne konstrukcje składniowe SQLJ: deklaracje i instrukcje SQLJ. Zarówno deklaracje, jak i instrukcje SQLJ rozpoczynają się słowem kluczowym `#sql`, a kończą się średnikiem. Treść poleceń SQL jest zagnieżdżana w instrukcjach SQLJ i musi być ujęta w nawiasy klamrowe. Deklaracje SQLJ służą do deklarowania klas iteratorów, wykorzystywanych przy przetwarzaniu wyników zapytań, oraz klas kontekstów połączeń, używanych w aplikacjach korzystających z więcej niż jednego połączenia z bazą danych jednocześnie. Przedstawione poniżej fragmenty kodu realizujące to samo zadanie ilustrują sposoby dostępu do bazy danych w oparciu o standardy JDBC i SQLJ.

```
// Dostęp do bazy danych poprzez interfejs JDBC
PreparedStatement ps = conn.prepareStatement(
 "SELECT ename, sal FROM emp WHERE sal > ?");
double minSal = 1000.00;
ps.setDouble(1, minSal);
ResultSet rs = ps.executeQuery();
while (rs.next()) {
    String n = rs.getString("ENAME");
    double s = rs.getDouble("SAL");
    System.out.println(n + "   " + s);
}
rs.close();
ps.close();

// Dostęp do bazy danych w standardzie SQLJ
#sql iterator EmpNameIter(String ename, double sal);
...
EmpNameIter iter;
double minSal = 1000.00;
#sql iter = { SELECT ename, sal FROM emp WHERE sal > :minSal };
while (iter.next()) {
    String n = iter.ename();
    double s = iter.sal();
    System.out.println(n + "   " + s);
}
iter.close();
```

Powyższe przykłady pokazują, że korzystanie ze standardu SQLJ nie tylko sprzyja tworzeniu programów bardziej odpornych na błędy programisty, ale dodatkowo prowadzi do bardziej zwartego i czytelnego kodu niż w przypadku korzystania z interfejsu JDBC. Mimo swoich niewątpliwych zalet standard SQLJ nie może jednak w pełni zastąpić JDBC, ponieważ oferuje on znacznie mniejszą elastyczność. SQLJ jest idealnym rozwiązaniem gdy instrukcja SQL wykorzystywana w programie ma charakter statyczny tzn. jej składnia jest znana w momencie tworzenia kodu. Oczywiście znana musi być tylko struktura składniowa instrukcji, a wartości na których dana instrukcja operuje mogą być wyznaczane dopiero w trakcie działania programu. Takie wartości są reprezentowane w instrukcji SQLJ poprzez tzw. wyrażenia zewnętrzne (ang. *host expressions*), mające postać wyrażeń języka Java poprzedzonych znakiem dwukropka. W najprostszej postaci wyrażenia te sprowadzają się do tzw. zmiennych zewnętrznych (ang. *host*

variables), stanowiących odwołanie do zmiennej w kodzie Java (w powyższym przykładzie wykorzystano odwołanie do zmiennej zewnętrznej `minSal`). Standard JDBC traktuje treść poleceń SQL jako łańcuchy znaków, w związku z czym pozwala na dynamiczne budowanie treści instrukcji SQL w trakcie działania programu. Standardy SQLJ i JDBC mogą współistnieć w programie, współdzieląc to samo połączenie z bazą danych. W przypadku statycznych instrukcji SQL zalecane jest korzystanie ze standardu SQLJ, ze względu na lepszą kontrolę błędów. Instrukcje SQL o charakterze dynamicznym, ze względu na ograniczenia standardu SQLJ, muszą być realizowane poprzez interfejs JDBC.

Przedstawiony wyżej fragment kodu SQLJ pokazuje jak w tym standardzie przetwarzane są zapytania zwracające więcej niż jeden rekord. W takim przypadku wynik zapytania musi zostać przypisany do iteratora, który jest wystąpieniem wcześniej zadeklarowanej klasy iteratora. Iteratory w SQLJ są odpowiednikami zbiorów wynikowych (ang. *result sets*) wykorzystywanych w standardzie JDBC, jednakże w przeciwieństwie do nich wspierają kontrolę typów. Standard SQLJ wprowadza dwa rodzaje iteratorów z silną kontrolą typów: iteratory nazwane (ang. *named iterators*) i iteratory pozycyjne (ang. *positional iterators*). Deklarując klasę iteratora podaje się typy jego pól, a w przypadku iteratorów nazwanych również ich nazwy. Typy pól iteratora (oraz ewentualnie ich nazwy) powinny być zgodne z typami pól (oraz ewentualnie nazwami) w rekordach zwracanych przez zapytanie. W trakcie nawigacji po rekordach iteratora w danej chwili dostępny jest jeden rekord. W przedstawionym wcześniej przykładzie wykorzystany został iterator nazwany, dlatego też dostęp do poszczególnych pól bieżącego rekordu realizowany był za pomocą metod iteratora o nazwach odpowiadających nazwom pól iteratora. W przypadku iteratorów pozycyjnych bieżący rekord musi być pobrany do zmiennych zewnętrznych za pomocą instrukcji FETCH (korzystanie z iteratorów pozycyjnych przypomina odczyt danych z kursora w PL/SQL). Oprócz iteratorów nazwanych i pozycyjnych, SQLJ oferuje również iteratory o słabej kontroli typów, które funkcjonalnie odpowiadają zbiorom wynikowym JDBC. Podstawowy sposób nawigacji po rekordach iteratora polega na przechodzeniu w pętli do kolejnych rekordów do momentu gdy przetworzony zostanie ostatni rekord. Dodatkowo, iteratory nazwane i pozycyjne mogą być „przewijalne” (ang. *scrollable iterators*), jeśli ich klasy implementują odpowiedni interfejs. Iteratory przewijalne umożliwiają bardziej elastyczną nawigację po liście rekordów (przesuwanie się w obu kierunkach o dowolną liczbę rekordów, przejście do rekordu o danym numerze).

Dla obsługi zapytań zwracających dokładnie jeden rekord, SQLJ posiada instrukcję SELECT INTO, pozwalającą na realizację takich zapytań na takiej samej zasadzie jak instrukcji DML (INSERT, UPDATE, DELETE), co ilustruje poniższy fragment kodu:

```
// Zapytanie zwracające dokładnie jeden rekord
int count;  double avgSal;
#sql { SELECT count(*), avg(sal)
       INTO :count, :avgSal
       FROM emp };
// Instrukcja DML
int num = 3347;
#sql { UPDATE emp SET sal = :avgSal
       WHERE empno = :num };
```

Poza opisanymi wyżej konstrukcjami, SQLJ oferuje również możliwość wywoływania składowanych procedur i funkcji oraz obsługuje instrukcję przypisania. Składnię tych operacji ilustruje poniższy przykładowy fragment kodu:

```
int a = 5;  double b;  int c = 0;
// Wywołanie procedury składowanej
#sql { CALL proc_1 (:IN a, :OUT b, :INOUT c) };
// Wywołanie funkcji składowanej
#sql a = { VALUES (proc_1 (:IN c) ) };
```

```
// Instrukcja przypisania
#sql { SET :c = 5 * :a + 10};
```

W przypadku wywoływania procedur i funkcji składowanych, dla każdej ze zmiennych zewnętrznych należy podać tryb w jakim ma być ona przekazana do instrukcji SQLJ (IN, OUT lub INOUT). Tryb przekazania zmiennej zewnętrznej można podawać we wszystkich typach instrukcji SQLJ, ale poza wywoływaniem podprogramów składowanych jest to opcjonalne i w większości przypadków nie jest konieczne, gdyż na ogół tryb przyjmowany domyślnie jest odpowiedni.

Wszystkie instrukcje SQLJ występujące w przykładach przedstawionych dotychczas korzystały z domyślnego kontekstu połączenia, przy założeniu, że z domyślnym kontekstem związane jest odpowiednie otwarte połączenie z bazą danych. W pewnych szczególnych przypadkach takie domyślne połączenie może być dostępne automatycznie od początku działania programu (np. połączenie z bieżącą bazą danych w procedurach składowanych w języku Java w Oracle8i/9i). Najczęściej jednak aplikacja musi sama otworzyć połączenie z bazą danych. Gdy aplikacja w danej chwili korzysta tylko z jednego połączenia z bazą danych, wygodnym rozwiązaniem jest ustawienie domyślnego kontekstu połączenia. Tworząc nowy kontekst połączenia z jednoczesnym otwarciem połączenia z bazą danych należy podać URL identyfikujący bazę danych, nazwę użytkownika i hasło oraz informację o tym, czy tryb automatycznego zatwierdzania (ang. *auto commit*) ma być włączony czy nie. Utworzenie nowego kontekstu połączenia, z jednoczesnym uczynieniem go kontekstem domyślnym pokazuje fragment kodu przedstawiony poniżej:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
DefaultContext.setDefaultContext(
  new DefaultContext("jdbc:oracle:thin:@localhost:1521:ORCL",
                              "scott", "tiger", true));
```

Jeśli aplikacja korzysta z kilku połączeń z bazami danych, konieczne jest utworzenie kilku obiektów reprezentujących konteksty połączeń. Gdy schematy (zbiory obiektów bazy danych) dostępne poprzez te połączenia są różne, konteksty połączeń powinny być wystąpieniami różnych, zadeklarowanych wcześniej, klas kontekstów połączeń. Dla instrukcji SQLJ, które mają wykonać się w innym niż domyślny kontekście połączenia, należy jawnie wyspecyfikować odpowiedni kontekst połączenia w nawiasach kwadratowych po słowie kluczowym `#sql` rozpoczynającym instrukcję SQLJ. Deklarację klasy kontekstu połączeń, utworzenie nazwanego kontekstu połączenia jako instancji tej klasy oraz sposób specyfikacji żądanego kontekstu połączenia dla danej instrukcji SQLJ pokazuje poniższy przykład:

```
#sql context Employees;
...
Class.forName("oracle.jdbc.driver.OracleDriver");
Employees ctxEmp = new Employees("jdbc:oracle:thin:@localhost:1521:ORCL",
                              "scott", "tiger", true);
...
#sql [ctxEmp] { DELETE FROM emp WHERE sal > 9000 };
```

Jeśli dla danego kontekstu połączenia ustawiony jest tryb automatycznego zatwierdzania, poszczególne instrukcje SQLJ wykonywane w tym kontekście stanowią oddzielne transakcje. Gdy tryb ten jest wyłączony, do sterowania przebiegiem transakcji używane są następujące instrukcje SQLJ:

```
// Zatwierdzenie transakcji
#sql { COMMIT };
// Wycofanie transakcji
#sql { ROLLBACK };
```

```
// Ustawienie właściwości transakcji (poziom izolacji, read only/read write)
#sql { SET TRANSACTION ... };
```

Dla każdej z instrukcji SQLJ zamiast lub oprócz kontekstu połączenia może pojawić się tzw. kontekst wykonania (ang. *execution context*). W przypadku gdy dla danej instrukcji specyfikowany jest zarówno kontekst połączenia, jak i kontekst wykonania, są one oddzielone przecinkiem, a kontekst połączenia musi być podany jako pierwszy. Kontekst wykonania pozwala na uzyskanie informacji o liczbie zmodyfikowanych rekordów, ostrzeżeniach SQL, a także umożliwia sterowanie przebiegiem wykonania instrukcji SQL (np. timeout dla zapytań). Poniższy przykład ilustruje wykorzystanie kontekstu wykonania:

```
sqlj.runtime.ExecutionContext ec = new sqlj.runtime.ExecutionContext();
#sql [ec] { UPDATE emp SET sal = sal * 1.1 WHERE sal < 1000 };
System.out.println(ec.getUpdateCount() + "rows updated.");
```

3. Translacja i uruchamianie programów SQLJ

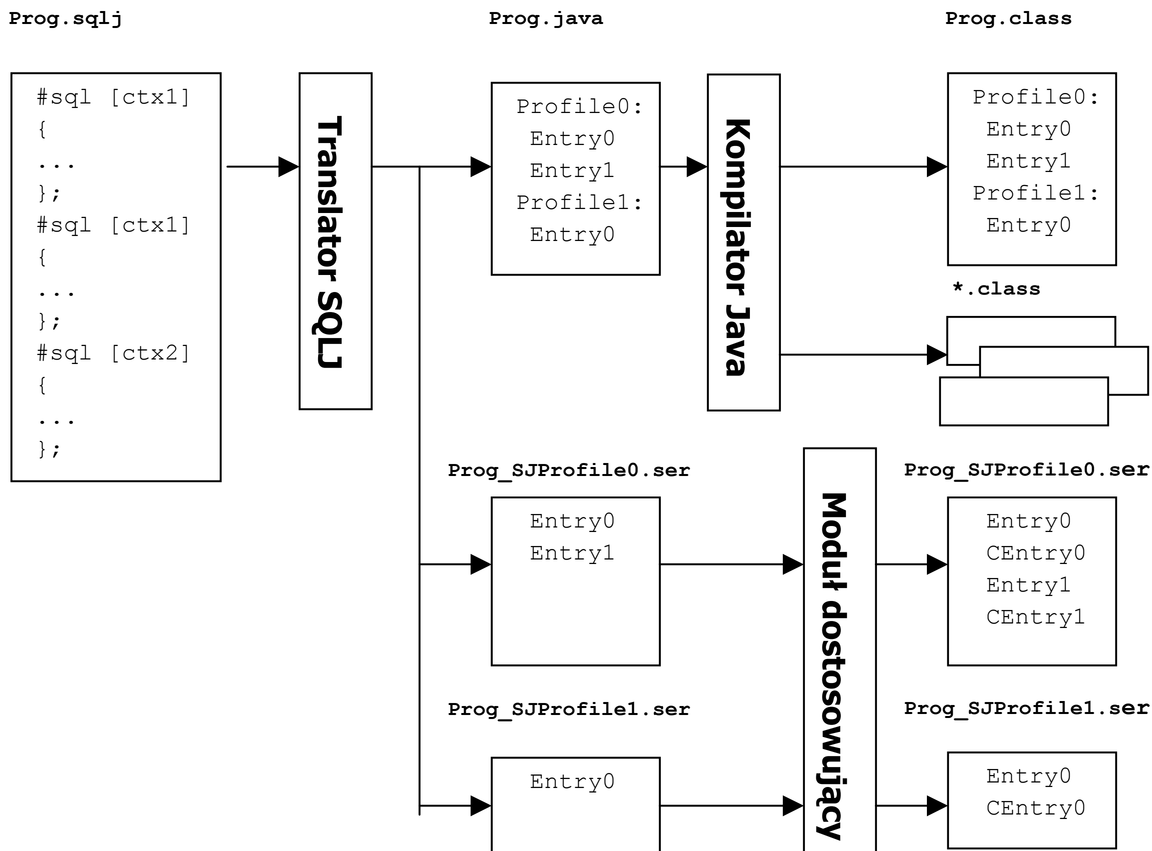
Program źródłowy w SQLJ musi być poddany translacji. Translator rozpoczyna pracę od sprawdzenia poprawności składni instrukcji SQLJ, z uwzględnieniem kontroli składni wyrażeń Java zagnieżdżonych w instrukcjach SQLJ. Następnie wywoływany jest moduł kontroli semantycznej (ang. *semantics-checker*). Kontrola semantyczna może odbywać się w trybie on-line lub off-line zależnie od parametrów wywołania translatora (w przypadku kontroli semantycznej w trybie on-line translator wymaga podania parametrów połączenia z bazą danych). Niezależnie od trybu w jakim jest przeprowadzana kontrola semantyczna instrukcji SQLJ, na tym etapie sprawdzana jest poprawność typów wyrażeń Java występujących w instrukcji SQL oraz określana jest natura instrukcji SQL (na podstawie występowania słów kluczowych `SELECT`, `INSERT`, itd.). W przypadku kontroli semantycznej w trybie on-line, dodatkowo sprawdzana jest składnia instrukcji SQL, zgodność typów SQL i Java oraz poprawność odwołań do obiektów bazy danych.

W przypadku pozytywnego wyniku wspomnianych testów, translator SQLJ tłumaczy pliki źródłowe SQLJ (posiadające standardowo rozszerzenie `.sqlj`) na czysty kod języka Java, zastępując zagnieżdżone w programie instrukcje SQL odwołaniami do środowiska uruchomieniowego SQLJ. Ponadto, translator generuje również tzw. pliki profili (jeden dla każdej klasy kontekstu połączenia wykorzystywanej w programie). Profile zawierają informacje o instrukcjach SQL występujących w programie, typach wykorzystywanych danych i trybach dostępu do nich. Domyślnie, generowane profile mają postać uszeregowanych zasobów Java, składowanych w plikach z rozszerzeniem `.ser`. Translator SQLJ posiada również opcję generacji profili w postaci plików `.class`.

Czysty kod Java powstały w wyniku działania translatora SQLJ poddawany jest kompilacji standardowym kompilatorem języka Java. W rezultacie powstaje zbiór skompilowanych klas Java (każda w osobnym pliku `.class`). Na ten zbiór składają się klasy Java jawnie zdefiniowane przez programistę w kodzie źródłowym, klasy reprezentujące wykorzystywane w programie klasy iteratorów i kontekstów połączenia oraz specjalna klasa służąca do obsługi profili.

Wygenerowane profile mogą być poddane dostosowaniu (ang. *customization*). Producenci systemów zarządzania bazami danych mogą dostarczać moduły dostosowujące (ang. *customizers*), których zadanie polega na dostrojeniu instrukcji SQL do konkretnej implementacji serwera bazy danych np. w celu poprawy efektywności ich wykonania.

Schemat procesu standardowej generacji kodu wynikowego dla aplikacji SQLJ przedstawia Rys. 1. Należy podkreślić, że w przypadku translatora SQLJ dostarczanego przez Oracle wszystkie etapy generacji kodu (translacja, kompilacja i dostosowanie za pomocą modułu dostosowującego Oracle) są wykonywane automatycznie po wywołaniu translatora.



Rys. 1. Standardowa generacja kodu wynikowego dla aplikacji SQLJ

Kod wynikowy SQLJ musi być uruchamiany w środowisku uruchomieniowym SQLJ (ang. *SQLJ runtime*). Środowisko uruchomieniowe realizuje instrukcje SQL programu wykorzystując informacje zawarte w profilach. Typowe implementacje środowiska uruchomieniowego SQLJ realizują dostęp do bazy danych poprzez sterownik JDBC, choć standard SQLJ tego nie wymaga. Standard SQLJ gwarantuje przenaszalność kodu wynikowego SQLJ (klas i profili) między różnymi platformami. Korzystanie z rozszerzeń standardu oferowanych przez różnych producentów (np. Oracle) ogranicza przenaszalność aplikacji do jednej, konkretnej platformy.

4. Rozszerzenia standardu SQLJ dostępne w Oracle9i

SQLJ dostępny w Oracle9i stanowi rozszerzenie standardu ISO. Oracle SQLJ akceptuje składnię SQL dostępną w serwerze bazy danych Oracle z uwzględnieniem konstrukcji specyficznych dla Oracle, stanowiących rozszerzenie standardu SQL. Ponadto, Oracle SQLJ pozwala na używanie anonimowych bloków PL/SQL jako zagnieżdżonych instrukcji SQL, na tej samej zasadzie jak np. instrukcji DML. W Oracle SQLJ programista ma do dyspozycji klasę `oracle.sqlj.runtime.Oracle` umożliwiającą wygodne otwieranie połączeń z bazą danych (szczególnie w przypadku bazy danych Oracle). Metoda `connect()` tej klasy przed otwarciem połączenia z bazą danych automatycznie rejestruje sterownik JDBC Oracle, a po otwarciu połączenia ustawia domyślny kontekst połączenia.

Dodatkowo, wersja SQLJ dostępna w Oracle9i oferuje specyficzną dla Oracle opcję generacji efektywnego kodu wynikowego, obsługuje więcej typów danych niż przewiduje to standard SQLJ oraz wprowadza konstrukcje językowe oferujące dodatkową funkcjonalność.

4.1. Generacja kodu specyficznego dla Oracle

Jak zostało to wspomniane wcześniej, standard SQLJ przewiduje translację kodu SQLJ do kodu Java zawierającego w miejscu zagnieżdżonych instrukcji SQL odwołania do środowiska uruchomieniowego SQLJ. W tym przypadku translator dodatkowo generuje pliki profili zawierające informacje o instrukcjach SQL używanych w programie. Profile są wykorzystywane przez środowisko uruchomieniowe SQLJ w trakcie wykonywania programu. W większości implementacji (np. w przypadku Oracle SQLJ) środowisko uruchomieniowe realizuje dostęp do bazy danych poprzez wywołania JDBC.

W wersji Oracle9i dostępna jest specyficzna dla Oracle opcja generacji kodu bez profili, polegająca na translacji programu SQLJ do kodu Java zawierającego jawne wywołania JDBC. W tym przypadku nie są generowane pliki profili, a rola środowiska uruchomieniowego jest znacznie ograniczona. Generacja kodu specyficznego dla Oracle posiada wiele zalet w porównaniu ze standardową generacją kodu wynikowego. Oto najważniejsze z nich:

- aplikacje działają bardziej efektywnie ze względu na bezpośrednie korzystanie z JDBC;
- rozmiar aplikacji jest mniejszy oraz mniejsza jest liczba komponentów aplikacji (nie ma plików profili).

Podstawową wadą generacji kodu specyficznego dla Oracle jest utrata możliwości przenoszenia postaci wynikowej aplikacji na inne platformy. Ponadto, niedostępna jest funkcjonalność dotycząca profili, np. dostosowywanie profili po generacji kodu wynikowego aplikacji.

4.2. Rozszerzenia dotyczące obsługiwanych typów danych

W zakresie obsługiwanych typów danych Oracle9i SQLJ wprowadza następujące rozszerzenia:

- obsługa instancji klas `oracle.sql.*` (klas otaczających dla typów danych Oracle SQL) oraz klas implementujących interfejsy `oracle.sql.ORAData` i `java.sql.SQLData`;
- możliwość używania instancji iteratorów i zbiorów wynikowych jako wejściowych i wyjściowych parametrów w dowolnym kontekście oraz instancji strumieni jako parametrów wyjściowych;
- obsługa typów znakowych Unicode.

4.3. Rozszerzenia funkcjonalne

Najbardziej istotnym i rewolucyjnym rozszerzeniem funkcjonalności SQLJ dostępnym w Oracle9i jest z pewnością wsparcie dla dynamicznych instrukcji SQL poprzez wprowadzenie nowych konstrukcji składniowych (patrz Rozdział 5). Ponadto, Oracle9i oferuje drobne rozszerzenia dotyczące funkcjonalności iteratorów o słabej kontroli typów (obsługuje przewijalne iteratory o słabej kontroli typów).

5. Dynamiczny SQL w Oracle9i

Większość instrukcji SQL zagnieżdżanych w aplikacjach ma charakter statyczny, tzn. ich składnia znana jest na etapie tworzenia programu, a w trakcie działania programu ustalane są jedynie wartości, na których te instrukcje operują. Niekiedy jednak w programach wymagana jest większa elastyczność. Może się zdarzyć tak, że składnia polecenia, bądź też nazwy kolumn lub tabel, do których ono się odwołuje, jest determinowana dopiero w trakcie działania programu. Niestety standard SQLJ obsługuje jedynie statyczny SQL. Dlatego też, w sytuacjach gdy w programie występują zarówno statyczne jak i dynamiczne instrukcje SQL, programiści korzystający ze standardu SQLJ muszą w przypadku dynamicznego SQL stosować wywołania JDBC. Takie rozwiązanie nie stanowi problemu od strony technicznej, gdyż instrukcje SQLJ mogą

być przeplatane wywołaniami JDBC, współdzieląc to samo połączenie z bazą danych. Jednakże nie ulega wątpliwości, że praca programisty jest łatwiejsza, a programy bardziej czytelne, gdy wszystkie instrukcje SQL w programie są realizowane za pomocą tego samego API.

W przypadku Oracle SQLJ, istnieje jeszcze jedna możliwość zagnieżdżania dynamicznych instrukcji SQL w programach SQLJ (dostępna już w wersjach serwera bazy danych poprzedzających Oracle9i). Polega ona na zagnieżdżeniu jako instrukcji SQL anonimowego bloku PL/SQL, zawierającego polecenie EXECUTE IMMEDIATE, uruchamiającego polecenie SQL o treści podanej w postaci łańcucha znaków, zbudowanego dynamicznie w trakcie działania programu. Poniższy fragment kodu ilustruje sposób korzystania z dynamicznego SQL w programach SQLJ z wykorzystaniem anonimowych bloków PL/SQL:

```
String polecenie;
polecenie = ...   // budowa treści dynamicznego polecenia SQL
#sql { begin
         execute immediate :polecenie;
       end
};
```

Wadą powyższego rozwiązania, występującą również w przypadku zastosowania wywołania JDBC, jest niemożność zweryfikowania składni polecenia SQL na etapie kompilacji programu. Oracle9i wprowadza nową, niestandardową składnię pozwalającą na zagnieżdżanie w programie Java dynamicznych poleceń SQL. Składnia ta jest zgodna z duchem standardu SQLJ i umożliwia częściowe sprawdzenie poprawności polecenia SQL już podczas translacji programu.

Nowe rozszerzenia standardu SQLJ dostępne w Oracle9i, wspierające dynamiczny SQL, polegają na możliwości zagnieżdżania w instrukcjach SQL dynamicznych wyrażeń SQL, określanych jako metawyrażenia związane (ang. *meta bind expressions*). Wyrażenia te mogą pojawiać się w miejscach, gdzie normalnie oczekiwane są statyczne klauzule SQL. Metawyrażenie związane zawiera identyfikator Java typu `String` lub wyrażenie Java zwracające wartość tego typu, wyznaczaną w trakcie działania programu. Dodatkowo, metawyrażenie związane może zawierać również statyczny tekst w języku SQL, którym ma być zastąpione wyrażenie podczas translacji, w celu umożliwienia kontroli semantycznej instrukcji SQLJ w trybie on-line.

Metawyrażenie związane może pojawić się w miejscu:

- nazwy tabeli;
- nazwy kolumny na liście SELECT;
- całości lub części klauzuli WHERE;
- nazwy roli, schematu, katalogu lub pakietu w instrukcji DDL lub DML;
- wyrażenia lub literału SQL.

Istnieją pewne ograniczenia dotyczące stosowania metawyrażeń związanych, wynikające z konieczności zapewnienia translatorowi SQLJ możliwości określenia natury instrukcji SQL, w celu przeprowadzenia analizy składniowej instrukcji SQLJ. Metawyrażenie związane nie może:

- rozpoczynać instrukcji SQL;
- zawierać słowa kluczowego INTO lub zwracać listy INTO polecenia SELECT INTO;
- występować w następujących instrukcjach i klauzulach: CALL, VALUES, PSM SET, COMMIT, ROLLBACK, FETCH INTO, CAST.

Składnia metawyrażeń związanych jest następująca:

```
:{ wyr_związane_Java }
```

lub

```
:{ wyr_związane_Java :: zastępczy_kod_SQL }
```

Wyrażenie związane Java musi spełniać kryteria składniowe języka Java, a zastępczy kod SQL musi być zgodny z regułami języka SQL (w szczególności może być pusty) i nie może zawierać zagnieżdżonych metawyrażeń związanych i wyrażeń Java. Podczas translacji programu SQLJ, w przypadku gdy metawyrażenie związane posiada zastępczy kod SQL, jest ono zastępowane tym kodem na potrzeby analizy semantycznej instrukcji SQLJ w trybie on-line (obejmującej analizę poprawności instrukcji SQL). W trakcie działania programu, przy przetwarzaniu dynamicznej instrukcji SQL metawyrażenia związane są zastępowane przez wyznaczone wartości ich wyrażeń związanych Java.

Rozważmy następujący przykład:

```
...
String table = "emp_boston";
String whereCond = "sal>2000";
#sql { DELETE FROM :{table :: emp}
       WHERE :{whereCond :: ename='BROWN'}
};
...
```

Podczas translacji, na potrzeby analizy semantycznej, zagnieżdżona dynamiczna instrukcja SQL przyjmie następującą postać:

```
DELETE FROM emp
WHERE ename='BROWN';
```

Z kolei w trakcie działania programu instrukcja ta zostanie wykonana w następującej postaci:

```
DELETE FROM emp_boston
WHERE sal>2000;
```

Taki sam efekt przyniosłoby wykonanie poniższego fragmentu kodu:

```
...
String table = "emp_boston";
String whereCond = "sal>2000";
#sql { DELETE FROM :{table}
       WHERE :{whereCond :: ename='BROWN'}
};
...
```

W tym przypadku jedyną różnicą w stosunku do przykładu rozważanego wcześniej jest brak zastępczego kodu SQL dla pierwszego z zagnieżdżonych metawyrażeń związanych. Taka zmiana nie może oczywiście wpływać na działanie programu, ale w jej wyniku translator nie będzie w stanie przeprowadzić analizy semantycznej instrukcji SQLJ w trybie on-line. Specyfikowanie zastępczego kodu SQL dla wszystkich metawyrażeń zagnieżdżonych na daje gwarancji, że zbudowana w trakcie działania programu dynamiczna instrukcja SQL będzie poprawna semantycznie (np. będzie odwoływać się jedynie do istniejących kolumn). Tym niemniej, nawet analiza semantyczna w trybie on-line przeprowadzona na zastępczym kodzie SQL może doprowadzić do wykrycia większej liczby błędów już na etapie translacji. Stosowanie zastępczego

kodu SQL jest tym bardziej wskazane, im lepiej można przewidzieć typową docelową strukturę instrukcji SQL będących wynikiem dynamicznej instrukcji SQL zawartej w programie.

Ta sama dynamiczna instrukcja SQL może w trakcie działania programu przyjmować różną postać, co ilustruje poniższy przykład:

```
double raise = 1.1;
String [] cities = {"boston", "ny", "detroit", "chicago"};
String salColName = "salary";
for (int i=0; i<cities.length; i++)
{
  #sql { UPDATE :{"emp_"+cities[i] :: emp}
         SET :{salColName :: sal} = :{ salColName :: sal} * :raise
  };
}
...
```

Podczas translacji, zagnieżdżona dynamiczna instrukcja SQL przyjmie postać:

```
UPDATE emp SET sal = sal * :raise;
```

Z kolei w trakcie działania programu instrukcja ta zostanie wykonana cztery razy:

```
UPDATE emp_boston SET salary = salary * 1.1;
UPDATE emp_ny SET salary = salary * 1.1;
UPDATE emp_detroit SET salary = salary * 1.1;
UPDATE emp_chicago SET salary = salary * 1.1;
```

6. Podsumowanie

Standard SQLJ pozwala na zagnieżdżanie instrukcji SQL w programach języka Java. Stanowi on alternatywę dla bezpośredniego korzystania z interfejsu JDBC. W porównaniu z JDBC, SQLJ pozwala na bardziej efektywne tworzenie aplikacji gdyż więcej błędów może być wykrytych już na etapie kompilacji programu, a nie dopiero przy jego uruchomieniu. W przypadku wywołań JDBC, zagnieżdżone instrukcje SQL nie są interpretowane przez kompilator. Natomiast w przypadku programów w SQLJ składnia poleceń SQL, a nawet poprawność odwołań do obiektów bazy danych i zgodność typów danych może być zweryfikowana już na etapie translacji. Ponadto, kod SQLJ jest bardziej zwarty niż JDBC. Pomimo swoich zalet standard SQLJ nie zawsze może zastąpić JDBC, ponieważ nie dorównuje mu funkcjonalnością. W przypadku wywołań JDBC, treść poleceń SQL może być budowana dynamicznie w trakcie działania programu. Standard SQLJ dotyczy zagnieżdżania statycznych poleceń SQL, czyli takich których składnia jest znana na etapie tworzenia kodu. Dodatkowo, w przypadku implementacji środowiska uruchomieniowego SQLJ korzystających z JDBC, efektywność programów SQLJ może być słabsza niż programów wykorzystujących interfejs JDBC bezpośrednio.

Implementacja SQLJ dostępna w Oracle9i oferuje wiele rozszerzeń standardu. Spośród nich na szczególną uwagę zasługują te, które stanowią odpowiedź na podstawowe wady technologii SQLJ. Oracle9i wspiera dynamiczny SQL w SQLJ, co powinno w większości przypadków wyeliminować konieczność bezpośredniego korzystania z JDBC w programach SQLJ. Ponadto, Oracle9i daje w trakcie generacji kodu wynikowego możliwość translacji kodu SQLJ do kodu zawierającego wywołania JDBC. W tym przypadku ograniczona jest rola środowiska uruchomieniowego SQLJ, co prowadzi do poprawy efektywności wykonywania programu. Oferowane przez Oracle9i SQLJ rozszerzenia standardu z pewnością należy uznać za atrakcyjne i przydatne, jednocześnie

pamiętając, że korzystanie z niestandardowych konstrukcji składniowych i mechanizmów ogranicza przenaszalność aplikacji.

Bibliografia

1. Basu J., Rohwedder E.: Using SQLJ for Enterprise Database Applications: Access, Procedures and Storage (Tutorial), VLDB'99, 1999
2. http://java.sun.com/products/jdbc/
3. http://www.sqlj.org/
4. Information Technology – Database Languages – SQL – Part 10: Object Language Bindings (SQL/OLB), ANSI specification X3.135.10-1998, 1998
5. Information Technology – Database Languages – SQL – Part 10: Object Language Bindings (SQL/OLB), ISO/IEC 9075-10:2000, 2000
6. Khan S., Kurian T., Wright B..: Zastosowanie SQLJ, Software 7/99, 1999
7. Oracle9i JDBC Developer's Guide and Reference, Release 1 (9.0.1), 2001
8. Oracle9i SQLJ Developer's Guide and Reference, Release 1 (9.0.1), 2001
9. Shaw P., Becker B., Klein J., Hapner M., Clossman G., Pledereder R.: Java and Relational Databases: SQLJ (Tutorial), SIGMOD'98, 1998
10. Stern S.A.: SQLJ: The 'open sesame' of Java database applications, JavaWorld May 1999, 1999