POLSKO–JAPOŃSKA WYŻSZA SZKOŁA
TECHNIK KOMPUTEROWYCH

Obiektowa baza danych jako repozytorium XML

PRACA MAGISTERSKA

Autorzy: *Michał Lentner*
Jakub Trzetrzelewski
Promotorzy: *dr Lech Banachowski, prof. nzw.*
dr hab. inż. Kazimierz Subieta, prof. nzw.
Specjalność: *Bazy Danych i Inżynieria Oprogramowania*

Warszawa, Wrzesień 2003

Abstrakt

Niniejsza praca magisterska stanowi próbę zbudowania prototypu nowoczesnego systemu zarządzania bazą danych, który byłby w stanie obsługiwać mechanizmy wprowadzane przez dwa najpopularniejsze obecnie trendy w rozwoju baz danych — obiektowość i XML. Zarówno model relacyjny, znane modele obiektowe, jak i struktury wprowadzane przez XML, traktowane w nim są jako przypadki szczególne obranego modelu danych, a obsługujący je język zapytań stanowi prawdopodobnie jedno z najnowocześniejszych osiągnięć w tej dziedzinie. Zakończona sukcesem próba implementacji tych, oraz kilku innych nowatorskich rozwiązań, pozwala stwierdzić, iż odpowiednio zaprojektowany system zarządzania obiektową bazą danych jest w stanie rozwiązać wiele problemów ograniczających zastosowania produktów reklamowanych jako systemy obiektowe, relacyjno–obiektowe, czy repozytoria XML.

Spis treści

Wprowadzenie

W ostatnich latach obserwuje się zwiększone zapotrzebowanie systemów informatycznych na coraz nowocześniejsze środki przetwarzania danych. Coraz częściej okazuje się, iż koncepcje stanowiące fundamenty relacyjnych baz danych okazują się niewystarczające w nowych warunkach. Wielu specjalistów zdaje sobie sprawę z braków tych systemów, szczególnie w zakresie modelowania pojęciowego, zdolności do opanowania nadmiernej złożoności, oraz języków i środowisk do tworzenia aplikacji. Niektórzy uważają wręcz, iż powstanie modelu relacyjnego na wiele lat opóźniło rozwój baz danych. Stąd właśnie bierze się zauważalny obecnie powrót do hierarchicznych struktur danych i rozwój takich ideologii, jak obiektowość, czy technologii związanych z XML. Te nowe podejścia do baz danych krytykowane są z kolei niejednokrotnie za wszystkoizm, mnożenie bytów, przypadkowość rozwiązań, niekonswekcję i ogólną niedojrzałość. Nie dziwi więc fakt, iż chaos panujący w tych dziedzinach stanowi jedną z przyczyn orgraniczających ich rozwój.

Niniejsza praca magisterska jest próbą zastosowania zupełnie nowego podejścia do budowy systemów zarządzania bazami danych i repozytoriów XML. Jako dowód słuszności opisywanych w niej rozwiązań, zaprezentowany został prototyp systemu zarządzania bazą danych, nazwanego przez autorów Yaod (*Yet Another Object Database*). Pomimo, iż system ten zaimplementowany został w nieco ponad dwa miesiące, już nawet w swej obecnej, bardzo uproszczonej postaci, potrafi realizować operacje, które są nie do pomyślenia w istniejących systemach obiektowych, relacyjno–obiektowych oraz repozytoriach XML. Przewagę tę przedstawiony system osiągnąć mógł tak łatwo w tak krótkim czasie dzięki wykorzystaniu mocnych podstaw naukowych, zbudowanych na przestrzeni lat wokół tematyki systemów zarządzania bazami danych oraz języków zapytań do baz danych.

Prototyp stworzonego systemu posiada następujące cechy:

- umożliwia przechowywanie, przetwarzanie i udostępnianie prostych i złożonych obiektów (z których każdy posiada wewnętrzny identyfikator), a także związków pointerowych między nimi;
- posługuje się uniwersalnym modelem danych, zdolnym do reprezentowania danych strukturalnych, półstrukturalnych, jak i nieustrukturalizowanych;
- stosuje się do relatywizmu obiektów (wszystko jest obiektem);
- posiada skład obiektów trwałych i nietrwałych, a także możliwość podłączania dowolnych innych rodzajów składów (np. systemów spadkowych, albo rozproszonych baz danych);
- zawartość dysku buforuje w pamięci operacyjnej (bufor bloków i bufor obiektów);
- obsługuje procedury i procedury funkcyjne, a co za tym idzie — również perspektywy (także, choć w ograniczonym stopniu, aktualizowalne);
- obsługuje wiele równoczesnych sesji, a z klientami komunikuje się przez sieć komputerową za pomocą specjalnie stworzonego protokołu;

- dysponuje jednym z najsilniejszych (jeśli nie najsilniejszym) znanym obecnie językiem zapytań (łącznie z konstrukcjami imperatywnymi);
- potrafi importować i eksportować dokumenty XML, zawierające najważniejsze konstrukcje tego języka (konstrukcje nieobsługiwane można bardzo łatwo dodać w przyszłości).

Yaod jest otwarty i podatny na rozbudowę, zatem bardzo prawodpodobne wydaje się, iż prace nad nim będą kontynuowane w przyszłości, a system wzbogaci się o szereg brakujących właściwości, np. system typów, klasy, dynamiczne role obiektów, optymalizator, katalog, transakcje, współbieżność, bezpieczeństwo i ochronę danych itd.

Tekstowa część praca podzielona została na 6 rozdziałów i 6 dodatków. W rozdziale pierwszym przestawiono szczegółowo motywacje, jakie przyświecały autorom podczas tworzenia systemu Yaod, a które mają swe źródła w niedoskonałościach współczesnych technologii baz danych. Rozdział drugi prezentuje najważniejsze problemy w zakresie budowy systemów zarządzania bazami danych, ze szczególnym uwzględnieniem osiągnięć w zakresie obiektowych baz danych. Rozdział trzeci opisuje budowę i działanie systemu Yaod, jako systemu zarządzania bazą danych zdolnego do obsługi mechanizmów związanych z obiektowością, oraz pełnienia roli nowoczesnego repozytorium XML z wyjątkowo silnym językiem zapytań, stworzonym w oparciu o podejście stosowe (SBA) do budowy tego typu języków. Rozdział czwarty dotyczy wyłącznie tego języka — przedstawia jego działanie i opisuje sposób, w jaki zaimplementowany został interpreter. W rozdziale piątym skrótowo opisano najważniejsze narzędzia wykorzystane do implementacji systemu Yaod, a w rozdziale szóstym podsumowano wyniki osiągnięte podczas tworzenia systemu, jak również naświetlono możliwe kierunki jego dalszego rozwoju. Pracę kończą dodatki uzupełniające niniejszą pracę o mniej istotne zagadnienia: słowa kluczowe języka Yaql (dodatek A), gramatyka tego języka (dodatek B), priorytety operatorów (dodatek C), specyfikacja skanera i parsera (dodatki D i E) oraz skrypt tworzący przykładową bazę danych (dodatek F). Kod źródłowy i wykonywalny systemu Yaod, razem ze szczegółową dokumentacją w formacie JavaDoc, umieszczony został na dołączonej do pracy płycie CD.

Ponieważ niniejsza praca magisterska tworzona była w zespole dwuosobowym, konieczne stało się rozdzielenie odpowiedzialności za poszczególne jej fragmenty pomiędzy członków tego zespołu. Autorzy zadecydowali o wprowadzeniu następującego podziału:

- Michał Lentner — implementacja jądra systemu i programu Yaod Workbench, abstrakt, wprowadzenie, zakończenie, rozdziały 1–3 (bez punktów 1.5 i 1.6) i 5.
- Jakub Trzetrzelewski — implementacja języka Yaql, rozdziały 1 (punkty 1.5 i 1.6) i 4–5, dodatki A–F.

Rozdział 1

XML, obiektowość, a bazy danych

1.1 Ważniejsze trendy we współczesnych bazach danych

Bazy danych stały się podstawowym składnikiem systemów informatycznych współczesnych organizacji. Zawartość baz danych, których rozmiary mierzone są coraz częściej w tera–, a nawet petabajtach, zwykle ma krytyczne znaczenie dla funkcjonowania większości firm i instytucji na całym świecie. Aby efektywnie zarządzać tak wielkimi zbiorami danych, potrzebne jest wysoce wyspecjalizowane oprogramowanie, zwane systemem zarządzania bazą danych.

1.1.1 Systemy zarządzania bazami danych

System zarządzania bazą danych (SZBD) jest zorganizowanym zbiorem narzędzi, umożliwiającym tworzenie i eksploatację jednej lub wielu baz danych. SZBD otacza bazę danych i dokonuje na niej wszystkich operacji. O przewadze baz danych i systemów zarządzania bazami danych nad innymi metodami składowania informacji decuduje wiedza i technologia rozwijana przez kilka ostatnich dziesięcioleci, a zwłaszcza następujące ich zalety:

- Niezależność danych — zwiększenie stopnia abstrakcji danych, prowadzące do uniezależnienia aplikacji od szczegółów reprezentacji tych danych oraz sposobu ich przechowywania.
- Szybki dostęp do danych — wykorzystanie wymyślnych oraz wysoce efektywnych algorytmów i struktur danych, umożliwiających błyskawiczne wyszukiwanie potrzebnych informacji.
- Ochrona danych — zastosowanie mechanizmów zabezpieczających bazę danych na wypadek wystąpienia awarii, kontrolujących dostęp do danych oraz wymuszających sprawdzanie ich integralności w taki sposób, by były wspólne dla wszystkich korzystających z nich aplikacji.
- Administracja danymi — możliwość zcentralizowanego zarządzania danymi, używanymi przez różnych użytkowników. Zmniejsza to redundancję oraz ułatwia dopasowanie struktury systemu do różnorodnych wymagań.
- Współbieżny dostęp — zaimplementowanie metod kontrolujących jednoczesny dostęp do danych przez wielu użytkowników. Każdy z tych użytkowników odnosi wrażenie, iż tylko on pracuje w danym momencie z bazą danych — w rzeczywistości system może obsługiwać tysiące równoczesnych połączeń.
- Skrócony czas tworzenia aplikacji — dostarczenie wspólnych interfejsów oraz gotowych bibliotek dla aplikacji korzystających z zawartości bazy danych.

System zarządzania bazą danych oraz baza danych muszą stosować się do pewnego modelu danych (sposobu patrzenia na dane). W wyniku pojawiających się potrzeb świata przemysłowego i osiągnięć naukowych, najpopularniejsze modele baz danych zmieniały się na przestrzeni lat, począwszy od sieciowych, poprzez hierarchiczne, aż po relacyjne (najbardziej dziś popularne).

1.1.2 Relacyjne bazy danych

Współczesne relacyjne bazy danych łączy kilka cech. Jedną z nich jest logiczna organizacja bazy danych, która widziana jest jako zbiór nazwanych, prostokątnych tablic, o nieograniczonej liczbie wierszy i o określonej z góry liczbie kolumn. Każda kolumna posiada przyporządkowaną jej nazwę oraz typ danych, a każda komórka na przecięciu kolumny i wiersza — jakąś wartość. Niektóre komórki nie muszą posiadać przypisanych wartości — sytuację taka określa się specjalnym znacznikiem Null, oznaczającym brak wartości. W każdej tabeli powinna istnieć natomiast wyróżniona kolumna lub grupa kolumn, której wartości jednoznacznie identyfikują poszczególne wiersze w tej tabeli. Kolumna (kolumny) ta nazywana jest kluczem głównym i wykorzystywana jest jako środek umożliwiający odwoływanie się do dowolnych wierszy zawierającej ją tabeli. Wartość klucza głównego jednej z tabel może być przechowywana w osobnej kolumnie tej samej lub innej tabeli. Kolumna taka nazywana jest kluczem obcym i wykorzystywana jest przy operacji złączenia tabel. Wynikiem złączenia dwóch lub więcej tabel jest jedna tabela, która w logiczny sposób reprezentuje informacje zawarte w tych tabelach. Systemy relacyjne implementują także inne operatory tzw. algebry relacyjnej: restrykcję (zwraca tabelę składającą się ze wszystkich wierszy, spełniających określone warunki), rzut (zwraca tabelę składającą się ze wszystkich wierszy wskazanej tabeli, ale tylko wybranych kolumn), iloczyn (zwraca tabelę składającą się ze wszystkich możliwych wierszy, będących kombinacjami dwóch wierszy, po jednym z każdej wskazanej tabeli), sumę (zwraca tabelę składającą się ze wszystkich wierszy występujących w jednej lub obu wskazanych tabelach), przecięcie (zwraca tabelę składającą się ze wszystkich wierszy występujących w pierwszej, ale nie występujących w drugiej ze wskazanych tabel).

1.1.3 Obiektowe bazy danych

Lata 80. ubiegłego wieku (1980 — powstanie Smalltalka) traktuje się jako początek ery obiektowości w informatyce. Obiektowość jest koncepcją bazująca na wyróżnianiu obiektów o dobrze określonych granicach oraz semantyce. Naczelną misją obiektowości jest walka z nadmierną złożonością metodyk, projektów, języków, systemów i zastosowań. Misję tę obiektowość realizuje poprzez nacisk na mechanizmy abstrakcji, mechanizmy kompozycji/dekompozycji złożonych struktur (obiektów), mechanizmy hermetyzacji i ukrywania niepotrzebnej informacji oraz własności sprzyjające abstrakcji i ponownemu użyciu. Uważa się, że dzięki mechanizmom takim, jak obiekty, klasy, hermetyzacja, polimorfizm, dziedziczenie czy późne wiązanie, obiektowość bardziej sprzyja naturalnym ludzkim własnościom, dostarczając silnych narzędzi do modelowania złożonych struktur i procesów świata rzeczywistego.

Początkowo obiektowość rozwijana była wyłącznie w kontekście języków programowania, wkrótce jednak jej zaletami zaczęli interesować się specjaliści z dziedziny baz danych. Corez częściej zaczęto zdawać sobie sprawę z ograniczeń modelu relacyjnego i opartych na nim produktów, szczególnie w zakresie modelowania pojęciowego, możliwościach opanowania nadmiernej złożoności oraz języków/środowisk do programowania aplikacji. Prostota modelu relacyjnego traktowana początkowo jako zaleta, okazała się jego największą wadą. Pojawiać się zaczęły opinie, iż model relacyjny odpowiedni był dla zastosowań tradycyjnych, takich jak zastosowania bankowe, ale jest nieadekwatny dla nowoczesnych zastosowań w takich dziedzinach, jak CAD, CAM, CASE, multimedia, OLAP, GIS, dane półstrukturalne i in. Dzięki

mechanizmom wprowadzonym w obiektowości możliwe jest pokonanie tych ograniczeń modelu relacyjnego, a także zwiększenie wydajności zespołów programistycznych (czynnik ludzki zawsze odgrywa największe znaczenie), szybkości działania systemów (nowe metody optymalizacyjne i implementacyjne, np. przemiana wskaźników, czy indeksy ścieżkowe) i utrzymania aplikacji baz danych (zmniejszenie kosztów dzięki lepszym koncepcjom związanym z modelowaniem danych). O sile tych mechanizmów świadczy chociażby to, że bazy obiektowe traktują struktury relacyjne jako przypadek szczególny, całkowicie go przykrywając (operacja odwrotna nie jest możliwa). Stworzone do tej pory produkty mają też sporo wad — nie dysponują odpowiednio wygodnymi środkami programowania aplikacji, a stosowane w nich modele obiektowe nie są dostatecznie uniwersalne.

1.1.4 Obiektowo–relacyjne bazy danych

Mimo że powolny, rozwój obiektowych baz danych wymusił na producentach systemów relacyjnych wprowadzenia pewnych elementów obiektowości w ich produktach. Z jednej strony wprowadzono zatem usprawnienia związane z obsługą danych multimedialnych (np. typy BLOB), informacji behawioralnej (reguł, metod itp., np. procedury składowane, wyzwalacze), modelowania pojęciowego (klasy, metody, dziedziczenie itp.) i środków programowania aplikacji (np. rozszerzenia SQL typu PL/SQL). To hybrydowe podejście do obiektowości spowodowało namnożenie się szeregu przygotowywanych ad hoc rozwiązań, których jedynym zadaniem jest wprowadzanie dodatkowej, „obiektowopodobnej” warstwy nad istniejącymi technologiami, a które zamiast ułatwić życie personelowi odpowiedzialnemu za tworzenie i utrzymywanie — wprowadzają zamęt w głowach programistów i administratorów, a pustki w portfelach prezesów. Przykładowo, chaos w tej dziedziny wywołał taką sytuację, iż obecnie od programisty tworzącego aplikacje w środowisku Oracle wymaga się znajomości SQL, PL/SQL, J2SE, J2EE i BC4J, a wszystko po to, aby stworzyć możliwość korzystania z niektórych właściwości obiektowości w bazach danych z poziomu języka Java.

W niniejszej pracy postaramy się udowodnić, że całkowite zerwanie z modelem relacyjnym, wprowadzenie nowych, spójnych koncepcji związanych z obiektowością, silnym językiem zapytań, jak również dążenie do minimalizacji wprowadzanych pojęć, jest w stanie zmienić w tak znaczący sposób obecny świat relacyjno–obiektowych baz danych, że wiele ze związanych z nim rozwiązań (np. prawie całe J2EE) może okazać się wartych wyrzucenia do kosza.

1.1.5 Repozytoria XML

XML (od ang. *eXtensible Markup Language*) jest językiem znaczników, służącym do definiowania innych języków tego typu. W ciągu ostatnich lat XML stał się już niemalże standardową metodą opisu danych w sieci Internet. Sukcesy tego języka w dziedzinie wymiany danych pomiędzy heterogenicznymi środowiskami sprawiły, że język ten ostatnio jest postrzegany jako panaceum na wszelkie bolączki systemów informatycznych, również systemów baz danych. W związku z tym, iż XML dotyczy jedynie plików tekstowych o składni określonej specjalnymi konstrukcjami (znacznikami) i jest pomyślany głównie jako środek służący do budowy standardów wymiany danych pomiędzy użytkownikami Webu, pojawia się coraz więcej opinii o potrzebie stworzenia nowego rodzaju baz danych, zdolnych do przechowywania dokumentów XML. Pomimo powołania grupy inicjatywnej o nazwie XML:DB, do tej pory nie udało się jednak zdefiniować takiej bazy danych. Mówi się jedynie, że system obsługujący taką bazę danych ma zapewnić możliwość obsługi XML na wejściu, jak i na wyjściu, niezależnie od tego, czy fizycznie dokumenty te przechowywane są w bazie obiektowej, relacyjnej, czy w formie plików systemu

operacyjnego.

W ostatnich latach pojawia się coraz więcej produktów zdolnych do pełnienia roli repozytoriów XML. Są to albo zupełnie nowe produkty (np. Tamino firmy Software AG), albo rozszerzenia systemów relacyjnych (np. XML DB w Oracle). Mimo osiągnięcia sukcesu komercyjnego tych produktów, na dłuższą metę rozwiązania te wydają się nieperspektywiczne i ograniczone. Ich wady mają swe źródło przede wszystkim w niedostatkach XML jako nowego modelu danych (mało dojrzałego w stosunku do stanu sztuki w dziedzinie baz danych), ale również w niedostatecznej efektywności (zdekomponowanie dokumentów do postaci elementów i zapisanie ich w tabelach relacyjnych) i w niewygodzie wykorzystania (przechowywanie w bazie danych całych dokumentów).

W niniejszej pracy pokażemy, iż istnieje możliwość stworzenia modelu danych posiadającego zalety modelu XML (np. sposób traktowania Null), ale nie posiadającego jego wad (np. brak wewnętrznej identyfikacji elementów). Zaprezentujemy również prototyp obsługującego go systemu zarządzania bazą danych (używając terminologii XML, można go nazwać „repozytorium”), oraz silny język zapytań, zostawiający daleko w tyle rozwiązania promowane przez konsorcjum W3C.

1.2 Podstawowe koncepcje w obiektowości

Obiektowość wprowadza szereg mechanizmów mających na celu stworzenie bazy intelektualnej i pojęciowej pozwalającej na budowę modeli struktur danych oraz na komunikację między ludźmi. W stosunku do modelu relacyjnego, model biektowy wprowadza znacznie więcej pojęć, które jednak posiadają nie zawsze jednakowo rozumianą semantykę. Ustandaryzowanie znaczenia poszczególnych pojęć oraz stworzenie powszechnie akceptowanego modelu obiektowego jest w związku z tym na razie mało prawdopodobne. Można natomiast mówić o pewnych typowych cechach, jakimi charakteryzują się języki i systemy określane jako „obiektowe”. Do tych cech należą m.in.:

- Złożone obiekty.
  Oprogramowanie i przechowywane dane powinno składać się ze zrozumiałych modułów zawierających struktury danych z przypisanymi do nich operacjami, czyli *obiektów*. Obiekty mogą być dowolnie duże i dowolnie złożone, ale reguły operowania obiektami nie powinny od tego zależeć. Obiekty mają przypisaną tożsamość, co oznacza, że istnieją i są identyfikowalne niezależnie od ich aktualnego stanu lub miejsca przechowywania.
- Związki.
  Obiekty mogą być powiązane związkami asocjacyjnymi, np. związek *Zatrudnia* łączy obiekt *Firma* z obiektami *Pracownik*. Powiązania są w sposób naturalny odwzorowane w strukturze danych, np. w postaci wskaźników prowadzących od obiektu do obiektu.
- Hermetyzacja.
  Hermetyzacja oznacza rozróżnienie pomiędzy interfejsem do obiektu opisującym zawartość obiektu i jego działanie, a implementacją definiującą jego budowę i sposób działania. Hermetyzacja oznacza także ukrywanie informacji niepotrzebnej na danym etapie projektowania lub programowania.
- Metody i komunikaty.
  Wszelkie operacje na obiektach wykonuje się przy pomocy metod, czyli procedur wykonywanych w środowisku wnętrza obiektu. Obiekt wykonuje jedną z przypisanych dla niego operacji po wysłaniu do niego komunikatu zawierającego jej nazwę (oraz parametry). Komunikat taki nie zależy

od szczegółów implementacji lub reprezentacji obiektu.

- Hierarchia klas i dziedziczenie.
  Klasy są organizowane w hierarchię (lub inną strukturę) zakresów znaczeniowych; klasy bardziej szczegółowe dziedziczą niezmienniki klas bardziej ogólnych. Konsekwencją tego założenia jest możliwość skorzystania z klas bardziej ogólnych przy definiowaniu klas będących ich specjalizacją: nowa klasa zawiera wszystkie wcześniej zdefiniowane cechy, plus niektóre nowe cechy. Z drugiej strony, daje to możliwość utworzenia klasy abstrakcyjnej, czyli „wyciągnięcia przed nawias" wspólnych cech pewnej grupy klas i utworzenia z nich nowej klasy.
- Przesłanianie, późne wiązanie, polimorfizm.
  Wybór nazwy dla operacji jest określony wyłącznie jej zewnętrznym, pojęciowym znaczeniem w ramach danej klasy obiektów. Wybór ten nie jest uwarunkowany jakimikolwiek właśnościami lub istnieniem innych klas. Ten sam komunikat wysłany do różnych obiektów może wywoływać różne operacje; tę własność określa się jako polimorfizm. Własność ta wymaga wiązania nazw na etapie wykonania (późnego wiązania). Przesłanianie oznacza, że metoda odziedziczona przez obiekt z klasy bardziej wyspecjalizowanej przesłania metodę o tej samej nazwie odziedziczoną z klasy bardziej ogólnej.

1.3 Podstawowe koncepcje w języku XML

Język XML jest odpowiedzią środowisk związanych z siecią WWW (skonsolidowanych w ramach konsorcjum W3C) na dające coraz częściej znać o sobie problemy związane z przetwarzaniem danych w tej sieci. Technologia ta stała się ostatnio tak popularna, iż jej rozwój wspierany jest przez największe firmy informatyczne, a niemal każda nowo tworzona aplikacja nie korzystająca w jakimś stopniu z jej dobrodziejstw, uważana jest za przestarzałą.

XML wyrósł z niedostatków języka HTML, który jest sercem sieci WWW, i jest do niego bardzo podobny składniowo. Podobieństwo to jest jednak pozorne, a swe źródło ma w pochodzeniu obu języków — oba wzorują się na starszym języku znaczników, o nazwie SGML.

Zadania stawiane przed językiem XML są zgoła odmienne od tych, które stanowią sens istnienia języka HTML. W przeciwieństwie do HTML, w XML znaczniki opisują dane, a nie ich wygląd w przeglądarce. Dzięki temu możliwe jest rozdzielenie zawartości (danych), od prezentacji (sformatowania ich w celu przedstawienia użytkownikowi, np. w przeglądace internetowej). Jest to ważna własność, ponieważ zawartość jest uniwersalna względem aplikacji i może zostać poddana dowolnemu formatowaniu. Prezentacja natomiast związana jest z konkretnym typem klienta: przeglądarką WWW (dokumenty HTML), telefonem komórkowym (dokumenty WML), aplikacją Javy (Swing) itp. Pomimo, iż dokument XML jest całkowicie odizolowany od warstwy prezentacji, może on zostać wykorzystany do wygenerowania takiej warstwy poprzez zastosowanie spokrewnionych technologii, np. XSLT. XSLT umożliwia tworzenie tzw. arkuszy stylów, których zadaniem jest opisanie reguł transformacji dokumentów XML na inne rodzaje dokumentów (np. HTML). W ten sposób twórca aplikacji może się skoncentrować na operacjach biznesowych, nie biorąc pod uwagę, jakiego rodzaju urządzenia będą wyświetlać dane — teraz czy w przyszłości.

Oprócz zadań w zakresie prezentacji danych, najważniejsze usprawnienia wprowadzone przez język XML dotyczą wymiany danych pomiędzy aplikacjami. Jednym z wyzwań, przed którymi stają twórcy współczesnych aplikacji bazodanowych, jest konieczność powiązania danych generowanych przez apli-

kacje pochodzące od różnych producentów, z różnych dziedzin aplikacyjnych, składowanych i przesyłanych pomiędzy różnymi węzłami sieci Internet. XML ułatwia tego rodzaju wymianę poprzez koncentrację na samych danych i ich kontekście, bez wiązania się z konkretnymi protokołami sieciowymi czy komunikacyjnymi. Przy użyciu XML i XSLT, aplikacje są w stanie wymieniać dane bez konieczności zarządzania i interpretowania wewnętrznych lub niezgodnych ze sobą formatów danych. XML zapewnia więc przenośność danych oraz możliwość ich efektywnej wymiany w środowiskach, gdzie występują one w różnych formatach, na różnych platformach, albo muszą być prezentowane w różnych formach czy na wielu różnorodnych urządzeniach. Właściwości XML w tej dziedzinie sprawiają, że bywa on często nazywany uniwersalnym formatem wymiany danych.

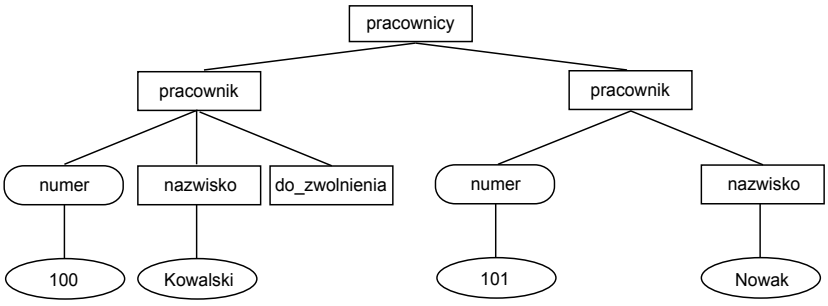
Najważniejszym pojęciem w dziedzinie XML jest dokument tego języka, który stanowi logicznie spójną porcję informacji, składającą się z jednego lub większej liczby elementów, mogących zawierać tekst oraz/lub inne elementy. Dokument XML jest więc strukturą hierarchiczną (daje się więc przedstawić w formie drzewa), którego podstawowym budulcem jest element. Zawartość każdego elementu (treść) ograniczona jest z jednej strony znacznikiem początkowym, z drugiej — znacznikiem końcowym. Oba znaczniki zawierają nazwę elementu objętą nawiasami kątowymi (< i >), przy czym w znaczniku końcowym nazwa jest zawsze poprzedzona znakiem ukośnika (/). Zarówno elementy posiadające zawartość, jak i elementy puste mogą zawierać dodatkowe informacje w formie atrybutów. Każdy atrybut posiada nazwę oraz wartość, a zapisywany jest wewnątrz znacznika i służy do zapisywania pewnych metainformacji.

W związku z tym, że ten sam dokument może zawierać elementy pochodzące z odrębnych języków (XSLT i HTML), istnieje potrzeba wprowadzenia rozróżnienia takich elementów. Przestrzeń nazw gwarantuje możliwość rozróżniania elementów pochodzących z różnych języków poprzez wprowadzenie prefiksów do nazw elementów i oddzielonych od nich dwukropkiem.

XML narzuca kilka prostych zasad dotyczących budowy dokumentów. Spełnienie tych reguł prowadzi do tzw. dokumentu poprawnego składniowo. Przykład takiego dokumenty przedstawiono poniżej:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<prac:pracownicy xmlns:prac="http://www.moj.jezyk.gr/pracownicy">
  <prac:pracownik numer="123">
    <prac:imie>Jan</prac:imie>
    <prac:nazwisko>Kowalski</prac:nazwisko>
    <prac:do_zwolnienia/>
  </prac:pracownik>
</prac:pracownicy>
```

Rysunek 1.1: Przykładowy dokument XML jako drzewo



Niektóre dokumenty XML nie posiadają tak jasno określonej struktury, jak w powyższym przykładzie.

Elementy w takich dokumentach oprócz podelementów mogą także posiadać zawartość tekstową. Dokumenty, w których występuje zagnieżdżenie elementów i tekstu w formie takiej jak powyższa, noszą nazwę dokumentów zawierających dane półstrukturalne. Oto przykład takiego dokumentu:

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<list>
  <powitanie>Szanowny Kliencie,</powitanie>
  Złożone przez Ciebie zamówienie dotyczące towaru o~nazwie
  <towar>klawiatura</towar> w~ilości <ilość>1</ilość> sztuk
  zostało przyjęte dnia <dataZamowienia>12.06.2002</dataZamowienia>.
</list>
```

Niezależnie od tego, czy dokument zawiera dane półstrukturalne czy ustrukturalizowane, niemal zawsze występuje konieczność zapewnienia zgodności tych dokumentów z pewnymi regułami określającymi ich pożądaną budowę. Reguły takie mogą na przykład określać, które elementy w danej klasie dokumentów muszą wystąpić, które są opcjonalne, jakie mają atrybuty i jak są ze sobą powiązane. Obecnie wykorzystywane są dwie podstawowe metody definiowania dopuszczalnej struktury dokumentu: za pomocą odziedziczonego po SGML mechanizmu określanego jako DTD (ang. *Document Type Definition*) lub nowszego, rozszerzającego możliwości DTD zapisu, noszącego nazwę XML Schema. Ze względu na znacznie większe możliwości drugiej z tych metod, przewiduje się, że wkrótce XML Schema całkowicie zastąpi DTD. Potrzeba zapewnienia zgodności dokumentu z jakimś DTD lub schematem XML Schema prowadzi do innego rodzaju poprawności dokumentów XML — poprawności strukturalnej.

Programista ma do dyspozycji wiele sposobów sprawdzania, czy dokument XML jest poprawny. Najczęściej czynność ta realizowana jest za pomocą programów zwanych parserami, dołączanymi zwykle do aplikacji jako biblioteki. Parser stanowi jedną z najważniejszych warstw aplikacji obsługującej XML. Oprócz sprawdzania poprawności składniowej (a czasem też strukturalnej), zadaniem tego modułu jest również taka analiza dokumentu, której wynikiem jest przeniesienie jego struktury i zawartości na grunt struktury danych jakiegoś języka programowania. Najnowsze parsery realizują zwykle dwa programistyczne modele dostępu do dokumentów: SAX i DOM. Dzięki ich wykorzystaniu programista może sięgać do dokumentów XML, operując na znanych sobie strukturach danego języka programowania, nie musząc skupiać się na składni XML.

1.4 Języki zapytań

Jednym z najważniejszych czynników, które zadecydowały o sukcesie relacyjnych baz danych jest ich język zapytań — SQL. Komercyjny sukces SQL jako wygodnego środka programowania aplikacji z bazą danych, spowodował ogromne zainteresowanie stworzeniem tego rodzaju języka również dla obiektowych baz danych oraz repozytoriów XML. Zbudowanie języka zapytań dla tych rozwiązań jest tak ważne ze względu na zdolność tej kategorii języków do [15]:

- obniżenia poziomu profesjonalizmu niezbędnego do programowania aplikacji baz danych;
- podwyższenia wydajności programistów poprzez dostarczenie do ich dyspozycji szeregu makroskopowych operacji, pozwalających zapisać złożone przetwarzanie w zwartej, czytelnej i zrozumiałej formie;
- zwiększenia niezawodności produktów programistycznych poprzez zwartość zapisu programu i konceptualizację myślenia programisty;

- zwolnienie projektanta i programisty z myślenia o mniej istotnych sprawach implementacyjnych, umożliwiając skupienie się na tym *co* powinno zostać zrobione, a nie *jak*.

Obecną systuację w zakresie języków zapytań do obiektowych baz danych można opisać jako próby dopasowania się do konwencji i konstrukcji językowych wypracowanych przez SQL, z jednoczesnym wprowadzeniem do języka istotnych cech obiektowości, takich jak złożone obiekty, związki, klasy, typy, metody, dziedziczenie i polimorfizm. Najbardziej rozwinęły się tutaj dwa kierunki, które można by określić jako matematyczne (tworzone głównie przez teoretyków) oraz niematematyczne (tworzone głównie przez praktyków).

Do pierwszej z tych grup należą teorie, takie jak różnego rodzaju algebry i logiki obiektowe. Podejścia te krytykowane są jednak za ogólną niedojrzałość, oderwanie od rzeczywistych problemów baz danych (nie obejmują wszystkich konstrukcji spotykanych w językach zapytań), brak podatności na rozszerzenia i zbytnie przywiązanie do sposobu myślenia wprowadzonego przez model relacyjny.

Istotny postęp w dziedzinie języków zapytań do języków zapytań dla obiektowych baz danych wprowadza dopiero drugi z omawianych kierunków, reprezentowany najpełniej przez język OQL, należący do standardu ODMG. OQL zachowuje składniową zgodność z SQL, wprowadzając jednak wiele nowości obsługujących mechnizmy związane z obiektowością. OQL jest jednak również powszechnie krytykowany za swoje liczne wady, z których najpoważniejszymi są: zbyt luźne zintegrowanie z konstrukcjami programistycznymi brak abstrakcji takich jak perspektywy, reguły czy też procedury składowane w bazie danych, oraz niedostatecznie precyzyjna semantyka. Programowanie w OQL wymaga zanurzania w uniwersalny jezyk programowania (np. C++, Java), co wiąże się jednak z wieloma trudnościami, określanymi jako „niezgodność impedancji”. Nieprecyzyjna semantyka powoduje trudności optymalizacyjne, a pozostałe wady tego języka powodują, że jest on niejednokrotnie uważany za nieimplementowalny. Mimo swoich wad, powstanie ODMG stanowi krok naprzód i jest bazą do budowy nowych rozwiązań, które jednak często stanowią krok do tyłu (np. koncepcje rozbudowania języka Java o elementy trwałości, takie jak PJama, czy JDO).

Prawdopodobnie jeszcze gorsza sytuacja w zakresie języków zapytań ma miejsce w stosunku do technologii XML. Dynamiczny rozwój języków uzupełniających XML o nieistniejące w nim cechy, jak również dążenie ich twórców do zbudowania teorii dotyczącej baz danych od nowa, spowodowało powstanie szeregu chaotycznych specyfikacji i „standardów” (XQL, XPath, XLink, XQuery i in.), o nikłych możliwościach, ale za to w wysokim stopniu nieprzyjemnych do implementacji i użytkowania. W istocie XML nie wnosi do tematu języków zapytań nic nowego: repozytoria XML są (ograniczonymi) hierarchicznymi bazami danych, dla ktorych języki zapytań istniały już w latach 70-ych [16]. Jak udowodniono chociażby w pracy [21], zastosowanie odpowiedniego podejścia, jest w stanie łatwo i przejrzyście rozwiązać większość problemów, z którymi borykają się twórcy języków zapytań do XML (i przy okazji — obiektowych baz danych). Podejście to wykorzystane zostało w systemie będącym tematem niniejszej pracy.

1.5 Podejście stosowe do budowy języków zapytań

Podejście stosowe do budowy języków zapytań (*Stack Based Approach*, SBA) stanowi zupełnie nowe spojrzenie na problematykę tworzenia tego rodzaju języków. Podejście to zakłada, iż języki zapytań są szczególnym przypadkiem języków programowania, w których zapytania pełnią rolę uogólnionych wyrażeń, za środowisko działania programu odpowiedzialna jest specjalna struktura danych — stos śro-

dowisk (*environmental stack*), a za wyniki przetwarzania odpowiedzialny jest stos rezultatów (*result stack*).

Teorie zbudowane na przestrzeni lat wokół SBA są podstawą, wokół której powstał również system będący tematem niniejszej pracy. Konieczne zatem wydaje się głębsze omówienie wprowadzanych przez niego pojęć (na podstawie [13, 14, 15]).

1.5.1 Stosy w SBA

Stos jest abstrakcyjną strukturą danych w którym rządzi dyscyplina kolejkowa FILO (*First In Last Out*). Operacje na stosie odbywają się zawsze na jego wierzchołku.

Standardowa obsługa stosu jest zrealizowana za pomocą czterech funkcji:

- $push(X)$ — funkcja wkładająca element X na czubek stosu,
- $pop()$ — funkcja zdejmująca ostatni element ze stosu,
- $top()$ — funkcja odczytująca ostatni element stosu,
- $empty()$ — funkcja sprawdzająca czy stos jest pusty.

Podstawą podejścia stosowego jest semantyka oparta na funkcjonowaniu dwóch stosów:

- stosu środowiskowego,
- stosu rezultatów.

1.5.2 Stos środowiskowy *ENVS*

Stos środowiskowy *ENVS* (*Environmental Stack*) reprezentuje aktualne środowisko przetwarzania. Składa się z sekcji (rekordów aktywacji) z których każda odpowiada poszczególnym środowiskom czasu wykonania. Każda taka sekcja stosu środowiskowego jest zbiorem binderów (zdefiniowanych poniżej) do bytów programistycznych.

Stos środowiskowy w językach programowania pełni następujące funkcje:

- jest odpowiedzialny za określenie oraz kontrolę zakresów nazw (*scoping*),
- jest odpowiedzialny za wiązanie nazw (*binding*),
- jest odpowiedzialny przechowywanie dynamicznych bytów programistycznych (wartości zmiennych w obrębie bloku, procedury, funkcji lub metody, parametrów procedury, funkcji lub metody, śladu powrotu z funkcji, wartości wynikające z przetwarzania operatora niealgebraicznego itp.).

Reguły zakresu

Reguły zakresu są częścią ograniczeń kontekstowych, których sprawdzenie jest zadaniem analizy semantycznej. Są odpowiedzialne za deklaracje oraz wystąpienia nazw użytych w programie.

Istnieją języki nie posiadające reguł zakresu (*Prolog*) lub posiadające reguły w znacznym zakresie ograniczonym. W językach tych wszystkie zmienne widziane są na tym samym globalnym poziomie (struktura monolityczna — *Basic*, *Cobol*) lub istnieje sztywny podział zakresu nazw na globalne i lokalne bez możliwości dowolnego zagnieżdżania bloków w blokach (struktura płaska — *Fortran*). Rozwiązania

takie znacznie komplikują czytelność, a przede wszystkim semantykę języka, co jest sprzeczne z naczelną zasadą podejścia stosowego dotycząca semantyki, która mówi, iż każdy, nawet najmniejszy błąd semantyczny, jest dużym problemem semantycznym.

W podejściu stosowym reguły zakresu są zdefiniowane rekurencyjnie na podstawie struktury zagnieżdżonych bloków (podobnie jak w *C/C++*, *Pascal*, *Java*). Oznacza to, iż w obrębie każdego bloku może być dowolnie zagnieżdżony każdy inny blok.

Regułami zakresu rządzą następujące zasady:

1. Zasada priorytetu lokalnego środowiska — przy wiązaniu nazw lokalne środowisko ma priorytet przed dowolnym środowiskiem bardziej globalnym.
2. Zasada leksykalnego/statycznego zakresu (*lexical/static scoping*) — nazwa nie może być wiązana do bytu programistycznego, którego nie mógł być świadomy programista w momencie pisania programu.
3. Zasada dowolnego zagnieżdżania wołań procedur — programista piszący procedurę może, bez żadnych ograniczeń, wołać w niej inne procedury (rekurencyjne, pośrednie lub bezpośrednie).

Wiązanie nazw

Elementarną strukturą przechowywaną na sekcjach stosu środowisk jest *binder*. Każdy binder reprezentowany jest jako para (n, x) lub $n(x)$, gdzie n jest zewnętrzną nazwą dostępna w programie (nazwa zmiennej, stałej, obiektu, procedury itd.), a x jest bytem czasu wykonania (referencją do obiektu, literałem itp.). W uogólnionej postaci bindera, x jest dowolnym rezultatem zwracanym przez wyrażenia. W innym kontekście *binder* jest traktowany jako nazwana wartość.

Zadaniem bindera $n(x)$ jest wiązanie (*binding*) nazw, czyli zastąpienie nazwy n występującej w programie na wartość x, będącą bytem czasu wykonania. Dla dowolnej nazwy n występującej w programie, na stosie środowiskowym musi istnieć odpowiedni binder posiadający tę nazwę, w przeciwnym przypadku nazwa taka nie może zostać związana (jest błędna).

W językach z silnym typowaniem (*statically/strongly typed languages*), czyli takich w których błędy typów są wykrywane przed uruchamianiem programu (na poziomie analizy semantycznej), sytuacja taka jest traktowana jako błędna i przerywa wykonywanie programu (kompilacje programu). Przykładami takich języków jest większość języków kompilowanych — *C*, *C++*, *Pascal*, *Java*, *Prolog*. W językach z luźnym typowaniem (*dynamically/loosely typed languages*), czyli takich w których błędy typów mogą być wykryte tylko podczas wykonywania programu (*runtime*), wiązanie takiej nazwy jest często dopuszczalne (wynik jest często wartością odpowiadająca zeru w zależności od kontekstu użycia — 0, 0.0, *""*, *null*, pusty zbiór itp.). Przykładami takich języków jest większość języków interpretowanych — *Perl*, *AWK*, *PHP*, *ASP*.

Wiązanie może być wczesne/statyczne (*early/static binding*), czyli takie w którym informacja o wiązanych bytach czasu wykonania może być wydobyta przed uruchomieniem programu (na poziomie analizy semantycznej), lub późne/dynamiczne (*late/dynamic binding*), czyli takie w którym informacja o wiązanych bytach czasu wykonania może być wydobyta tylko podczas wykonywania programu.

Reguły wiązania nazw na stosie środowiskowym są następujące:

- Dla wiązanej nazwy n, stos środowiskowy jest przeszukiwany od czubka w dół aż do znalezienia sekcji w której znajduje się binder opatrzony nazwą n.
- Podczas przechodzenia przez sekcje, stosowane są reguły zakresu, które nakazują omijanie pewnych sekcji stosu.
- Po znalezieniu takiej sekcji, przeszukiwanie jest zakończone.
- Wszystkie bindery w danej sekcji oznaczone nazwą n uczęstniczą w tworzeniu rezultatu przeszukiwania.
- Rezultatem wiązania jest zbiór (kolekcja) wartości wszystkich znalezionych binderów.

W podejściu stosowym funkcja realizująca wiązanie nazw została nazwana *bind*. Przeszukuje ona stos środowiskowy od jego wierzchołka do jego podstawy, z pominięciem niektórych sekcji, stosując ustalone reguły zakresu oraz wiązania. Argumentem tej funkcji jest nazwa n, a wynikiem rezultat wiązania zgodny z opisanymi wyżej zasadami.

Nowe sekcje stosu środowiskowego

W wiekszości języków programowania, nowa sekcja stosu środowisk jest otwierana wtedy, gdy zostaje wywołana procedura lub gdy zostaje otwarty nowy blok. Analogicznie skasowanie tej sekcji, następuje w momencie zakończenia procedury lub zamknięcia bloku.

W podejściu stosowym istnieje jeszcze dodatkowa sytuacja powodująca otworzenie nowej sekcji na stosie środowisk, która stanowi istotę tego podejścia do języków zapytań. Jest nią ewaluacja wyrażeń zawierających pewne operatory (nazwane niealgebraicznymi), których realizacja jest podobna do wywołania procedury lub otwierania nowych bloków. Istota takiej ewaluacji polega na tym, iż wyrażenie będące po prawej stronie takiego operatora jest ewaluowane w środowisku określonym przez „wnętrze" wyrażenia znajdującego się po lewej stronie takiego operatora (z reguły będzie to identyfikator obiektu).

Funkcja *nested* jest właśnie odpowiedzialna za określenie „wnętrza" wyrażenia znajdującego się po lewej stronie tego operatora. W najprostszej postaci, funkcja taka oczekuje identyfikatora obiektu, a jej wynikiem jest wewnętrzne środowisko tego obiektu (zbiór identyfikatorów podobiektów). Dodatkowo z wszystkich identyfikatorów są tworzone bindery w których nazwa jest nazwą wydobytą z obiektu na podstawie danego identyfikatora, a wartość to po prostu dany identyfikator.

Powyższa definicja funkcji *nested* jest wystarczająca do prostych instrukcji, jednakże w przypadku bardziej zaawansowanych instrukcji, istnieje potrzeba uogólnienia tej definicji:

- dla dowolnej wartości atomowej $v \in V \rightarrow$ *nested*$(v) = \emptyset$ (zbiór pusty),
- dla identyfikatora i obiektu atomowego $\langle i, n, v\rangle \rightarrow$ *nested*$(i) = \emptyset$,
- dla identyfikatora i obiektu pointerowego $\langle i, n, i_1\rangle$, dla którego istnieje w składzie obiekt $\langle i_1, n_1, v_1\rangle \rightarrow$ *nested*$(i) = \{n_1(i_1)\}$,
- dla identyfikatora i obiektu złożonego $\langle i, n, \{\langle i_1, n_1, \rangle, \langle i_2, n_2, \rangle, \ldots, \langle i_k, n_2, \rangle\}\rangle \rightarrow$ *nested*$(i) = \{n_1(i_1), n_2(i_2), \ldots, n_k(i_k)\}$,
- dla dowolnego bindera $n(x) \rightarrow$ *nested*$(n(x)) = \{n(x)\}$,
- dla dowolnej struktury, wynik jest sumą mnogościową rezultatów funkcji *nested*$(\ldots)$ wywołanej dla pojedynczych elementów tej struktury $\rightarrow$ *nested*$(struct\{x_1, x_2, \ldots, x_k\}) =$ *nested*$(x_1) \cup$ *nested*$(x_2) \ldots \cup \ldots$ `nested`(x_k).

1.5.3 Stos rezultatów *QRES*

Stos rezultatów *QRES* (*Query Result Stack*) jest uogólnionym stosem arytmetycznym znanym w większości języków programowania. Jest odpowiedzialny za przetrzymywanie pośrednich oraz końcowych wyników poszczególnych wyrażeń.

Z semantycznego punktu widzenia, stos rezultatów ma znaczenie drugorzędne, gdyż semantyka języka równie dobrze może być wyrażona przy pomocy funkcji rekurencyjnych zwracających rezultaty. Takie rozwiązanie też jest możliwe, ale jest dość trudne oraz mało elastyczne, podczas gdy posłużenie się zewnętrzną pamięcią jaką jest stos rezultatów, poprawia czytelność, zwiększa modularność oraz uwalnia programistę od myślenia jaka i jakiego typu wartość powinna być zwrócona przez daną funkcję.

Elementy jakie będą przechowywane na stosie rezultatów to elementy zbioru *REZULTAT* zdefiniowanego poniżej.

Rezultaty zwracane przez zapytania

W podejściu stosowym zapytania nigdy nie zwracają obiektów, lecz referencje (identyfikatory) do obiektów. Dodatkowymi wartościami jakie mogą być przetwarzane w trakcie ewaluacji programu są wartości atomowe (liczby rzeczywiste, liczby całkowite, wartości logiczne, łańcuchy znaków) oraz kolekcje (struktura, zbiór, sekwencja). Podejście stosowe wprowadza jeszcze jeden typ rezultatu, nie spotykany *explicite* w innych podejściach. Jego istota polega na tym, iż każdy rezultat może być opatrzony nazwą, która podlega takim samym regułom zakresu i wiązania, które zostały zrealizowane w postaci mechanizmu stosu środowisk.

Podział wstępnie przedstawiony powyżej, został zdefiniowany rekurencyjnie jako zbiór *REZULTAT*:

- rezultat będący wartością pojedynczą należy do zbioru *REZULTAT*:
  - każda wartość atomowa należąca do zbioru V (np. 25, „Kowalski”, 3.14) należy do zbioru *REZULTAT*,
  - każda referencja (identyfikator) do obiektu dowolnego typu (obiekt prosty, pointerowy, złożony, procedura/funkcja, perspektywa itp.) należący do zbioru I, należy do zbioru *REZULTAT*,
  - jeżeli $x \in$ *REZULTAT*, zaś $n \in N$ jest dowolną nazwą, wówczas każda para $n(x)$ należy do zbioru *REZULTAT*. Taki rezultat (będący nazwaną wartością), będziemy nazywać *binder*.
- rezultat będący wartością złożoną należy do zbioru *REZULTAT*:
  - jeżeli $x_1, x_2, \ldots \in$ *REZULTAT*, wówczas *struct*$(x_1, x_2, \ldots) \in$ *REZULTAT*. *struct* jest kolekcją rezultatów będącą strukturą, kolejność elementów w takiej kolekcji może mieć znaczenia.
  - jeżeli $x_1, x_2, \ldots \in$ *REZULTAT*, wówczas *bag*$(x_1, x_2, \ldots) \in$ *REZULTAT*. *bag* jest kolekcją rezultatów będącą zbiorem, kolejność elementów w takiej kolekcji nie ma znaczenia.
  - jeżeli $x_1, x_2, \ldots \in$ *REZULTAT*, wówczas *sequence*$(x_1, x_2, \ldots) \in$ *REZULTAT*. *sequence* jest kolekcją rezultatów będącą sekwencją, kolejność elementów w takiej kolekcji ma znaczenie.
- zbiór *REZULTAT* nie zawiera innych rezultatów.

1.5.4 Założenia syntaktyczne i semantyczne języka

W podejściu stosowym pojęcie stanu jest definiowane jako aktualny stan składu obiektów (omówiony w nastepnym punkcie) wraz z aktualnym stanem stosu środowisk. Semantyka konstrukcji języka, jest funkcją odwzorowywująca stan, w rezultat takiej konstrukcji, przy czym gdy konstrukcja taka jest imperatywna (aktualizująca), to wynikiem takiej konstrukcji, jest nowy stan, a jeżeli konstrukcja taka posiada efekty uboczne (np. jest to procedura powodująca zmiany w bazie danych), to wynikiem jest rezultat tej konstrukcji oraz nowy stan. Powyższa koncepcja została formalnie zapisana poniżej (q oznacza pewną konstrukcje programistyczną (wyrażenie, instrukcje), a $|q|$ jej semantykę):

- $|q|$: *Stan* $\rightarrow$ *Rezultat*,
- $|q|$: *Stan* $\rightarrow$ *Stan*,
- $|q|$: *Stan* $\rightarrow$ (*Rezultat* $\times$ *Stan*).

Główną zasadą, która przyświeca konstrukcji składni oraz semantyki języka w podejściu stosowym jest zasada kompozycyjności, która mówi, że semantyka całości wyrażenia jest funkcją semantyk wszystkich części tego wyrażenia. Podstawowe założenia syntaktyczne, oparte na tej zasadzie, są następujące:

- literały reprezentujące wartości atomowe (2, „Kowalski”, 3.14) są elementarnymi zapytaniami (zbiór L),
- nazwy zmiennych (`x`, `EMPLOYEE`) są elementarnymi zapytaniami (zbiór N),
- zapytania można łączyć w większe zapytania przy pomocy operatorów algebraicznych (unarnych, binarnych lub ternarnych),
- zapytania można łączyć w większe zapytania przy pomocy operatorów niealgebraicznych (binarnych),
- zapytania można łączyć w większe zapytania przy pomocy instrukcji.

Definicja semantyki języka zapytań opartego na podejściu stosowym została zaproponowana w sposób operacyjny jako procedura rekurencyjna *eval*. Procedura ta korzysta ze składu obiektów, stosu *ENVS* oraz *QRES* i jest realizacją zasady kompozycyjności. Argumentem takiej procedury może być dowolne zapytanie/wyrażenie lub instrukcja. W przypadku wyrażeń wynikiem jest rezultat tego wyrażenia wkładany na wierzchołek stosu *QRES*. W przypadku instrukcji, wynikiem jest odpowiednia zmiana (lub też żadna) spowodowana charakterem instrukcji na stosie *ENVS* oraz w składzie danych.

Kompletny opis semantyki języka za pomocą procedury *eval*, jest następujący:

- jeżeli argumentem jest literał $l \in L$, to na stos *QRES* wkładana jest odpowiadająca mu wartość atomowa $l \in V$,
- jeżeli argumentem jest nazwa $n \in N$, to na stos *QRES* wkładany jest wynik wywołania funkcji *bind(*n*)* dla tej nazwy,
- jeżeli argumentem jest operator algebraiczny:
  - i jest on unarny, to następuje ewaluacja operanda (wynik znajdzie się na stosie *QRES*). Następnie wynik tej ewaluacji zostaje zdjęty ze stosu *QRES* i wykorzystany w celu obliczenia ostatecznego wyniku, który zostaje włożony na stos *QRES*. Sposób obliczenia tego wyniku uzależniony jest od konkretnego operatora unarnego,

  - i jest on binarny, to następują ewaluacje operandów (wyniki znajdą się na stosie *QRES*). Następnie wyniki tych ewaluacji zostają zdjęte ze stosu *QRES* i wykorzystane w celu obliczenia ostatecznego wyniku, który zostaje włożony na stos *QRES*. Sposób obliczenia tego wyniku uzależniony jest od konkretnego operatora binarnego,
  - i jest on ternarny, to następują ewaluacje operandów (wyniki znajdą się na stosie *QRES*). Następnie wyniki tych ewaluacji zostają zdjęte ze stosu *QRES* i wykorzystane w celu obliczenia ostatecznego wyniku, który zostaje włożony na stos *QRES*. Sposób obliczenia tego wyniku uzależniony jest od konkretnego operatora ternarnego,

- jeżeli argumentem jest binarny operator niealgebraiczny to wykonywane są następujące kroki:
  1. dokonywana jest ewaluacja operanda znajdującego się po lewej stronie operatora (wynik znajdzie się na stosie *QRES*),
  2. dla każdego elementu e należącego do wyniku tego zapytania, wykonywane są następujące czynności:
     - obliczana jest wartość *nested(*e*)*,
     - wynik jest wkładany jako nowa sekcja na stos *ENVS*,
     - dokonywana jest ewaluacja operatora znajdującego się po prawej stronie operatora,
     - w pewien zależny od operatora sposób, obliczany jest wynik cząstkowy poprzez połączenie elementu e z wynikiem ewaluacji operanda znajdującego się po prawej stronie operatora,
     - usuwana jest nowo wstawiona górna sekcja stosu *ENVS*,
  3. w pewien zależny od operatora sposób, obliczany jest wynik ostateczny na podstawie wszystkich wyników cząstkowych,

- jeżeli argumentem jest instrukcja, to wykonywane wszystkie instrukcje oraz wyrażenia, z których owa instrukcja się składa. Wynik ewaluacji nie pozostawia rezultatu na stosie *QRES*.

1.6 Modele składu obiektów w SBA

Kluczową role w modelach składu obiektów zaproponowanych w podejściu stosowym, pełnią dwie zasady:

- Zasada wewnętrznej identyfikacji — każdy byt programistyczny, który może być w jakikolwiek sposób przetwarzany (wyszukiwany, wiązany, aktualizowany, usuwany, wstawiany, indeksowany, autoryzowany, zabezpieczany, blokowany itp.) musi posiadać swój unikalny wewnętrzny identyfikator (referencje),
- Zasada relatywizmu obiektów — każdy obiekt złożony jest zestawem pod–obiektów, które mogą być złożone lub atomowe. Każdy z podobiektów może być traktowany jako samodzielny obiekt. Ogólne własności obiektów na dowolnych poziomach hierarchii są identyczne.

We wszystkich modelach obiektów zostały zastosowane trzy zbiory:

- I — zbiór wewnętrznych identyfikatorów obiektów,
- N — zbiór zewnętrznych nazw obiektów,
- V — zbiór atomowych wartości (liczby rzeczywiste, liczby całkowite, łańcuchy znakowe, wartości logiczne, wartości BLOB, skompilowane procedury itp.).

Każdy obiekt składu może być trwały lub ulotny oraz posiada następujące cechy:

- wewnętrzny identyfikator — identyfikujący w sposób jednoznaczny każdy obiekt ze składu. Identyfikatory są nadawane przez system, muszą być unikalne na każdym poziomie hierarchii danych oraz nie mogą być bezpośrednio użyte w zapytaniach. Identyfikator taki jest obywatelem drugiej kategorii obywatelskiej,
- zewnętrzna nazwa — nadawana przez programistę lub projektanta bazy danych, umożliwia dostęp do obiektów składu z programu. W odróżnieniu od identyfikatora, zewnętrzna nazwa nie musi być unikalna na każdym poziomie hierarchii danych. Nazwa taka jest obywatelem pierwszej kategorii obywatelskiej,
- zawartość obiektu — wartością taką może być wartość atomowa należąca do zbioru V, referencja innego obiektu należąca do zbioru I lub wartość złożona (zbiór obiektów).

1.6.1 Model składu *M0*

Model *M0* obejmuje dowolnie powiązane hierarchicznie struktury danych, przykrywa struktury implikowane przez *XML*, koncepcję zagnieżdżonych relacji oraz dane określane jako pół–strukturalne. Model ten nie zakłada abstrakcji programistycznych bliższych programowaniu obiektowo zorientowanemu (klasy, dziedziczenie, hermetyzacja itp.).

W modelu *M0* każdy obiekt reprezentowany jest jako trójka $\langle i, n, v\rangle$. Przy założeniu że $i, i_1, i_2, \ldots, i_k \in I$, $n, n_1, n_2, \ldots, n_k \in N$, $v, v_1, v_2, \ldots, v_k \in V$ możliwe obiekty modelu *M0* zamodelowane są w następujący sposób:

- $\langle i, n, v\rangle$ — trójka reprezentująca *obiekt atomowy*,
- $\langle i, n, i_1\rangle$ — trójka reprezentująca *obiekt pointerowy* lub *referencyjny*,
- $\langle i, n, \{\langle i_1, n_1, v_1\rangle, \langle i_2, n_2, v_2\rangle, \ldots, \langle i_k, n_k, v_k\rangle\}\rangle$ — trójka reprezentująca *obiekt złożony* (wartością jest zbiór innych obiektów o liczności k).

W modelu *M0* skład obiektów jest zdefiniowany jako para uporządkowana $\langle S, R\rangle$, gdzie:

- S to zbiór obiektów,
- R to zbiór identyfikatorów obiektów korzeniowych (startowych).

Definicja obiektu jest rekurencyjna i umożliwia swobodne tworzenie złożonych obiektów na dowolnych poziomach hierarchii. Związki są modelowane za pomocą obiektów pointerowych.

Propozycja powyższego modelu przykrywa wszystkie relacyjne, relacyjno–obiektowe struktury danych oraz model danych zaproponowany w języku XML.

1.6.2 Model składu *M1*

Model *M1* uzupełnia model *M0* o pojęcia klasy, dziedziczenia oraz wielokrotnego dziedziczenia. Klasa jest obiektem, podobnym do wprowadzonych poprzednio obiektów, który przechowuje inwarianty innych obiektów.

W modelu *M1* skład obiektów jest zdefiniowany jako uporządkowana czwórka $\langle S, R, KK, OK\rangle$, gdzie:

- S to zbiór obiektów (rozszerzonym o klasy),
- R to zbiór identyfikatorów obiektów korzeniowych (startowych),
- $KK \subseteq I \times I$ wyznacza związki dziedziczenia pomiędzy klasami,
- $OK \subseteq I \times I$ wyznacza związki dziedziczenia pomiędzy obiektami i klasami.

Dla każdej pary $\langle i_L, i_R \rangle \in OK$, i_L oznacza identyfikator obiektu, który dziedziczy z klasy oznaczonej identyfikatorem i_R.

Dla każdej pary $\langle i_L, i_R \rangle \in KK$, i_L oznacza identyfikator klasy, która dziedziczy z klasy oznaczonej identyfikatorem i_R.

1.6.3 Model składu *M2*

Model *M2* uzupełnia model *M1* oraz nieco go modyfikuje wprowadzając dziedziczenie pomiędzy obiektami oraz dynamiczne role. W modelu tym obiekty mogą dziedziczyć z innych obiektów na takiej samej zasadzie jak obiekty dziedziczą z klas. Obiekty dziedziczące z innych obiektów, w terminologii SBA, nazywane są rolami

W modelu *M2* skład obiektów jest zdefiniowany jako uporządkowana piątka $\langle S, R, KK, OK, OO \rangle$, gdzie:

- S to zbiór obiektów (rozszerzonym o klasy),
- R to zbiór identyfikatorów obiektów korzeniowych (startowych),
- $KK \subseteq I \times I$ wyznacza związki dziedziczenia pomiędzy klasami,
- $OK \subseteq I \times I$ wyznacza związki dziedziczenia pomiędzy obiektami i klasami.
- $OO \subseteq I \times I$ wyznacza związki dziedziczenia pomiędzy obiektami.

Dla każdej pary $\langle i_L, i_R \rangle \in OK$, i_L oznacza identyfikator obiektu, który dziedziczy z klasy oznaczonej identyfikatorem i_R.

Dla każdej pary $\langle i_L, i_R \rangle \in KK$, i_L oznacza identyfikator klasy, która dziedziczy z klasy oznaczonej identyfikatorem i_R.

Dla każdej pary $\langle i_L, i_R \rangle \in OO$, i_L oznacza identyfikator obiektu, który dzidziczy z obiektu oznaczonego identyfikatorem i_R.

1.6.4 Model składu *M3*

Model *M3* uzupełnia model *M1* oraz *M2* o pojęcie hermetyzacji czyli podział własności klas i obiektów na publiczne i prywatne. Model ten uzupełnia skład w taki sposób, iż każda klasa wyposażona jest w dodatkowy inwariant — listę eksportową. Jest ona zbiorem nazw własności tej klasy i jej obiektów (metod, atrybutów), które będą widoczne z zewnątrz.

Idea hermetyzacji polega na tym, aby w określonych sytuacjach zabronić dostępu do pewnych własności obiektów, określanych jako „prywatne”. Istota tego dostępu polega na tym, iż własności prywatne dostępne są z „wnętrza” obiektu, zaś niedostępne są z „zewnątrz”.

Rozdział 2

Podstawowe problemy projektowe w systemach zarządzania obiektowymi bazami danych

Wiele decyzji, jakie musi podjąć projektant systemu zarządzania bazą danych jest wspólna dla każdego takiego systemu i nie zależy od wybranego modelu danych. W niniejszym rozdziale opisano najważniejsze problemy projektowe, z jakimi musieli zmierzyć się autorzy niniejszej pracy: ich symptopmy, znane rozwiązania, oraz rozwiązania wprowadzone w systemie Yaod.

2.1 Budowa i funkcjonowanie systemów zarządzania bazami danych

Istniejące na rynku systemy zarządzania bazami danych różnią się od siebie w znaczącym stopniu. Wydaje się jednak, iż wśród wiądących systemów wyróżnić można pewne wspólne elementy w ich budowie i funkcjonowaniu.

Użytkownik korzystający z zawartości bazy danych wykorzystuje przeznaczone do tego celu oprogramowanie, zwane oprogramowaniem klienckim. Oprogramowanie to może być dostarczone przez producenta SZBD, albo stworzone przez inną firmę (np. za pomocą języka Java). Pozornie oprogramowanie to może nie różnić się prawie niczym od innych aplikacji systemu operacyjnego — użytkownik może nawet nie wiedzieć, że pracując z aplikacją (np. sklepem internetowym), w rzeczywistości korzysta z usłuch SZBD.

Program po stronie klienta łączy się (zwykle poprzez sieć komputerową) z serwerem bazy danych, czyli komputerem, na którym znajduje się baza danych oraz jej system zarządzania. Po uzyskaniu połączenia z bazą danych, użytkownik jest uwierzytelniany, a następnie tworzona jest tzw. sesja, która trwa do momentu rozłączenia się użytkownika z bazą danych. W ramach sesji użytkownik może wysyłać do serwera polecenia oraz odczytywać z niego wyniki realizacji tych poleceń. Zwykle wszelkie polecenia wysyłane do serwera zapisywane są w specjalnie przeznaczonym do tego celu języku, noszącym nazwę języka zapytań (zwykle jest to SQL). Zapytania są przetwarzane przez kompilator zapytania, który generuje plan wykonania zapytania. Plan taki zawiera szczegółową listę niskopoziomowych operacji, jakie muszą w odpowiedniej kolejności zostać wykonane przez system w celu zrealizowania zapytania. Ze względu na nieproceduralny charakter języków zapytań, istnieje potencjalnie wiele semantycznie równoważnych planów wykonania zapytania, jednak wykonywany jest tylko ten, który oceniony zostanie najlepiej pod

względem szybkości realizacji i stopnia wykorzystania zasobów. Oceny planów wykonania zapytań dokonuje moduł SZBD noszący nazwę optymalizatora, posługując się zestawem ściśle określonych reguł i rankingiem operacji (optymalizacja regułowa), albo/i zebranym uprzednio zestawem statystyk opisujących zawartość bazy danych (optymalizacja kosztowa).

Wygenerowany plan wykonania zapytania trafia następnie do silnika bazy danych, który krok po kroku realizuje zapisane w nim informacje, żądając od modułu zarządzania zasobami danych w postaci małych ich porcji. Moduł zarządzający zasobami posiada przede wszystkim wiedzę o plikach danych, jak i formacie i rozmiarach poszczególnych rekordów. System próbuje przewidywać które fragmenty bazy danych wykorzystywane będą w najbliższej przyszłości, buforując je w pamięci operacyjnej.

Operacje wykonywane przez użytkowników na bazie danych grupowane są w ramach tzw. transakcji. Transakcja jest logiczną jednostką pracy, zawierającą jedną lub więcej instrukcji języka zapytań (zwłaszcza instrukcji aktualizujących). Transakcja jest również jednostką atomową — efekt działania transakcji może zostać jedynie w całości zatwierdzony (utrwalony w bazie danych), albo wycofany (usunięty z bazy danych). Wycofanie transakcji po awarii systemu gwarantują specjalne dzienniki transakcji (logi), w których system zapisuje (zwykle z wyprzedzeniem) wszystkie modyfikacje wprowadzone w bazie danych.

Mechanizm transakcji jest ściśle powiązany z modułem sterowania współbieżnością, który zapewnia możliwość pracy jednoczesnym transakcjom. Ponieważ nie jest możliwe efektywne zaimplementowanie obsługi współbieżności w taki sposób, by każda transakcja była całkowicie odizolowana od innych transakcji (zgodnie z zasadą ACID), dlatego systemy zarządzania wprowadzają rozwiązania pośrednie, nazywane poziomami izolacji transakcji. Standard SQL definiuje cztery tryby izolacji transakcji (serializable, repeatable read, read commited, read uncommied)[1], różniące się restrykcyjnością izolacji (bardziej restrykcyjne — mniejsza współbieżność, mniej restrykcyjne — możliwość powstawania niespójności).

Głównym składnikiem implementacji idei izolacji transakcji jest mechanizm tzw. blokad (zamków). Poprzez ustanowienie blokady na danym elemencie bazy danych (np. obiekcie) tymczasowo uniemożliwia się równoczesnym transakcjom przeprowadzanie na nim pewnych operacji mogących zakłócić spójność bazy danych. Blokady różnią się od siebie stopniem restrykcyjności — bardziej restrykcyjne mogą np. zablokować zapis i odczyt obiektu, mniej — tylko odczyt.

2.2 Przechowywanie danych

W typowym systemie komputerowym jest wiele części, które mogą być przeznaczone do przechowywania danych: od rejestrów CPU, poprzez pamięć podręczną procesora, pamięć operacyjną, pamięć dyskową, taśmową, po sieć komputerową. Części te różnią się się od siebie znacząco pod względem objętości przechowywanych danych, szybkości funkcjonowania oraz ceny w przeliczeniu na 1 bajt. Obowiązuje zasada mówiąca, iż im szybsze działanie, tym większa cena i mniejsza maksymalna objętość przechowywanych danych. Przykładowo (wg danych z 2000 r.), typowy czas dostępu między CPU i pamięcią operacyjną wynosi ok. 100 ns, a między pamięcią i lokalnym dyskiem — $3 \cdot 10^7$ ns. Typowy zakres pojemności pierwszego z tych rodzajów pamięci wynosi za to zaledwie kilka kilobajtów, podczas gdy drugiego — liczona jest w gigabajtach. Również cena przechowywania jednego bajta na dysku jest wie-

[1] Niektóre systemy zarządzania bazami danych (np. Oracle) wprowadzają własne poziomy izolacji transakcji.

lokrotnie mniejsza, niż w pamięci SRAM.

W kontekście współczesnych systemów zarządzania bazami danych największe znaczenie mają dwa rodzaje urządzeń służących do przechowywania danych:

1. dysk twardy,
2. pamięć operacyjna.

2.2.1 Dysk twardy

Ze względu na wymaganie obsługi bardzo dużych zbiorów danych, od wielu lat podstawowym urządzeniem służącym do przechowywania danych jest dysk twardy. Do zalet dysków twardych należą niska cena i możliwość przechowywania dużej ilości danych, a do wad — prędkość działania i wysoka awaryjność[2]. Wady dysków są na tyle istotne, iż zazwyczaj mają decydujący wpływ na wiele decyzji architektonicznych podejmowanych przy projektowaniu systemu zarządzania bazą danych.

Dysk twardy złożony jest z kilku talerzy ułożonych jeden na drugim, z zachowaniem odstępu umożliwiającego ruch głowic odczytująco–zapisujących. Dane są przechowywane na talerzu w ciągu koncentrycznych pierścieni, określanych jako ścieżki. Każda ścieżka składa się z sektorów. Sektor może mieć pojemność np. 512 bajtów i reprezentuje najmniejszą jednostkę, jaką dysk może przesłać. Grupa ścieżek o tym samym numerze, ale umieszczonych na osobnych talerzach, nosi nazwę cylindra.

Charakterystyka budowy dysku twardego ma wpływ na dwa podstawowe wskaźniki, które powinny znajdować się w centrum uwagi przy projektowaniu systemu wejścia–wyjścia:

1. Czas przeszukiwania.
   By uzyskać dostęp do określonych danych, głowice dysku muszą przemieścić się nad scieżkę zawierającą potrzebne dane. Ta operacja nazywana jest przeszukiwaniem (*seek*).

   Średni czas przeszukiwania dla większości współczesnych dysków wynosi kilka ms. Na czas ten wpływ mają dwa podstawowe czynniki:

   (a) Profil wykorzystania dysku.
   Dostęp do dysku może mieć charakter sekwencyjny (dostęp do kilku kolejnych bloków) oraz swobodny (punktowy dostęp do pojedynczych bloków, np. w wyniku wykorzystania indeksu). W pierwszym przypadku czas przeszukiwania jest minimalny, ponieważ przesunięcie głowicy do następnej ścieżki może mieć miejsce tylko wtedy, gdy odczytano ostatni blok ścieżki bieżącej. Przypadek drugi jest bardziej kosztowny, ponieważ oznacza konieczność dużo częstszej zmiany ścieżki.

   (b) Rozłożenie danych na dysku.
   W celu zminimalizowania przeszukiwanej przestrzeni, dane powinny być zapisywane na dysku w taki sposób, by były rozłożone jak najbliżej siebie. Nowoczesne dyski twarde zapisują dane na dysku stosując tzw. strefowy zapis bitowy (*zoned bit recording*), dzięki czemu dla każdego centymetra kwadratowego dysku zapisywana jest ta sama liczba bitów, niezależnie od umiejscowienia na dysku. Oznacza to, że ścieżki znajdujące się na obrzeżu talerza mają większą pojemność w porównaniu do ścieżek zlokalizowanych przy wewnętrznej jego

[2] Już w 1987 roku wykazano wręcz, że chociaż prędkość procesora i pamięci znacząco się zwiększa z roku na rok, to szybkość dysku twardego praktycznie pozostaje bez zmian.

krawędzi (zawierają więcej sektorów). Umieszczenie najczęściej używanych danych na krawędzi talerzy, owocuje zatem zmniejszoną liczbą operacji przeszukiwania (statystycznie jest to mniej prawodopodobne) oraz uzyskaniem większych prędkości przesyłu (nawet o 50%).

2. Opóźnienie rotacyjne.
   Po zakończeniu przeszukiwania głowica musi odczekać, aż żądany sektor znajdzie się pod nią. Ten okres oczekiwania określany jest jako opóźnienie rotacyjne (*rotational delay*). Gdy odpowiedni fragment dysku znajdzie się pod głowicą, wówczas inicjowana jest operacja odczytu lub zapisu.

   Czas opóźnienia rotacyjnego dla większości dysków również wynosi kilka ms. Zasadniczo wpływ na niego ma tylko jeden czynnik — prędkość obrotowa dysku. Jego negatywne skutki zniwelować można jedynie umieszczając kolejno odczytywane dane blisko siebie, w kolejności odczytywania (a nie np. odwrotnej — znacznie zwiększyłoby to czas oczekiwania na obrót dysku przed każdym odczytem), a operacje I/O wykonywać większymi partiami zamiast bajt po bajcie (m.in. stąd wzięło się pojęcie bloku w systemach plików, czyli logicznej jednostki grupującej kilka sektorów).

O ile niektóre systemy zarządzania bazami danych mają bardzo duży wpływ na to, jak wykorzystywane są dyski (wykorzystywanie dysków surowych, odwoływanie się bezpośrednio do kontrolerów dysków, a nie API do zarządzania I/O systemów operacyjnych, itp.), programista tworzący system zarządzania bazą danych w języku Java odcięty jest (maszyną wirtualną i systemem operacyjnym) od wpływu na tak niskopoziomowe aspekty funkcjonowania dysku. Przedstawione wyżej cechy dysków pozwoliły autorom docenić wagę jedynie dwóch mechanizmów — modelu blokowego w dostępie do dysku oraz technik klastrowania danych (opisanych w dalszej części pracy). Zdaniem autorów, niskopoziomowe optymalizacje dostępu do dysków mogą przynieść jedynie marginalne wzrosty funkcjonowania systemu (w przeciwieństwie do dobrego optymalizatora języka zapytań), dlatego ich implementacja nie będzie miała dla nas pierwszorzędnego znaczenia.

2.2.2 Pamięć operacyjna

Spadające ceny pamięci operacyjnych w ostatnich latach oraz rozwój technologii typu UPS spowodowały próby wyeliminowania dysków twardych z bieżącego funkcjonowania systemów baz danych i zredukowania ich zastosowania wyłącznie do urządzeń składujących kopie bezpieczeństwa. Uważa się, że przechowywanie baz danych w pamięci operacyjnej znacznie uprościłoby budowę systemów nimi zarządzających oraz przyspieszyłoby ich funkcjonowanie.

Póki co, zastosowanie pamięci operacyjnej jako głównego nośnika danych ma znaczenie marginalne (na rzecz ciągle jeszcze dysków twardych), ale jak pokażemy w dalszej części pracy — nasz system równie dobrze może funkcjonować jako tradycyjny system zarządzania bazą danych, jak również utrzymywać całe bazy danych w pamięci, bez wykorzystania dysku twardego.

2.3 Struktura danych

Podstawową strukturą fizyczną w istniejących obecnie systemach zarządzania bazami danych jest plik danych. Plik danych podzielony jest na logiczne fragmenty wielkości zwykle 4-32 KB (wielokrotność bloku systemu operacyjnego), noszące nazwę bloków. Bloki bazy danych mogą mieć różną strukturę wewnętrzną, w zależności od struktury, którą opisują. Przykładowo, inną strukturę mają bloki danych, bloki indeksów, bloki nagłowków plików, bloki używane do sortowania na dysku itp. Najważniejszym

rodzajem bloku jest blok danych, który zwykle składa się przynajmniej z 4 części: nagłówka, katalogu przesunięć (przyrasta w dół bloku, wypełniając od góry sekcję pustą), sekcji pustej, sekcji danych (przyrasta w górę bloku, wypełniając od dołu sekcję pustą). W sekcji danych zapisywane są rekordy. Struktura rekordów ma bezpośrednie przełożenie na strukturę tabeli. Jeżeli schemat tabeli przechowywany jest jako metadane w katalogu, wówczas struktura rekordu może przyjąć formę ciągu wartości występujących w kolumnach. Jeśli kolumna zawiera dane zmiennej długości (np. VARCHAR2), wówczas wartość poprzedzana jest długością ciągu. Jeśli rekord nie mieści się w bloku, wówczas na jego końcu dodawany jest wskaźnik do innego rekordu, będącego jego dalszym ciągiem.

Opisana wyżej struktura rekordów komplikuje się znacznie, gdy trzeba zbudować rekord z powtarzającymi się polami, gdy rekord nie ma ustalonego schematu, i gdy każde pole w rekordzie musi być identyfikowane z zewnątrz (dla zapewnienia relatywizmu obiektów). Istniejące systemy obiektowe rzadko jednak wymagają tak elastycznej struktury, ze względu na obierane przez ich producentów uproszczenia modelu danych, traktując klasę jako pewną formę tabeli relacyjnej, co umożliwia wykorzystanie sprawdzone w systemach relacyjnych metody zapisu na dysk.

Alternatywną formą zapisu obiektów na dysku (przyjętą w systemie Yaod) jest zapis zdekomponowany obiektów. Oznacza to, że na dysk trafiają rekordy odpowiadające obiektom złożonym, prostym i referencyjnym. Rozwiązanie to wymaga utrzymywania dużej liczby wskaźników pomiędzy obiektami oraz zastosowania technik zapobiegających rozproszeniu obiektów na dysku. Prawdopodobnie jeśli wpisy będą ze sobą dobrze sklastrowane, wówczas można będzie osiągnąć dobrą wydajność I/O.

2.4 Klastrowanie danych

Efektywność SZBD zależy w dużej mierze od tego, ile bloków musi zostać wczytanych w celu uzyskania dostępu do wszystkich obiektów wymaganych do ewaluacji zapytania. W celu utrzymania tej liczby na najniższym z możliwych poziomów, różne obiekty przechowuje się w tych samych blokach. Dobór obiektów przechowywanych razem zależy od spodziewanej ścieżki dostępu do nich. Proces ten nazywany jest klastrowaniem (gronowaniem)[3] obiektów i może być realizowany przez twórcę aplikacji, administratora bazy danych, albo automatycznie, przez system zarządzania bazą danych.

Przykładowo w systemie Oracle istnieją dwa mechanizmy służące do zapewnienia dobrego klastrowania danych [18]:

- Segmenty.
  Segment jest strukturą składającą się z kilku bloków przeznaczonych do przechowywania danych wyłącznie jednego obiektu bazy danych, np. tabeli. Segment tabeli składa się zatem z bloków zawierających wyłącznie rekordy odwzorowujące wiersze tej tabeli. Zatem jeśli system musi wykonać operację pełnego przeglądu tabeli, wówczas wystarczy że wczyta wszystkie rekordy z bloków należących do segmentu o nazwie takiej, jak nazwa tabeli. Ponieważ bloki te formowane są w grupy ciągłych obszarów (tzw. ekstenty), operacja ta wykonywana jest szybko i sprawnie.
- Klastry.
  Wiedząc w jaki sposób aplikacja najczęściej złącza tabele, administrator ma możliwość wpłynięcia na sposób fizycznego zapisu rekordów, tworząc struktury zwane klastrami. Klaster przechowuje dane z osobnych segmentów w tych samych blokach, zatem w idealnej sytuacji można doprowa-

[3] Klastrowanie danych w tym przypadku nie ma nic wspólnego z klastrowaniem w zakresie *data mining*.

dzić do takiej sytuacji, gdy np. złączenie trzech tabel spowoduje wczytanie trzykrotnie mniejszej ilości bloków.

Ponieważ w systemie Yaod wybrano zapis obiektów po ich zdekomponowaniu, istnieje możliwość, iż np. obiekt `osoba(imie, nazwisko, wiek)` rozrzucony zostanie pomiędzy 4 różne bloki. Przydałby się zatem mechanizm zapewniający przynajmniej, że:

- wszystkie podobiekty obiektu złożonego będą zawsze znajdowały się w tym samym bloku, co obiekt złożony;
- podobiekty obiektów złożonych o tej samej nazwie i tej samej pozycji w hierarchii obiektów, będą wypełniały tę samą grupę bloków (struktura a'la segment[4]).

Mechanizm ten jednak potraktujemy jako optymalizację nie mającą zasadniczego wpływu na funkcjonowanie systemu, i nie będziemy przywiązywać do niej uwagi na obecnym etapie budowy naszej implementacji.

2.5 Identyfikatory obiektów

W obiektowej bazie danych każdemu obiektowi obowiązkowo musi być przypisany unikalny identyfikator obiektu (OID). Identyfikator ten jest odpowiednikiem klucza głównego znanego z baz relacyjnych — służy do jednoznacznego identyfikowania każdego obiektu. W odróżnieniu od baz relacyjnych, identyfikator obiektu ukryty jest przed użytkownikiem — korzysta z niego wyłącznie system zarządzania bazą danych. W toku rozwoju obiektowych baz danych wykształciły się trzy rodzaje identyfikatorów obiektów:

- identyfikatory fizyczne,
- identyfikatory logiczne,
- identyfikatory fizyczno–logiczne.

2.5.1 Identyfikatory fizyczne

Ten typ identyfikatorów powszechny był w pierwszych bazach obiektowych, ale funkcjonuje również w niektórych bazach używanych współcześnie.

Identyfikator fizyczny wskazuje bezpośrednio położenie obiektu na dysku i jest w związku z tym najszybszym sposobem dostępu do niego. Jeżeli stosowana jest metoda ciągłego przydziału przestrzeni, wówczas identyfikator fizyczny może przyjąć formę przesunięcia (*offset*) od początku pliku, w którym zapisano obiekt.

Fizyczny OID zmienia swoją wartość tylko wówczas, gdy obiekt zmieni swoją lokalizację na dysku. Sytuacja taka występuje jednak stosunkowo często (np. powiększenie obiektu może spowodować, że trzeba przenieść go w nową lokalizację. Powiadamianie o takim zdarzeniu wszystkich aplikacji odwołujących się do takiego obiektu nie jest efektywne, dlatego często w miejscu poprzedniej lokalizacji obiektu umieszcza się specjalny odnośnik, wskazujący na jego nowe położenie. Duża ilość takich odnośników może powodować dodatkowe problemy z ich kasowaniem, dlatego w niektórych kręgach naukowych i przemysłowych popularność zdobyło inne rozwiązanie — identyfikatory logiczne.

[4]Problemem są tutaj jednak nieregularności w danych — jeśli obiekt złożony o tej samej nazwie występuje tylko raz (np. w dokumencie XML), wówczas tworzenie dla niego osobnego segmentu, byłoby stratą miejsca.

2.5.2 Identyfikatory logiczne

Logiczny identyfikator jest bardziej uniwersalny, gdyż nie jest przywiązany do fizycznej lokalizacji obiektu, a więc jego wartość nigdy się nie zmienia. Rozwiązanie to prowadzi do konieczności zbudowania efektywnej struktury indeksującej (np. zwykle w formie B–drzewa), której zadaniem jest odpowiednie odwzorowanie identyfikatora logicznego na fizyczną lokalizację obiektu. Każda próba dostępu do dowolnego obiektu wymaga zatem skorzystania z takiego indeksu, co wydaje się trudne do efektywnego zaimplementowania (olbrzymia ilość danych, wąskie gardło przy współbieżnym dostępie, bardzo wolne aktualizacje itp.).

2.5.3 Idetyfikatory fizyczno–logiczne

Wady obu poprzednich rozwiązań częściowo niweluje kombinacja ich zalet, czyli identyfikatory fizyczno–logiczne, powszechnie wykorzystywane w systemach relacyjnych, a także w niektórych systemach obiektowych. Rozwiązanie to ma zastosowanie w przypadku przyjęcia modelu blokowego w dostępie do dysku. Identyfikator w takim przypadku składa się z dwóch części — fizycznej i logicznej. Część fizyczna identyfikuje blok, a część logiczna rekord w bloku. Przykładowo, OID wykorzystywany w Objectivity/DB jest 64–bitową strukturą, składającą się z czterech 16–bitowych sekcji:

1. identyfikator bazy danych,
2. identyfikator kontenera,
3. numer bloku w ramach kontenera,
4. numer slotu w ramach bloku.

Slot jest numerem wskazującym na pozycję w tablicy przesunięć, utrzymawanej dla każdego bloku danych. Obiekt może dzięki temu dowolnie zmieniać swoją lokalizację w ramach bloku — przeniesienie obiektu (np. wskutek operacji aktualizacji zwiększającej jego rozmiar) nie powoduje zmiany identyfikatora obiektu, a jedynie wartość offsetu w tablicy przesunięć danego bloku.

Ten właśnie typ identyfikatorów wykorzystano w systemie Yaod.

2.6 Ziarnistość danych

Problem ziarnistości danych pojawia się w tematyce związanej z SZOBD w kilku kontekstach. Z punktu widzenia języka zapytań, dostęp do danych przebiega zwykle na poziomie obiektu, jednak z punktu widzenia wewnętrznych mechanizmów systemowych, dane bardzo często odczytywane i zapisywane są w większych partiach — blokach. Praktyka pokazuje, że w zależności od zastosowania, operowanie większymi partiami danych jest często skuteczniejsze, od posługiwania się małymi strukturami (i odwrotnie).

Ziarnistość danych ma największe znaczenie dla trzech aspektów.

1. Transferu danych między klientem i serwerem.
   Problem ziarnistości danych w tym kontekście wywołał powstanie dwóch koncepcji: serwera bloków (stron) i serwera obiektów. Różnica między nimi jest taka, iż serwer bloków traktuje za namniejszą jednostkę transferu danych do klienta blok, a serwer obiektów — obiekt. Przypomina to zatem nieco techniki transferu pomiędzy dyskiem i pamięcią operacyjną, ale okazuje się że różnice

są na tyle duże, że możliwe stało się udowodnienie, iż w większości przypadków serwer obiektów może być co najmniej tak efektywny jak serwer stron, a w niektórych sytuacjach — nawet efektywniejszy.

System Yaod w swej obecnej postaci jest bliższy działaniu serwerowi obiektów, niż serwerowi stron.

2. Zarządzania buforami.
   Pod względem ziarnistości buforowanych danych, w SZOBD rozpatruje się najczęściej możliwość zastosowania jednego z trzech rozwiązan: dotyczących buforów:

   (a) Bufor bloków.
   Bufor bloków wykorzystywany jest przez większość wiodących systemów zarządzania bazami danych. Rozwiązanie to jest efektywne, gdy obiekty są dobrze sklastrowane w ramach bloków. Bloki o standardowych rozmiarach są również łatwe w zarządzaniu — przydzielanie i zwalnianie bloków o jednakowych rozmiarach jest proste i nie powoduje fragmentacji pamięci.

   (b) Bufor obiektów.
   Bufor obiektów zakłada przechowywanie obiektów jako osobnych stuktur, odseparowanych od bloków. Podejście to jest szczególnie użyteczne, gdy obiekty w blokach są źle sklastrowane (nie trzeba marnować pamięci na buforowanie nieużywanych danych). Wadą tego rozwiązania jest powstająca w buforze fragmentacja pamięci (obiekty mają różne rozmiary), oraz utrudnienia w aktualizacji (zmodyfikowany obiekt musi najpierw trafić do bloku, a dopiero później na dysk). Oczywiście, jeśli system nie korzysta z podejścia blokowego w dostępie do dysku, wówczas bufor obiektów jest jedynym rozwiązaniem.

   (c) Podwójny bufor.
   Idea podwójnego bufora zakłada, że system zaopatrzony jest zarówno w bufor obiektów, jak i bufor bloków. Odmiana tego rozwiązania zakłada, że obiekty dobrze sklastrowane mogą być składowane razem ze swoimi blokami, a źle — osobno.

   W systemie Yaod stworzono bufor stron i bufor obiektów, choć w zależności od składu, istnieje również możliwość wykorzystania tylko jednego, albo wręcz żadnego z nich.

3. Sterowania współbieżnością.
   Możliwość współbieżnego wykonywania operacji ma krytyczne znaczenie dla każdego systemu wielodostępowego. Negatywną stroną współbieżności jest konieczność synchronizowania wykonywanych równocześnie prac. Synchronizacja taka zazwyczaj realizowane jest poprzez czasowe zablokowanie działania niektórych procesów. Im większa ziarnistość blokad (np. na poziomie pliku, bloku) tym współbieżność mniejsza, im mniejsza (np. na poziomie obiektu) — tym współbieżność większa, ale kosztem zasobów potrzebnych do utrzymywania blokad. Najniższym poziomem blokad w większości systemów relacyjnych jest blokada na poziomie wiersza. W systemach obiektowych możliwe są natomiast zupełnie nowe tryby blokowania, np. blokada na poziomie atrybutu, blokada części drzewa obiektów i in.

   W systemie Yaod zastosowano bardzo uproszczony model współbieżności, skutkiem czego było powstanie tylko jednego typu blokady, blokady na poziomie polecenia (tylko jedna sesja może wykonywać w tym samym czasie zapytanie).

2.7 Buforowanie

Buforowanie danych polega na zmniejszaniu liczby dostępów do dysku poprzez utrzymywanie najczęściej używanych danych w pamięci operacyjnej. Efektywne algorytmy zarządzania buforami mają krytyczne znaczenie dla funkcjonowania systemu, stąd waga jaką przywiązuje się do zaprojektowania odpowiednich algorytmów przydziału i zwalniania miejsca w buforach (możliwość wystąpienia fragmentacji) oraz zastępowania jego zawartości (zwykle jest to jakaś forma algorytmu LRU).

Wydaje się, iż system zarządzania bazą danych taki jak Yaod, przystosowany do przetwarzania danych półstrukturalnych i operujący na bardzo małych strukturach danych powiązanych wskaźnikami, stawia szczególnie wysokie wymagania w stosunku do sposobu działania buforów. Wydaje się bowiem, iż system taki nigdy nie będzie w stanie sprostać serwerom relacyjnym w efektywności odczytu i zapisu danych na dysku (ze względu na prosty model danych tych systemów). Jego siłą jest natomiast sposób w jaki dane przetwarzane są po umieszczeniu ich w pamięci operacyjnej, zatem nadzieją na efektywne wykorzystanie tych możliwości jest właśnie odpowiedni mechanizm buforowania.

W systemie Yaod wykorzystano bardzo uproszczony model buforowania — buforowane elementy utrzymywane są jako obiekty Javy, zapisane w strukturach `Hashtable`. Nie zaimplementowano żadnego algorytmu zastępowania zawartości buforów, wskutek czego bufory mają teoretycznie nieograniczony rozmiar, a w razie braku pamięci operacyjnej — ich zawartość przesyłana jest na dysk przez standardowy mechanizm pamięci wirtualnej systemu operacyjnego.

2.8 Wielkie obiekty

Wielki obiekt to struktura danych zawierająca dane binarne o znacznej wielkości (nawet wielu megabajtów). Wielkie obiekty odgrywają szczególną rolę w multimedialnych zastosowaniach baz danych, a ich obsługa wymaga specjalnych mechanizmów związanych z zapisem (np. w systemach relacyjnych są one w jakiś sposób oddzielone od pozostałych danych), odczytem (zwykle nie można wprowadzić całego obiektu do pamięci operacyjnej, zatem obchodzi się ten problem dostarczając osobnego interfejsu, np. strumieniowego do jego zawartości). Niektóre zastosowania multimedialnych baz danych (np. tzw. *video on demand*) wprowadzają tutaj konieczność stosowania pewnych mechanizmów przetwarzania czasu rzeczywistego, ze względu na wysokie wymagania względem czasu dostępu do poszczególnych fragmentów tego obiektu.

System Yaod nie obsługuje obecnie wielkich obiektów w żaden szczególny sposób. Istnieje co prawda możliwość zapisania obiektu liczącego wiele megabajtów, ale operacja ta nie zostanie zrealizowana w żaden szczególny sposób w stosunku do obiektów o mniejszych rozmiarach. Oczywiście wymagania dotyczące sposobu przetwarzania wielkich obiektów były brane pod uwagę podczas projektowania systemu i najprawdopodobniej nie podjęto żadnych decyzji architektonicznych, które uniemożliwiałyby wprowadzenie opisanych wyżej mechanizmów w kolejnych wersjach systemu.

2.9 Optymalizacja zapytań

Zdolność do optymalizacji zapytań jest jednym z najważniejszych zadań stawianych przed każdym systemem zarządzania bazą danych. Ze względu na nieproceduralny charakter języków zapytań, są one skrajnie nieefektywne przy braku takiej optymalizacji. Do niedawna uważano, że obiektowość i optymalizacja

zapytań to pojęcia wykluczające się (głównie ze względu na własność hermetyzacji). Jak udowodniono chociażby w pracy [8], podobne twierdzenia nie mogą być już dłużej traktowane poważnie. Najpopularniejsze metody optymalizacyjne (nie tylko w systemach obiektów) podzielić można na następujące grupy:

- Metody dotyczące fizycznej organizacji danych — indeksowanie, partycjonowanie i in.
- Metody oparte na przepisywaniu (*rewriting*) — przekształcają zapytania na takie semantycznie równoważne postacie, które rokują lepszy czas przetwarzania. Przykładowo, istotne efekty przynieść może przesunięcie wykonania operatorów selekcji i projekcji przed operator złączenia.
- Metody wykorzystujące wysoko wyspecjalizowane algorytmy dla niektórych operatorów. Na przykład operację złączenia można wykonać metodą *nested–loops*, *sort–merge*, *hash–join*) itd.
- Metody wybierające optymalny sposób ewaluacji zapytania spośród wielu semantycznie równoważnych.
- Zapamiętywanie wyników poprzednio obliczonych zapytań. Niektóre szczególnie często spotykane zapytania są „materializowane”, dzięki czemu nie jest potrzebna powtórna ich ewaluacja. Struktury te nazywane są zmaterializowanymi perspektywami i wykorzystywane są głównie w rozproszonych bazach danych i hurtowniach danych.
- Metody oparte na równoległym wykonywaniu zapytań. Polegają one na podzieleniu pojedynczego polecenia SQL na niezależne zadania, z których każde wykonywane jest oddzielnie. Mechanizm ten warto stosować w systemach wieloprocesorowych, posiadających co najmniej kilka niezależnych dysków.

Jedną z najciekawszych technik dotyczących optymalizacji w zakresie obiektowych baz danych jest tzw. przemiana wskaźników (*pointer swizzling*). Przemiana wskaźników jest metodą, która w stosunku do baz relacyjnych umożliwia podwyższenie wydajności niektórych aplikacji nawet tysiąckrotnie. Działanie tej metody polega na tym, iż po wprowadzeniu bloku danych lub obiektu z dysku do pamięci, następuje analiza jego zawartości, a następnie wszelkie zapisane w nim trwałe wskaźniki (identyfikatory obiektów) przekształcane są na wskaźniki danego języka programowania (np. na referencje Javy), prowadzące do identyfikowanych przez nie obiektów, wczytanych wcześniej do pamięci operacyjnej. Jeśli obiekt zostanie później zmodyfikowany, wówczas przed zapisaniem go na dysku musi zajść proces odwrotny — tłumaczenie przemienionych wskaźników na ich trwałą postać. Przemiana wskaźników może być więc korzystna tylko w niektórych przypadkach (wielokrotne odczyty i rzadkie aktualizacje), co potwierdzają istniejące strategie implementacji tego mechanizmu.

System prezentowany w niniejszej pracy nie posiada niestety optymalizatora zapytań, ani żadnych specjalnych mechanizmów przyspieszających wykonywanie zapytań. Nic nie stoi jednak na przeszkodzie, by dodać te mechanizmy w późniejszym okresie. Umożliwia to zarówno architektura samego systemu, jak i język Java.

2.10 Przetwarzanie transakcji

Pojęcie transakcji jest podstawą wszelkich mechanizmów odpowiedzialnych za niezawodne działanie systemu. Transakcja jest logiczną jednostką pracy, posiadającą następujące cechy[5]:

[5] Właściwości te określane są skrótem ACID.

- Atomowość (ang. *atomicity*) — wykonane są wszystkie czynności, które wchodzą w skład transakcji, albo też żadna z nich.
- Spójność (ang. *consistency*) — o ile transakcja zastała bazę danych w spójnym stanie, po jej zakończeniu stan bazy jest również spójny (chociaż w trakcie działania transakcji stan ten może być chwilowo niespójny).
- Izolacja (ang. *isolation*) — transakcja nie wie nic o innych transakcjach i nie musi uwzględniać ich działania. Czynności wykonane przez daną transakcję są niewidoczne dla innych transakcji aż do jej zakończenia.
- Trwałość (ang. *durability*) — po zatwierdzeniu transakcji jej skutki w zakresie zmiany stanu bazy danych są trwałe i nie mogą być zniszczone przez jakiekolwiek zdarzenia.

Z mechanizmem transakcji bezpośrednio wiążą się dwa dodatkowe pojęcia:

- współbieżność,
- ochrona danych.

2.10.1 Współbieżność

Współbieżność jest zdolnością systemu do jednoczesnej obsługi wielu użytkowników, transakcji lub procesów. Im większa liczba współbieżnie wykonywanych zadań, tym większe prawdopodobieństwo, że będą one żądały w tym samym czasie dostępu do tych samych danych. Jednym z najważniejszych zadań systemu zarządzania bazą danych jest więc stworzenie i wyekzekwowanie reguł rządzących współbieżnym dostępem do danych. Gdyby system nie posiadał tego typu mechanizmów, wówczas mogłoby dojść m.in. do sytuacji przedstawionej poniżej.

Tabela 2.1: Sekwencja zdarzeń prowadzących do utraty jednej modyfikacji

Czas	**Użytkownik** A	**sal**$_A$	**Użytkownik** B	**sal**$_B$
t_0,	czyta `sal`	100		
t_1, $t_1 > t_0$		100	czyta `sal`	100
t_2, $t_2 > t_1$	zwiększa `sal` o 10	110		100
t_3, $t_3 > t_2$		110	zwiększa `sal` o 10	110
t_4, $t_4 > t_3$	zapisuje `sal`	110		110
t_5, $t_5 > t_4$		110	zapisuje `sal`	110

Użytkownik A i użytkownik B modyfikują jednocześnie pensję pracownika Jones. Obaj wprowadzają następującą instrukcję (zapisaną w języku Yaql):

```
(EMP where ename == "Jones").sal += 10;
```

Po wykonaniu obu tych instrukcji, pensja pracownika o nazwisku Jones powinna zwiększyć się o 20 w stosunku do pierwotnej sumy. Gdyby nie istniały żadne reguły rządzące jednoczesnym dostępem do bazy danych, wynik mógłby być zupełnie inny. W tabeli 2.1 pokazano przykładową sekwencję zdarzeń, prowadzących do utraty jednej modyfikacji.

Najlepszym sposobem uniknięcia powyższego problemu (oraz innych, znanych z systemów relacyjnych pod nazwami: brudne odczyty, niepowtarzalne odczyty, fantomy itp.) jest całkowita serializacja transakcji, tzn. wykonywanie ich po kolei. Niestety ze względów wydajnościowych nie jest to możliwe, zatem wprowadza się pewne poziomy izolacji transakcji, implementowane zwykle za pomocą tzw. blokad

(zamków). Zadaniem blokady jest tymczasowe uniemożliwienie wykonywania pewnych operacji na zablokowanym obiekcie. Ponieważ zablokowanie wszystkich operacji zwykle jest bardzo niekorzystne, dlatego wprowadza się różnorodne typy blokad, z których najbardziej znane, to blokada wyłączna i blokada współdzielona.

Jak widać z powyższego opisu, zapewnienie mechanizmów realizujących mechanizm współbieżności nie jest zadaniem prostym. Dlatego właśnie w systemie Yaod zaproponowano bardzo proste rozwiązanie tego problemu — w tym samym czasie wykonywane może być tylko jedno polecenie (pozostałe sesje czekają na jego realizację).

2.10.2 Ochrona danych

Normalna praca bazy danych może zostać zakłócona poprzez wystąpienie różnego rodzaju awarii. Wynikiem wystąpienia awarii jest zazwyczaj konieczność odtworzenia zawartości bazy danych do jakiegoś momentu z przeszłości. Celem odtwarzania jest doprowadzenie bazy danych do takiego stanu, by żadna z transakcji zatwierdzonych przed wystąpieniem awarii nie została utracona, oraz by każda z niezatwierdzonych transakcji została wycofana. Czynność oddtwarzania systemy zarządzania bazami danych wykonywać mogą przede wszystkim dzięki wykorzystaniu struktur zwanych dziennikiem powtórzeń (logiem) oraz kopii archiwalnych bazy danych. Pierwsza z tych struktur przydatna jest dzięki temu, iż rejestruje (z wyprzedzeniem, przed zasygnalizowaniem sukcesu w realizacji transakcji) wszystkie operacje modyfikujące zawartość bazy danych, zatem umożliwia powtórzenie dowolnych takich operacji zrealizowanych w przeszłości. Kopie archiwalne przydają się w sytuacjach skrajnych — znajdują zastosowanie w najpoważniejszych awariach (np. dysku twardego z bazą danych). Przy odtwarzaniu bazy danych wykorzystywane są dosyć złożone algorytmy, dzielące wszystkie transakcje na takie, które muszą zostać wycofane oraz takie, które powinny zostać wykonane jeszcze raz.

Ochrona danych jest prawdopodobnie najpoważniejszą wadą systemu Yaod. Nie obsługuje on obecnie mechanizmu transakcji, w związku z czym zawartość bazy danych narażona jest na wszelkiego rodzaju (nawet najdrobniejsze) awarie.

2.11 Bezpieczeństwo

Bezpieczeństwo jest jednym z ważniejszych aspektów związanych z funkcjonowaniem systemów zarządzania bazami danych w środowiskach, do których dostęp może mieć wielu użytkowników. Mechanizmy wykorzystywane do zapewnienia bezpieczeństwa zawartości bazy danych pozwalają określić kto ma dostęp do jej zawartości, jaką część zasobów SZBD może wykorzystać, jakie operacje może wykonywać, i jak śledzić jego poczynania. Najprostszym rodzajem zapewnienia bezpieczeństwa dowolnego jest wprowadzenie systemu kont chronionych hasłem i charakteryzowanych uprawnieniami do wykonywania różnych operacji. Jak pokazuje przykład serwerów usług katalogowych LDAP, dla baz danych o strukturze hierarchicznej (a więc także obiektowych) możliwe jest zbudowanie mechanizmów bezpieczeństwa o bardzo interesującym charakterze.

Póki co, serwer Yaod nie posiada żadnych możliwości związanych z zapewnieniem bezpieczeństwa — dostęp do bazy danych ma każdy, kto zdoła się z nią połączyć.

Rozdział 3

Architektura i implementacja systemu Yaod

System zarządzania bazą danych Yaod jest grupą procesów i struktur pamięci, które współdziałając ze sobą, tworzą razem system zarządzania bazą danych. W niniejszym rozdziale przedstawiono architekturę tego systemu oraz naświetlono jego sposób implementacji.

3.1 Menedżer danych, baza danych, a skład danych

W środowisku systemu Yaod, bazę danych nazywać będziemy uporządkowanym zestawem logicznie powiązanych ze sobą danych, zapisanym w jednym lub kilku składach danych. Skład danych jest modułem, którego zadaniem jest fizyczne zorganizowanie obiektów w plikach dyskowych, w pamięci operacyjnej, rozproszonej bazie danych, heterogenicznym źródle danych itp. Skład danych jest niezależnym modułem, który można wpinać i wypinać z bazy danych.

Użytkownik nie odczytuje bezpośrednio struktur należących do składów danych (np. plików), ale korzysta z pośrednictwa specjalnego oprogramowania — menedżera danych. Menedżer danych składa się głównie z kilku struktur pamięci oraz wyspecjalizowanych procesów. Pamięć na jego struktury jest rezerwowana, a procesy startują w momencie, gdy administrator uruchamia menedżera danych. Po uruchomieniu menedżer danych związywany jest z jedną i tylko jedną bazą danych (baza danych jest montowana). W przyszłości wyobrazić można sobie możliwość uruchomienia wielu menedżerów danych obsługujących jedną bazę (środowiska klastrowe — kilka komputerów współdzielących jeden system dyskowy).

Menedżer bazy danych, baza danych, oraz składy danych stanowią główne składniki każdego systemu bazodanowego opartego na Yaod. Uproszczona struktura takiego systemu przedstawiona została na rysunku 3.1, a poszczególne jego składniki opisano w dalszej części niniejszego rozdziału.

3.2 Struktura menedżera danych

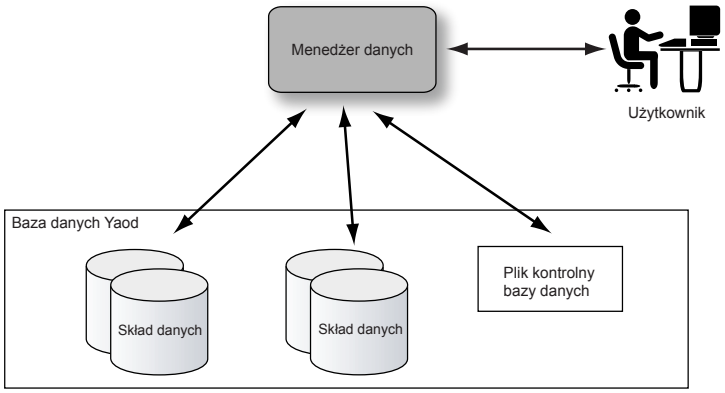
Aby baza dana danych Yaod mogła być dostępna dla użytkowników, musi być zamontowana w menedżerze danych. Menedżer danych systemu Yaod jest grupą procesów i struktur pamięci, zapewniających usługi gromadzenia, składowania, przetwarzania i udostępniania danych. Jeden egzemplarz menedżera

danych może operować tylko na jednej bazie danych, a jednocześnie na tym samym komputerze może pracować kilka takich menedżerów.

3.2.1 Obszary pamięci

Podstawowe struktury pamięci wykorzystywane przez Yaod to bufory. System stara się przechowywać tak dużo danych w pamięci operacyjnej, jak to możliwe. Utrzymywanie informacji w pamięci radykalnie zwiększa prędkość działania systemu, ponieważ dostęp do jej zawartości jest wielokrotnie szybszy, niż np. dostęp do dysku.

Rysunek 3.1: Działanie systemu Yaod



W obecnej wersji systemu zaimplementowano dwa rodzaje buforów:

- bufor bloków,
- bufor obiektów.

Buforowanie jest usługą oferowaną składom danych. To, czy skład korzysta z buforów zależy wyłącznie od programisty implementującego ten skład. Skład może korzystać z obu rodzajów buforów, albo tylko z jednego z nich. Wyobrazić sobie można (i zaimplementować w przyszłości) również inne rodzaje buforów, np. bufor planów wykonania zapytań, czy bufor dziennika powtórzeń.

Bufor bloków

Bufor bloków zawiera kopie bloków odczytanych z dysku (przydaje się, jeśli skład wykorzystuje blokowy dostęp do dysku). Gdy istnieje potrzeba dostępu do dysku, oprogramowanie zarządzające składem może najpierw sprawdzić, czy odpowiedni blok nie znajduje się już w buforze. Jeśli nie, wówczas blok musi zostać odczytany z dysku, a następnie umieszczony w buforze.

Bufor obiektów

Bufor obiektów zawiera obiekty, pojmowane jako wewnętrzne struktury danych, nazywane atomami. Ten rodzaj buforów nie występuje w bazach relacyjnych, ale w systemie Yaod jest konieczny, m.in. z następujących powodów:

- niektóre rodzaje składów mogą nie korzystać z bloków (np. skład systemu Loqis [9, 10, 11, 12]);

- wyodrębnienie ciągu bajtów z bloku danych, a następnie wygenerowanie na tej podstawie atomu (jako obiektu Javy), jest zadaniem długotrwałym, co można przyspieszyć przez ich buforowanie.

3.2.2 Procesy

W celu maksymalizacji efektywności oraz umożliwienia jednoczesnej pracy wielu użytkownikom, system Yaod wykorzystuje szereg wyspecjalizowanych procesów, zaimplementowanych jako wątki Javy. W obecnej wersji systemu, jednocześnie mogą pracować trzy rodzaje procesów:

1. Proces sekretarza bazy danych (*Database Writer*, DBWR).
   Proces DBWR zarządza buforami danych, zlecając poszczególnym składom zapisanie na trwałym nośniku zmodyfikowane („brudne") struktury umieszczone w buforach. Jego celem powinno być doprowadzenie do takiej sytuacji, by przy ograniczonym rozmiarze buforów, najczęściej używane fragmenty bazy danych znajdowały się w pamięci operacyjnej. Brudne bufory powinny trafiać na dysk w dwóch sytuacjach:

   (a) gdy w buforze brakuje wolnego miejsca na zapisanie w nim nowej struktury,
   (b) gdy wystąpi określone zdarzenie, zwane punktem kontrolnym[1].

   Ze względu na prototypowy charakter systemu Yaod oraz bardzo krótki czas jego tworzenia, zastosowano uproszczony mechanizm zarządzania buforami, w którym zapewniono obsługę jedynie drugiej z tych sytuacji. Przyjęto, iż bufory mają nieograniczony rozmiar (w przyszłości należałoby zaimplementować zapewne jakąś formę algorytmu LRU), a punkt kontrolny realizowany jest co 3 sekundy i jest to jedyna sytuacja, w której dane trafiają na dysk. Jeśli pomiędzy dwoma punktami kontrolnymi wystąpi awaria powodująca zatrzymanie systemu, wówczas wprowadzone wtedy dane ulegają utracie, a system może znaleźć się w stanie niespójności (implementacja mechanizmu transakcji rozwiązałaby ten problem).

2. Proces nasłuchu (*Listener*, LSNR).
   Zadaniem procesu LSNR jest wychwytywanie połączeń sieciowych nadchodzących od klientów oraz tworzenie dla każdego z nich osobnego procesu usługowego. W momencie utworzenia procesu usługowego, proces LSNR przestaje zajmować się danym połączeniem. LSNR domyślnie nasłuchuje na porcie o numerze 1521 i zakłada, że klient jest w stanie komunikować się z serwerem za pomocą protokołu Y.NET (punkt 3.6).

   Kolejne wersje systemu mogą zawierać dobudowaną możliwość komunikacji za pomocą innych protokołów (np. HTTP, IIOP, SOAP).

3. Proces usługowy (*Server Process*, SVRP).
   Proces SVRP reprezentuje klienta po stronie serwera, wykonując w jego imieniu wszelkie operacje na bazie danych. Gdy klient chce odczytać lub zmodyfikować zawartość bazy danych, wówczas proces usługowy przekazuje to żądanie do odpowiedniego składu obiektów, a następnie wysyła

[1] Punkt kontrolny jest zdarzeniem systemowym mającym swe źródło w modułach zarządzania transakcjami. We wiodących systemach zarządzania bazami danych zdarzenie to ma miejsce w regularnych odstępach czasu i powoduje zapisanie zawartości wszystkich buforów na dysku. Dzięki temu tworzy się pewien stan systemu, od którego można rozpocząć odtwarzanie bazy danych po ewentualnej, późniejszej awarii. W ten sposób skraca się proces odtwarzania, gdyż nie trzeba przeglądać całego dziennika powtórzeń (logu).

odpowiedź do zleceniodawcy.

W przyszłości funkcjonalność procesów usługowych może zostać rozbudowana (np. poprzez dodanie funkcji uwierzytelniania użytkowników).

Można wyobrazić sobie również wprowadzenie w przyszłości innych procesów, np. sekretarza dziennika powtórzeń (LGWR), zajmującego się zapisywaniem zawartości plików dziennika powtórzeń (logu).

3.3 Struktura bazy danych

Baza danych Yaod posiada strukturę warstwową. Wyróżnić w niej można następujące warstwy:

- fizyczną,
- logiczną,
- pojęciową.

Poszczególne warstwy są praktycznie niezależne od siebie, dzięki czemu np. przeniesienie pliku danych na inny dysk (część warstwy fizycznej), nie wpływa w żaden sposób na istnienie obiektów bazy danych (warstwa logiczna i pojęciowa). Komunikacja pomiędzy warstwami zachodzi wyłącznie za pośrednictwem API, składającego się prawie wyłącznie z bardzo niewielkiej ilości uniwersalnych funkcji CRUD (*Create*, *Retrieve*, *Update*, *Delete*).

3.3.1 Warstwa fizyczna

Zadaniem warstwy fizycznej jest zapis i odczyt na nośniku danych dowolnych bytów programistycznych, traktowanych w systemie Yaod zgodnie z zasadą relatywizmu obiektów („wszystko jest obiektem”). Warstwę fizyczną tworzą przede wszystkim poszczególne składy obiektów. System zbudowany jest w taki sposób, by mógł pracować praktycznie z dowolnym rodzajem składu. Każdy ze składów przyłączonych do bazy danych może stosować zupełnie odmienną organizację fizyczną danych, ważne by udostępniał swoją zawartość warstwie logicznej za pomocą dobrze zdefiniowanego interfejsu, składającego się z zaledwie kilku prostych funkcji. Składem obiektów może być zatem zarówno moduł zapisujący dane w strukturach niskiego poziomu (pliki, bloki, rekordy itp.), jak również wysokiego poziomu (np. baza danych Oracle, skład obiektów systemu Loqis, baza danych Yaod traktowana jako baza rozproszona itp.). Obiekty znajdujące się we wszystkich składach tworzących bazę danych dostępne są dla warstwy logicznej w taki sposób, jak gdyby znajdowały się w jednej, wirtualnej puli obiektów.

W systemie Yaod zaimplementowano trzy przykładowe składy obiektów: skład trwałych obiektów, skład nietrwałych obiektów i skład obiektów Javy (punkt 3.4). Istnieje oczywiście możliwość podpięcia do bazy danych innych rodzajów składu. Operacja taka składa się głównie z implementacji klasy potomnej w stosunku do jednej z klas abstrakcyjnych. System potrafi podłączać nowe składy w czasie wykonania, znając wyłącznie nazwę klasy, ścieżkę dostępu do pliku konfiguracyjnego składu oraz nazwę odróżniającą dany skład od innych. Operacja ta jest na tyle prosta, że np. zbudowanie modułu umożliwiającego komunikację z dowolną relacyjną bazą danych nie powinno zająć więcej, niż 2 dni.

Oprócz poszczególnych składów, częścią warstwy fizycznej są specjalne pliki, nazywane plikami kontrolnymi. Zadaniem pliku kontrolnego jest przechowywanie informacji konfiguracyjnych dotyczących bazy danych (plik kontrolny bazy danych) oraz składu (plik kontrolny składu). Plik kontrolny bazy danych zawiera nazwę bazy, datę jej utworzenia oraz informacje pozwalające na uruchomienie wszystkich

składów przyporządkowanych danej bazie danych. Plik kontrolny składu zawiera dowolne dane, które autor składu za ważne z punktu widzenia konfiguracji działania składu (np. ścieżki dostępu do plików danych).

3.3.2 Warstwa logiczna

Warstwa logiczna stanowi najważniejszą częścią systemu. Jej zadaniem jest wprowadzenie pewnego modelu obiektowego (modelu M0 w SBA), który spełniałby dwie role:

1. ukrywał złożoność warstwy fizycznej,
2. wprowadzał możliwość tworzenia złożonych obiektów.

Każda struktura warstwy logicznej jest obiektem w sensie relatywizmu obiektów. Aby jednak wprowadzić rozróżnienie pomiędzy obiektami warstwy logicznej, a obiektami warstwy pojęciowej (instancjami klas), te pierwsze nazywać będziemy dalej atomami. Atom jest zatem elementarną jednostką reprezentującą dane, i złożoną z trzech składników:

1. Nazwy.
   W systemie mogą występować atomy o tych samych nazwach, ale o zupełnie innej zawartości.
2. Jednoznacznego identyfikatora.
   Identyfikator atomu skada się z dwóch części: numeru jednoznacznie identyfikującego skład oraz jednego lub kilku elementów identyfikujących atom w danym składzie. Druga z tych części może mieć strukturę inną w każdym składzie, np. <*liczba*> (offset) w przypadku składu z Loqisa, <*nazwa_tabeli, klucz_główny*> w przypadku bazy relacyjnej, <*plik, blok, slot*> w przypadku domyślnego składu trwałych obiektów Yaod.
3. Typ.
   Typ określa rodzaj atomu. Generalnie, istnieją trzy rodzaje atomów:
   (a) Atomy proste — <*identyfikator*, *nazwa, wartość*>,
   (b) Atomy złożone — <*identyfikator*, *nazwa*, *sekwencja_identyfikatorów*>,
   (c) Atomy referencyjne — <*identyfikator*, *nazwa*, *identyfikator*>.

Struktury tworzone przez atomy złożone mogą zostać zaimplementowane na kilka sposobów. W tym celu na poziomie implementacyjnym wprowadzono dodatkowe rozróżnienie na atomy nazwane:

1. „pająkami".
   Atom nadrzędny posiada sekwencję identyfikatorów atomów podrzędnych, a każdy atom podrzędny posiada identyfikator atomu nadrzędnego. Struktura ta jest szczególnie predysponowana do budowy struktur hierarchicznych, mniej nadaje się do budowy kolekcji (ponieważ odczytanie kolejnego atomu powoduje w rzeczywistości konieczność załadowania dwóch atomów).
2. „pierścieniami” — cykliczna lista dwukierunkowa, w której każdy atom posiada identyfikator poprzedniego i kolejnego atomu (ogon listy połączony jest z jej głową). Struktura ta jest wydajnym sposobem tworzenia kolekcji przeszukiwanych sekwencyjnie. Nie jest wydajna w dostępie swobodnym.
3. „trójkątami” — atom nadrzędny posiada sekwencję identyfikatorów atomów podrzędnych, a atomy podrzędne połączone są ze sobą za pomocą listy dwukierunkowej. Struktura łączy zalety obu poprzednich, kosztem zwiększonego zapotrzebowania na pamięć.

Również atomy proste występują na poziomie implementacyjnym w kilku odmianach. Osobne ich klasy służą do przechowywania np. łańcuchów znakowych, liczb całkowitych oraz zmiennoprzecinkowych.

Ważnym elementem warstwy logicznej jest sposób zarządzania nieużytkami i ochrona przed zjawiskiem zwisających wskaźników. Yaod wykorzystuje do tego celu mechanizm zwrotnych referencji, co oznacza, iż dla każdej referencji tworzona jest referencja zwrotna. Referencja zwrotna zapisana jest w specjalnym atomie systemowym o nazwie `$BACKWARD`, umieszczonym w grafie warstwy logicznej bezpośrednio za wskazywanym obiektem. Dzięki temu nigdy nie dochodzi do pojawienia się nieużytków ani wiszących referencji, ponieważ skasowanie wszystkich referencji do danego obiektu może zostać łatwo wykryte (reakcja: usunięcie obiektu znajdującego się poza grafem powiązań), podobnie jak usunięcie wskazywanego obiektu (reakcja: usunięcie wszystkich referencji do niego).

3.3.3 Warstwa pojęciowa

Warstwa pojęciowa definiuje struktury oparte o zawartość warstwy logicznej (głównie klasy i role). Warstwa ta niestety nie jest zaimplementowana w prezentowanym systemie ze względu na konieczność uprzedniego zbudowania wielu dodatkowych struktur (np. katalogu, czy systemu typów). W pracy [1] pokazano, w jaki sposób zaimplementować klasy oraz dynamiczne role obiektów, dysponując jedynie możliwościami warstwy logicznej systemu Yaod.

3.4 Składy obiektów

W systemie Yaod zaimplementowano trzy składy obiektów:

- skład obiektów trwałych,
- skład obiektów nietrwałych,
- skład obiektów Javy.

3.4.1 Skład obiektów trwałych

Skład obiektów trwałych jest najważniejszym rodzajem składu. Jego zadaniem jest przechowywanie obiektów nieulotnych, czyli takich, które nie mogą przestać istnieć po zatrzymaniu systemu.

Skład ten widziany jest z poziomu systemu operacyjnego jako zbiór:

- Jednego lub kilku plików danych, posiadających rozszerzenie `dbf` (od *database file*).
  Każdy z tych plików podzielony jest logicznie na kilkukilobajtowe obszary, nazywane dalej blokami (punkt 3.5). Rozmiar bloku jest wspólny dla całej bazy danych i ustala go administratora w czasie jej zakładania w taki sposób, by był on wielokrotnością. bloku systemu plików danego systemu operacyjnego. Mniejsze bloki nadają się bardziej do zastosowań OLTP, większe — DSS. Pierwszy blok każdego pliku danych jest blokiem wolnej przestrzeni i służy do szybkiego wyszukiwania bloku posiadającego wystarczająco dużo miejsca, aby można było do niego wstawić rekord o podanej wielkości. Pozostałe bloki są blokami danych, a ich rolą jest przechowywanie i udostępnianie rekordów, czyli fizycznej formy atomów. Rekord tworzony jest poprzez zserializowanie (zapisanie w formie tablicy `byte[]`) obiektu Javy reprezentującego atom za pomocą standardowego mechanizmu serializacji Javy. Dostępne są bardziej efektywne metody serializacji atomów nie oparte na możliwościach języka Java w tej materii, ale nie są one w tej chwili zaimplementowane.

- Jedgnego pliku kontrolnego składu, posiadającego rozszerzenie `ctl` (od *control file*).
  Plik `ctl` zawiera informacje konfiguracyjne, niezbędne do funkcjonowania składu. Przede wszystkim zapisana jest w nim lista plików danych oraz identyfikator atomu wejściowego do danego składu (jego korzeń).

Każdy atom w składzie trwałych obiektów identyfikowany jest za pomocą trójki: $\langle p, b, r \rangle$, gdzie p jest identyfikatorem pliku danych w składzie, b jest numerem bloku w pliku, r jest numerem rekordu w bloku. Identyfikator atomu jest podstawą do budowy identyfikatora obiektu. OID (wprowadzony na wyższym poziomie) składa się oprócz tego z identyfikatora składu.

3.4.2 Skład obiektów nietrwałych

Skład obiektów nietrwałych służy do przechowywania obiektów ulotnych, tzn. takich, które przestają istnieć po zatrzymaniu systemu (lub nawet wcześniej, np. po wyjściu z procedury). Skład ten może mieć zarówno charakter prywatny w stosunku do danej sesji użytkownika, jak również globalny, dostępny dla wszystkich użytkowników. Pierwsza z tych właściwości może być użyteczna do tworzenia obiektów lokalnych (np. w procedurach), a druga — zapewnia możliwość utworzenia bazy danych znajdującej się w całości w pamięci operacyjnej (bardzo popularny ostatnio temat).

Rozpatrywano dwie możliwości zaimplementowania tego składu:

1. Alokacja dużego fragmentu pamięci, a następnie jego podział na bloki i dostęp do danych w sposób podobny do tego, który przyjęto w składzie obiektów trwałych. Zaletą tego rozwiązania jest możliwość szybkiej wymiany dysk–pamięć, gdyby było to konieczne w przypadku braku pamięci.
2. Przechowywanie atomów jako obiektów Javy w strukturze `Hashtable`. Zaletą tego rozwiązania jest oszczędność zasobów, i to właśnie ono zostało wybrane do budowy składu obiektów nietrwałych.

3.4.3 Skład obiektów Javy

Katalogi wielu systemów zarządzania bazami danych zawierają specjalny rodzaj perspektyw (np. w Oracle tzw. dynamiczne perspektywy wydajnościowe), które pozwalają na dostęp z poziomu języka zapytań (zwykle jest to tylko odczyt) do wewnętrznych struktur języka programowania, w którym stworzony został system. Właściwość taka jest szczególnie użyteczna dla administratora zajmującego się optymalizacją pracy systemu, gdyż może np. za pomocą instrukcji `SELECT` uzyskać informacje na temat pracy wewnętrznych elementów systemu (zapisywanie w składzie trwałych obiektów takich informacji jak np. ilość wolnego miejsca w buforze nie miałoby większego sensu).

Skład obiektów Javy jest wewnętrznym składem systemu Yaod, który rozwiązuje ten problem. Jego zadaniem jest wykorzystanie mechanizmu refleksji Javy do reprezentowanie obiektów znajdujących się na stercie Javy w taki sposób, by były one widoczne dla języka zapytań. Skład ten obecnie nie pełni pierwszoplanowej roli, ale powinien przydać się w przyszłości.

3.5 Bloki

Blok stanowi najmniejszą jednostkę rezerwacji przestrzeni dyskowej i operacji wejścia–wyjścia w trwałym składzie obiektów. Suma bloków wszystkich plików danych tworzy jednolitą, logiczną przestrzeń składu. Obecnie istnieją dwa rodzaje bloków:

- Bloki danych.
  Zadaniem bloków danych jest przechowywanie atomów, traktowanych na poziomie fizycznym bazy danych jako nic nie znaczące ciągi bajtów (rekordy). Każdy blok danych składa się z kilku sekcji. Rozmiar każdej sekcji jest dynamiczny i praktycznie w każdym bloku różny. Liczba sekcji jest stała i taka sama dla każdego bloku. Obecnie bloki danych składają się z pięciu sekcji:

  1. Sekcji nagłówka bloku.
     Sekcja zawiera identyfikator bloku (numer pliku i numer bloku w pliku) oraz jego typ.
  2. Sekcji nagłówka bloku danych.
     Obszar zawiera sześć dwubajtowych wartości, określających początek (względem początku bloku) i długość sekcji katalogu rekordów, sekcji pustej i sekcji danych.
  3. Katalogu rekordów.
     Obszar składa sie z listy slotów. Każdy slot jest liczbą (dwa bajty) przyporządkowaną dla każdego rekordu zapisanego w bloku i określającą początek tego rekordu w bloku względem początku bloku. Numer slotu stanowi część identyfikatora atomu. Wartość zapisana w slocie może się zmieniać jeśli rekord zostaje przemieszczony w bloku na nową pozycję (np. po zmianie wielkości).
  4. Sekcji pustej.
     Obszar, w którym nie są zapisane żadne dane. Może być przydzielony powiększającej się sekcji slotów (przydział od początku sekcji) oraz sekcji danych (przydział od dołu sekcji).
  5. Sekcji danych.
     Obszar ten przeznaczony jest do zapisu rekordów. Zapisy wykonywane są od końca bloku w kierunku katalogu rekordów. Każdy rekord rejestrowany jest w katalogu rekordów. Jeśli rekord zmieni swój rozmiar, wówczas przenoszony jest na szczyt sekcji danych, a pozostawiona przez niego „dziura” wypełniana jest poprzez przeniesienie na jego miejsce poprzedzających go rekordów.

     Jeśli rekord rozrośnie się na tyle, że nie mieści się już w bloku, albo gdy do bloku wstawiany jest rekord większy, niż rozmiar bloku, wówczas rekord ten staje się rekordem łańcuchowanym, czyli zapisanym w kilku blokach.

     Każdy rekord posiada nagłówek, składający się z dwóch fragmentów:
     (a) 15 bitów określających długość rekordu w danym bloku,
     (b) 1 bit określający, czy blok jest łańcuchowany,

     Jeśli blok jest łańcuchowany, to posiada również stopkę. Stopką jest identyfikator dalszej części rekordu. System odczytując rekord łańcuchowany, czyta wszystkie jego fragmenty, a następnie łączy w całość, i dopiero wówczas zwraca.

  Rozbudowa systemu o nowe funkcje może spowodować konieczność stworzenia dodatkowych pól i sekcji. Przykładowo implementacja współbieżności mogła by się wiązać z wprowadzeniem sekcji blokad, zawierajacej informacje o zamkach założonych na atomach zapisanych w danym bloku (np. w formie par <*slot*, *tryb_blokady*>).

- Bloki wolnej przetrzeni.
  System mając za zadanie wstawienie nowego rekordu do jakiegoś bloku, musi dysponować informacją na temat wolnej przestrzeni. Niedopuszczalna jest taka sytuacja, aby w takim momencie bloki wczytywane zostawałyby po kolei, a ich wolna przestrzeń porównywana z przestrzenią

wymaganą na wstawienie rekordu. Bloki wolnej przestrzeni rozwiązują ten problem, dysponując informacją zbiorczą na temat wolnej przestrzeni we wszystkich blokach danych danego pliku.

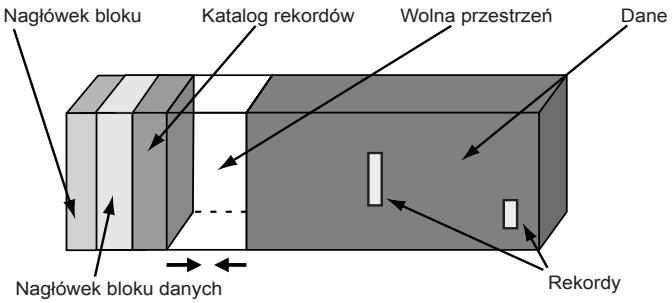
Każdy blok wolnej przestrzeni składa się z dwóch sekcji:

1. Sekcji nagłówka bloku.
   Sekcja zawiera identyfikator bloku i jego typ.
2. Sekcji danych.
   Sekcja zawiera ciąg 2–bajtowych wartości, określających ilość wolnej przestrzeni w kolejnych blokach pliku danych.

Zakładając, że wielkość bloku w składzie wynosi 8K, można przyjąć, że jeden blok wolnej przestrzeni jest w stanie opisać ok. 4000 bloków, czyli plik wielkości 30MB. W przyszłości rozpatrzyć można implementację łańcuchów bloków wolnej przestrzeni, albo zbudować bardziej złożony model zarządzania wolną przestrzenią (np. oparty o tzw. znaczniki wysokiej wody).

W kolejnych wersjach systemu łatwo będzie zaimplementować inne rodzajów bloków, np. bloków indeksów, bloki służące do sortowania dyskowego itp.

Rysunek 3.2: Struktura bloku danych



3.6 Interakcja użytkownika z bazą danych

Komunikacja aplikacji klienckiej z bazą danych Yaod przebiega w sposób tradycyjny. Klient oddzielony od serwera siecią komputerową wykorzystuje oprogramowanie umożliwiające uzyskanie połączenia z serwerem. Ze względu na to, iż Yaod w swej obecnej formie nie obsługuje mechanizmów związanych z bezpieczeństwem, dostęp do bazy danych otrzymuje każdy użytkownik. Po uzyskaniu połączenia klient może wysyłać do bazy danych zapytania oraz otrzymywać na nie odpowiedzi. Komunikacja zrealizowana jest za pomocą specjalnego protokołu, nazwanego roboczo Y.NET (choć bardzo łatwo można wprowadzić obsługę innych protokołów). Y.NET jest bardzo prostym protokołem opartym o mechanizm serializacji Javy (pomiędzy klientem i serwerem przesyłane są serializowane obiekty tego języka). Istnieje kilka rodzajów komunikatów:

- `QueryRequest` — żądanie klienta, zawierające treść zapytania do wykonania przez serwer,
- `QueryResult` — odpowiedź serwera, zawierająca wynik zapytania,
- `DumpRequest` — żądanie klienta, zawierające zlecenie wykonania zrzutu całej bazy danych,
- `DumpReply` — odpowiedź serwera, zawierająca wszystkie atomy bazy danych oraz związki między nimi,

- `XMLLoadRequest` — żądanie klienta, zawierające treść dokumentu XML, który ma zostać załadowany do bazy danych,
- `ErrorReply` — odpowiedź serwera, zawierająca informację o błędzie (np. składniowym w zapytaniu).

Wynik zapytania może być dwojakiego rodzaju:

1. wartość prosta: ciąg znaków, liczba całkowita, liczba zmiennoprzecinkowa, wartość boolowska, identyfikator, binder;
2. wartość złożona: wielozbiór, sekwencja, struktura.

Yaod obsługuje wielodostęp w bardzo uproszczonej formie. Możliwe jest co prawda jednoczesne połączenie dowolnej liczby użytkowników, ale w danej chwili wykonywane mogą być zapytania tylko jednego z nich (blokada na poziomie polecenia, zrealizowana za pomocą statycznej metody synchronizowanej). System zaprojektowany jest jednak tak, by po implementacji transakcji możliwe było stworzenie wydajnego mechanizmu zarządzania współbieżnymi sesjami, przy zachowaniu wszelkich reguł znanych we wiodących systemach zarządzania bazami danych (blokady, poziomy izolacji transakcji itp.).

3.7 Program Yaod Workbench

Program Yaod Workbench jest przykładowym klientem bazy danych Yaod. Program ten potrafi połączyć się z każdą bazą danych dostępną poprzez sieć komputerową. Dodanie do niego nowej bazy danych polega na wybraniu węzła **Databases**, wybraniu prawym przyciskiem myszki menu **Register database**, oraz podaniu hosta i portu (zwykle 1521). Po uzyskaniu połączenia z bazą danych, użytkownik otrzymuje możliwość wprowadzania zapytań Yaql oraz odczytywania ich wyników, drzew AST oraz stosu ENVS. Wyniki zapytań mogą być wyświetlane w dwóch formach: wg schematu języka XML oraz wg schematu Yaod. Dodatkowe możliwości Workbencha wiążą się z jego zdolnością do wyświetlenia wszystkich atomów bazy danych oraz ładowania dokumentów XML do repozytorium.

Yaod uruchamia się po wprowadzeniu w linii poleceń następującej instrukcji:

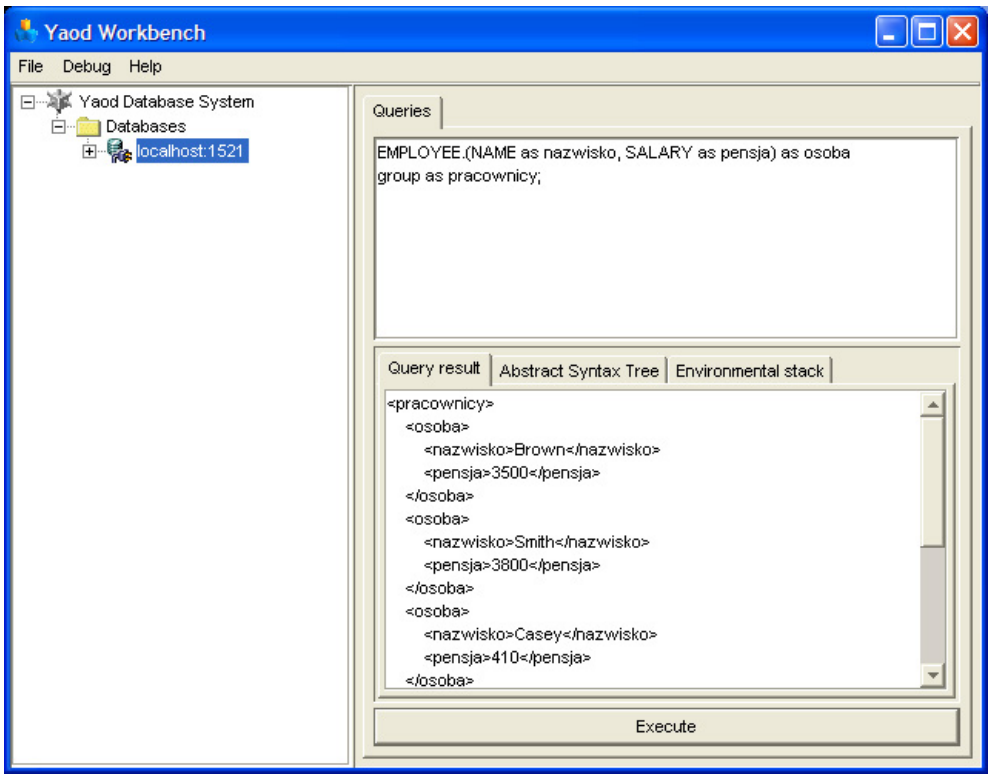
```
java pl.edu.pjwstk.yaod.workbench.Workbench
```

3.8 Administracja bazą danych

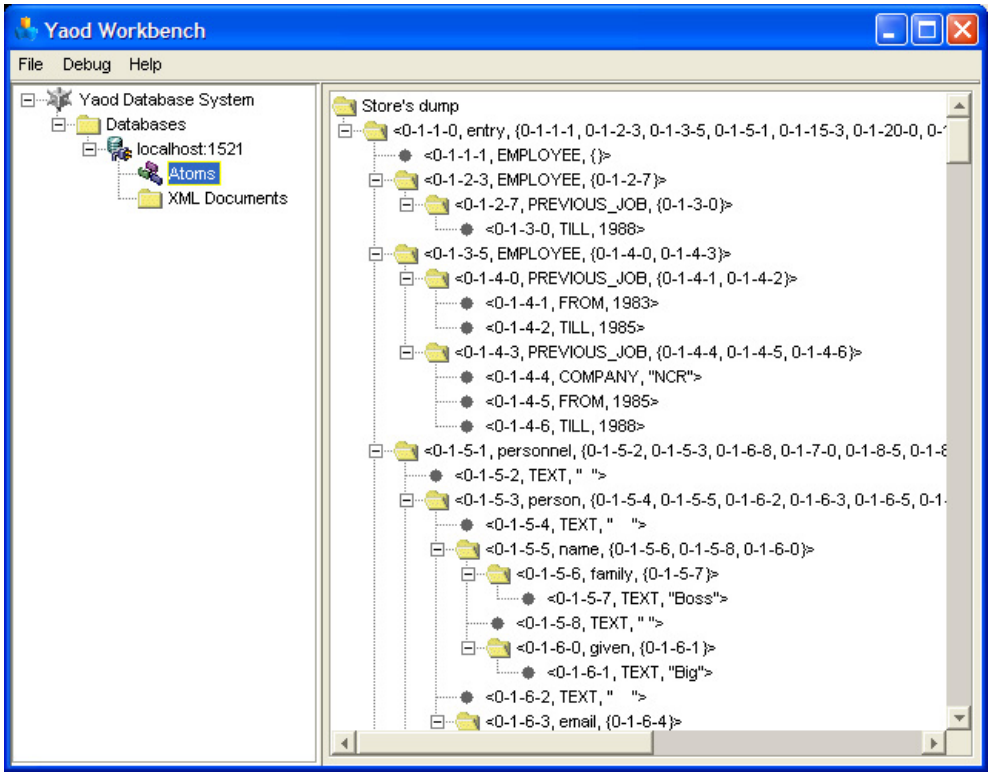
Administracja bazą danych w wersji 1.0 systemu Yaod możliwa jest wyłącznie za pomocą udostępnianego API. Nie istnieją niestety żadne narzędzia, ani języki wspomagające ten proces, za wyjątkiem jednego — kreatora bazy danych. Kreator bazy danych to prosta aplikacja, której zadaniem jest zainicjalizowanie małej bazy danych wykorzystującej domyślny skład trwałych obiektów. Przy uruchamianiu programu podaje się jedynie najprostsze parametry dotyczące tworzonej bazy danych — jej rozmiar i ścieżkę dostępu dla tworzonych plików danych. System automatycznie utworzy dwa pliki danych (każdy wielkości 50% podanego rozmiaru bazy i bloku wielkości 8192B), a także plik kontrolny bazy danych, w którym zarejestruje jeden skład trwałych obiektów. Bardziej szczegółowa konfiguracja wymaga rekompilacji programu tworzącego bazę danych.

Kreator bazy danych uruchamia się za pomocą następującego polecenia:

Rysunek 3.3: Workbench – wykonywanie zapytań



Rysunek 3.4: Workbench – przeglądanie bazy danych



```
java pl.edu.pjwstk.yaod.creators.DatabaseCreator
```

Ścieżka dostępu do pliku kontrolnego bazy danych powinna zostać podana przy uruchamianiu serwera.

Serwer baz danych Yaod uruchamia się wprowadzając w linii poleceń systemu operacyjnego następującą instrukcję:

```
java pl.edu.pjwstk.yaod.system.Yaod
```

3.9 Obsługa XML

Dzięki uniwersalnym założeniom modelu M0 z SBA, model danych przyjęty w systemie Yaod jest w stanie łatwo reprezentować dokumenty XML. XML może pojawiać się jednak wyłącznie na wejściu oraz wyjściu tego systemu — wewnętrzne struktury przetwarzania nie są od niego w żaden sposób uzależnione. Dokumenty zapisywane są w repozytorium w formie zdekomponowanej, czyli każdy element stanowi osobny obiekt. Yaod obecnie akceptuje tylko bardzo proste dokumenty. Nie jest w stanie jeszcze obsługiwać niektórych kluczowych mechanizmów tego języka. Ograniczenia te mają nie mają jednak swych źródeł w podstawowych koncepcjach przyjętych w tym systemie (np. w modelu danych), ale w stopniu zaawansowania ich implementacji. Rozbudowa systemu o obsługę linków (np. w formie zaproponowanej w pracy [21]), czy atrybutów, jest obecnie przedsięwzięciem właściwie tak trywialnym, że nie powinno ono zająć nawet jednego dnia.

3.10 Yaod w przykładach

Poniżej opsiano kilka typowych sytuacji, występujących w pracy systemu Yaod oraz ich wewnętrzny sposób realizacji. Ze względu na stosunkowo duże rozmiary kodu (prawie 300 klas), szczegółowy sposób implementacji systemu odczytać można z dołączonej do niego dokumentacji w formacie JavaDoc.

3.10.1 Uruchomienie serwera

Podczas uruchamiania serwer wykonuje następujące czynności:

1. Odczytuje plik kontrolny bazy danych.
2. Uruchamia menedżera bazy danych.
3. Menedżer danych danych analizuje zawartość pliku kontrolnego, a następnie otwiera wszystkie zapisane w nim składy:
   (a) Pobiera nazwę klasy skladu.
   (b) Tworzy nową instancję składu.
   (c) Otwiera skład, wywołując jego metodę `open()`. W przypadku składu trwałych obiektów, reakcją na wywołanie tej metody jest wykonanie następujacych kroków przez skład:
      i. Wczytanie prywatnego pliku kontrolnego.
      ii. Uruchomienie menedżera plików, który otwiera wszystkie pliki danych.
4. Menedżer danych uruchamia proces nasłuchu (LSNR). Proces ten otwiera gniazdko i rozpoczyna nasłuch sieci w celu wykrycia nadchodzących połączeń.
5. Menedżer danych uruchamia proces sekretarza bazy danych (DBWR). Sekretarz bazy danych usypia się na 3 sekundy, po czym wykonuje punkt kontrolny, znowu się usypia itd.

3.10.2 Wysłanie żądania przez klienta

Klient wykonuje następujące operacje:

1. Tworzy instancję klasy `YCommunicator`, podając host serwera i port, na którym nasłuchuje proces LSNR.
2. Tworzy instancję klasy `QueryRequest`, podając treść zapytania (np. `EMPLOYEE.NAME`).
3. Wysyła żądanie, wywołując metodę `YCommunicator.putMessage(QueryRequest)`.
4. Odbiera obiekt `QueryReply`, wywołując metodę `YCommunicator.getMessage()`.
5. Odbiera obiekt `Result`, wywołując metodę `QueryReply.getResult()`.
6. Przetwarza wyniki, przechodząc po drzewie wyniku zapytania.

3.10.3 Realizacja żądania przez menedżera bazy danych

Jeśli proces LSNR wykryje nadchodzące połączenie, wówczas tworzy i uruchamia nowy proces usługowy (SVRP), który przejmuje dalszą komunikację z danym klientem. Proces SVRP wykonuje następujące kroki w celu zrealizowania żądania klienta:

1. Pobiera z menedżera bazy danych listę wszystkich składów, a następnie każdemu z nich zleca utworzenie osłony.
2. Tworzy menedżera obiektów.
3. Inicjalizuje interpreter Yaql.
4. Pobiera z gniazdka żądanie klienta, a następnie sprawdza jego rodzaj (zapytanie, zrzut bazy danych, załadowanie pliku XML).
5. Jeśli jest to zapytanie, wówczas przekazuje je do wykonania interpreterowi Yaql.
6. Interpreter wykonuje zapytanie, pobierając zawartość bazy danych za pomocą metod menedżera obiektów. Przykładowo, w celu wykonania zapytania `EMPLOYEE.NAME`, interpreter realizuje ciąg następujących operacji:
   (a) Uruchamia pętlę iterującą po wszystkich osłonach składów.
   (b) Dla każdej osłony pobiera atom wejściowy składu.
   (c) Dla każdego atomu wejściowego wywołuje metodę: `YComplexObject.findMemberByName("EMPLOYEE")`.
   (d) Dla każdego obiektu zwróconego przez powyższe wywołanie, wywołuje metodę: `YComplexObject.findMemberByName("NAME")`.
7. Wynik odczytany z interpretera przesyła klientowi.

3.10.4 Realizacja żądania przez skład trwałych obiektów

Gdy wykonywana jest któraś z metod obiektu potomnego `YObject`, wówczas zazwyczaj wywołanie to jest tłumaczone na jedno lub więcej wywołań do osłony składu. Osłona składu tłumaczy udostępniane przez siebie, standardowe funkcje CRUD na serię wywołań związanych z niskopoziomowym funkcjonowaniem danego składu. Gdy przykładowo obsłona trwałego składu otrzyma zlecenie wyszukania atomu o podanym identyfikatorze (metoda `PersistentStoreWrapper.retrieveAtom()`, wówczas wykonywane są następujące kroki:

1. Osłona sprawdza w buforze atomów, czy atom o podanym identyfikatorze znajduje się już w pamięci operacyjnej. Jeśli tak, wówczas jest on zwracany.
2. Jeśli nie, wówczas wywołuje metodę `getRecord()` menedżera rekordów.
3. Menedżer rekordów wykorzystuje menedżera bloków odszukania potrzebnego bloku danych.
4. Menedżer bloków sprawdza, czy blok o podanym identyfikatorze (część identyfikatora atomu) znajduje się już w buforze bloków. Jeśli tak, wówczas odczytuje rekord (punkt 6).
5. Jeśli nie, wtedy menedżer bloków wykorzystuje menedżera plików do odczytania potrzebnego fragmentu pliku danych o odpowiednim identyfikatorze (część identyfikatora atomu), tworzy z niego obiekt Javy reprezentujący blok danych, umieszcza go w buforze bloków, a następnie zwraca menedżerowi rekordów.
6. Menedżer rekordów pobiera rekord z odczytanego bloku danych i sprawdza czy jest on łańcuchowany. Jeśli nie, wówczas zwraca go osłonie składu.
7. Jeśli tak, wówczas odczytuje adres kolejnej części, powtarza krok 3, po czym łączy otrzymane do tej pory części. Gdy nie ma już więcej części, wówczas zwraca cały rekord menedżerowi rekordów.
8. Osłona dysponując rekordem w postaci tablicy `byte[]`, deserializuje go, a następnie zwraca otrzymany w ten sposób obiekt Javy reprezentujący atom jako rezultat metody `retrieveAtom()`.

Podobnie realizowane są operacje tworzenia nowych obiektów. Osłona serializuje obiekt, przekazuje tablicę `byte[]` do menedżera rekordów, który wczytuje blok wolnej przestrzeni, wyszukuje w nim blok posiadający wystarczającą ilość wolnego miejsca, wczytuje ten blok i tworzy w nim rekord. Jeśli rekord jest większy, niż blok, wówczas jest on łańcuchowany między kilka bloków.

Operacja aktualizacji również przebiega podobnie. Różnica polega jedynie na sposobie traktowania rekordów łańcuchowanych. Jeśli rekord jest łańcuchowany, wówczas wówczas najpierw kasowane są wszystkie jego dodatkowe rekordy–części, pierwsza część jest zastępowana nowym rekordem, a jeśli to konieczne — pozostałe części tworzone od początku.

Kasowanie rekordów jest najprostszą operacją. Menedżer rekordów sprawdza czy rekord jest łańcuchowany, jeśli tak, to dla każdego rekordu–części usuwa jego slot (jeśli slotu nie można go usunąć wówczas ustawia adres rekordu na 0, co traktowane jest jako slot nieużywany), następnie wypełnia dziurę powstałą po usuniętym rekordzie, przenosząc wszystkie rekordy w danym bloku o offsecie mniejszym od usuwanego rekordu, o wartość równą długości usuniętego rekordu w kierunku dołu bloku (wypełniając w ten sposób powstałą dziurę).

Rozdział 4

Język Yaql

Język Yaql (*Yet Another Query Language*) jest językiem zapytań opartym na tym podejściu stosowym, zaimplementowanym pomyślnie w systemie Yaod. W niniejszym rozdziale opisano składnie oraz semantykę języka, a także omówiono najważniejsze aspekty implementacji interpretera tego języka.

4.1 Główne założenia implementacji

Podczas implementacji języka kierowaliśmy się następującymi założeniami projektowymi:

- składnia języka będzie oparta na składni języka SBQL ([10, 13, 14, 15]),
- semantyka języka będzie kierowana składnią,
- język będzie zakładał luźną kontrole typów oraz późne wiązanie,
- istnienie konstrukcji deklaratywnych oraz imperatywnych,
- istnienie procedur właściwych oraz funkcyjnych (funkcji),
- brak konstrukcji obiektowo–zorientowanych (klasy, dziedziczenia, hermetyzacja itp.),

4.2 Konstrukcja drzewa rozbioru programu

Źródło programu wewnątrz interpretera języka, jest reprezentowane za pomocą drzewiastej struktury danych (drzewo składni abstrakcyjnej). Przyjeliśmy konwencję, iż gramatyke języka będziemy bezpośrednio przekształcić w taką strukture stosując poniższe zasady dotyczące reprezentacji struktur drzewiastych w Javie (na podstawie [4], [6]):

1. Drzewa są opisane za pomocą gramatyki.
2. Drzewo jest opisane przez jedną lub wiele klas abstrakcyjnych, z których każda odpowiada nieterminalowi w gramatyce.
3. Każda klasa abstrakcyjna rozszerza jedną lub wiele podklas, przy czym dla każdej produkcji gramatyki istnieje dokładnie jedna taka klasa.
4. Dla każdej prawej strony produkcji w której występuje symbol będący nieterminalem lub terminalem, zostanie stworzone odpowiednie pole w aktualnej klasie.
5. Każda z klas będzie posiadać konstruktor inicjalizujacy wszystkie pola.

6. Dane są inicjalizowane gdy są tworzone i nigdy nie są modyfikowane później.

Drzewo takie jest tworzone w trakcie parsowania tekstu źródłowego. Parser to realizujący został stworzony za pomocą generatora parserów LALR dla Javy — *CUP*, który współpracuje ze skanerem stworzonym za pomocą generatora analizotorów leksykalnych dla Javy - *JFlex*.

Generator skanerów (analizatorów leksykalnych) jest odpowiedzialny za składnię języka. Wszystkie symbole (tokeny) użyte w języku są definiowane jako wyrażenia regularne w pliku konfiguracyjnym, na podstawie którego zostaje wygenerowany skaner. Słowa kluczowe języka zostały przedstawione w dodatku A. Plik konfiguracyjny wykorzystany stworzenia skanera został dołączony w dodatku D.

Generator parserów jest odpowiedzialny za gramatyke języka. Wszystkie terminale (odpowiadające tokenom ze skanera) i nieterminale stanowiące gramatykę bezkontekstową języka (zob. dodatek B) są definiowane jako ciąg produkcji w pliku konfiguracyjnym. W pliku tym określa się także priorytety operatorów (zob. dodatek C) oraz akcje semantyczne. Wynikiem jest plik pełniący funkcje parsera. Plik konfiguracyjny wykorzystany do stworzenia parsera został dołączony w dodatku E.

Poza wartościami oczywistymi dla każdego węzła (operandy, operatory, wartości literałów, nazwy identyfikatorów itp.), przy każdym węźle posiadającym terminale zostaje dołączona dodatkowa informacja związana z położeniem danego terminala w tekście źródłowym (kolumna, wiersz). Rozwiązanie takie umożliwia generowanie raportów o błedach z dokładnością co do pozycji w tekście źródłowym.

Wynikiem działania parsera na tekście źródłowym jest drzewo składni abstrakcyjnej (drzewo pozbawione lukru syntaktycznego). Przykładowe drzewo rozbioru dla wyrażenia:

```
( (1 as X, 1 as LICZNIK)
    close by
  (((4/X + X)/2 as X, LICZNIK+1 as LICZNIK) where LICZNIK <= 10)
).(X where LICZNIK == 10)
```

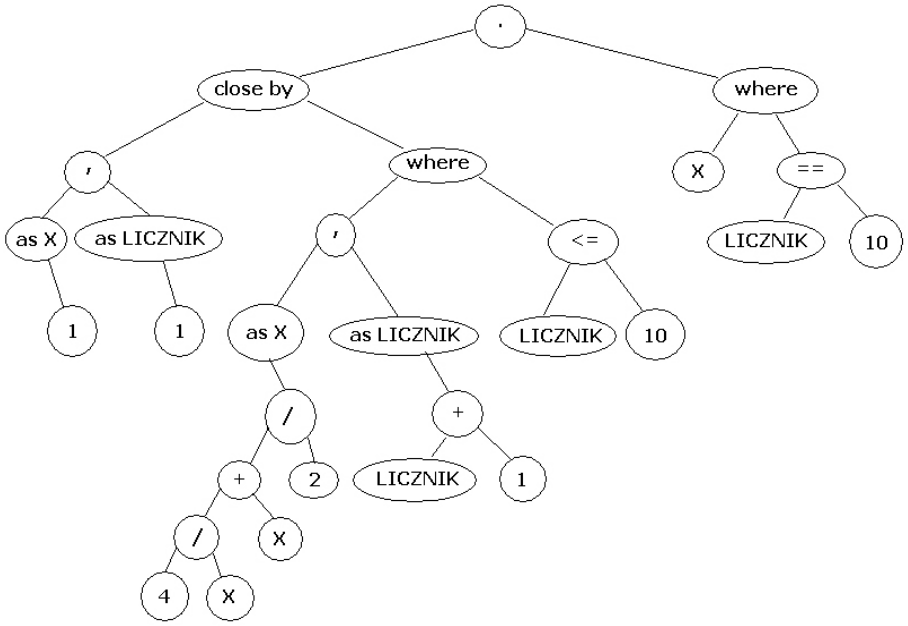
zostało przedstawione na rysunku 4.1.

4.3 Ewaluacja (interpretacja) programu

Ewaluacja programu polega na przechodzeniu przez drzewo składni abstrakcyjnej w porządku postfixowym, a następnie dla każdego węzła uruchamiana jest odpowiednia procedura ewaluacji. Każdy taki węzeł jest reprezentowany poprzez klasę która posiada w sobie odpowiednią metodę `eval()`. Takie podejście jest typowe w projektowaniu zorientowanym obiektowo.

W naszej implementacji zdecydowaliśmy sie na oddzielenie składni od interpretacji, wykorzystując wzorzec projektowy `Visitor`. Specyfika tego rozwiązania polega na tym, iż istnieje pewna klasa `EvalVisitor`, która w sobie zawiera interpretacje wszystkich możliwych konstrukcji programistycznych (węzłów drzewa składni abstrakcyjnej). Głównym powodem takiego rozwiązania była czytelność, oraz modularność i łatwość dodawania nowych interpretacji (w chwili obecnej wykorzystywana jest tylko ewaluacja, ale przy rozbudowie języka o system typów, optymalizacje, generowanie kodu pośredniego itp. wystarczy dodać nową klasę wizytującą reprezentującą odpowiednią interpretacje, która implementuje interfejs `Visitor`). Z drugiej strony, dodanie nowego typu węzła (wyrażenia, instrukcji) nie jest aż tak modularne (względem interpretacji), gdyż sprowadza się do dodania odpowiedniej klauzuli w każdej funkcji interpretującej (w naszym przypadku jest to tylko jedna interpretacja — ewaluacja). W podejściu

Rysunek 4.1: Przykładowe drzewo rozbioru wyrażenia.



zorientowanym obiektowo wady i zalety są odwrotne. Powyższe rozwiązanie jest szeroko stosowane przy konstrukcji kompilatorów (por. [4, 6]).

4.4 Rezultaty zwracane przez wyrażenia

Główna klasa reprezentująca zbiór rezultatów została nazwana `QueryResult`. Z niej dziedziczy klasa reprezentująca wyniki pojedyncze — `SingleValue`, oraz klasa reprezentująca wyniki złożone — `ComplexValue`. Dalsze możliwe rezultaty, ich podział oraz sposób dziedziczenia został przedstawiony poniżej:

- Wyniki pojedyncze (`SingleValue`) moga być:
  - wartością nazwaną - binderem (`BinderValue`), która posiada nazwę oraz wartość będącą elementem zbioru rezultatów,
  - identyfikatorem obiektu w składzie (`IdentifierValue`),
  - wartością atomową (`AtomicValue`) która może być:
    * wartością logiczną (`BooleanValue`),
    * liczbą całkowitą (`IntegerValue`),
    * liczbą rzeczywistą (`DoubleValue`),
    * łańcuchem znakowym (`StringValue`).
- Wyniki złożone - kolekcje (`ComplexValue`) moga być:
  - strukturą (`StructValue`),
  - zbiorem (`BagValue`),
  - sekwencją (`SequenceValue`).

4.5 Stos środowisk oraz stos rezultatów

Stos środowisk został zaimplementowany jako klasa `EnvironmentalStack`. Jedyne elementy z jakich się składa to sekcje będące zbiorami binderów. Sekcja taka będąca rekordem aktywacji została zaimplementowana jako klasa `ActivationRecord`, a elementy sekcji (bindery) istnieją jako obiekty klasy `BinderValue`, która została już omówiona.

Stos rezultatów został zaimplementowany jako klasa `QueryResultStack`. Sekcje z których się składa, są elementami zbioru rezultat `QueryResult`, omówionym poprzednio.

Dla obydwu stosów zostały zdefiniowane funkcje typowe dla funkcjonowania takich struktur danych (`push()`, `pop()`, `empty()`, `top()`). W przypadku nieprawidłowego odwołania sie do elementów stosu, zostaje podniesiony wyjątek `EmptyStackException`.

Dodatkowo dla stosu środowiskowego została zdefiniowana funkcja `bind()`, wiążąca nazwe z bytem czasu wykonania, oraz zestaw funkcji uaktualniających środowisko:

- `deleteElement(binder)` - funkcja usuwająca binder ze stosu,
- `renameElement(binder, nazwa)` - funkcja zmieniająca nazwe bindera,
- `addAtBottom(binder)` - funkcja dodający nowy binder na dole stosu,
- `addAtTop(biner)` - funkcja dodająca nowy binder na górze stosu,
- `deleteAtBottom(binder)` - funkcja usuwająca binder z dołu stosu,
- `deleteAtTop(binder)` - funkcja usuwająca binder z góry stosu.

Funkcje te są wykorzystywane przede wszystkim przez konstrukcje imperatywne.

Funkcja `nested()`, zwracająca „wnętrze” przetwarzanego obiektu, jest wywoływana zawsze przed włożeniem nowej sekcji na stos środowiskowy. Jej implementacja ma charakter obiektowy, a więc jej definicja znajduje się w każdej klasie reprezentującej rezultat z domyślnie zwracaną wartością będącą pustą kolekcję typu `bag`. Tylko w trzech przypadkach jej definicja jest inna:

- dla wartości typu `IdentifierValue` — zwracany jest odpowiedni rezultat zgodny z definicją funkcji `nested()` zaprezentowaną w rozdziale pierwszym,
- dla wartości typu `BinderValue` — zwracany jest ten binder,
- dla wartości typu `StructValue` — zwracany jest rezultat będący sumą mnogościową elementów struktury.

4.6 Drzewo składni abstrakcyjnej

Drzewo składni abstrakcyjnej zaimplementowane zostało jest jako klasa `AbstractSyntaxTree`. Podstawowe dwie grupy węzłów jakie posiada to wyrażenia (klasa `Expression`) oraz instrukcje (klasa `Statement`).

Wyrażenia to takie konstrukcje programistyczne, których ewaluacja pozostawia pojedynczy wynik na stosie rezultatów *QRES*. Z syntaktycznego punktu widzenia wyrażenia zostały podzielone na pojedyncze (literały, identyfikatory, konstruktory pustych kolekcji) oraz na złożone (unarne, binarne oraz ternarne).

Dodatkowo wyrażenia binarne zostały podzielone na algebraiczne i niealgebraiczne, a wyrażenia unarne na parametryzowane i nieparametryzowane nazwą.

Instrukcje to takie konstrukcje programistyczne, których ewaluacja nie pozostawia wyniku na stosie rezultatów *QRES*. Zazwyczaj mają charakter imperatywny (zmieniają stan).

Wszystkie możliwe węzły drzewa składni abstrakcyjnej, wraz z ich składnią i semantyką zostały przedstawione w kolejnych punktach.

4.7 Wyrażenia pojedyncze

Wyrażenia pojedyncze to takie których ewaluacja nie zależy od innych wyrażeń, czyli po prawej stronie produkcji nigdy nie będzie nieterminala reprezentującego wyrażenie. Wyrażenia takie zostały zaimplementowane jako klasa `SingleExpression`.

Literały

Składnia:

`wyrażenie` → `bool | integer | double | string`

Klasy implementujące:

- `BooleanExpression`
- `IntegerExpression`
- `DoubleExpression`
- `StringExpression`

Semantyka:

Literały są wartościami których ewaluacja polega na włożeniu na stos rezultatów wartości odpowiedniego typu wywnioskowanego z wyniku parsowania. Możliwymi wartościami są:

- `bool` - reprezentuje wartość logiczną której wartości to `true` lub `false`,
- `integer` - reprezentuje liczby całkowite,
- `double` - reprezentuje liczby rzeczywiste,
- `string` - reprezentuje łańcuchy znakowe zawarte pomiędzy znakami cudzysłowu (`"..."`).

Przykłady:

- `true`
- `123`
- `3.14`
- `"To jest łańcuch znakowy"`

Identyfikatory

Składnia:

`wyrażenie → identyfikator`

Klasa implementująca:

- `IdentifierExpression`

Semantyka:

Identyfikatory są to nazwy, których ewaluacja powoduje zbindowanie odpowiednich wartości na stosie ENVS, a następnie włożenie tych wartości na stos QRES. Wynikiem jest kolekcja typu `bag` wartości zależnych od typów obiektów.

Przykłady:

- `EMPLOYEE`
- `DEPARTMENT`

Konstruktory pustych kolekcji

Składnia:

`wyrażenie → bag() | struct() | sequence()`

Klasy implementujące:

- `EmptyBagExpression`
- `EmptyStructExpression`
- `EmptySequenceExpression`

Semantyka:

Ewaluacja powyższych konstruktorów powoduje włożenie na stos QRES odpowiednią kolekcje, bez żadnych wartości.

Przykłady:

- `struct()`
- `bag()`
- `sequence()`

Wywołanie procedury/funkcji bezparametrowej

Składnia:

`wyrażenie → identyfikator ()`

Klasa implementująca:

- `EmptyProcedureCallExpression`

Semantyka:

Ewaluacja powyższego wyrażenia powoduje wywołanie procedury/funkcji, której rezultat zostanie włożeny na stos QRES.

Przykłady:

- `test()/2`
- `foo() + foo()`

Operator gwiazdki

Składnia:

`wyrażenie` → `*`

Klasa implementująca:

- `AsteriskExpression`

Semantyka:

Ewaluacja powyższego operatora zwraca kolekcje typu `bag` składająca się z wartości binderów znajdujących sie na czubku stosu ENVS.

Przykłady:

- `*`
- `EMPLOYEE.*`
- Podaj średnią liczbę atrybutów w obiektach EMPLOYEE:
  `avg(EMPLOYEE.count(*))`

4.8 Wyrażenia unarne

Wyrażenia unarne to takie których ewaluacja jest uzależniona od jednego wyrażenia, czyli po prawej stronie produkcji, oprócz terminali, znajduje się dokładnie jeden nieterminal reprezentujący wyrażenie. Wyrażenia takie zostały zaimplementowane jako klasa `UnaryExpression`.

Ewaluacja wyrażeń unarnych przebiega w następujących krokach:

1. Na początku następuje ewaluacja wyrażenia znajdującego się po prawej stronie produkcji.
2. Zostaje obliczony wynik na podstawie rezultatu zwróconego przez poprzednie wyrażenie. Specyfika obliczenia wyniku (funkcja `compute()`) zależy od wyrażenia unarnego.

4.8.1 Operator logicznej negacji

Składnia:

`wyrażenie → ! wyrażenie`

Klasa implementująca:

- `NegationExpression`

Semantyka:

Ewaluacja tego operatora polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to wartość logiczna (`bool`), dokonywana jest jej negacja. Wynikiem jeste wartość logiczna. W pozostałych przypadkach zachodzi błąd.

Przykłady:

- `! true`
- `! ! false`

4.8.2 Operatory arytmetyczne

Składnia:

`wyrażenie → - wyrażenie | + wyrażenie`

Klasy implementujące:

- `UMinusExpression`
- `UPlusExpression`

Semantyka:

Ewaluacja tych operatorów polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to wartość liczbowa (`integer` lub `double`), to następuje zamiana znaku liczby na przeciwny. Wynikiem jest wartość liczbowa. W pozostałych przypadkach zachodzi błąd.

Przykłady:

- `- 3`
- `+ - 4`

4.8.3 Operatory inkrementacji/dekrementacji

Składnia:

`wyrażenie → -- wyrażenie | ++ wyrażenie`

Klasy implementujące:

- `PreDecrementExpression`

- `PreIncrementExpression`

Semantyka:

Ewaluacja tych operatorów polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to identyfikator obiektu liczbowego (`integer` lub `double`), to następuje zmniejszenie/zwiększenie go o 1. W pozostałych przypadkach zachodzi błąd. Wynikiem jest wartość liczbowa. Operatory te zostały zaimplementowane tylko w wersji pre(inkrementacji/dekrementacji).

Przykłady:

- `-(--i)`
- `++(-zmienna)`

4.8.4 Funkcje liczbowe

Składnia:

```
wyrażenie → nazwa_funkcji ( wyrażenie )
nazwa_funkcji → sqrt | abs | sin | cos | log | sign
```

Klasy implementujące:

- `SquareRootExpression`
- `AbsoluteValueExpression`
- `SineExpression`
- `CosineExpression`
- `LogarithmExpression`
- `SignumExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to wartość liczbowa (`integer` lub `double`), to w zależności od funkcji następuje odpowiednie wyliczenie wartości. Wynikiem jest wartość liczbowa. W pozostałych przypadkach zachodzi błąd.

- `sqrt` - funkcja zwraca pierwiastek kwadratowy,
- `abs` - funkcja zwraca wartość absolutną,
- `sin` - funkcja zwraca sinus,
- `cos` - funkcja zwraca kosinus,
- `log` - funkcja zwraca logarytm,
- `sign` - funkcja zwraca `1` lub `-1` w zależności od znaku liczby.

Przykłady:

- `sqrt(4)`
- `cos(sin(0))`
- `sign(log(exp(3)))`

4.8.5 Funkcje arytmetyczne zagregowane

Składnia:

```
wyrażenie → nazwa_funkcji ( wyrażenie )
nazwa_funkcji → sum | avg | min | max
```

Klasy implementujące:

- `SumExpression`
- `AverageExpression`
- `MinimumExpression`
- `MaximumExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to kolekcja wartości liczbowych (`integer` lub `double`), to w zależności od funkcji następuje odpowiednie wyliczenie wartości. W pozostałych przypadkach zachodzi błąd. Wynikiem jest wartość liczbowa.

- `sum` - funkcja zwraca sumę z liczb w kolekcji,
- `avg` - funkcja zwraca średnią arytmetyczną z liczb w kolekcji,
- `min` - funkcja zwraca najwiekszą liczbę z liczb w kolekcji,
- `max` - funkcja zwraca najmniejszą liczbę z liczb w kolekcji.

Przykłady:

- `avg(bag(2,3,4,5))`
- `sum(bag(2,3,4,5))/4`
- `sum(max(EMPLOYEE.SALARY) + min(EMPLOYEE.SALARY))`

4.8.6 Funkcje operujące na kolekcjach

Składnia:

```
wyrażenie → nazwa_funkcji ( wyrażenie )
nazwa_funkcji → count | unique | exists
```

Klasy implementujące:

- `CounteExpression`
- `UniqueExpression`
- `ExistsExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to kolekcja dowolnych wartości, to w zależności od funkcji następuje odpowiednie wyliczenie wartości. W pozostałych przypadkach zachodzi błąd. Wynik jest uzależniony od funkcji.

- `count` - funkcja zlicza liczbę elementów w kolekcji (wynikiem jest liczba),
- `unique` - funkcja usuwa duplikaty z kolekcji (wynikiem jest kolekcja),
- `exists` - funkcja sprawdzająca czy kolekcja nie jest pusta (wynikiem jest wartość logiczna).

Przykłady:

- `count(EMPLOYEE)`
- `unique(EMPLOYEE.JOB)`
- `exists(EMPLOYEE where JOB == "Programmer")`

4.8.7 Funkcje wyznaczające klucz sortowania

Składnia:

```
wyrażenie → nazwa_funkcji ( wyrażenie )
nazwa_funkcji → asc | desc
```

Klasy implementujące:

- `AscendingExpression`
- `DescendingExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia znajdującego się po prawej stronie produkcji. Wynik jest uzależniony od funkcji:

- `asc` (*ascending*) - jest funkcją identyczności, wynikiem jest argument bez zmian,
- `desc` (*descending*) - jest funkcją dopełnienia, wynikiem jest dopełnienie argumentu.

Przykłady:

- `3; asc(3); desc(3);`
- `EMPLOYEE order by NAME`
- `EMPLOYEE order by desc(NAME), asc(SALARY)`

4.8.8 Operatory koercji

Składnia:

```
wyrażenie → nazwa_funkcji ( wyrażenie )
nazwa_funkcji → toInteger | toDouble | toString | toSingle
              | toStruct | toBag | toSequence
```

Klasy implementujące:

- `ToIntegerExpression`
- `ToDoubleExpression`

- `ToStringExpression`
- `ToSingleExpression`
- `ToStructExpression`
- `ToBagExpression`
- `ToSequenceExpression`

Semantyka:

Operatory koercji to operatory zmiany typów wartości. Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia po prawej stronie produkcji i w zależności od funkcji następuje odpowiednie wyliczenie wartości. Argument oraz wynik jest uzależniony od funkcji.

- `toInteger` - operator ten oczekuje liczby całkowitej lub rzeczywistej, wynikiem jest liczba całkowita `integer`,
- `toDouble` - operator ten oczekuje liczby całkowitej lub rzeczywistej, wynikiem jest liczba rzeczywista `double`,
- `toString` - operator ten oczekuje dowolnej wartości atomowej (`integer`, `double`, `string`, `bool`), wynikiem jest łańcuch znaków `string`.
- `toSingle` - operator ten oczekuje dowolnej kolekcji jednoelementowej, wynikiem jest zmiana typu z kolekcji na pojedynczą wartość. Operator ten w przypadku argumentu będącego dowolną inną wartością, zwraca ją bez zmian,
- `toStruct` - operator ten oczekuje dowolnej kolekcji, wynikiem jest zmiana typu tej kolekcji na kolekcję typu `struct`. Operator ten w przypadku argumentu nie będącego kolekcją, wkłada ten argument w kolekcje typu `struct`,
- `toBag` - operator ten oczekuje dowolnej kolekcji, wynikiem jest zmiana typu tej kolekcji na kolekcję typu `bag`. Operator ten w przypadku argumentu nie będącego kolekcją, wkłada ten argument w kolekcje typu `sequence`,
- `toSequence` - operator ten oczekuje dowolnej kolekcji, wynikiem jest zmiana typu tej kolekcji na kolekcję typu `sequence`. Operator ten w przypadku argumentu nie będącego kolekcją, wkłada ten argument w kolekcje typu `sequence`,

Przykłady:

- `toInteger(3.14)`
- `toString(2 * 3.14 - 3)`
- `toSingle(EMPLOYEE where NAME == "Smith")`
- `toStruct(EMPLOYEE)`

4.8.9 Konstruktory obiektów

Składnia:

`wyrażenie` → `create [ permanent | local ] wyrażenie`

Klasy implementujące:

- `CreateExpression`
- `CreatePermanentExpression`
- `CreateLocalExpression`

Semantyka: Ewaluacja tego wyrażenia polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli jest to binder albo kolekcja binderów, to tworzone są obiekty w składzie odpowiadające strukturze binderów. Wynikiem jest identyfikator albo kolekcja identyfikatorów utworzonych obiektów.

Przykłady:

- `create 3 as Liczba;`
- `create permanent ("Jan" as Imie, "Kowalski" as Nazwisko) as Osoba;`

4.8.10 Konstruktory wartości złożonych (kolekcji)

Składnia:

`wyrażenie` → `nazwa_funkcji ( wyrażenie )`
`nazwa_funkcji` → `struct | bag | sequence`

Klasy implementujące:

- `StructExpression`
- `BagExpression`
- `SequenceExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia po prawej stronie produkcji i w zależności od konstruktora następuje odpowiednie wyliczenie wartości. Argumentem konstruktora może być dowolna wartość. Wynik konstruktora jest uzależniony od funkcji.

- `bag` - wynikiem tego konstruktora jest kolekcja typu `bag`. Konstrukcja `bag(`r_1`, `r_2`, ..., `r_k`)` jest równoważna `bag(`r_1 `union` r_2 `union ... union` r_k`)`,
- `struct` - wynikiem tego konstruktora jest kolekcja typu `struct`. Konstrukcja `struct(`r_1`, `r_2`, ..., `r_k`)` jest równoważna `(`r_1`, `r_2`, ..., `r_k`)`,
- `sequence` - wynikiem tego konstruktora jest kolekcja typu `sequence`.

Przykłady:

- `struct(1,2,3)`
- `bag("a", "b", 3, 3.14, false)`
- `bag(struct(1, "Atos"), struct(2, "Portos"), struct(3, "Aramis"))`
- `sequence(10, 20, 21, 30)`

4.8.11 Wywołanie procedury/funkcji z parametrami

Składnia:

`wyrażenie` → `identyfikator ( wyrażenie )`

Klasy implementujące:

- `ProcedureCallExpression`

Semantyka:

Ewaluacja powyższego wyrażenia polega na ewaluacji wyrażenia po prawej stronie produkcji, a następnie na uruchomieniu procedury/funkcji, której rezultat zostanie włożony na stos QRES.

Przykłady:

- `test(3+2, 4)/2`
- `foo(x*2) + foo(x/2)`

4.8.12 Operator dereferencji

Składnia:

`wyrażenie` → `deref ( wyrażenie )`

Klasa implementująca:

- `DereferenceExpression`

Semantyka:

Ewaluacja operatora dereferencji polega na ewaluacji wyrażenia znajdującego się po prawej stronie produkcji i jeżeli jest to identyfikator obiektu ze składu to następuje wydobycie jego wartości. W przypadku gdy wynikiem wyrażenia nie jest identyfikator, zostanie zwrócony ten wynik bez zmian. Dereferencja dla większości operatorów jest wywoływana *implicite*.

Definicja dereferencji obiektu o została przedstawiona poniżej:

- gdy o jest obiektem atomowym $\langle i, n, v\rangle$ to zwracana jest jego wartość atomowa v,
- gdy o jest obiektem pointerowym $\langle i, n, i_1\rangle$ to zwracany jest wartość bedąca identyfikatorem i_1,
- gdy o jest obiektem złożonym $\langle i, n, \{\langle i_1, n_1, v_1\rangle, \langle i_2, n_2, v_2\rangle, \ldots, \langle i_k, n_k, v_k\rangle\}\rangle$ to zwracany jest kolekcja typu `struct` zawierająca elementy bedące wynikami referencji podobiektów o identyfikatorach $i_1, i_2, \ldots, i_k$,
- gdy o jest obiektem reprezentującym procedure/funkcję to zgłaszany jest błąd.

Przykłady:

- `deref(2), deref("asdf"), deref(true)`
- `deref(bag("a", "b", 3, 3.14, false))`
- `EMPLOYEE.(deref(NAME), deref(SALARY))`

4.9 Wyrażenia unarne parametryzowane nazwą

Oprócz powyższych operatorów unarnych, istnieją jeszcze wyrażenia które z syntaktycznego punktu widzenia są binarne, natomiast z punktu widzenia semantyki są traktowane jako unarne. Są to wyrażenia parametryzowane nazwą. Wyrażenia takie zostały zaimplementowane jako klasa `ParametricExpression`.

Składnia:

```
wyrażenie → wyrażenie operator identyfikator
operator → as | group as
```

Klasy implementujące:

- `AsExpression`
- `GroupAsExpression`

Semantyka:

Ewaluacja takiego wyrażenia powoduje włożenie na stos QRES wyniku ewaluacji wyrażenia, po czym dany rezultat opatrzony jest nazwą w sposób zależny od operatora:

- w przypadku operatora `as` etykietowana jest każda wartość zwracana przez wyrażenie. Operator ten nie zmienia charakteru kolekcji tj. `bag(1,2,3) as T → bag(T(1), T(2), T(3))`
- w przypadku operatora `group as` działanie jest podobne do działania operatora `as` z różnicą w przypadku kolekcji. Operator `group as` przypisuje nazwę do całości wyniku wyrażenia, a nie do poszczególnych elementów kolekcji tj. `bag(1,2,3) as T → T(bag(1, 2, 3))`

Operatory te mogą być dowolnie zagnieżdżane.

Przykłady

- `bag(struct(1, "Jan"), struct(2, "Mirek")) as kolega group as koledzy`
- `("Jan" as Imie, "Kowalski" as Nazwisko, 25 as Wiek) group as Osoba`
- Podwyższ wszystkim pracownikom zarobek o 100:
  `for each EMPLOYEE as E do E.SALARY += 100;`

4.10 Wyrażenia binarne

Wyrażenia binarne to takie których ewaluacja jest uzależniona od dwóch wyrażeń, czyli po prawej stronie produkcji, oprócz terminali, znajdują się dokładnie dwa nieterminale reprezentujące wyrażenia. Wyrażenia takie zostały zaimplementowane jako klasa `BinaryExpression`.

Ewaluacja wyrażeń binarnych przebiega w następujących krokach:

1. Na początku następuje ewaluacja wyrażeń znajdujących się po prawej stronie produkcji (kolejność ich ewaluacji nie ma znaczenia).
2. Zostaje obliczony wynik na podstawie dwóch ostatnich rezultatów zwróconych przez poprzednie wyrażenia. Specyfika obliczenia wyniku (funkcja `compute()`) zależy od wyrażenia binarnego.

4.10.1 Operatory arytmetyczne

Składnia:

```
wyrażenie → wyrażenie operator wyrażenie
operator → + | - | * | / | % | ^
```

Klasy implementujące:

- `PlusExpression`
- `MinusExpression`
- `TimesExpression`
- `DivideExpression`
- `ModuloExpression`
- `PowerExpression`

Semantyka:

Ewaluacja tych operatorów polega na ewaluacji wyrażeń po prawej stronie produkcji. Z wyjątkiem operatora `+`, który działa także jako operator konkatenacji łańcuchów znakowych, pozostałe operatory wymagaja aby operandy były wartościami liczbowymi (`integer` lub `string`). W pozostałych przypadkach zachodzi błąd. Operator `^` jest operatorem potęgowania.

Przykłady:

- `2 + 3 * 4 ^ 5`
- `"Ala ma " + 2 + " koty"`
- `--(-2 - -2)`
- `(256 % 2) * (255 % 2)`

4.10.2 Operatory porównania

Składnia:

```
wyrażenie → wyrażenie operator wyrażenie
operator → == | != | compare
```

Klasy implementujące:

- `EqualExpression`
- `NotEqualExpression`
- `CompareExpression`

Semantyka:

Ewaluacja tych operatorów polega na ewaluacji wyrażeń po prawej stronie produkcji. Z wyjątkiem operatora `compare`, operatory te wymagaja aby operandy były wartościami atomowymi (`bool`, `int`, `double`, `string`) oraz porównywalnymi (ich typy muszą być zgodne). W pozostałych przypadkach

zachodzi błąd. Operator `compare` jest uogólnieniem operatora `==`. Jego specyfika polega na tym, iż potrafi porównywać dowolnie złożone wartości oraz w przypadku niepowodzenia zgodności typów, zwraca wartość `false` zamiast zgłaszać błąd.

Przykłady:

- `log(exp(3)) == exp(log(3))`
- `3 as L compare 3 as L`
- `count(EMPLOYEE where SALARY > 3000) != 0`

4.10.3 Operatory porównania na liczbach

Składnia:

```
wyrażenie → wyrażenie operator wyrażenie
operator → > | < | >= | <=
```

Klasy implementujące:

- `GreaterExpression`
- `LowerExpression`
- `GreaterEqualExpression`
- `LowerEqualExpression`

Semantyka:

Ewaluacja tych operatorów polega na ewaluacji wyrażeń po prawej stronie produkcji, i w przypadku gdy są to wartości porównywalne (ich typy są zgodne) zostaje wykonane odpowiednie wyliczenie. Wynikiem jest wartość logiczna. W pozostałych przypadkach zachodzi błąd.

Przykłady:

- `4 < 5 || 5 < 4`
- `sin(x) >= -1 && sin(x) <= 1`
- `for all EMPLOYEE (SALARY > 1000)`

4.10.4 Operatory logiczne

Składnia:

```
wyrażenie → wyrażenie operator wyrażenie
operator → && | ||
```

Klasy implementujące:

- `ConjunctionExpression`
- `DisjunctionExpression`

Semantyka:

Ewaluacja tych operatorów polega na ewaluacji wyrażeń po prawej stronie produkcji i w przypadku, gdy są to wartości logiczne `bool` zostaje wykonane odpowiednie wyliczenie (konkatenacja && lub alternatywa `||`). Wynikiem jest wartość logiczna. W pozostałych przypadkach zachodzi błąd.

Przykłady:

- `true || false`
- `!true && !false`
- `log(exp(3)) && exp(log(3))`

4.10.5 Operatory działające na zbiorach

Składnia:

```
wyrażenie → wyrażenie operator wyrażenie
operator → in | union | intersect | minus | symmetric minus | ,
```

Klasy implementujące:

- `InclusionExpression`
- `UnionExpression`
- `IntersectionExpression`
- `DifferenceExpression`
- `SymmetricDifferenceExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażenia po prawej stronie produkcji i w zależności od funkcji następuje odpowiednie wyliczenie wartości. W pozostałych przypadkach zachodzi błąd. Arguemnt oraz wynik jest uzależniony od funkcji.

- `union` - operator realizujący sumę teorio-mnogościową na operandach, argumenty mogą być dowolnymi wartościami, wynikiem jest kolekcja typu `bag`,
- `intersect` - operator realizujący przecięcie teorio-mnogościowe na operandach argumenty mogą być dowolnymi wartościami, wynikiem jest kolekcja typu `bag`,
- `in` - operator realizujący zawieranie teorio-mnogościowe na operandach, argument po lewej stronie może być dowolną wartością, argument po prawej stronie musi być kolekcją typu `sequence` lub `bag`, wynikiem jest wartością logiczną,
- `minus` - operator realizujący różnice teorio-mnogościową na operandach argumenty muszą być kolekcjami typu `bag`, wynikiem jest kolekcja typu `bag`,
- `symmetric minus` - operator realizujący różnice symetryczną teorio-mnogościową na operandach argumenty muszą być kolekcjami typu `bag`, wynikiem jest kolekcja typu `bag`,
- `,` - operator realizujący produkt kartezjański na operandach argumenty mogą być dowolnymi wartościami, wynik jest uzależniony od operandów,

Przykłady:

- `(struct(10)) union (struct(20))`
- `struct(1,2,3) in bag(struct(1), struct(2), struct(3), struct(1,2,3))`
- `(bag(1,2,3) minus bag(2,3,4)) union (bag(2,3,4) minus bag(1,2,3))`
- `(bag(1,2,3) symmetric minus bag(2,3,4))`

4.10.6 Operatory podstawienia

Składnia:

```
wyrażenie → wyrażenie operator wyrażenie
operator → = | += | -= | *= | /= | %=
```

Klasy implementujące:

- `AssignExpression`
- `PlusAssignExpression`
- `MinusAssignExpression`
- `TimesAssignExpression`
- `DivideAssignExpression`
- `ModuloAssignExpression`

Semantyka:

Operatory podstawienia są operatorami imperatywnymi (zmieniającymi stan). Ewaluacja tych operatorów polega na ewaluacji wyrażeń po prawej stronie produkcji i jeżeli lewym operandem jest identyfikator obiektu, a prawym operandem jest wartość odpowiadająca typowi obiektu to następuje podstawienie tej wartości pod ten obiekt. W każdym innym przypadku zachodzi błąd. Wynikiem jest identyfikator obiektu znajdujący sie po lewej stronie.

Przykłady:

- `i %= 2`
- `EMPLOYEE.SALARY -= 100`
- `(EMPLOYEE where NAME == "Smith").SALARY += 100`

4.10.7 Operator wstawiania

Składnia:

```
wyrażenie → wyrażenie <- wyrażenie
```

Klasa implementująca:

- `InsertExpression`

Semantyka:

Operator wstawienia jest operatorem imperatywnym (zmieniającym stan). Ewaluacja tego operatora polega na ewaluacji wyrażeń po prawej stronie produkcji i jeżeli zarówno lewym i prawym operandem jest identyfikator obiektu, przy czym identyfikator obiektu po lewej stronie jest identyfikatorem obiektu złożonego, to następuje wstawienie obiektu po prawej stronie do obiektu po lewej stronie. W każdym innym przypadku zachodzi błąd. Wynikiem jest identyfikator obiektu znajdujący sie po lewej stronie.

Przykłady:

- `EMPLOYEE <- create 3000 as SALARY`
- ```
  (EMPLOYEE where NAME == "Smith")
       <- (EMPLOYEE where NAME == "Brown").WORKS_IN
  ```

4.10.8 Operator zakresu

Składnia:

`wyrażenie → wyrażenie [ wyrażenie ]`

Klasa implementująca:

- `BRangeExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażeń po prawej stronie produkcji. Wyrażenie stojące po lewej stronie operatora `[]` musi zwrócić kolekcję typu `sequence`. Wyrażenie znajdujące się wewnątrz operatora `[]` musi zwrócić wartość będącą liczbą całkowitą. Wynikiem jest kolekcja typu `bag` zawierająca jeden element pasujący do podanego indeksu. W sytuacji gdy podany indeks wykracza poza dostępny zakres zostaje zgłoszony odpowiedni błąd. W pozostałych przypadkach zachodzi błąd.

Przykłady:

- `sequence(10,20,30,40)[2]`
- `(sequence(10,20,100,200) group as P).P[count(P)]`

4.11 Wyrażenia binarne niealgebraiczne

Oprócz powyższych operatorów, istnieje grupa operatorów binarnych, których semantyka nie sprowadza się w prosty sposób do algebry, stąd też nazwa - operatory niealgebraiczne. Główna różnica pomiędzy operatorami algebraicznymi a niealgebraicznymi polega na ich stosunku do stosu ENVS. Operatory algebraiczne, w trakcie ewaluacji wykorzystują tylko stos QRES, podczas gdy operatory niealgebraiczne, oprocz stosu QRES, wykorzystują także stos ENVS. Ponadto w operatorach niealgebraicznych kolejność ewaluacji operandów ma znaczenie (zawsze będa wykonywane od lewej do prawej). Wyrażenia takie zostały zaimplementowane jako klasa `AlgebraicExpression`.

Ewaluacja wyrażeń posiadających operatory niealgebraiczne, przebiega w następujących krokach:

1. Na początku następuje ewaluacja wyrażenia znajdującego sie po lewej stronie operatora (wynik jest kolekcją typu `bag`).
  ```

2. Dla każdego elementu e należącego do tej kolekcji, zostaną wykonane następujące czynności:
   (a) Na stos ENVS zostaje włożona nowa sekcja, która jest wnętrzem elementu e (wywołana zostaje funkcja `nested(`e`)`).
   (b) W nowo powstałym środowisku, następuje ewaluacja wyrażenia po prawej stronie operatora niealgebraicznego.
   (c) Zostaje obliczony wynik pośredni będący pewnym połączeniem elementu e z rezultatem zwróconym przez ostatnie wyrażenie. Specyfika połączenia (funkcja `combine()`) zależy od operatora niealgebraicznego.
   (d) Ze stosu ENVS usuwana jest ostatnia sekcja.

3. Wszystkie wyniki pośrednie zostaną w pewien sposób zsumowane w wynik końcowy, który zostaje włożony na stos QRES. Specyfika zsumowania wyników pośrednich (funkcja `merge()`) zależy od operatora niealgebraicznego.

Dla niektórych operatorów semantyka zostanie przedstawiona bardziej szczegółowo za pomocą dodatkowej tabeli. W każdym takim przypadku:

- `e` - oznacza wartość pojedynczą czyli binder, identyfikator lub wartość atomową,
- `str` - oznacza kolekcję typu `struct`,
- `bag` - oznacza kolekcję typu `bag`,
- `seq` - oznacza kolekcję typu `sequence`.

4.11.1 Operator nawigacji

Składnia:

`wyrażenie` → `wyrażenie . wyrażenie`

Klasa implementująca:

- `NavigationExpression`

Semantyka:

- funkcja `combine()` - ignoruje e, zwraca wynik wyrażenia po prawej stronie operatora niealgebraicznego,
- funkcja `merge()` - sumuje mnogościowo wszystkie wyniki pośrednie.

Opis działania oraz zwracanych wyników został bliżej przedstawiony w tabeli 4.11.1.

Przykłady:

- `EMPLOYEE.SALARY`
- `(EMPLOYEE where NAME == "Smith").(SALARY, JOB)`
- `(EMPLOYEE where NAME == "Smith").WORKS_IN.DEPARTMENT.NAME`

Tabela 4.1: Działanie operator nawigacji.

L	R	L . R
e	e	e
e	str	str
e	bag	bag
e	seq	seq
str	e	e
str	str	str
str	bag	bag
str	seq	seq
bag(l_1, l_2, ..., l_m)	e	bag(e_1, e_2, ..., e_m)
bag(l_1, l_2, ..., l_m)	str	bag(str_1, str_2, ..., str_m)
bag(l_1, l_2, ..., l_m)	bag	bag(bag_1, bag_2, ..., bag_m)
bag(l_1, l_2, ..., l_m)	seq	bag(seq_1, seq_2, ..., seq_m)
seq(l_1, l_2, ..., l_m)	e	seq(e_1, e_2, ..., e_m)
seq(l_1, l_2, ..., l_m)	str	seq(str_1, str_2, ..., str_m)
seq(l_1, l_2, ..., l_m)	bag	bag(bag_1, bag_2, ..., bag_m)
seq(l_1, l_2, ..., l_m)	seq	bag(seq_1, seq_2, ..., seq_m)

4.11.2 Operator selekcji

Składnia:

`wyrażenie` → `wyrażenie where wyrażenie`

Klasa implementująca:

- `SelectionExpression`

Semantyka:

- Ograniczenie - wynik wyrażenia po prawej stronie operatora musi zwrócić wartość logiczna `true` ub `false`, w przeciwnym przypadku zachodzi błąd.
- funkcja `combine()` - dla każdego e będącego stanowiącego wynik zapytania po lewej stronie operatora, funkcja zwraca:
  - `bag()` - pustą kolekcje, gdy wynikiem zapytania po prawej stronie operatora jest wartość logiczna `false`,
  - `bag(`e`)` - jednoelementową kolekcje, gdy wynikiem zapytania po prawej stronie operatora jest wartość logiczna `true`.
- funkcja `merge()` - sumuje mnogościowo wszystkie wyniki pośrednie.

Przykłady:

- `EMPLOYEE where SALARY > 1000`
- `EMPLOYEE where WORKS_IN.DEPARTMENT.NAME == "Sales"`

4.11.3 Operator zależnego złączenia

Składnia:

`wyrażenie` → `wyrażenie join wyrażenie`

Klasa implementująca:

- `DependentJoinExpression`

Semantyka:

- funkcja `combine()` - zarówno e jak i każdy element e_2 zwracany przez wyrażenie po prawej stronie operatora traktuje jako struktury (jednoelementowe lub wieloelementowe). Struktura taka jest tworzona poprzez połączenie e z e_2. Specyfika tego połączenia zależy od typu rezultatu. Wynikiem jest kolekcja typu `struct` lub kolekcja typu `bag` kolekcji typu `struct`.
- funkcja `merge()` - sumuje mnogościowo wszystkie wyniki pośrednie.

Opis działania oraz zwracanych wyników został bliżej przedstawiony w tabeli 4.11.3.

Tabela 4.2: Działanie operator zależnego złączenia.

L	R	L **join** R
e	e	str(e, e)
e	str(r_1, ..., r_n)	str(e, r_1, ..., r_n)
e	bag(r_1, ..., r_n)	bag(str(e,r_1), ..., str(e,r_n))
e	seq(r_1, ..., r_n)	bag(str(e,r_1), ..., str(e,r_n))
str(l_1, ..., l_m)	e	str(e, e)
str(l_1, ..., l_m)	str(r_1, ..., r_n)	str(r_1, ..., r_n, e)
str(l_1, ..., l_m)	bag(r_1, ..., r_n)	bag(str(l_1, ..., l_m, r_1), str(l_1, ..., l_m, r_2), str(l_1, ..., l_m, r_n))
str(l_1, ..., l_m)	seq(r_1, ..., r_n)	bag(str(l_1, ..., l_m, r_1), str(l_1, ..., l_m, r_2), str(l_1, ..., l_m, r_n))
bag(l_1, ..., l_m)	e	bag(str(l_1, e), ..., str(l_m, e))
bag(l_1, ..., l_m)	str(r_1, ..., r_n)	bag(str(l_1, r_1, ..., r_n), str(l_2, r_1, ..., r_n), str(l_m, r_1, ..., r_n))
bag(l_1, ..., l_m)	bag(r_1, ..., r_n)	bag(str(l_1, r_1), ..., str(l_1, r_n), str(l_2, r_1), ..., str(l_2, r_n), str(l_m, r_1), ..., str(l_m, r_n))
bag(l_1, ..., l_m)	seq(r_1, ..., r_n)	bag(str(l_1, r_1), ..., str(l_1, r_n), str(l_2, r_1), ..., str(l_2, r_n), str(l_m, r_1), ..., str(l_m, r_n))
seq(l_1, ..., l_m)	e, str, seq, bag	wyniki dla sekwencji są takie same jak dla zbioru bag

Przykłady:

- `EMPLOYEE join SALARY`

- `(EMPLOYEE where NAME == "Smith").(SALARY join JOB)`
- `DEPARTMENT as D join EMPLOYEE where WORKS_IN.DEPARTMENT.NAME == D`

4.11.4 Operator sortowania

Składnia:

`wyrażenie` → `wyrażenie order by wyrażenie`

Klasa implementująca:

- `SortingExpression`

Semantyka:

- Ograniczenie - wyrażenie po prawej stronie operatora wyznacza klucz sortowania dlatego wynik zapytania po prawej stronie operatora musi zwracać wartości dla których istnieje pewien porządek liniowy. W naszej implementacji wartościami takimi mogą być liczby rzeczywiste, liczby całkowite, łańcuchy znakowe i wartości logiczne.
- funkcja `combine()` - działa tak samo jak dla operatora zależnego złaczenia przy czym na wartości po prawej stronie operatora jest dokonywana automatycznie dereferencja,
- funkcja `merge()` - sumuje mnogościowo wszystkie wyniki pośrednie a następnie dokonywane jest sortowanie powstałego zbioru. Sortowanie następuje w ramach pierwszego klucza, następnie w ramach identycznych wartości pierwszego klucza — według drugiego klucza, itd. Ostatecznie zostają usunięte klucze sortowania a wynik jest kolekcją typu `sequence`.

Opis działania oraz zwracanych wyników został bliżej przedstawiony w tabeli 4.11.4.

Tabela 4.3: Działanie operator sortowania.

L	R	L **order by** R
e	e	seq(e)
e	str(r_1, ..., r_n)	seq(e)
e	bag(r_1, ..., r_n)	seq(e_1, ..., e_n)
e	seq(r_1, ..., r_n)	seq(e_1, ..., e_n)
str	e	seq(str)
str	str(r_1, ..., r_n)	seq(str)
str	bag(r_1, ..., r_n)	seq(str_1, ..., str_n)
str	seq(r_1, ..., r_n)	seq(str_1, ..., str_n)
bag	e	seq(bag)
bag	str(r_1, ..., r_n)	seq(bag)
bag	bag(r_1, ..., r_n)	seq(bag_1, ..., bag_n)
bag	seq(r_1, ..., r_n)	seq(bag_1, ..., bag_n)
seq	e	seq(seq)
seq	str(r_1, ..., r_n)	seq(seq)
seq	bag(r_1, ..., r_n)	seq(seq_1, ..., seq_n)
seq	seq(r_1, ..., r_n)	seq(seq_1, ..., seq_n)

Przykłady:

- `EMPLOYEE order by SALARY, NAME`
- `EMPLOYEE order by WORKS_IN.DEPARTMENT.NAME`
- `DEPARTMENT order by count(EMPLOYS)`

4.11.5 Operator tranzytywnego domknięcia

Składnia:

`wyrażenie` → `wyrażenie close by wyrażenie`

Klasa implementująca:

- `TransitiveClosureExpression`

Semantyka:

- funkcja `combine()` - do sekcji stosu QRES, której elementem jest e, dostawiany jest rezultat wyrażenia po prawej stronie operatora. Dostawianie polega na sumowaniu mnogościowym.
- funkcja `merge()` - sumuje mnogościowo wszystkie wyniki pośrednie.

Przykłady:

- Pokaż wszystkie podczęści dysku twardego:

```
((PART where NAME == "disk drive")
  close by (CONSISTS_OF.DETAIL.PART)).NAME
```

- Pokaż wszystkie podczęści dysku twardego wraz z ich poziomem wystąpienia:

```
( (PART where NAME == "disk drive", 1 as LEVEL)
   close by (CONSISTS_OF.(DETAIL.PART, (LEVEL + 1) as LEVEL))
).(NAME, LEVEL)
```

- Pokaż wszystkie liściowe podczęści dysku twarego:

```
(((PART where NAME == "disk drive")
  close by (CONSISTS_OF.DETAIL.PART)
 ) where !exists CONSISTS_OF).NAME
```

4.11.6 Operator kwantyfikatora egzystencjalnego

Składnia:

`wyrażenie` → `wyrażenie for any wyrażenie`
`wyrażenie` → `for any wyrażenie ( wyrażenie )`

Klasa implementująca:

- `ForAnyExpression`

Semantyka:

- Ograniczenie - wynik wyrażenia po prawej stronie operatora musi zwrócić wartość logiczna `true` ub `false`, w przeciwnym przypadku zachodzi błąd.
- funkcja `combine()` - ignoruje e, zwraca wynik wyrażenia po prawej stronie operatora niealgebraicznego,
- funkcja `merge()` - zwraca wartość logiczną `true` jeżeli co najmniej jeden wynik pośredni zwrócony przez zapytanie po prawej stronie operatora jest wartością logiczna `true`, w przeciwnym przypadku zwraca wartość logiczną `false`.

Przykłady:

- `EMPLOYEE for any SALARY > 1000`
- `for any DEPARTMENT (count(EMPLOYS) == count(DEPARTMENT))`

4.11.7 Operator kwantyfikatora uniwersalnego

Składnia:

```
wyrażenie → wyrażenie for all wyrażenie
wyrażenie → for all wyrażenie ( wyrażenie )
```

Klasa implementująca:

- `ForAllExpression`

Semantyka:

- Ograniczenie - wynik wyrażenia po prawej stronie operatora musi zwrócić wartość logiczna `true` ub `false`, w przeciwnym przypadku zachodzi błąd.
- funkcja `combine()` - ignoruje e, zwraca wynik wyrażenia po prawej stronie operatora niealgebraicznego,
- funkcja `merge()` - zwraca wartość logiczną `false` jeżeli co najmniej jeden wynik pośredni zwrócony przez zapytanie po prawej stronie operatora jest wartością logiczna `false`, w przeciwnym przypadku zwraca wartość logiczną `true`.

Przykłady:

- `EMPLOYEE for all SALARY > 1000`
- `for all DEPARTMENT (count(EMPLOYS) == count(DEPARTMENT))`

4.12 Wyrażenia ternarne

Wyrażenia ternarne to takie których ewaluacja jest uzależniona od trzech wyrażeń, czyli po prawej stronie produkcji, oprócz terminali, znajdują się dokładnie trzy nieterminale reprezentujące wyrażenia. Wyrażenia takie zostały zaimplementowane jako klasa `TernaryExpression`.

Ewaluacja wyrażeń ternarnych przebiega w następujących krokach:

1. Na początku następuje ewaluacja wyrażeń znajdujących się po prawej stronie produkcji (kolejność ich ewaluacji nie ma znaczenia).
2. Zostaje obliczony wynik na podstawie trzech ostatnich rezultatów zwróconych przez poprzednie wyrażenia. Specyfika obliczenia wyniku (funkcja `compute()`) zależy od wyrażenia ternarnego.

4.12.1 Operator warunkowy

Składnia:

`wyrażenie` → `wyrażenie ? wyrażenie : wyrażenie`

Klasa implementująca:

- `ConditionExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażeń po prawej stronie produkcji. Wyrażenie stojące po lewej stronie operator `?` musi zwrócić wartość logiczną `bool`. W pozostałych przypadkach zachodzi błąd. Jeżeli wartość ta jest prawdą `true` zwrócone zostanie środkowe wyrażenie, w przeciwnym przypadku zwrócone zostanie wyrażenie znajdujące się po prawej stronie operatora `:`. Wynik jest uzależniony od wyniku zwróconego wyrażenia.

Przykłady:

- `x < y ? x : y`
- `exists(EMPLOYEE where exists WORKS_IN) ? "tak" : "nie"`

4.12.2 Operator zakresu

Składnia:

`wyrażenie` → `wyrażenie [ wyrażenie .. wyrażenie ]`

Klasa implementująca:

- `RangeExpression`

Semantyka:

Ewaluacja tych wyrażeń polega na ewaluacji wyrażeń po prawej stronie produkcji. Wyrażenie stojące po lewej stronie operatora `[]` musi zwrócić kolekcję typu `sequence`. Wyrażenia znajdujące się wewnątrz operatora `[]` muszą zwrócić wartości będące liczbami całkowitymi. Wynikiem jest kolekcja typu `sequence` zawierająca elementy pasujące do podanego zakresu. W sytuacji gdy podany zakres został przekroczony zostaje zgłoszony odpowiedni błąd. W pozostałych przypadkach zachodzi błąd.

Przykłady:

- `sequence(10,20,30,40)[2..3][2]`
- `(sequence(10,20,100,200) group as P).P[2 .. count(P)]`
- Zwróć trzech najwięcej zarabiających pracowników:
  `(EMPLOYEE order by SALARY)[1 .. 3]`

4.13 Instrukcje

4.13.1 Pusta instrukcja

Składnia:

```
instrukcja → ;
```

Klasa implementująca:

- `EmptyStatement`

Semantyka:

Powyższa instrukcja znajduje swoje odbicie tylko w drzewie rozbioru programu (podczas parsingu). W żaden sposób nie jest ewaluowana.

Przykłady:

- `;;;;`
- `3;;;;4;;;;`

4.13.2 Instrukcja–wyrażenie

Składnia:

```
instrukcja → wyrażenie ;
```

Klasa implementująca:

- `ExpressionStatement`

Semantyka:

Ewaluacja powyższej instrukcji polega najpierw na ewaluacji wyrażenia, którego wynik znajdzie się na stosie QRES, a następnie na jego zdjęciu. W chwili obecnej dodatkowo zdejmowany wynik jest wyświetlany na ekran.

Przykłady:

- `1; 2; 3; 4;`
- `true; false; 3 as Z; struct(1, "a", 3.14);`

4.13.3 Bloki

Składnia:

```
instrukcja → { instrukcje }
instrukcja → { }
```

Klasy implementujące:

- `BlockStatement`

- `EmptyBlockStatement`

Semantyka: Ewaluacja powyższej instrukcji dla niepustego bloku przebiega w kilku krokach:

1. Otwierana jest nowa sekcja na stosie ENVS.
2. W ramach tej sekcji zostają ewaluowane instrukcje, których ewaluacja może powołać do życia zmienne lokalne.
3. Usuwane są wszystkie obiekty lokalne znajdujące się na czubku stosu ENVS.
4. Zdejmowana jest sekcja stosu ENVS.

Pusty blok ma znaczenie syntaktyczne (podczas parsingu) a ewaluacja dla niego nie zachodzi.

Przykłady:

- `{}{1;}{2;}{}`
- `if (true) { false; }`

4.13.4 Instrukcje warunkowe

Składnia:

```
instrukcja →  if ( wyrażenie ) instrukcja
instrukcja →  if ( wyrażenie ) instrukcja else instrukcja
```

Klasy implementujące:

- `IfStatement`
- `IfElseStatement`

Semantyka:

Ewaluacja powyższych instrukcji polega najpierw na ewaluacji wyrażenia a następnie na ewaluacji jednej z instrukcji (lub żadnej). Wyrażenie musi zwracać wartość logiczną.

- W przypadku instrukcji `if` bez `else` gdy wyrażenie zwróci wartość `true` zostaje wykonana `instrukcja`, w przeciwnym przypadku `instrukcja` nie zostaje ewaluowana,
- W przypadku instrukcji `if` z `else` gdy wyrażenie zwróci wartość `true` zostaje wykonana pierwsza `instrukcja`, w przeciwnym przypadku zostaje ewaluowana druga `instrukcja`.

Problem zwisającego `else` został rozwiązany w taki sposób, iż `else` zawsze jest przyporządkowane do ostatniej instrukcji `if`.

Przykład:

- ```
  if (x > y)
      res = x;
  res = y;
  ```
- `if(-1 <= 0) if (-1 == 0) 0; else -1;`
  jest równoważne
  `if(-1 <= 0) { if(-1 == 0) 0; else -1; }`
  ```

4.13.5 Instrukcje pętli

Składnia:

```
instrukcja → while ( wyrażenie ) instrukcja
instrukcja → do instrukcja ( wyrażenie ) ;
```

Klasy implementujące:

- `WhileStatement`
- `DoWhileStatement`

Semantyka:

Ewaluacja powyższych instrukcji polega na wykonaniu instrukcji po prawej stronie produkcji, przy czym wyrażenie musi zwracać wartość logiczną. Instrukcja zostaje wykonana dopóki wyrażenie będzie zwracać wartość logiczną `true`, inaczej jest przerywana. Różnica pomiędzy pętlami polega na tym iż w pętli `do ... while` instrukcja jest wykonywana co najmniej raz, podczas gdy w pętli `while` tak nie jest.

Przykład:

- ```
  while((EMPLOYEE where NAME == "Smith").SALARY > avg(EMPLOYEE.SALARY))
      (EMPLOYEE where NAME == "Smith").SALARY -= 100;
  ```
- ```
  create 1 as i;
  do
      (EMPLOYEE order by SALARY)[i].SALARY += avg(EMPLOYEE.SALARY)/i;

  while (++i < count(EMPLOYEE));
  ```

4.13.6 Instrukcja iteracji

Składnia:

```
instrukcja → for each wyrażenie do instrukcja
```

Klasa implementująca:

- `ForEachStatement`

Semantyka:

Ewaluacja powyższej instrukcji jest bardzo podobna do ewaluacji operatora niealgebraicznego:

1. Na początku następuje ewaluacja wyrażenia (wynik jest kolekcją typu `bag`).
2. Dla każdego elementu e należącego do tej kolekcji, zostaną wykonane następujące czynności:
   - Na stos ENVS zostaje włożona nowa sekcja, będąca „wnętrzem” elementu e (wywołana zostaje funkcja `nested(`e`)`).
   - W nowo powstałym środowisku, następuje ewaluacja instrukcji.
   - Następuje zdjęcie sekcji ze stosu środowiskowego, oraz powrót do punktu 2.

Przykłady:

- ```
  for each (EMPLOYEE where JOB == "salesman") as E do
       E.WORKS_IN = (DEPARTMENT where NAME == "Sales");
  ```
- ```
  for each EMPLOYEE as E do
      while ((E.SALARY) > 0)
         E.SALARY -= 100;
  ```

4.13.7 Instrukcja przerwania pętli

Składnia:

```
instrukcja → break ;
```

Klasa implementująca:

- `BreakStatement`

Semantyka:

Ewaluacja powyższej instrukcji polega na przerwaniu ewaluacji dowolnej instrukcji będącej blokiem.

Przykład:

- ```
  while (i<10) {
      if (i == 3) break;
      ++i;
  }
  ```

4.13.8 Instrukcja deklaracji procedury/funkcji

Składnia:

```
instrukcja → procedure identyfikator ( ) { instrukcje }
instrukcja → procedure identyfikator ( parametry ) { instrukcje }
instrukcja → function identyfikator ( ) { instrukcje }
instrukcja → function identyfikator ( parametry ) { instrukcje }
```

Klasy implementujące:

- `EmptyProcedureDeclarationStatement`
- `ProcedureDeclarationStatement`

Semantyka:

Ewaluacja powyższej instrukcji polega na stworzeniu nowego obiektu w składzie, bedącego procedurą/funkcją, którego wartością jest:

- w przypadku procedury/funkcji bezparametrowej, obiekt taki posiada tylko wartość będącą drzewem rozbioru instrukcji,
- w przypadku procedury/funkcji posiadającej parametry, obiekt taki posiada dwie wartości z których jedna jest drzewem rozbioru listy parametrów, a druga jest jest drzewem rozbioru listy instrukcji.
```

Przy deklaracji procedury/funkcji, jej ciało oraz parametry nie są ewaluowane. Ich ewaluacja następuje w momencie wywołania procedury/funkcji.

Przykłady:

- ```
  function silnia(n) {
      if (n > 1)
          return n * silnia(n-1);
      return 1;
  }
  ```
- ```
  procedure oblicz(n) {
       silnia(n) * silnia(n+1);
  }
  ```

4.13.9 Instrukcja powrotu z procedury/funkcji

Składnia:

```
instrukcja → return ;
instrukcja → return wyrażenie ;
```

Klasy implementujące:

- `EmptyReturnStatement`
- `ReturnStatement`

Semantyka:

Ewaluacja powyższej instrukcji jest uzależniona od rodzaju.

- w przypadku pustej instrukcji `return`, wynikiem jest pusta kolekcja typu `bag`,
- w przypadku niepustej instrukcji `return`, wynikiem jest rezultat ewaluacji wyrażenia po prawej stronie.

Przykład:

- ```
  procedure sprawdz(Liczba) {
      if (Liczba > 5000)
          return ;
      return Liczba;
  }
  ```

4.13.10 Instrukcja usuwania obiektów

Składnia:

```
instrukcja → delete wyrażenie ;
```

Klasa implementująca:

- `DeleteStatement`
```

Semantyka:

Ewaluacja powyższej instrukcji polega na ewaluacji wyrażenia po prawej stronie produkcji i jeżeli wyrażenie to zwraca dowolną kolekcję identyfikatorów obiektów to następuje usunięcie tych obiektów ze składu danych. W każdym innym przypadku zachodzi błąd.

Przykłady:

- `delete (EMPLOYEE where NAME == "Brown").WORKS_IN;`
- `delete EMPLOYEE join DEPARTMENT;`

4.13.11 Instrukcja zmiany nazwy obiektów

Składnia:

`instrukcja` → `rename wyrażenie to identyfikator ;`

Klasa implementująca:

- `RenameToStatement`

Semantyka:

Ewaluacja powyższej instrukcji polega na ewaluacji wyrażeń po prawej stronie produkcji i jeżeli lewym operandem jest identyfikator dowolnego obiektu lub kolekcja typu `bag` takich identyfikatorów a prawym jest identyfikator to następuje zmiana nazwy wszystkim tym obiektom na nazwe będącą identyfikatorem. W każdym innym przypadku zachodzi błąd.

Przykłady:

- `rename EMPLOYEE.PREVIOUS_JOB to WORKED_FOR;`
- `rename EMPLOYEE to EMP;`

4.13.12 Instrukcja aktualizacji obiektów

Składnia:

`instrukcja` → `update wyrażenie to wyrażenie ;`

Klasa implementująca:

- `UpdateToStatement`

Semantyka:

Ewaluacja powyższej instrukcji polega na ewaluacji wyrażeń po prawej stronie produkcji i jeżeli lewym operandem jest identyfikator dowolnego obiektu lub kolekcja typu `bag` takich identyfikatorów a prawym jest wartość której typ jest zgodny z typami obiektu/obiektów po lewej stronie to następuje przypisanie. W każdym innym przypadku zachodzi błąd.

Przykłady:

- `update (EMPLOYEE where NAME != "Brown")`
  `to (EMPLOYEE where NAME == "Brown");`
- `update EMPLOYEE.SALARY to avg(EMPLOYEE.SALARY);`

4.13.13 Instrukcja wstawiania obiektów

Składnia:

`instrukcja` → `insert wyrażenie into wyrażenie ;`

Klasa implementująca:

- `InsertIntoStatement`

Semantyka:

Ewaluacja powyższej instrukcji polega na ewaluacji wyrażeń po prawej stronie produkcji i jeżeli prawym operandem jest identyfikator obiektu złożonego a lewym jest identyfikator lub kolekcja typu `bag` identyfikatorów, to następuje wstawienie obiektu/obiektów po lewej stronie do obiektu po prawej stronie. W każdym innym przypadku zachodzi błąd.

Przykłady:

- `insert EMPLOYEE.* into (EMPLOYEE where NAME == "Brown");`
- `insert (EMPLOYEE where NAME == "Brown").WORKS_IN`
  `into (EMPLOYEE where NAME == "Brown");`

4.14 Rozbudowa języka Yaql o nowe instrukcje/wyrażenia/rezultaty

Naczelną zasadą przyświecająca autorowi podczas konstrukcji oraz późniejszej rozbudowy języka jest zasada korespondecji, która mówi, że wraz z wprowadzeniem do języka pewnej cechy X należy precyzyjnie określić inne cechy języka w taki sposób, aby cecha X współdziałała z już istniejącymi konstrukcjami, została wtopiona w istniejące lub zmodyfikowane mechanizmy nazywania, typowania, zakresu i wiązania, oraz miała zapewnioną uniwersalną obsługę.

Przy rozszerzaniu funkcjonalności języka o dodatkowe wyrażenia, instrukcje lub rezultaty należy przede wszystkim stosować się do powyższej zasady a dodatkowym drogowskazem są poniższe wskazówki.

W przypadku dodania nowej instrukcji należy:

- stworzyć klase reprezentującą tą instrukcje (konwencja nazewnicza stosowana w implementacji to nazwa instrukcji wraz z dołączonym słowem `Statement` np. `IfStatement`, `WhileStatement`),
- klasa ta powinna dziedziczyć z klasy `Statement`,
- klasa ta powinna zostać umieszczona w pakiecie `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.statements`,
- gdy instrukcja wprowadza nowe słowo kluczowe lub rozbudowywuje gramatykę powinny zostać odpowiednio zmodyfikowane pliki konfiguracyjne generatora skanera oraz parsera,

- klasa ta, powinna w odpowiedni sposób zapewnić poprawną ewaluację instrukcji, jaką reprezentuje. Dokonywane zmiany należy uczynić w klasie reprezentującej ewaluacje `EvalVisitor`.

W przypadku dodania nowego wyrażenia należy:

- stworzyć klase reprezentującą to wyrażenie (konwencja nazewnicza stosowana w implementacji to nazwa wyrażenia wraz z dołączonym słowem `Expression` np. `MinusExpression`),
- klasa ta w zależności od charakteru wyrażenia powinna dziedziczyć jedną z poniższych klas:
  - `SingleExpression` — gdy dodawane wyrażenie nie zawiera w sobie innych wyrażeń,
  - `ParametricExpression` — gdy dodawane wyrażenie jest wyrażeniem unarnym parametryzowanym nazwą,
  - `NonParametricExpression` — gdy dodawane wyrażenie jest wyrażeniem unarnym nie parametryzowanym nazwą,
  - `UnaryExpression` — gdy dodawane wyrażenie jest wyrażeniem unarnym, które nie kwalifikuje się do powyższych wyrażeń unarnych,
  - `AlgebraicExpression` — gdy dodawane wyrażenie jest wyrażeniem binarnym algebraicznym,
  - `NonAlgebraicExpression` — gdy dodawane wyrażenie jest wyrażeniem binarnym niealgebraicznym,
  - `BinaryExpression` — gdy dodawane wyrażenie jest wyrażeniem binarnym, które nie kwalifikuje się do powyższych wyrażeń binarnych,
  - `TernaryExpression` — gdy dodawane wyrażenie jest wyrażeniem ternarnym,
  - `Expression` — gdy dodawane wyrażenie jest wyrażeniem które nie kwalifikuje się do powyższych wyrażeń,
- klasa ta w zależności od charakteru powinna zostać umieszczona w jednym z poniższych pakietów:
  - gdy dodawane wyrażenie jest wyrażeniem pojedynczym: `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.single`,
  - gdy dodawane wyrażenie jest wyrażeniem unarnym: `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.unary`,
  - gdy dodawane wyrażenie jest wyrażeniem binarnym: `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.binary`,
  - gdy dodawane wyrażenie jest wyrażeniem ternarnym: `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.ternary`,
  - gdy dodawane wyrażenie jest wyrażeniem nie kwalifikującym się do powyższych: `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions`,
- gdy wyrażenie wprowadza nowe słowo kluczowe lub rozbudowywuje gramatykę powinny zostać odpowiednio zmodyfikowane pliki konfiguracyjne generatora skanera oraz parsera,
- klasa ta, powinna w odpowiedni sposób zapewnić poprawną ewaluację wyrażenia, jakie reprezentuje. Dokonywane zmiany należy uczynić w klasie reprezentująca ewaluacje `EvalVisitor`. Dla wyrażeń polegać to będzie na odpowiednim zdefiniowaniu funkcji:
  - `combine` oraz `merge` w przypadku wyrażenia posiadającego operator niealgebraiczny,

  - `compute` — we wszystkich pozostałych przypadkach wyrażeń.

W przypadku dodania nowej konstrukcji nie zaliczającej się do powyższych, należy:

- stworzyć klase reprezentującą tą konstrukcje,
- klasa ta powinna dziedziczyć z klasy `AbstractSyntaxTree`, lub z dowolnej innej nowo utworzonej klasy (w zależności od charakteru konstrukcji). Przykładem może tu być klasa reprezentująca deklaracje.
- klasa ta powinna zostać umieszczona w pakiecie `pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree`, lub w dowolnej innej nowo utworzonej,
- gdy konstrukcja wprowadza nowe słowo kluczowe lub rozbudowywuje gramatykę powinny zostać odpowiednio zmodyfikowane pliki konfiguracyjne generatora skanera oraz parsera,
- klasa ta, powinna w odpowiedni sposób zapewnić poprawną ewaluację konstrukcji, jaką reprezentuje. Dokonywane zmiany należy uczynić w klasie reprezentującej ewaluacje `EvalVisitor`.

W przypadku dodania nowego typu zwracanego przez zapytanie należy:

- stworzyć klase reprezentującą ten rezultat,
- klasa ta w zależności od rodzaju rezultatu powinna dziedziczyć jedną z poniższych klas:
  - `ComplexValue` — gdy dodawany rezultat stanowi wartość złożoną (kolekcje),
  - `SingleValue` — gdy dodawany rezultat stanowi wartość pojedynczą,
  - `AtomicValue` — gdy dodawany rezultat stanowi wartość pojedynczą będącą wartością atomową. Przykładem może tu być wartość reprezentująca daty,
  - `QueryResult` — gdy dodawany rezultat jest rezultatem nie kwalifikującym się do powyższych,
- klasa ta w zależności od rodzaju rezultatu powinna zostać umieszczona w jednym z poniższych pakietów:
  - gdy dodawany rezultat stanowi wartość złożoną (kolekcje): `pl.edu.pjwstk.yaod.yaql.engine.queryresulttypes.complex`,
  - gdy dodawany rezultat stanowi wartość pojedynczą: `pl.edu.pjwstk.yaod.yaql.engine.queryresulttypes.single`,
  - gdy dodawany rezultat jest nowym rodzajem rezultatu nie kwalifikującym się do powyższych: `pl.edu.pjwstk.yaod.yaql.engine.queryresulttypes`,
- dla nowo wprowadzego typu rezultatu powinna zostać określona semantyka dla wszystkich istniejących wyrażeń oraz instrukcji. Domyślnie w każdym takim przypadku podnoszony jest wyjątek `TypeMismatchException` wyświetlający informacje o błędnym użyciu.

4.15 Podsumowanie

Zaprezentowany język zapytań Yaql posiada następujące cechy:

- jest zformalizowanym językiem zapytań opartym na bardzo mocnych podstawach teoretycznych podejścia stosowego,
- jest językiem w pełni ortogonalnym, łatwo poddającym się rozbudowie,
- jest językiem zwięzłym oraz czytelnym posiadającym precezyjną oraz intuicyjną składnie i semantyke,
- jest językiem łatwym do nauczenia oraz użycia,
- jest językiem relewantnym do relacyjnych, relacyjno–obiektowych oraz obiektowo–zorientowanych modeli. Wszystkie podstawowe konstrukcje OQL oraz SQL są bezpośrednio w nim wyrażalne,
- nie zmusza ani nie zabrania używania zmiennych iteracyjnych (tj. pomocniczych nazw),
- posiada składnie abstrakcyjną odzwierciedlającą reguły semantyczne, unika lukru syntaktycznego.

Rozdział 5

Narzędzia zastosowane w implementacji

5.1 Język Java

Java jest obiektowym językiem programowania stworzonym przez firmę Sun Microsystems w roku 1995. U źródeł powstania Javy leżała potrzeba zbudowania środowiska do tworzenia aplikacji, który spełniałby wyzwania ery Internetu — zdolność do pracy w heterogenicznym i rozproszonym środowisku, przy zachowaniu niezbędnych środków bezpieczeństwa, wykorzystaniu minimalnych zasobów, możliwości dynamicznej rozbudowy i uruchamiania na dowolnej platformie programowej. Obecnie jest to jeden z najbardziej popularnych języków programowania, a ze względu na bardzo wysokopoziomowy charakter — idealny do szybkiego tworzenia aplikacji.

Wszelkie informacje na temat języka Java uzyskać można na stronie:
`http://java.sun.com`.

5.2 Oracle XML Parser

Parser XML jest oprogramowanie analizującym dokumenty tego języka oraz udostępniającym aplikacjom ich strukturę i zawartość w łatwej do przetwarzania formie. Zwalniając programistów z konieczności tworzenia własnych procedur przeprowadzających analizę składniową i leksykalną dokumentów, parsery XML umożliwiają również sprawdzenie ich poprawności składniowej oraz strukturalnej.

Moduł ładujący dokumenty XML do bazy Yaod korzysta z jednego z najlepszych dostępnych parserów XML — Oracle XML Parser. Parser ten udostępnia zawartość analizowanych przez siebie dokumentów za pomocą interfejsów DOM i SAX, może pracować jako parser walidujący, jak i niewalidujący.

Oracle XML Parser dostępny jest (wraz z dokumentacją) na stronie:
`http://technet.oracle.com`.

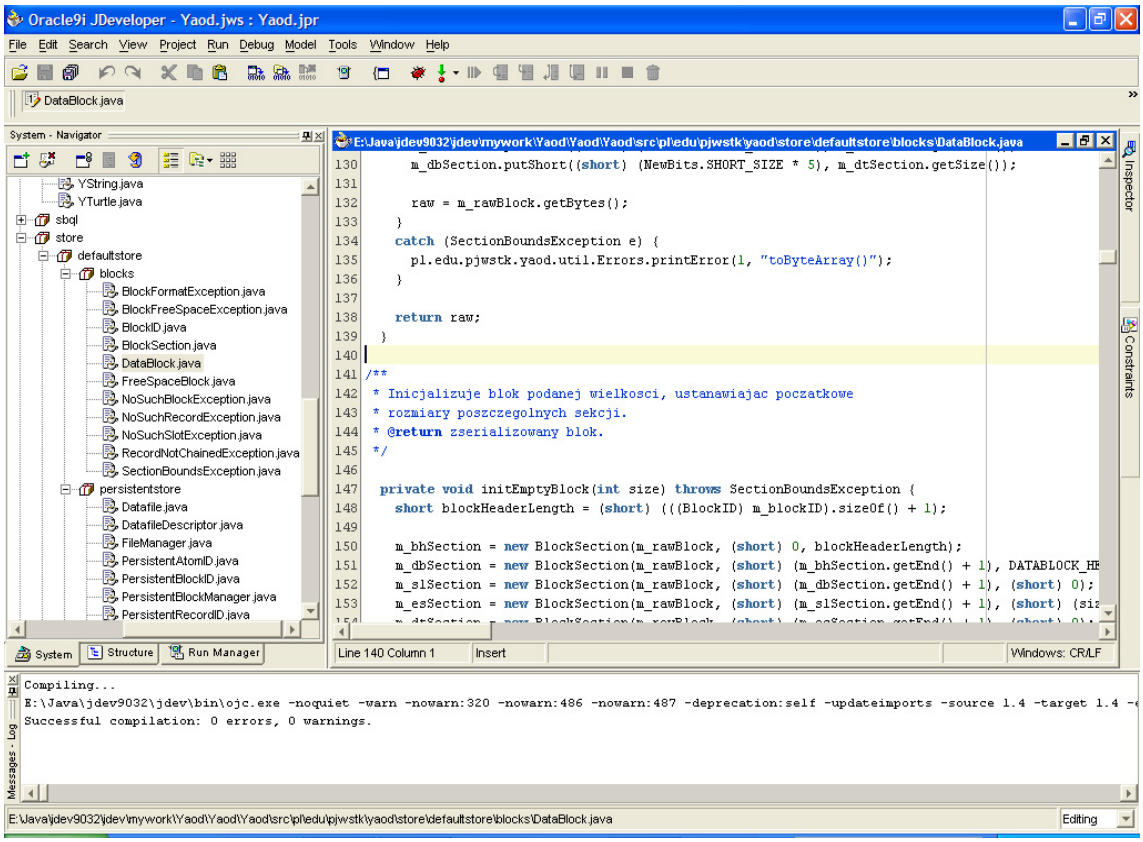
5.3 Środowisko programistyczne JDeveloper

Oracle9*i* JDeveloper jest jednym z jednym z najnowocześniejszych zintegrowanych środowisk programistycznych (IDE) dla Javy. JDeveloper wspomaga developerów w ich pracy za pomocą zbioru zintegroawnych narzędzi, umożliwiających produktywne wytwarzanie i publikowanie aplikacji. Środowisko

to wspomaga programistę na każdym etapie jego pracy, m.in. modelowanie, generację kodu, kodowanie, debugowanie, testowanie, optymalizję i publikację. Gdyby nie możliwości JDevelopera (zwłaszcza wspaniały debugger), wówczas najprawdopodobniej projekt Yaod nigdy nie zostałby ukończony w tak krótkim czasie.

Więcej informacji na temat Środowiska JDeveloper uzyskać można na stronie:
`http://technet.oracle.com`.

Rysunek 5.1: Projekt Yaod w środowisku programistycznym JDeveloper



5.4 Serwer CVS

CVS (*Concurrent Versions System*) jest darmowym oprogramowaniem wykorzystywanym przez autorów jako system kontroli wersji. CVS wprowadza centralne repozytorium dla wszystkich oficjalnie wydanych źródeł i zmian, jednocześnie pozwalając poszczególnym programistom programistom na zarządzanie lokalnymi kopiami kodu źródłowego wraz z roboczymi zmianami. CVS pozwala m.in. monitorować i rejestrować zmiany wprowadzone przez pracujących równolegle developerów, łączyć je i wykrywać konflikty między nimi. Bez wykorzystania CVS, praca kilku programistów nad jednym projektem polegałaby na uciążliwym „umawianiu się" kto i gdzie w danej chwili może wprowadzać zmiany do kodu źródłowego, a w przypadku awarii, powrót do ostatniej działającej wersji byłby niemożliwy. Dzięki modułowi programu JDeveloper, integrującemu to środowisko z serwerem CVS, kontrola wersji w projekcie

Yaod przebiegała szybko i sprawnie.

Strona domowa projektu CVS, dostępna jest pod adresem:
`http://www.cvshome.org`.

5.5 JFlex

JFlex jest generatorem analizatorów leksykalnych, stworzonym przez G. Kleina. Pomimo, iż generuje kod w języku Java, jego sposób funkcjonowania zbliżony jest do popularnego generatora analizatorów leksykalnych Lex/FLex. JFlex został zaprojektowany z myślą o łatwej integracji m.in. generatorem parserów o nazwie CUP. Bez wykorzystania obu tych narzędzi, czas implementacji języka Yaql znacznie by się wydłużył.

Więcej informacji na temat generator JFlex uzyskać można na stronie domowej projektu, dostepnej pod adresem:
`http://www.jflex.de`.

5.6 CUP

CUP jest generatorem parserów LALR, worzonym na przestrzeni lat głównie przez trzy osoby: S. Hudsona, F. Flannery i C.S. Ananiana. Pomimo, iż generuje kod w języku Java, jego sposób funkcjonowania zbliżony jest do popularnego generatora parserów o nazwie YACC. CUP oferuje programistom kilka bardzo użytecznych udogodnień, a wygenerowany przez niego kod łatwo integruje się z kodem wygenerowanym przez JFlex.

Generator CUP ściągnąć można ze strony:
`http://www.cs.princeton.edu/ appel/modern/java/CUP`.

Podsumowanie

Systemy zarządzania bazami danych są obecnie uważane za jeden z najbardziej zaawansowanych typów oprogramowania tworzonego przez człowieka. Aplikacje te budowane są zwykle w dużych zespołach programistycznych, powoływanych przez wielkie korporacje, zdolne do wydawania milionów dolarów dziennie na same badania naukowe. Nie sposób jednak czasami oprzeć się wrażeniu, że sponsorowane przez te firmy badania nie zawsze podążają w odpowiednim kierunku.

W niniejszej pracy magisterskiej pokazano, że przy zastosowaniu odpowiedniego podejścia, dwóch programistów jest w stanie w bardzo krótkim czasie przygotować system zarządzania bazą danych zdolny rozwiązać wiele problemów, z jakimi od lat borykają się największe koncerny informatyczne na świecie. Najważniejsze aspekty, w których Yaod już teraz jest lepszy od wielu konkurencyjnych rozwiązań, to przede wszystkim: jego język zapytań, perspektywy, obsługa danych półstrukturalnych, Null, czy integracja danych. A jest to zaledwie wierzchołek góry lodowej.

Zbudowany system dysponuje zdaniem autorów praktycznie nieograniczonym potencjałem rozwoju. Jego rozbudowa chociażby o własności sygnalizowane niemal na każdym kroku w poprzednich rozdziałach pracy, mogłaby spowodować powstanie produktu będącego zupełnie nową jakością w swojej dziedzinie. Ta nowa jakość jest bardzo potrzebna, gdyż bez niej systemy informatyczne mogą na wieki pogrążyć się w totalnym chaosie.

Dodatek A

Słowa kluczowe języka Yaql

Wszystkie słowa kluczowe języka Yaql prezentuje poniższa tabela.

Słowo kluczowe	Znaczenie
`abs`	funkcja arytmetyczna
`all`	klauzula operatora `for`
`any`	klauzula operatora `for`
`as`	operator definiowania pomocniczej nazwy lub klauzula polecenia `group`
`asc`	operator identyczności
`avg`	funkcja arytmetyczna zagregowana
`bag`	konstruktor kolekcji typu `bag`
`break`	opuszcza pętle
`by`	klauzula operatora `close` lub `order`
`close`	rozpoczyna operator tranzytywnego domknięcia `close by`
`compare`	generyczny operator porównania
`cos`	funkcja arytmetyczna
`count`	funkcja zliczająca liczbę elementów w kolekcji
`create`	rozpoczyna polecenie tworzenia obiektu `create ...`
`delete`	polecenie usunięcia obiektu
`deref`	operator dereferencji
`desc`	operator dopełnienia
`do`	rozpoczyna pętle `do ... while ...`
`each`	klauzula polecenia `for each`
`else`	klauzula polecenia `if`
`exists`	funkcja sprawdzająca czy kolekcja nie jest pusta
`exp`	funkcja arytmetyczna
`for`	rozpoczyna polecenia iteracji `for each ... do ...` lub jeden z operatorów `for all`, `for any`
`function`	rozpoczyna polecenie deklaracji funkcji `function ... (...) {...}`
`group`	rozpoczyna operator definiowania pomocniczej nazwy `group as`
`if`	polecenie warunku

Słowo kluczowe	Znaczenie
`in`	operator zawierania
`insert`	polecenie wstawiania obiektów
`intersect`	operator przecięcia
`into`	klauzula polecenia `insert`
`join`	operator zależnego złaczenia
`local`	klauzula polecenia `create`
`log`	funkcja arytmetyczna
`max`	funkcja arytmetyczna zagregowana
`min`	funkcja arytmetyczna zagregowana
`minus`	operator różnicy teorio–mnogościowej lub klauzula operatora `symmetric`
`order`	rozpoczyna operator sortowania `order by`
`permanent`	klauzula polecenia `create`
`print`	polecenie wyświetlania wyniku
`procedure`	rozpoczyna polecenie deklaracji procedury `procedure ... (...) {...}`
`rename`	rozpoczyna polecenie zmiany nazwy obiektu `rename ... to ...`
`return`	kończy metodę, zwracając wartość
`sequence`	konstruktor kolekcji typu `sequence`
`struct`	konstruktor kolekcji typu `struct`
`toBag`	operator koercji
`toDouble`	operator koercji
`toInteger`	operator koercji
`toSequence`	operator koercji
`toSingle`	operator koercji
`toString`	operator koercji
`toStruct`	operator koercji
`this`	niejawny argument reprezentujący wnętrze przetwarzanego obiektu złożonego
`to`	klauzula polecenia `update` i `rename`
`show`	polecenie wyświetlania wyniku w postaci surowej
`sign`	funkcja arytmetyczna
`sin`	funkcja arytmetyczna
`sqrt`	funkcja arytmetyczna
`sum`	funkcja arytmetyczna zagregowana
`unique`	funkcja usuwująca duplikaty z kolekcji
`union`	operator sumy teorio–mnogościowej
`update`	rozpoczyna polecenie aktualizacji obiektów `update ... to ...`
`where`	operator selekcji
`while`	rodzaj pętli
`xml`	polecenie wyświetlania wyniku w postaci XML

Dodatek B

Gramatyka języka Yaql

Gramatyka języka Yaql została przedstawiona poniżej za pomocą notacji BNF.

```
program ::= lista_instrukcji

lista_instrukcji ::= instrukcja
                   | instrukcja lista_instrukcji

instrukcja ::= ;
             | wyrażenie ;
             | { lista_instrukcji }
             | { }
             | if ( wyrażenie ) instrukcja
             | if ( wyrażenie ) instrukcja else instrukcja
             | while ( wyrażenie ) instrukcja
             | do instrukcja while ( wyrażenie ) ;
             | for each wyrażenie do instrukcja
             | break ;
             | delete wyrażenie ;
             | rename wyrażenie to identyfikator ;
             | update wyrażenie to wyrażenie ;
             | insert wyrażenie into wyrażenie ;
             | print ( wyrażenie ) ;
             | show  ( wyrażenie ) ;
             | xml   ( wyrażenie ) ;
             | procedure identyfikator ( ) { lista_instrukcji }
             | procedure identyfikator ( lista_parametrów ) { lista_instrukcji }
             | return ;
             | return wyrażenie ;

lista_parametrów ::= identyfikator
                   | identyfikator , lista_parametrów

wyrażenie ::= wyrażenie_pojedyncze
            | wyrażenie_unarne
            | wyrażenie_binarne
            | wyrażenie_ternarne
            | ( wyrażenie )

wyrażenie_pojedyncze ::= identyfikator
                       | literał
                       | bag ()
                       | struct ()
                       | sequence ()
                       | identyfikator ()
                       | *

literał ::= bool
```

```
          | integer
          | double
          | string

wyrażenie_unarne ::= operator_unarny wyrażenie
                   | wyrażenie as identyfikator
                   | wyrażenie group as identyfikator
                   | identyfikator ( wyrażenie )

operator_unarny ::= ! | + | - | ++ | -- |
                  | sqrt | abs | sin | cos | exp | log | sign
                  | exists | unique | count
                  | sum | avg | min | max
                  | asc | desc
                  | toInteger | toDouble | toString | toSingle
                  | toBag | toStruct | toSequence
                  | deref

wyrażenie_binarne ::= wyrażenie operator_algebraiczny wyrażenie
                    | wyrażenie operator_niealgebraiczny wyrażenie
                    | for all wyrażenie ( wyrażenie )
                    | for any wyrażenie ( wyrażenie )
                    | wyrażenie [ wyrażenie ]

operator_algebraiczny ::= + | - | * | / | % | ^
                        | == | != | > | >= | < | <=
                        | && | || | , | compare
                        | in | union | intersect | minus | symmetric minus
                        | = | += | -= | *= | /= | %= | <-

operator_niealgebraiczny ::= . | where | join
                           | for all | for any
                           | close by | order by

wyrażenie_ternarne ::= wyrażenie ? wyrażenie : wyrażenie
                     | wyrażenie [ wyrażenie .. wyrażenie ]
```

Dodatek C

Priorytety operatorów

Priorytety operatorów zaimplementowanych w języku Yaql prezentuje poniższa tabela, przy czym najwyższy priorytet jest u góry tabeli, a najniższy na dole.

Operatory	Łączność
`[]`, `()`, `.`	lewostronna
`deref`, `-` (unarny), `+` (unarny), `!`, `++`, `--`	prawostronna
`exists`, `asc`, `desc`	prawostronna
`^`	prawostronna
`*`, `/`, `%`	lewostronna
`+`, `-`	lewostronna
`>`, `<`, `>=`, `<=`	lewostronna
`==`, `!=`	lewostronna
`&&`	lewostronna
`||`	lewostronna
`as`, `group as`	lewostronna
`sqrt`, `abs`, `sin`, `cos`, `exp`, `log`	lewostronna
`union`, `intersect`, `minus`, `symmetric minus`, `in`, `compare`	lewostronna
`count`	lewostronna
`sum`, `avg`, `min`, `max`	lewostronna
`where`, `join`, `for any`, `for all`, `close by`, `order by`	lewostronna
`struct`, `bag`, `sequence`	lewostronna
`else`	lewostronna
`?:`	lewostronna
`..`	prawostronna
`for each`, `while`, `delete`, `create`, `update`, `insert`, `<-`	prawostronna
`=`, `+=`, `-=`, `*=`, `/=`, `%=`	prawostronna
`,`	lewostronna
`;`	prawostronna

Dodatek D

Konstrukcja skanera

Plik konfiguracyjny dla generatora analizatorów leksykalnych (*JFlex*) został przedstawiony poniżej:

```
package pl.edu.pjwstk.yaod.yaql.parser;

import java_cup.runtime.Symbol;

%%

%public
%class Lexer
%implements Tokens
%cupsym Tokens
%unicode
%line
%column
%cup
%initthrow LexerException
%scanerror Exception
%yylexthrow LexerException
// %debug

%{
  StringBuffer string = new StringBuffer();

  public int getLine() {
    return yyline;
  }
  public int getColumn() {
    return yycolumn;
  }
  public String getText() {
    return yytext();
  }
  private Symbol token(int type) {
    return new Symbol(type, getLine(), getColumn(), getText());
  }
  private Symbol token(int type, Object value) {
    return new Symbol(type, getLine(), getColumn(), value);
  }
%}

INTEGER    = [0-9]+
DOUBLE     = {INTEGER}("."{INTEGER})?
BOOLEAN    = "true"|"false"
IDENTIFIER = [:jletter:][:jletterdigit:]*

LineTerminator      = \r|\n|\r\n
```

```
WhiteSpace          = {LineTerminator} | [ \t\f]
OneLineComment      = "//"[^\n]* {LineTerminator}
TraditionalComment  = "/*"([^*]|"*"+[^"/"])*"*"+"/"
Comment             =  {TraditionalComment} | {OneLineComment}

%state STRING

%%

<YYINITIAL> {
  ";"          { return token(SEMICOLON);        }
  "("          { return token(LROUND);           }
  ")"          { return token(RROUND);           }
  "{"          { return token(LCURLY);           }
  "}"          { return token(RCURLY);           }
  "["          { return token(LSQUARE);          }
  "]"          { return token(RSQUARE);          }

  "abs"        { return token(KEY_ABS);          }
  "all"        { return token(KEY_ALL);          }
  "any"        { return token(KEY_ANY);          }
  "as"         { return token(KEY_AS);           }
  "asc"        { return token(KEY_ASC);          }
  "avg"        { return token(KEY_AVG);          }
  "bag"        { return token(KEY_BAG);          }
  "break"      { return token(KEY_BREAK);        }
  "by"         { return token(KEY_BY);           }
  "close"      { return token(KEY_CLOSE);        }
  "compare"    { return token(KEY_COMPARE);      }
  "cos"        { return token(KEY_COS);          }
  "count"      { return token(KEY_COUNT);        }
  "create"     { return token(KEY_CREATE);       }
  "delete"     { return token(KEY_DELETE);       }
  "deref"      { return token(KEY_DEREF);        }
  "desc"       { return token(KEY_DESC);         }
  "do"         { return token(KEY_DO);           }
  "each"       { return token(KEY_EACH);         }
  "else"       { return token(KEY_ELSE);         }
  "exists"     { return token(KEY_EXISTS);       }
  "exp"        { return token(KEY_EXP);          }
  "for"        { return token(KEY_FOR);          }
  "function"   { return token(KEY_FUNCTION);     }
  "group"      { return token(KEY_GROUP);        }
  "if"         { return token(KEY_IF);           }
  "in"         { return token(KEY_IN);           }
  "insert"     { return token(KEY_INSERT);       }
  "intersect"  { return token(KEY_INTERSECT);    }
  "into"       { return token(KEY_INTO);         }
  "join"       { return token(KEY_JOIN);         }
  "local"      { return token(KEY_LOCAL);        }
  "log"        { return token(KEY_LOG);          }
  "max"        { return token(KEY_MAX);          }
  "min"        { return token(KEY_MIN);          }
  "minus"      { return token(KEY_MINUS);        }
  "order"      { return token(KEY_ORDER);        }
  "permanent"  { return token(KEY_PERMANENT);    }
  "print"      { return token(KEY_PRINT);        }
  "procedure"  { return token(KEY_PROCEDURE);    }
  "rename"     { return token(KEY_RENAME);       }
  "return"     { return token(KEY_RETURN);       }
  "sequence"   { return token(KEY_SEQUENCE);     }
  "show"       { return token(KEY_SHOW);         }
  "sign"       { return token(KEY_SIGN);         }
  "sin"        { return token(KEY_SIN);          }
  "sqrt"       { return token(KEY_SQRT);         }
  "struct"     { return token(KEY_STRUCT);       }
```

```
  "sum"        { return token(KEY_SUM);          }
  "symmetric"  { return token(KEY_SYMMETRIC);    }
  "to"         { return token(KEY_TO);           }
  "toBag"      { return token(KEY_TOBAG);        }
  "toDouble"   { return token(KEY_TODOUBLE);     }
  "toInteger"  { return token(KEY_TOINTEGER);    }
  "toSequence" { return token(KEY_TOSEQUENCE);   }
  "toSingle"   { return token(KEY_TOSINGLE);     }
  "toString"   { return token(KEY_TOSTRING);     }
  "toStruct"   { return token(KEY_TOSTRUCT);     }
  "union"      { return token(KEY_UNION);        }
  "unique"     { return token(KEY_UNIQUE);       }
  "update"     { return token(KEY_UPDATE);       }
  "where"      { return token(KEY_WHERE);        }
  "while"      { return token(KEY_WHILE);        }
  "xml"        { return token(KEY_XML);          }

  "="          { return token(OP_ASSIGN);        }
  "+="         { return token(OP_PLUS_ASSIGN);   }
  "-="         { return token(OP_MINUS_ASSIGN);  }
  "*="         { return token(OP_TIMES_ASSIGN);  }
  "/="         { return token(OP_DIVIDE_ASSIGN); }
  "%="         { return token(OP_MODULO_ASSIGN); }
  "++"         { return token(OP_INCREMENT);     }
  "--"         { return token(OP_DECREMENT);     }
  ".."         { return token(OP_RANGE);         }
  "?"          { return token(OP_QUESTION);      }
  ":"          { return token(OP_COLON);         }
  "&&"         { return token(OP_CONJUNCTION);   }
  "||"         { return token(OP_DISJUNCTION);   }
  "!"          { return token(OP_NEGATION);      }
  "=="         { return token(OP_EQUAL);         }
  "!="         { return token(OP_NOT_EQUAL);     }
  "<"          { return token(OP_LOWER);         }
  "<="         { return token(OP_LOWER_EQUAL);   }
  ">"          { return token(OP_GREATER);       }
  ">="         { return token(OP_GREATER_EQUAL); }
  "+"          { return token(OP_PLUS);          }
  "-"          { return token(OP_MINUS);         }
  "*"          { return token(OP_TIMES);         }
  "/"          { return token(OP_DIVIDE);        }
  "%"          { return token(OP_MODULO);        }
  "^"          { return token(OP_POWER);         }
  ","          { return token(OP_COMMA);         }
  "."          { return token(OP_NAVIGATION);    }
  "<-"         { return token(OP_INSERT);        }

  {BOOLEAN}    { return token(LIT_BOOLEAN, yytext());  }
  {INTEGER}    { return token(LIT_INTEGER, yytext());  }
  {DOUBLE}     { return token(LIT_DOUBLE,  yytext());  }
  \"           { string.setLength(0); yybegin(STRING); }
  {IDENTIFIER} { return token(IDENTIFIER,  yytext());  }

  {WhiteSpace} { /* ignore white spaces */ }
  {Comment}    { /* ignore comments     */ }
}

<STRING> {
  \"                 { yybegin(YYINITIAL);
                       return token(LIT_STRING, string.toString()); }
  {STRING}           { string.append(yytext());  }
  "\\b"              { string.append('\b');      }
  "\\t"              { string.append('\t');      }
  "\\n"              { string.append('\n');      }
  "\\f"              { string.append('\f');      }
  "\\r"              { string.append('\r');      }
```

```
  "\\\""              { string.append('\"');      }
  "\\\'"              { string.append('\'');      }
  "\\\\"              { string.append('\\');      }
  \\.                 { throw new LexerException(
                        "LEXER: Illegal escape sequence \"" + yytext()
                        + "\"", yyline, yycolumn); }
  {LineTerminator}    { throw new LexerException(
                        "LEXER: Unterminated string at end of line",
                        yyline, yycolumn); }
}
<YYINITIAL> {
  .|\n   { throw new LexerException("LEXER: Illegal character <" + yytext()
                                    + ">", yyline, yycolumn); }
}
```

Dodatek E

Konstrukcja parsera

Plik konfiguracyjny dla generatora parserów (*CUP*) został przedstawiony poniżej:

```
package pl.edu.pjwstk.yaod.yaql.parser;

import java_cup.runtime.Symbol;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.statements.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.terminals.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.single.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.unary.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.binary.*;
import pl.edu.pjwstk.yaod.yaql.abstractsyntaxtree.expressions.ternary.*;

import java.io.*;
import java.lang.reflect.*; // Field, Class
import java.util.Stack;

init with {:
// for operatorCOMMA() and its ambiguities
  FlagStack = new Stack();
  FlagStack.push(new Integer(FLAG_NOTDEFINED));
:}

action code {:
  public void pushFlag(int flag) {
    parser.FlagStack.push(new Integer(flag));
  }
  public void popFlag() {
    parser.FlagStack.pop();
  }
  public int topFlag() {
    return ((Integer)(parser.FlagStack.peek())).intValue();
  }
:};

parser code {:
  public final static int FLAG_NOTDEFINED = -1;
  public final static int FLAG_BAG        = 0;
  public final static int FLAG_STRUCT     = 1;
  public final static int FLAG_SEQUENCE   = 2;
  public final static int FLAG_PROCEDURE  = 3;
  public Stack FlagStack;

  public void report_error(String message, Object info) {
    StringBuffer m = new StringBuffer("Error ");
    if (info instanceof Symbol)
```

```
      m.append( "("+info.toString()+")" );
    m.append(" : "+message);
    System.out.println(m);
  }

  public void report_fatal_error(String message, Object info) throws ParserException {
    report_error(message, info);
    Lexer lexer = (Lexer) getScanner();
    throw new ParserException("PARSER: (<" + cur_token.value + ">)",
      lexer.getLine(), lexer.getColumn());
  }

  public void syntax_error(Symbol current) {
    Lexer lexer = (Lexer) getScanner();
    report_error("Syntax error, Token: (" + current.sym + ") -> "
     + ", line: " + lexer.getLine() + ", column: " + lexer.getColumn(), current);
  }
:}

// TERMINALS !!!
terminal String SEMICOLON, UMINUS, UPLUS,
  LROUND, RROUND, LCURLY, RCURLY, LSQUARE, RSQUARE,
  KEY_ABS, KEY_ALL, KEY_ANY, KEY_AS,
  KEY_ASC, KEY_AVG, KEY_BAG, KEY_BREAK,
  KEY_BY, KEY_CLOSE, KEY_COMPARE, KEY_COS,
  KEY_COUNT, KEY_CREATE, KEY_DELETE, KEY_DEREF,
  KEY_DESC, KEY_DO, KEY_EACH, KEY_ELSE,
  KEY_EXISTS, KEY_EXP, KEY_FOR, KEY_FUNCTION,
  KEY_GROUP, KEY_IF, KEY_IN, KEY_INSERT,
  KEY_INTERSECT, KEY_INTO, KEY_JOIN, KEY_LOG,
  KEY_LOCAL, KEY_MIN, KEY_MINUS, KEY_MAX,
  KEY_ORDER, KEY_PERMANENT, KEY_PRINT, KEY_PROCEDURE,
  KEY_RENAME, KEY_RETURN, KEY_SEQUENCE, KEY_SHOW,
  KEY_SIGN, KEY_SIN, KEY_SQRT, KEY_STRUCT,
  KEY_SUM, KEY_SYMMETRIC, KEY_TO, KEY_TOBAG,
  KEY_TODOUBLE, KEY_TOINTEGER, KEY_TOSEQUENCE, KEY_TOSINGLE,
  KEY_TOSTRING, KEY_TOSTRUCT, KEY_UNIQUE, KEY_UNION,
  KEY_UPDATE, KEY_WHERE, KEY_WHILE, KEY_XML,
  OP_ASSIGN, OP_COLON, OP_COMMA, OP_CONJUNCTION,
  OP_DECREMENT, OP_DISJUNCTION, OP_DIVIDE, OP_DIVIDE_ASSIGN,
  OP_EQUAL, OP_GREATER, OP_GREATER_EQUAL, OP_INCREMENT,
  OP_INSERT, OP_LOWER, OP_LOWER_EQUAL, OP_MINUS,
  OP_MINUS_ASSIGN, OP_MODULO, OP_MODULO_ASSIGN, OP_NAVIGATION,
  OP_NEGATION, OP_NOT_EQUAL, OP_PLUS, OP_PLUS_ASSIGN,
  OP_POWER, OP_QUESTION, OP_RANGE, OP_TIMES,
  OP_TIMES_ASSIGN, LIT_BOOLEAN, LIT_DOUBLE, LIT_INTEGER,
  LIT_STRING, IDENTIFIER;

// NONTERMINALS !!!
non terminal                          set_bag_flag, set_str_flag, set_seq_flag,
                                      set_proc_flag, reset_flag;
non terminal AbstractSyntaxTree       ntGOAL;
non terminal StatementList            ntSTMLIST;
non terminal Statement                ntSTM;
non terminal Expression               ntEXP;
non terminal UnaryExpression          ntUNAEXP;
non terminal BinaryExpression         ntBINEXP;
non terminal TernaryExpression        ntTEREXP;
non terminal SingleExpression         ntSINGLEEXP;
non terminal AlgebraicExpression      ntALGEXP;
non terminal NonAlgebraicExpression   ntNONALGEXP;
non terminal ParametricExpression     ntPAREXP;
non terminal NonParametricExpression  ntNONPAREXP;
non terminal FormalParametersList     ntFORPARLIST;

// LOWEST PRECEDENCE !!!
```

```
precedence right  SEMICOLON;
precedence left   OP_COMMA;
precedence right  OP_ASSIGN, OP_PLUS_ASSIGN, OP_MINUS_ASSIGN, OP_TIMES_ASSIGN,
                  OP_DIVIDE_ASSIGN, OP_MODULO_ASSIGN;
precedence right  OP_INSERT, KEY_EACH, KEY_WHILE, KEY_DELETE, KEY_CREATE, KEY_UPDATE;
precedence right  OP_RANGE;
precedence left   OP_QUESTION, OP_COLON;
precedence left   KEY_STRUCT, KEY_BAG, KEY_SEQUENCE;
precedence left   KEY_WHERE, KEY_JOIN, KEY_FOR, KEY_ANY, KEY_ALL, KEY_CLOSE,
                  KEY_ORDER, KEY_BY;
precedence left   KEY_SUM, KEY_AVG, KEY_MIN, KEY_MAX;
precedence left   KEY_COUNT;
precedence left   KEY_UNION, KEY_INTERSECT, KEY_SYMMETRIC, KEY_MINUS, KEY_IN, KEY_COMPARE;
precedence left   KEY_SQRT, KEY_ABS, KEY_SIN, KEY_COS, KEY_EXP, KEY_LOG;
precedence left   KEY_AS, KEY_GROUP;
precedence left   KEY_ELSE;
precedence left   OP_DISJUNCTION;
precedence left   OP_CONJUNCTION;
precedence left   OP_EQUAL, OP_NOT_EQUAL;
precedence left   OP_GREATER, OP_LOWER, OP_GREATER_EQUAL, OP_LOWER_EQUAL;
precedence left   OP_PLUS, OP_MINUS;
precedence left   OP_TIMES, OP_DIVIDE, OP_MODULO;
precedence right  OP_POWER;
precedence right  KEY_EXISTS, KEY_ASC, KEY_DESC;
precedence right  KEY_DEREF, UMINUS, UPLUS, OP_NEGATION, OP_INCREMENT, OP_DECREMENT;
precedence left   LSQUARE, RSQUARE, LROUND, OP_NAVIGATION;
// HIGHEST PRECEDENCE !!!


start with ntGOAL;

ntGOAL    ::=
  ntSTMLIST:s   {: RESULT = s; :}
;

ntSTMLIST ::=
  ntSTM:s               {: RESULT = new LastStatementList(s);      :}
| ntSTM:s1 ntSTMLIST:s2 {: RESULT = new PairStatementList(s1, s2); :}
;

ntSTM ::=
  SEMICOLON
  {: RESULT = new EmptyStatement(); :}
| ntEXP:e SEMICOLON
  {: RESULT = new ExpressionStatement(e); :}
| LCURLY:lk ntSTMLIST:s RCURLY:rk
  {: RESULT = new BlockStatement(s, new KeywordTerminal(lk, lkleft, lkright),
                                    new KeywordTerminal(rk, rkleft, rkright)); :}
| LCURLY:lk RCURLY:rk
  {: RESULT = new EmptyBlockStatement(new KeywordTerminal(lk, lkleft, lkright),
                                      new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_IF:k LROUND ntEXP:e RROUND ntSTM:s
  {: RESULT = new IfStatement(e, s, new KeywordTerminal(k, kleft, kright)); :}
| KEY_IF:lk LROUND ntEXP:e RROUND ntSTM:s1 KEY_ELSE:rk ntSTM:s2
  {: RESULT = new IfElseStatement(e, s1, s2, new KeywordTerminal(lk, lkleft, lkright),
                                             new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_PRINT:k LROUND ntEXP:e RROUND SEMICOLON
  {: RESULT = new PrintStatement(e, new KeywordTerminal(k, kleft, kright)); :}
| KEY_SHOW:k LROUND ntEXP:e RROUND SEMICOLON
  {: RESULT = new ShowStatement(e, new KeywordTerminal(k, kleft, kright)); :}
| KEY_XML:k LROUND ntEXP:e RROUND SEMICOLON
  {: RESULT = new XmlPrintStatement(e, new KeywordTerminal(k, kleft, kright)); :}
| KEY_WHILE:k LROUND ntEXP:e RROUND ntSTM:s
  {: RESULT = new WhileStatement(e, s, new KeywordTerminal(k, kleft, kright)); :}
| KEY_DELETE:k ntEXP:e SEMICOLON
  {: RESULT = new DeleteStatement(e, new KeywordTerminal(k, kleft, kright)); :}
```

```
| KEY_DO:lk ntSTM:s KEY_WHILE:rk LROUND ntEXP:e RROUND SEMICOLON
  {: RESULT = new DoWhileStatement(s, e, new KeywordTerminal(lk, lkleft, lkright),
                                         new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_FOR:lk KEY_EACH:mk ntEXP:e KEY_DO:rk ntSTM:s
  {: RESULT = new ForEachStatement(e, s, new KeywordTerminal(lk, lkleft, lkright),
                                         new KeywordTerminal(mk, mkleft, mkright),
                                         new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_RENAME:lk ntEXP:e1 KEY_TO:rk IDENTIFIER:i SEMICOLON
  {: RESULT = new RenameToStatement(e1, new IdentifierTerminal(i, ileft, iright),
                                        new KeywordTerminal(lk, lkleft, lkright),
                                        new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_UPDATE:lk ntEXP:e1 KEY_TO:rk ntEXP:e2 SEMICOLON
  {: RESULT = new UpdateToStatement(e1, e2, new KeywordTerminal(lk, lkleft, lkright),
                                            new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_INSERT:lk ntEXP:e1 KEY_INTO:rk ntEXP:e2 SEMICOLON
  {: RESULT = new InsertIntoStatement(e1, e2, new KeywordTerminal(lk, lkleft, lkright),
                                              new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_BREAK:k  SEMICOLON
  {: RESULT = new BreakStatement(new KeywordTerminal(k,  kleft,  kright)); :}
| KEY_RETURN:k SEMICOLON
  {: RESULT = new EmptyReturnStatement(new KeywordTerminal(k,  kleft,  kright)); :}
| KEY_RETURN:k ntEXP:e SEMICOLON
  {: RESULT = new ReturnStatement(e, new KeywordTerminal(k,  kleft,  kright)); :}
| KEY_PROCEDURE:k IDENTIFIER:i LROUND RROUND LCURLY ntSTMLIST:s RCURLY
  {: RESULT = new EmptyProcedureDeclarationStatement(s, new KeywordTerminal(k, kleft, kright),
                                                        new IdentifierTerminal(i, ileft, iright)); :}
| KEY_PROCEDURE:k IDENTIFIER:i LROUND ntFORPARLIST:p RROUND LCURLY ntSTMLIST:s RCURLY
  {: RESULT = new ProcedureDeclarationStatement(p, s, new KeywordTerminal(k, kleft, kright),
                                                      new IdentifierTerminal(i, ileft, iright)); :}
| KEY_FUNCTION:k IDENTIFIER:i LROUND RROUND LCURLY ntSTMLIST:s RCURLY
  {: RESULT = new EmptyProcedureDeclarationStatement(s, new KeywordTerminal(k, kleft, kright),
                                                        new IdentifierTerminal(i, ileft, iright)); :}
| KEY_FUNCTION:k IDENTIFIER:i LROUND ntFORPARLIST:p RROUND LCURLY ntSTMLIST:s RCURLY
  {: RESULT = new ProcedureDeclarationStatement(p, s, new KeywordTerminal(k, kleft, kright),
                                                      new IdentifierTerminal(i, ileft, iright)); :}
;

ntFORPARLIST ::=
  IDENTIFIER:i
  {: RESULT = new LastFormalParametersList(new IdentifierTerminal(i, ileft, iright)); :}
| IDENTIFIER:i OP_COMMA ntFORPARLIST:p
  {: RESULT = new PairFormalParametersList(new IdentifierTerminal(i, ileft, iright), p); :}
;

ntEXP ::=
  ntSINGLEEXP:s              {: RESULT = s; :}
| ntTEREXP:t                 {: RESULT = t; :}
| ntBINEXP:b                 {: RESULT = b; :}
| ntUNAEXP:u                 {: RESULT = u; :}
| LROUND ntEXP:e RROUND      {: RESULT = e; :}
;

ntSINGLEEXP ::=
  IDENTIFIER:i set_proc_flag reset_flag
  {: RESULT = new IdentifierExpression(new IdentifierTerminal(i, ileft, iright)); :}
| LIT_INTEGER:l
  {: RESULT = new IntegerExpression(new LiteralTerminal(l, lleft, lright, new Integer(l))); :}
| LIT_DOUBLE:l
  {: RESULT = new DoubleExpression (new LiteralTerminal(l, lleft, lright, new Double(l))); :}
| LIT_STRING:l
  {: RESULT = new StringExpression (new LiteralTerminal("\"" + l + "\"", lleft, lright, new String(l))); :}
| LIT_BOOLEAN:l
  {: RESULT = new BooleanExpression(new LiteralTerminal(l, lleft, lright, new Boolean(l))); :}
| OP_TIMES:o
  {: RESULT = new AsteriskExpression(new OperatorTerminal(o, oleft, oright)); :}
| KEY_BAG:k set_bag_flag LROUND RROUND reset_flag
```

```
  {: RESULT = new EmptyBagExpression(new KeywordTerminal(k, kleft, kright)); :}
| KEY_STRUCT:k set_str_flag LROUND RROUND reset_flag
  {: RESULT = new EmptyStructExpression(new KeywordTerminal(k, kleft, kright)); :}
| KEY_SEQUENCE:k set_seq_flag LROUND RROUND reset_flag
  {: RESULT = new EmptySequenceExpression(new KeywordTerminal(k, kleft, kright)); :}
| IDENTIFIER:i set_proc_flag LROUND RROUND reset_flag
  {: RESULT = new EmptyProcedureCallExpression(new IdentifierTerminal(i, ileft, iright)); :}
;

ntUNAEXP ::=
  ntPAREXP:p     {: RESULT = p; :}
| ntNONPAREXP:n  {: RESULT = n; :}
;

ntBINEXP ::=
  ntALGEXP:a     {: RESULT = a; :}
| ntNONALGEXP:n  {: RESULT = n; :}
;

ntTEREXP ::=
  ntEXP:l OP_QUESTION:lo ntEXP:m OP_COLON:ro ntEXP:r
  {: RESULT = new ConditionExpression(l, m, r, new OperatorTerminal(lo, loleft, loright),
                                               new OperatorTerminal(ro, roleft, roright));    :}
| ntEXP:l LSQUARE:lo ntEXP:m OP_RANGE:mo ntEXP:r RSQUARE:ro
  {: RESULT = new RangeExpression(l, m, r, new OperatorTerminal(lo, loleft, loright),
                                           new OperatorTerminal(mo, moleft, moright),
                                           new OperatorTerminal(ro, roleft, roright)); :}
;

ntALGEXP ::=
  ntEXP:l OP_PLUS:o ntEXP:r
  {: RESULT = new PlusExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_MINUS:o ntEXP:r
{: RESULT = new MinusExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_TIMES:o ntEXP:r
{: RESULT = new TimesExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_DIVIDE:o ntEXP:r
{: RESULT = new DivideExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_MODULO:o ntEXP:r
{: RESULT = new ModuloExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_POWER:o ntEXP:r
{: RESULT = new PowerExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_EQUAL:o ntEXP:r
{: RESULT = new EqualExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_NOT_EQUAL:o ntEXP:r
{: RESULT = new NotEqualExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_GREATER:o ntEXP:r
{: RESULT = new GreaterExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_LOWER:o ntEXP:r
{: RESULT = new LowerExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_GREATER_EQUAL:o ntEXP:r
{: RESULT = new GreaterEqualExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_LOWER_EQUAL:o ntEXP:r
{: RESULT = new LowerEqualExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_CONJUNCTION:o ntEXP:r
{: RESULT = new ConjunctionExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_DISJUNCTION:o ntEXP:r
{: RESULT = new DisjunctionExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l KEY_IN:k ntEXP:r
{: RESULT = new InclusionExpression(l, r, new KeywordTerminal (k, kleft, kright)); :}
| ntEXP:l KEY_UNION:k ntEXP:r
{: RESULT = new UnionExpression(l, r, new KeywordTerminal (k, kleft, kright)); :}
| ntEXP:l KEY_INTERSECT:k ntEXP:r
{: RESULT = new IntersectionExpression(l, r, new KeywordTerminal (k, kleft, kright)); :}
| ntEXP:l KEY_MINUS:k ntEXP:r
{: RESULT = new DifferenceExpression(l, r, new KeywordTerminal (k, kleft, kright)); :}
```

```
| ntEXP:l KEY_SYMMETRIC:lk KEY_MINUS:rk ntEXP:r
{: RESULT = new SymmetricDifferenceExpression(l, r, new KeywordTerminal (lk, lkleft, lkright),
                                                    new KeywordTerminal (rk, rkleft, rkright)); :}
| ntEXP:l OP_INSERT:o ntEXP:r
{: RESULT = new InsertExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_ASSIGN:o ntEXP:r
{: RESULT = new AssignExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_PLUS_ASSIGN:o ntEXP:r
{: RESULT = new PlusAssignExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_MINUS_ASSIGN:o ntEXP:r
{: RESULT = new MinusAssignExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_TIMES_ASSIGN:o ntEXP:r
{: RESULT = new TimesAssignExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_DIVIDE_ASSIGN:o ntEXP:r
{: RESULT = new DivideAssignExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_MODULO_ASSIGN:o ntEXP:r
{: RESULT = new ModuloAssignExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l OP_COMMA:o ntEXP:r {:
    switch(topFlag()) {
      case parser.FLAG_PROCEDURE:
        RESULT = new ListExpression (l, r, new OperatorTerminal(o, oleft, oright));
      break;
      case parser.FLAG_SEQUENCE:
      case parser.FLAG_BAG:
        RESULT = new UnionExpression(l, r, new KeywordTerminal (o, oleft, oright));
      break;
      case parser.FLAG_STRUCT:
      default:
        RESULT = new CommaExpression(l, r, new OperatorTerminal(o, oleft, oright));
    }
  :}
| ntEXP:l KEY_COMPARE:k ntEXP:r
  {: RESULT = new CompareExpression(l, r, new KeywordTerminal (k, kleft, kright)); :}
| ntEXP:l LSQUARE:lo ntEXP:r RSQUARE:ro
  {: RESULT = new BRangeExpression(l, r, new OperatorTerminal(lo, loleft, loright),
                                         new OperatorTerminal(ro, roleft, roright)); :}
;

ntNONALGEXP ::=
  ntEXP:l OP_NAVIGATION:o ntEXP:r
  {: RESULT = new NavigationExpression(l, r, new OperatorTerminal(o, oleft, oright)); :}
| ntEXP:l KEY_WHERE:k ntEXP:r
  {: RESULT = new SelectionExpression(l, r, new KeywordTerminal(k, kleft, kright)); :}
| ntEXP:l KEY_JOIN:k ntEXP:r
  {: RESULT = new DependentJoinExpression(l, r, new KeywordTerminal(k, kleft, kright)); :}
| ntEXP:l KEY_FOR:lk KEY_ANY:rk ntEXP:r
  {: RESULT = new ForAnyExpression(l, r, new KeywordTerminal(lk, lkleft, lkright),
                                         new KeywordTerminal(rk, rkleft, rkright)); :}
| ntEXP:l KEY_FOR:lk KEY_ALL:rk ntEXP:r
  {: RESULT = new ForAllExpression(l, r, new KeywordTerminal(lk, lkleft, lkright),
                                         new KeywordTerminal(rk, rkleft, rkright)); :}
| ntEXP:l KEY_CLOSE:lk KEY_BY:rk ntEXP:r
  {: RESULT = new TransitiveClosureExpression(l, r, new KeywordTerminal (lk, lkleft, lkright),
                                                    new KeywordTerminal (rk, rkleft, rkright)); :}
| ntEXP:l KEY_ORDER:lk KEY_BY:rk ntEXP:r
  {: RESULT = new SortingExpression(l, r, new KeywordTerminal(lk, lkleft, lkright),
                                          new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_FOR:lk KEY_ALL:rk ntEXP:l LROUND ntEXP:r RROUND
  {: RESULT = new ForAllExpression(l, r, new KeywordTerminal(lk, lkleft, lkright),
                                         new KeywordTerminal(rk, rkleft, rkright)); :}
| KEY_FOR:lk KEY_ANY:rk ntEXP:l LROUND ntEXP:r RROUND
  {: RESULT = new ForAnyExpression(l, r, new KeywordTerminal(lk, lkleft, lkright),
                                         new KeywordTerminal(rk, rkleft, rkright)); :}
;

ntPAREXP ::=
```

```
  ntEXP:l KEY_AS:k IDENTIFIER:i
  {: RESULT = new AsExpression(l, new KeywordTerminal(k,  kleft,  kright),
                                  new IdentifierTerminal(i, ileft, iright)); :}
| ntEXP:l KEY_GROUP:lk KEY_AS:rk IDENTIFIER:i
  {: RESULT = new GroupAsExpression(l, new KeywordTerminal(lk, lkleft, lkright),
                                       new KeywordTerminal(rk, rkleft, rkright),
                                       new IdentifierTerminal(i, ileft, iright)); :}
// conflict with procedures
//| IDENTIFIER:i LROUND ntEXP:l RROUND
//  {: RESULT = new EmptyAsExpression(l, new IdentifierTerminal(i, ileft, iright)); :}
| IDENTIFIER:i LCURLY ntEXP:l RCURLY
  {: RESULT = new EmptyGroupAsExpression(l, new IdentifierTerminal(i, ileft, iright)); :}
;

ntNONPAREXP ::=
  OP_NEGATION:o ntEXP:r
  {: RESULT = new NegationExpression(r, new OperatorTerminal(o, oleft, oright)); :}
| OP_MINUS:o ntEXP:r
  {: RESULT = new UMinusExpression(r, new OperatorTerminal(o, oleft, oright)); :} %prec UMINUS
| OP_PLUS:o ntEXP:r
  {: RESULT = new UPlusExpression(r, new OperatorTerminal(o, oleft, oright)); :} %prec UPLUS
| OP_INCREMENT:o ntEXP:r
  {: RESULT = new PreIncrementExpression(r, new OperatorTerminal(o, oleft, oright)); :}
| OP_DECREMENT:o ntEXP:r
  {: RESULT = new PreDecrementExpression(r, new OperatorTerminal(o, oleft, oright)); :}
| KEY_DEREF:k LROUND ntEXP:r RROUND
  {: RESULT = new DereferenceExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_SQRT:k LROUND ntEXP:r RROUND
  {: RESULT = new SquareRootExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_ABS:k LROUND ntEXP:r RROUND
  {: RESULT = new AbsoluteValueExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_SIN:k LROUND ntEXP:r RROUND
  {: RESULT = new SineExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_COS:k LROUND ntEXP:r RROUND
  {: RESULT = new CosineExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_EXP:k LROUND ntEXP:r RROUND
  {: RESULT = new ExponentExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_LOG:k LROUND ntEXP:r RROUND
  {: RESULT = new LogarithmExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_EXISTS:k ntEXP:r
  {: RESULT = new ExistsExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_UNIQUE:k LROUND ntEXP:r RROUND
  {: RESULT = new UniqueExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_COUNT:k LROUND ntEXP:r RROUND
  {: RESULT = new CountExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_SUM:k LROUND ntEXP:r RROUND
  {: RESULT = new SumExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_AVG:k LROUND ntEXP:r RROUND
  {: RESULT = new AverageExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_MIN:k LROUND ntEXP:r RROUND
  {: RESULT = new MinimumExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_MAX:k LROUND ntEXP:r RROUND
  {: RESULT = new MaximumExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_SIGN:k LROUND ntEXP:r RROUND
  {: RESULT = new SignumExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_ASC:k LROUND ntEXP:r RROUND
  {: RESULT = new AscendingExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_DESC:k LROUND ntEXP:r RROUND
  {: RESULT = new DescendingExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_TOINTEGER:k LROUND ntEXP:r RROUND
  {: RESULT = new ToIntegerExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_TODOUBLE:k LROUND ntEXP:r RROUND
  {: RESULT = new ToDoubleExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_TOSTRING:k LROUND ntEXP:r RROUND
  {: RESULT = new ToStringExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_TOSTRUCT:k LROUND ntEXP:r RROUND
```

```
  {: RESULT = new ToStructExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_TOBAG:k LROUND ntEXP:r RROUND
  {: RESULT = new ToBagExpression(r, new KeywordTerminal (k, kleft, kright)); :}
| KEY_TOSEQUENCE:k LROUND ntEXP:r RROUND
  {: RESULT = new ToSequenceExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_TOSINGLE:k LROUND ntEXP:r RROUND
  {: RESULT = new ToSingleExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_CREATE:k ntEXP:r
  {: RESULT = new CreateExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_CREATE:lk KEY_PERMANENT:rk ntEXP:r
  {: RESULT = new CreatePermanentExpression(r, new KeywordTerminal (lk, lkleft, lkright),
                                               new KeywordTerminal (rk, rkleft, rkright)); :}
| KEY_STRUCT:k set_str_flag LROUND ntEXP:r RROUND reset_flag
  {: RESULT = new StructExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_BAG:k set_bag_flag LROUND ntEXP:r RROUND reset_flag
  {: RESULT = new BagExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| KEY_SEQUENCE:k set_seq_flag LROUND ntEXP:r RROUND reset_flag
  {: RESULT = new SequenceExpression(r, new KeywordTerminal(k, kleft, kright)); :}
| IDENTIFIER:i set_proc_flag LROUND ntEXP:p RROUND reset_flag
  {: RESULT = new ProcedureCallExpression(p, new IdentifierTerminal(i, ileft, iright)); :}
;

set_bag_flag  ::= {: pushFlag(parser.FLAG_BAG);       :};
set_str_flag  ::= {: pushFlag(parser.FLAG_STRUCT);    :};
set_seq_flag  ::= {: pushFlag(parser.FLAG_SEQUENCE);  :};
set_proc_flag ::= {: pushFlag(parser.FLAG_PROCEDURE); :};
reset_flag    ::= {: popFlag();                       :};
```

Dodatek F

Przykładowa baza danych

Skrypt tworzący przykładową baze danych w języku Yaql został przedstawiony poniżej:

```
/* EMPLOYEE - DEPARTMENT */

create (
  EMPLOYEE{NAME{"Brown"}, SALARY{3500}, JOB{"clerk"},
           PREVIOUS_JOB{COMPANY{"CDC"}, FROM{1979}, TILL{1982}}},
  EMPLOYEE{NAME{"Smith"}, SALARY{3800}, JOB{"clerk"}},
  EMPLOYEE{NAME{"Casey"}, SALARY{410},  JOB{"programmer"},
           PREVIOUS_JOB{COMPANY{"IBM"}, FROM{1982}, TILL{1988}}},
  EMPLOYEE{NAME{"Lewis"}, SALARY{3800}, JOB{"programmer"}},
  EMPLOYEE{NAME{"Jones"}, SALARY{700},  JOB{"engineer"},
           PREVIOUS_JOB{COMPANY{"IBM"}, FROM{1983}, TILL{1985}},
           PREVIOUS_JOB{COMPANY{"NCR"}, FROM{1985}, TILL{1988}}}
);

create (
  DEPARTMENT {NAME{"Sales"},
    LOCATION{"Rome"}, LOCATION{"Paris"},
    EMPLOYS{EMPLOYEE where NAME == "Jones"},
    EMPLOYS{EMPLOYEE where NAME == "Lewis"}},
  DEPARTMENT { NAME{"Service"},
    LOCATION{"Rome"}, LOCATION{"Tokyo"}, LOCATION{"London"},
    EMPLOYS{EMPLOYEE where NAME == "Brown"},
    EMPLOYS{EMPLOYEE where NAME == "Smith"},
    EMPLOYS{EMPLOYEE where NAME == "Casey"}}
);

for each DEPARTMENT as D do
  for each D.EMPLOYS.EMPLOYEE as E do
    insert create WORKS_IN{D} into E;


/* PART */

create (
  PART{NAME{"disk drive"}},
  PART{NAME{"chassis"},      WEIGHT{250}},
  PART{NAME{"motor"}},
  PART{NAME{"head arm"}},
  PART{NAME{"bolt M2 x 10"}, WEIGHT{1}},
  PART{NAME{"bolt M3 x 20"}, WEIGHT{3}},
  PART{NAME{"connector"}},
  PART{NAME{"corpus"}},
  PART{NAME{"rotor"},        WEIGHT{30}},
  PART{NAME{"head"},         WEIGHT{5}},
  PART{NAME{"arm"},          WEIGHT{25}},
```

```
  PART{NAME{"binding"},      WEIGHT{20}},
  PART{NAME{"pin"},          WEIGHT{1}},
  PART{NAME{"cylinder"},     WEIGHT{30}},
  PART{NAME{"cover"},        WEIGHT{10}}
);

insert create (
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "chassis"}},
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "motor"}},
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "head arm"}},
      CONSISTS_OF{QUANTITY{4}, DETAIL{PART where NAME == "bolt M2 x 10"}},
      CONSISTS_OF{QUANTITY{6}, DETAIL{PART where NAME == "bolt M3 x 20"}},
      CONSISTS_OF{QUANTITY{2}, DETAIL{PART where NAME == "connector"}})
into  PART where NAME == "disk drive";

insert create (
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "corpus"}},
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "rotor"}},
      CONSISTS_OF{QUANTITY{3}, DETAIL{PART where NAME == "bolt M2 x 10"}})
into  PART where NAME == "motor";

insert create (
      CONSISTS_OF{QUANTITY{9}, DETAIL{PART where NAME == "head"}},
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "arm"}})
into  PART where NAME == "head arm";

insert create (
      CONSISTS_OF{QUANTITY{1},  DETAIL{PART where NAME == "binding"}},
      CONSISTS_OF{QUANTITY{2},  DETAIL{PART where NAME == "bolt M2 x 10"}},
      CONSISTS_OF{QUANTITY{20}, DETAIL{PART where NAME == "pin"}})
into  PART where NAME == "connector";

insert create (
      CONSISTS_OF{QUANTITY{1}, DETAIL{PART where NAME == "cylinder"}},
      CONSISTS_OF{QUANTITY{2}, DETAIL{PART where NAME == "cover"}})
into  PART where NAME == "corpus";

for each (PART where exists CONSISTS_OF) as A do
  for each (A.CONSISTS_OF.DETAIL.PART) as B do
    insert create IS_DETAIL_OF{A} into B;
```

Bibliografia

[1] A. Jodłowski, *Dynamic Object Roles in Conceptual Modelling and Databases*, praca doktorska, Instytut Podstaw Informatyki, Polska Akademia Nauk, 2002.

[2] A. Silberschatz, P.B. Galvin, *Podstawy systemów operacyjnych*, WNT, 2000.

[3] A.V. Aho, R. Sethi, J.D. Ullman, *Kompilatory - Reguły, metody i narzędzia*, WNT, 2002.

[4] A.W. Appel, *Modern compiler implementation in Java*, Cambridge University Press, 2002.

[5] C.J. Date, *Wprowadzenie do systemów baz danych*, WNT, 2000.

[6] D.A. Watt, D.F. Brown, *Programming language processors in Java, Compilers and Interpreters*, Prentice Hall, 2000.

[7] H. Garcia–Molina, J.D. Ullman, J. Widom, *Implementacja systemów baz danych*, WNT, 2003.

[8] J. Płodzień, *Optimization Methods in Object Query Languages*, praca doktorska, Instytut Podstaw Informatyki, Polska Akademia Nauk, 2000.

[9] K. Subieta, *Object-Based Virtual Memory for PC-s*, ICS PAS report 696, Warsaw, 1990.

[10] K. Subieta, M. Missala, K. Anacki, *The LOQIS System*, ICS PAS report 695, Warsaw, 1990.

[11] K. Subieta, *Virtual Persisten Object Store Package (Sun Version)*, 1991.

[12] K. Subieta, *Dynamic Memory Package*, 1991.

[13] K. Subieta, Y. Kambayashi, J. Leszczyłowski, *Procedures in Object-Oriented Query Languages*, Proc. 21-st VLDB Conf., Zurich, 1995.

[14] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt, *A Stack–Based Approach to Query Languages*, ICS PAS, report 738, Warszawa, 1993.

[15] K. Subieta, *Języki i środowiska programowania baz danych*, materiały do wykładu prowadzonego w Polsko–Japońskiej Wyższej Szkole Technik Komputerowych na studiach magisterskich o specjalizacji „Bazy danych i inżynieria oprogramowania” w roku akademickim 2001/2002.

[16] K. Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, (w przygotowaniu) Warszawa, 2003.

[17] K. Subieta, *Słownik terminów z zakresu obiektowości*. Akademicka Oficyna Wydawnicza PLJ, 1999.

[18] M. Lentner, *Oracle*9i. *Kompletny podręcznik użytkownika*. Wydawnictwo Polsko–Japońskiej Wyższej Szkoły Technik Komputerowych, 2003.

[19] M. Michalak, B. Bogucki, *Implementacja SBQL dla ooPortal*, praca magisterska (w przygotowaniu), Polsko–Japońska Wyższa Szkoła Technik Komputerowych, 2003.

[20] P. Habela, *Metamodel for Object–Oriented Database Management Systems*, praca doktorska, Polsko–Japońska Wyższa Szkoła Technik Komputerowych, 2002.

[21] T. Pieciukiewicz, R. Hryniów, *Jezyk zapytań dla XML oparty na podejściu stosowym*, praca magisterska, Polsko–Japońska Wyższa Szkoła Technik Komputerowych, 2002.