

**POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK
KOMPUTEROWYCH**

PRACA MAGISTERSKA

Nr

Optymalizator zapytań obiektowego języka SBQL

Student

Tomasz Łazarczyk

Nr albumu

1538

Promotor

prof. Dr hab. Kazimierz Subieta

Specjalność

Inżynieria Oprogramowania i Baz Danych

Katedra

Systemów Informacyjnych

Data zatwierdzenia tematu

YY-YY-YYYY

Data zakończenia pracy

XX-XX-XXXX

Podpis promotora pracy

.....

Podpis kierownika katedry

.....

Streszczenie

Autor pracy jako cel postawił sobie zaimplementowanie optymalizatora zapytań języka SBQL. Wybór powyższego celu zbiegł się w czasie z zakończeniem prac nad wspomnianym w streszczeniu prototypem języka. Istniała potrzeba jego wzbogacenia, tak aby w szybki sposób można było odpytywać repozytorium Office Object Portal.

Prototyp ten nie posiadał wbudowanych metod optymalizacji. Istotnym więc było wyposażenie go w wybrane rozwiązania optymalizacyjne, tak aby ewaluacja zapytań odbywała się w akceptowalnym czasie wykonania.

Autor przystępując do realizacji przyjął następujące założenia dotyczące optymalizatora:

- zaimplementować metodę opartą na przepisywaniu niezależnych pod-zapytań (ang. *rewriting independent subqueries*)
- zintegrować optymalizator z istniejącą strukturą prototypu SBQL'a
- optymalizować wszelkie konstrukcje językowe (m. in. różne operatory)

Wybraną przez autora metodę optymalizacji jak i budowę samego optymalizatora opracował Kazimierz Subieta i opisał w książce “Teoria i konstrukcja obiektowych języków zapytań” [Subieta 2004].

W związku z wymogiem integracji z prototypem SBQL'a optymalizator został napisany w języku Java.

Wynikiem pracy jest w pełni działająca optymalizacja zapytań języka SBQL. Zapytania, zadawane do repozytorium Office Object Portal są teraz optymalizowane, co w przypadku systemu inteligentnego zarządzania wiedzą nie pozostaje bez znaczenia.

Przejrzysta budowa oraz logiczne działanie optymalizatora zapytań jest dowodem na to, że optymalizacja języka mającego mocne podstawy teoretyczne, wcale nie musi być zbiorem reguł matematycznych.

Spis treści

1. Wstęp
2. Stan sztuki
2.1 ICONS
2.2 OfficeObjects® Portal
2.3 Interpreter Języka Zapytań SBQL dla Office Object Portal
2.4 Co to jest optymalizacja zapytań ?
2.4.1 SQL
2.4.2 XQuery
2.4.3 Optimization Methods in Object Query Languages
2.5 Metody optymalizacji
2.5.1 Metoda niezależnych pod-zapytań – drugi sposób
2.5.2 Metody niskopoziomowe
2.5.2.1 Indeksowanie
2.5.2.2 Bezpośrednia nawigacja
2.5.3 Metody optymalizacji wyrażeń ścieżkowych
2.5.3.1 „Wyciąganie” wspólnych wyrażeń ścieżkowych
2.5.3.2 Relacje wsparcia dostępu
2.5.4 Metody optymalizacji zapytań wywołujących perspektywy
2.5.4.1 Usuwanie zbędnych pomocniczych nazw
2.5.4.2 Zastępywanie złączenia operatorem projekcji
3. Język SBQL (Stack-Based Query Language)
3.1 Składnia SBQL
3.2 Stosy
3.2.1 Stos środowisk (ENVS)
3.2.1.1 Wiązanie (ang. <i>binding</i>)
3.2.1.2 Tworzenie nowej sekcji na stosie środowisk
3.2.1.3 Funkcja <i>nested</i>
3.2.2 Stos rezultatów (QRES)
3.3 Rezultaty zapytań
3.4 Ewaluacja zapytań
4. Optymalizacja zapytań języka SBQL
4.1 Metoda niezależnych pod-zapytań – pierwszy sposób
4.1.1 Omówienie
4.1.2 Przykład
4.2 Budowa optymalizatora
4.2.1 Meta-baza
4.2.2 Stosy statyczne
4.2.3 Drzewo syntaktyczne
4.3 Analiza statyczna zapytania
4.3.1 Algorytm

4.3.2	Funkcja <i>static_nested</i>
4.3.3	Funkcja <i>static_eval</i>
4.3.4	Przykład
4.3.5	Wyznaczanie niezależnych pod-zapytań
5.	Implementacja
5.1	Metabaza
5.2	Drzewo
5.3	Optymalizator
5.3.1	Sygnatury
5.3.2	Stosy
5.4	Przykłady
6.	Podsumowanie
7.	Literatura
Dodatek A

1. Wstęp

Niniejsza praca przedstawia implementację optymalizatora zapytań języka SBQL, opartego na teorii stosów. Stanowi ona integralną część implementacji języka SBQL (co było tematem pracy magisterskiej autorstwa Bogdana Boguckiego i Marcina Michalaka – patrz rozdział 2.3) stworzonej dla Office Object Portal – Systemu Inteligentnego Zarządzania Zawartością, tworzonego w ramach piątego ramowego projektu (ICONS).

“Wstęp” przedstawia ogólne założenia oraz cel pracy.

Rozdział “Stan sztuki” ukazuje ogólne spojrzenie na kwestię optymalizacji zapytań na przykładzie kilku wybranych języków.

W rozdziale dotyczącym języka SBQL zostaje on dokładnie przedstawiony wraz z zasadą jego działania.

Rozdział „Metody optymalizacyjne” dokonuje przeglądu metod optymalizacyjnych polegających na przepisywaniu.

„Optymalizacja zapytań języka SBQL” stanowi centralny punkt pracy, który to opisuje nie tylko budowę, działanie optymalizatora, ale również skupia się na metodzie niezależnych pod-zapytań i statycznej analizie.

Przedostatni rozdział dotyczy rozwiązań implementacyjnych tj. przedstawia budowę struktur danych (metabaza, stosy i optymalizator). Dodatkowo znajdują się tutaj zrzuty ekranów oraz przykłady pokazujące efekty optymalizacji.

W ostatnim rozdziale znajduje się podsumowanie pracy. Autor opisuje tam trudności jakie autor, wnioski wypływające z implementacji oraz możliwości dalszej rozbudowy optymalizatora zapytań.

2. Stan sztuki

Zadaniem poniższego rozdziału jest przedstawienie czytelnikowi pojęcia optymalizacji na przykładzie części najbardziej popularnych rozwiązań dotyczących optymalizacji zapytań ponieważ obszar związany z optymalizacją zapytań jest tak olbrzymi, że stworzenie pełnego ich przeglądu jest niemożliwe.

Dodatkowo 2.4.3 autor krótko przedstawia pracę doktorską poświęconą tematowi optymalizacji.

Na początku jednak zostanie krótko opisany projekt ICONS, system OfficeObjects® Portal, praca magisterska p. Bogdana Boguckiego i Marcina Michalaka, jako że praca ta jest ich składową.

2.1 ICONS

Projekt ICONS (Intelligent CONTENT Management System) ma za zadanie połączenie zaawansowanych rezultatów badawczych, technologii i standardów w dziedzinie reprezentacji wiedzy i istniejących źródeł informacji w jednorodną i internetową architekturę.

Wypracowane metody będą stanowiły podstawy do budowy prototypu systemu zarządzania wiedzą i zawartością multimedialną. Integracja i rozszerzenie rezultatów z dziedzin sztucznej inteligencji oraz baz danych połączone z zaawansowanymi cechami nowo powstających technologii informatycznych da w rezultacie nowatorską architekturę systemu ICONS.

Badania w zakresie reprezentacji wiedzy będą pokrywały takie paradygmaty jak logika (disjunctive Datalog), sieci semantyczne (semantyczne modele UML oraz standard RDF) oraz wiedzę proceduralną o procesach działalności opartą o grafy skierowane zgodnie z zaleceniami koalicji (WfMC).

Prototyp systemu ma zarządzać multimedialnym repozytorium zawartości, przechowywać złożone obiekty informacyjne i reprezentacje zewnętrznych informacji znajdujących się w rozproszonych bazach danych.

ICONS ma zarządzać opartym na XML, multimedialnym repozytorium zawartości, przechowywać złożone obiekty informacyjne i reprezentacje (proxy) zewnętrznych informacji znajdujących się w heterogenicznych bazach, danych generowanych na wyjściach innych systemów przetwarzających, stronach Web oraz odpowiednich ontologii dziedzinowych. Portal Zarządzania Wiedzą o Projektach Funduszy Strukturalnych będzie dostępny poprzez Internet i będzie najlepszą demonstracją wyników projektu ICONS.

[ICONS_1, ICONS_2]

2.2 OfficeObjects® Portal

OfficeObjects® Portal jest korporacyjnym portalem, dzięki któremu pracownicy, management oraz partnerzy handlowi i klienci w kontrolowany sposób w pełni korzystają z zasobów wiedzy firmy przez 24h na dobę. Wystarczy tylko dostęp do sieci Internet i komputer, aby możliwym było korzystanie z firmowych "skarbów" we wszystkich oddziałach rozproszonych w różnych częściach kraju, czy świata.

Z korporacyjnego punktu widzenia kluczowe znaczenia ma elastyczność w dostosowaniu portalu do specyfiki firmy, jej charakteru (produkcja przemysłowa, handel, usługi) oraz wielkości (wysoka skalowalność).

Poniżej wypunktowano główne cechy portalu:

- pełne zintegrowanie wszystkich działających w firmie systemów informatycznych

i powiązanie ze sobą rozdzielonych dotychczas zasobów danych za pośrednictwem Internetu

- pobieranie aktualnych danych z zewnętrznych źródeł, jak np. notowania giełdowe czy informacje z agencji prasowych i ich integrowanie z zasobami wiedzy firmy
- zarządzanie obiegiem dokumentów i wewnętrzną komunikacją w firmie (dzięki pośrednictwu wewnętrznej sieci - *intranetu*)
- porządkowanie i kategoryzacja danych wg ustalonych i precyzyjnych kryteriów
- wymiana na bieżąco aktualizowanych informacji ze światem (z klientem, partnerem handlowym, mediami)
- nawiązywanie nowych kontaktów handlowych (z możliwością udzielenia natychmiastowej odpowiedzi)
- dokonywanie za pośrednictwem Internetu transakcji handlowych (zamówienia *on line*)
- zapewnienie dobrego wizerunku firmy na zewnątrz oraz wśród pracowników (działania z zakresu Public Relations)
- dynamiczne wsparcie wszystkich działań promocyjnych i marketingowych firmy

2.3 Interpreter Języka Zapytań SBQL dla Office Object Portal

Implementacja języka SBQL była tematem pracy magisterskiej autorstwa p. Bogdana Boguckiego i Marcina Michalaka. Zostaną tu opisane jej podstawowe założenia, zastosowane narzędzia i techniki implementacyjne.

Tematem pracy było zaimplementowanie prototypu języka opartego o podejście stosowe dla wspomnianego w poprzednim rozdziale repozytorium Office Object Portal. Celem nadrzędnym było zaopatrzenie tego repozytorium (do tej pory nie posiadało ono sprawnego języka zapytań) w język zapytań, który jest:

- łatwo rozbudowywalny,
- przystosowany do modelu obiektowego,
- prosty w implementacji,
- łatwy w użytkowaniu,
- w pełni implementowalny

oraz ma mocne podstawy teoretyczne.

Następujące narzędzia zostały użyte w implementacji:

- Java – język programowania użyty do napisania języka zapytań; ponieważ część ooPortal została napisana właśnie w javie to oczywistym było wybranie javy jako narzędzia; pozwoliło to na prostą integrację SBQL'a z ooPortal
- JFlex – generator skanerów leksykalnych w javie; format specyfikacji leksykalnej został oparty o specyfikację jaką przyjmowały generatory skanerów dla C/C++ (LEX orz FLEX)
- CUP – generator parserów LALR napisany w języku java; specyfikacja gramatyki jest oparta na specyfikacji z generatorów YACC oraz Bison; parsery wygenerowane przez CUP'a łatwo się integruje ze skanerami leksykalnymi wygenerowanymi przez program JFlex

Autorzy tej pracy wykonali postawione przed sobą zadanie. Ponadto magistranci stworzyli interfejs graficzny pozwalający zadawać zapytania z poziomu strony WWW. Stworzony został także moduł pozwalający na użycie SBQL dla ooPortal przez Structural Knowledge Graph Manager (SKGM - modułu pozwalającego na przeglądanie zawartości

repozytorium w sposób graficzny, nawigując po grafie). Poprzez swoją pracę pokazali, że język mający mocne podstawy teoretyczne doskonale się sprawdza w przypadku dużych systemów.

Zadaniem na przyszłość jest wzbogacenie języka o równania stało-punktowe, kontrolę typów, optymalizację poprzez przepisywanie oraz indeksy.

2.4 Co to jest optymalizacja zapytań ?

Język zapytań jest to narzędzie występujące w systemach zarządzania bazami danych (SZB) służące do wyszukiwania informacji wg zadanego przez użytkownika zapytania.

Współczesne SZB posiadają wiele wspaniałych cech, które potrafią uprzyjemnić życie użytkownikowi, jednakże najbardziej pożądaną cechą jest szybkość. Priorytetem jest, aby ewaluacja zapytań dokonywała się jak najbardziej efektywnie, a co za tym idzie szybko. Końcowego użytkownika SZB nie interesuje jak jest zbudowany (czarna skrzynka); chce, aby jego zapytanie wykonało się w akceptowalnym czasie.

Na szybkość wykonywania zapytań mają wpływ oczywiście takie czynniki jak szybkość dostępu do dysku, przepustowość łącza internetowego (dla baz rozproszonych).

Jednak to optymalizator jest fundamentalną składową dowolnego języka zapytań. Bez niego bezpośrednia ewaluacja zapytań doprowadziłaby do niewyobrażalnie długich czasów wykonania i dużej konsumpcji pamięci.

Oto prosty przykład zaczerpnięty z [Subieta 2002].

Proste zapytanie w SQL (podaj nazwiska dostawców dostarczających części o nazwie "zawór"):

```
select Dostawca.nazwisko from Dostawca, Produkt, DP  
where Dostawca.NrDost = DP.NrDost and DP.NrProd = Produkt.NrProd  
and Produkt.nazwa = "zawór"
```

wymaga wykonania produktu kartezyjskiego relacji wymienionych w klauzuli **from**. Przyjmując (dość rozsądnie) 10000 dostawców, 10000 produktów, 100000 krotek w relacji DP (wiążącej dostawców z produktami) i średnio 100 bajtów w każdej krotce tych relacji, produkt kartezyjski miałby 10^{13} elementów i zajmowałby 10^{15} bajtów, czyli ponad 930000 GB. Jeżeli sprawdzenie warunku w klauzuli **where** dla pojedynczej krotki trwałoby jedną tysięczną sekundy, to wyselekcjonowanie z produktu kartezyjskiego właściwych krotek trwałoby 10^{10} sekund, czyli 317 lat.

Wykonanie takiego zapytania bez pomocy optymalizatora i operowanie na tak dużej ilości wyników pośrednich w pamięci jest praktycznie niemożliwe. Natomiast wykorzystując metody optymalizacyjne wykonanie potrwałoby zaledwie kilka sekund.

Dzielią się one na następujące rodzaje:

- **metody oparte na przepisaniu** (ang. *rewriting*) – metody te polegają na przekształceniu zapytania (lub jego części) w równoważne zapytanie o mniejszym czasie wykonania
- **zastosowanie specjalnych struktur danych lub organizacji danych** – mówiąc szczegółowiej indeksów, tablic wskaźników, kodowania mieszającego
- **pomijanie podczas ewaluacji specyficznych fragmentów zapytań** – mowa tutaj o takich fragmentach zapytań, które nie mają wpływu na końcowy wynik
- **zapamiętywanie wyników poprzednio obliczonych zapytań** – zapytania, które są często wykonywane są „materializowane” (zapamiętywane) i przez to już później nie wykonywane

- **równoległa optymalizacja kilku zapytań** – sytuacja taka ma miejsce, gdy serwer obsługuje jednocześnie wiele zapytań od użytkowników; istnieje wtedy możliwość znalezienia części wspólnej tych zapytań i jednorazowej ich ewaluacji
- **wybór plan ewaluacji zapytania** – w przypadku kiedy istnieje wiele dróg ewaluacji zapytania trzeba oszacować czas wykonania każdej z nich i wybrać najszybszą

Pierwsza spośród wyżej wymienionych metod jest metodą najlepiej optymalizującą w większości przypadków (poza sytuacjami skrajnymi). Nigdy lub prawie nigdy nie pogarsza ona czasu odpowiedzi na zapytanie. Z tego powodu autor zdecydował się na implementację tej metody.

Pozostałe metody są również efektywne, ale tylko w specyficznych sytuacjach.

2.4.1 SQL

SQL (Structured Query Language) jest językiem współczesnych systemów relacyjnych baz danych. Został opracowany w latach 70-tych w firmie IBM. Od tamtej pory stał się tak popularny, iż stał się standardem w komunikacji z serwerami relacyjnych baz danych, takimi jak IBM-owskie DB2 i SQL/DS, a także MySQL, PostgreSQL, Interbase SQL, Oracle, Microsoft SQL Server, Microsoft Access, Sybase Adaptive Server Enterprise, Sybase SQL Anywhere, Computer Associates Ingres, Informix, mSQL, First SQL i inne. Można powiedzieć, że korzystanie z relacyjnych baz danych, to korzystanie z SQLa.

W systemach relacyjnych optymalizacja kładzie głównie nacisk na operatory złączenia (ang. *join operators*).

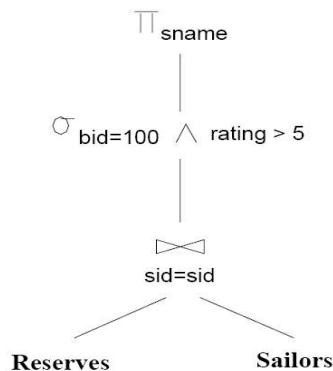
Zapytanie traktowane jest jako algebraiczne wyrażenie $\sigma - \Pi - X$. Przykładowo zapytanie:

```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
AND R.bid = 100 AND S.rating > 5
```

wyrażone jest w algebrze relacji jako:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating > 5}(Reserves \bowtie_{sid=sid} Sailors))$$

zaś w postaci drzewa jako:



Rys. 2.1 Zapytanie w algebrze relacji rozpisane w postaci drzewa

Optymalizacja relacyjnej algebry sprowadza się do dwóch rzeczy:

- policzenia alternatywnych planów ewaluacji zapytania; najczęściej optymalizator bierze pod uwagę tylko podzbiór wszystkich planów ze względu na ich ogromną ilość
- oszacowania kosztu wykonania każdego z tych planów i wybrania najtańszego planu

W większości relacyjnych systemów optymalizator zaimplementowany jest na wzór optymalizatora w Systemie R firmy IBM. Jego główne cechy to:

- zapytania są optymalizowane podczas kompilacji a później tylko powtarzane
- statystyki bazy danych są używane do obliczenia kosztu ewaluacji zapytania
- model kosztu bierze pod uwagę zarówno CPU jak i I/O
- głównym celem optymalizacji są niezagnieżdżone zapytania
- dla operatora projekcji powtarzające się karotki nie są usuwane
- przy tworzeniu alternatywnych planów ewaluacji dla zapytań typu *join query* uwzględniane są tylko lewe poddrzewa

2.4.2 XQuery

W ciągu ostatnich paru lat XML zdobył ogromną popularność jako uniwersalny format wymiany danych. Wraz z upływem czasu gdy stawał się standardem dla e-biznesu, rosła potrzeba jego efektywnego odpytywania oraz przechowywania.

Na tą potrzebę odpowiedziały takie języki zapytań jak Lorel, Quilt, XQL, XML-QL, XPath oraz XQuery. Najpopularniejszymi językami są XQuery oraz jego podzbiór XPath.

Dotychczasowe zainteresowanie optymalizowaniem XQuery i XPath było niewielkie. Spowodowane to było faktem, iż XML był głównie używany do przechowywania niewielkich zbiorów informacji. Optymalizowanie takich dokumentów nie ma najmniejszego sensu.

Rozwój XML'a zdaje się kierować w stronę bazodanowych paradygmatów, a co za tym idzie coraz więcej starań jest czynionych, aby zapewnić efektywną ewaluację zapytań języków XPath/XQuery.

XQuery to typowo funkcyjny język umożliwiający zarówno transformowanie jak i odpytywanie XML'a. Pozwala również na wyciąganie danych za pomocą wyrażeń XPath, łączenie ze sobą dokumentów oraz tworzenie całkowicie nowych.

Poniżej przedstawiono różne podejścia do tematu optymalizacji XQuery.

- **przetwarzanie równoległe** - budowanie drzewa oraz parser działają równoległe z engine transformacji; gdy arkusz stylów chce odwołać się do nie istniejącego w drzewie węzła engine transformacji czeka dopóki parser nie dostarczy takowego
- **optymalizacja wg DTD (definicja typu dokumentu)** – na podstawie DTD można wyeleminować takie zapytania (ścieżki) XPath, które są niemożliwe do przejścia, napewno będą puste lub redundantne
- **formalne semantyki** – przekształcenie zapytań wg formalnej semantyki na język XQuery Core (niższego poziomu, słabo czytelny), umożliwi wykorzystanie wielu technik dostępnych dla zwykłych języków bazodanowych
- **przepisywanie (ang. *rewriting*)**
- **„leniwe” konstruowanie drzewa** – węzeł drzewa (oraz całe poddrzewo pod nim) może być reprezentowany jako zamknięcie (ang. *closure*) zawierające wyrażenie oraz kontekst dla wykonania tego wyrażenia

2.4.3 Optimization Methods in Object Query Languages

Nazwa niniejszego rozdziału to tytuł pracy doktorskiej J. Płodzenia poświęconej

tematowi optymalizacji, a wykorzystaną również jako źródło wiedzy ([Płodzień 2000]) do stworzenia niniejszej pracy.

Autor opisuje ogólne strategie optymalizacyjne jakie występują w językach zapytań, zarówno relacyjnych jak i obiektowych.

Praca została zakończona implementacją składającą się z następujących modułów:

- parser lekko zmodyfikowanego języka SBQL
- parser Loqis DDL
- silnik do statycznej analizy
- metoda niezależnych podzapytań oparta na przepisywaniu

Jak widać powyższe założenia są bardzo podobne do założeń niniejszej pracy. Zasadnicza różnica dotyczy się wykorzystanej bazy danych, tj. Loqis (w porównaniu do ICONS).

W obu przypadkach zaimplementowano tą samą metodę optymalizacji.

2.5 Metody optymalizacji

Rozdział ten poświęcony metodom optymalizacji zapytań opartych na przepisywaniu. Omówione zostaną:

- przepisywanie niezależnych pod-zapytań
- przepisywanie za pomocą niskopoziomowych technik
- przepisywanie wyrażeń ścieżkowych
- przepisywanie zapytań wywołujących perspektywy

2.5.1 Metoda niezależnych pod-zapytań – drugi sposób

Metoda ta ma swoje dwie odmiany: „wyciąganie” niezależnych pod-zapytań przed operator niealgebraiczny (ang. *factoring out*) oraz „wypychanie” selekcji przed operator niealgebraiczny (ang. *pushing out*). Ten rozdział poświęcony został tylko drugiemu sposobowi. Pierwszy zaś zostanie omówiony w rozdziale 4.1, ponieważ zaimplementowany optymalizator oparty jest na sposobie jego działania.

Zanim autor opisze drugi sposób należy przedstawić zasadę dystrybucyjności operatorów, która to bardzo usprawnia jego działanie. Niealgebraiczny operator Θ jest dystrybucyjny, jeśli dla składniowo poprawnych zapytań $q1$, $q2$, $q3$ oraz dla dowolnego stanu bazy danych i stosu środowiskowego zachodzi:

$$(q1 \text{ union } q2) \Theta q3$$

jest semantycznie równoważne

$$(q1 \Theta q3) \text{ union } (q2 \Theta q3)$$

Innymi słowy niealgebraiczny operator Θ jeśli dla każdego zapytania

$$(q1 \Theta q3)$$

jego rezultat może być obliczony jako złączenie wszystkich rezultatów

$$r \Theta q2$$

, gdzie r jest rezultatem $q1$. Końcowy rezultat jest zawsze złączeniem rezultatów cząstkowych.

Zachodzą następujące reguły:

- $(q1.q2).q3$ jest równoważne $q1.(q2.q3)$

- $(q1 \text{ join } q2) \text{ join } q3$ jest równoważne $q1 \text{ join } (q2 \text{ join } q3)$
- $(q1 \text{ join } q2).q3$ jest równoważne $q1.(q2.q3)$
- $(q1 \text{ join } q2) \text{ where } q3$ jest równoważne $q1 \text{ join } (q2 \text{ where } q3)$

Operatory **where**, projekcji oraz złączenia są dystrybucyjne. Dystrybucyjnymi nie są kwantyfikatory, operatory sortowania oraz grupowania.

„Wypychanie” selekcji przed złączenie było znane już w relacyjnych systemach baz danych. W niniejszym podejściu została uogólniona jako „wypychanie” selekcji przed operator niealgebraiczny: jeżeli predykat selekcji jest niezależny od **where**, wtedy selekcja nie powinna być „wyciągnięta”, lecz „wypchnięta” przed ten operator.

Poniższe zapytanie pomoże w zrozumieniu ogólnej idei tej metody:

Pracownik.(*Zarobek where Nazwisko = „Kowalski”*)
 (1,1) 2 (2,2) 3 (3,2)

Proszę zwrócić uwagę na nazwę *Nazwisko* występującą w tym zapytaniu jako część predykatu *Nazwisko = „Kowalski”*

Wiązana ona jest w sekcji nr 2, pomimo iż rozmiar stosu wynosi 3. Ponieważ sekcja nr 3 otwierana jest przez operator **where**, predykat jest niezależny od tego operatora (jest zależny od operatora projekcji). Uwzględniając właściwość dystrybucyjności tej selekcji i biorąc pod uwagę semantykę tego operatora daje się zauważyć, że możliwe jest „wypchnięcie” predykatu przed **where**:

(Pracownik where Nazwisko = „Kowalski”).(Zarobek)
 (1,1) 2 (2,2) 2 (2,2)

[P.2.1]

W większości przypadków bardziej opłaca się „wypchnąć” predykat przed selekcję niż „wyciągnąć” go używając pierwszej odmiany metody niezależnych pod-zapytań (patrz rozdział 4.1). Dzieje się tak dlatego, że „wypychanie” redukuje rozmiar ewaluacji częściowych zapytań, co powoduje obniżenie kosztu ewaluacji całego zapytania. Ponadto po „wypchnięciu” predykatu inna metoda optymalizacyjna może zostać użyta, np. indeksowanie (ang. *indexing*). Dla P.2.1 naturalnym miejscem do założenia indeksu byłoby podzapytanie *Pracownik where Nazwisko = „Kowalski”* (więcej na temat użycia indeksów w rozdziale 2.5.2).

W P.2.1 cały predykat został „wypchnięty” przed selekcję. Możliwe jest jednak „wypchnięcie” tylko części predykatu. Pokazuje to niniejszy przykład P.2.2. Zapytanie zwraca 20 letnich pracowników razem z działami sprzedaży (znajdującymi się w Warszawie), w których to pracują.

Pracownik join (PracujeW.(Dział where Nazwa = „sprzedaż” and Wiek = 20)
 (1,1) 2 (2,2) 3 (3,3) 4 (4,4) (4,2)

[P.2.2]

Predykat selekcji nie jest niezależnym pod-zapytaniem jako całość, ponieważ nazwa *Nazwa* wiązana jest w sekcji otwieranej przez operator **where**. Nazwa *Wiek* znajdująca się w podpredykatcie

Wiek = 20

[P.2.3]

jest niezależna od **where**; jest również niezależna od operatora projekcji (jest za to zależna od **join**). Zgodnie z właściwością dystrybucyjności operatora **where**, projekcji i **join** oraz faktem, że podpredykat P.2.3 jest połączony z resztą predykatu selekcji poprzez **and**, warunek (P.2.3) może być „wypchnięty” i przeniesiony do lewego pod-zapytania operatora **join**, tj. pod-zapytania *Pracownik*. I tak zapytanie P.2.2 może być przepisane do następującej postaci:

(*Pracownik where Wiek = 20*) **join** (*PracujeW.(Dział where Nazwa = „sprzedaż”)*)
 (1,1) 2 (2,2) 2 (2,2) 3 (3,3) 4 (4,4)

Podsumowując, w metodzie tej dwie rzeczy są istotne. Po pierwsze, można „wypychać” niezależny predykat, tylko jeżeli wszystkie niealgebraiczne operatory, wobec których jest niezależny i przed które chcemy go „wypchnąć” są dystrybucyjne. W przeciwnym wypadku takie przekształcenie jest niepoprawne. Po drugie, niezależny podpredykat może zostać „wypchnięty” tylko jeżeli jest połączony z resztą predykatu za pomocą operatora logicznego **and** (P.2.2), inaczej może zostać tylko „wyciągnięty”.

2.5.2 Metody niskopoziomowe

Metody niskopoziomowe dowiodły swej skuteczności i przydatności zarówno w relacyjnych jak i obiektowych systemach baz danych dzięki temu, że są używane na poziomie fizycznym.

Poniżej autor zaprezentuje dwóch przedstawicieli tych metod: indeksowanie (ang. *indexing*) oraz bezpośrednią nawigację (ang. *direct navigation*).

2.5.2.1 Indeksowanie

Pojedynczy index założony na atrybucie *Atr* obiektu *Obiekt* można zdefiniować jako procedurę o sygnaturze

index_Object_Atr(par)
 [P.2.4]

którą można wywoływać jako zwykłe (pod)zapytanie. Parametr *par* jest zapytaniem zwracającym pojedynczą atomową wartość, której typ jest kompatybilny z typem *Atr*.

Procedura P.2.4 działa w następujący sposób: odczytuje z właściwego indeksu wszystkie identyfikatory obiektów *Obiekt*, dla których atrybut *Atr* równy jest rezultatowi zwracanemu przez *par*, a następnie wkłada te identyfikatory na stos wyników. Z powodu zastosowania indeksów, ewaluacja takiej procedury jest znacząco szybsza (czasem o parę rzędów wielkości) niż ewaluacja odpowiadającego (pod)zapytania

Obiekt where Atr = par

Użycie indeksów zawsze polepsza wydajność. Jeśli *Obiekt* posiada indeksy założone na paru atrybutach (przynajmniej dwóch) powinno się użyć tego indeksu, który jest bardziej selektywny. Do tego służy właśnie model kosztów.

Pojęcie selektywności definiowane jest poprzez czynniki redukcji. Czynniki redukcji danego predykatu selekcji jest to szacowana procentowa ilość wyników zwróconych ze stosu rezultatów dla podpredykatu, dla którego predykat jest zastosowany. Indeks, który jest bardziej selektywny ma liczbowo mniejszy czynnik redukcji i tego indeksu używamy.

Procedura P.2.4 działa tylko dla operatora równości. Można jednak zdefiniować ją również dla innych operatorów relacji. Przykładowo dla operatora $>$ procedura miałaby postać

index_Objekt_Atr_wiekszy_od(par)

Analogicznie do P.2.4 wybiera ona identyfikatory tych obiektów *Objekt*, dla których wartość *Atr* jest większa od wartości zwróconej przez *par*.

Poniżej zawarto przykłady ilustrujące użycie indeksów w metodzie przepisywania zapytań.

(1) *Pracownik where Nazwisko = „Kowalski”*

zostaje przepisane do postaci (indeks założony na atrybucie *Nazwisko*)
index_Pracownik_Nazwisko(„Kowalski”)

(2) (*Pracownik where Nazwisko = „Kowalski”*) **join** *PracujeW.Dzial.Nazwa*

zostaje przepisane do postaci (indeks założony na atrybucie *Nazwisko*)
index_Pracownik_Nazwisko(„Kowalski”) join PracujeW.Dzial.Nazwa

(3) (*Pracownik where Zarobek > ((Pracownik where Nazwisko = „Kowalski”).Zarobek)*).
Nazwisko

zostaje przepisane do postaci (indeks założony na atrybucie *Zarobek*)
index_Pracownik_Zarobek_wiekszy_od((Pracownik where name = „Kowalski”).Zarobek).
Nazwisko

(4) *Pracownik where (Nazwisko = „Kowalski” and Urodzony > '1980-01-01' and Zarobek > 2500)*

zostaje przepisane do postaci (indeks założony na atrybucie *Nazwisko*)
index_Pracownik_Nazwisko(„Kowalski”) where (Urodzony > '1980-01-01' and Zarobek > 3000)

Warto tutaj pokazać jak wyglądałoby przepisane zapytanie, jeśli indeks byłby założony na atrybucie *Urodzony*:

index_Pracownik_Urodzony_wiekszy_od('1980-01-01') where (Nazwisko = „Kowalski” and Zarobek > 2500)

Jeśli na obu atrybutach założone by były indeksy o wyborze zadecydowałby ich parametr selektywności.

2.5.2.2 Bezpośrednia nawigacja

Kolejnym sposobem optymalizowania zapytań jest wykonywanie pewnych operacji bezpośrednio na obiektach, nie modyfikując przy tym stosu środowiskowego. Ponownie jak w przypadku indeksowania operacja ta wykonują się na niskim poziomie.

Bezpośrednia nawigacja (ang. *direct navigation*) dotyczy sytuacji kiedy nawiguje się po atrybucie *n* obiektów *Objekt*, których identyfikatory zostały zwrócone przez podzapytanie *q*:

q.n
[P.2.5]

Jak wiadomo z zasad działania SBQL procedura *eval* wykonuje P.2.5 następująco: oblicza podzapytanie *q* w celu uzyskania identyfikatorów obiektów *Objekt*, tworzy sekcje na stosie środowiskowym zawierające atrybuty tych obiektów i w końcu przeszukuje te sekcje w

poszukiwaniu identyfikatorów atrybutu n . Koszt takiej bezpośredniej ewaluacji jest znaczący.

Metoda bezpośredniej nawigacji dla zapytań typu P.2.5 nie musi modyfikować stosu środowiskowego ani pobierać obiektów po to żeby w rezultacie otrzymać identyfikatory n . Założenie jest takie, że identyfikator atrybutu obiektu może zostać wyznaczony poprzez identyfikator obiektu oraz offset atrybutu w tym obiekcie (wartości takich offsetów powinny być trzymane w bazie). Wprowadźmy specjalną procedurę bezpośredniej nawigacji:

dnav(offset)

[P.2.6]

Działa ona następująco: dla każdego identyfikatora obiektu ze szczytu stosu rezultatów pobiera identyfikator tego atrybutu, którego offset równy jest parametrowi *offset*, usuwa najwyższą sekcję ze stosu a wkłada tam identyfikator atrybutu.

Zapytanie P.2.5 można przepisać stosując procedurę P.2.6 w taki oto sposób:

q dnav(offset_n)

,gdzie *offset_n* jest offsetem atrybutu n w obiektach *Obiekt*. Stosując to co przed chwilą zostało napisane zapytanie

Pracownik.Urodzony

zostanie przepisane do postaci

Pracownik dnav(offset_Urodzony)

zaś zapytanie

(Pracownik where Zarobek >= 2000).PracujeW.Dział.Lokacja

do

((Pracownik where Zarobek >= 2000) dnav(offset_PracujeW).Dział dnav(offset_Lokacja)

, gdzie offsety *offset_PracujeW*, *offset_Lokacja*, *offset_Urodzony* są offsetami odpowiadających atrybutów w obiektach *Pracownik*.

2.5.3 Metody optymalizacji wyrażeń ścieżkowych

Jedną z podstawowych właściwości modelu obiektowego są wyrażenia ścieżkowe, tj. (pod)zapytań postaci

n₁.n₂.n₃. ... n_{k-1}.n_k

gdzie n_i (i należy do $\{1, 2, \dots, k\}$) jest zewnętrzną nazwą obiektu lub jego atrybutu.

Przykładowo zapytanie

Pracownik.PracujeW.Dział

nawiguje od od pracowników do działów, w których oni pracują.

W tym rozdziale zostaną opisane dwie metody tego typu, które można zastosować do zapytań zawierających wyrażenia ścieżkowe.

2.5.3.1 „Wyciąganie” wspólnych wyrażeń ścieżkowych

Znajdowanie wspólnego podzapytania w zapytaniu, tak aby wykonać je tylko raz a później użyć raz obliczony rezultat jest popularną techniką optymalizacji (zazwyczaj niskopoziomową). W rozdziale tym zostanie opisane znajdowanie oraz „wyciąganie” wspólnych wyrażeń ścieżkowych. Proszę spojrzeć na poniższy przykład:

*Wykład where (ProwadzonyPrzez.Profesor.PracujeW.Katedra.Nazwa = „Fizyka”) and
(„Warszawa” in ProwadzonyPrzez.Profesor.PracujeW.Katedra.Lokacja)*

[P.2.7]

Da się zauważyć dwa wyrażenia ścieżkowe w klauzuli **where**, tj:

ProwadzonyPrzez.Profesor.PracujeW.Katedra.Nazwa
[P.2.8]

oraz

ProwadzonyPrzez.Profesor.PracujeW.Katedra.Lokacja
[P.2.9]

są liczone oddzielnie. Można jednak z nich wyodrębnić część wspólną:

ProwadzonyPrzez.Profesor.PracujeW.Katedra
[P.2.10]

Ponieważ P.2.10 jest obliczane zarówno dla P.2.8 i P.2.9 w dokładnie tym samym środowisku (tworzonym przez operator **where**, **and** nie ma wpływa, ponieważ jest operatorem algebraicznym, a takowe nie modyfikują stosu środowiskowego), rezultat końcowy jest takim sam w obu przypadkach. Wystarczy więc raz obliczyć P.2.10, a następnie obliczyć resztę z P.2.8 i P.2.9:

Wykład where ProwadzonyPrzez.Profesor.PracujeW.Katedra.((Nazwa = „Fizyka”) and („Warszawa” in Lokacja))

2.5.3.2 Relacje wsparcia dostępu

Relacja wsparcia dostępu (ASR) jest specjalną strukturą indeksową, która znacznie zwiększa wydajność poruszania po wyrażeniach ścieżkowych.

Jeżeli dane zapytanie zawiera wyrażenie ścieżkowe

$n_1.n_2.n_3. \dots n_{k-1}.n_k$

[P.2.11]

oraz gdy istnieje ASR modelująca nawigację od encji n_1 (obiekt, podobiekt itp.) do encji n_k (obiekt, podobiekt itp.) to wtedy wyrażenie P.2.11 można zastąpić wywołanej procedury

ASR_n1_nk

[P.2.12]

P.2.12 wykonuje się tak: poprzez użycie odpowiedniej ASR otrzymuje identyfikatory encji n_k nawigowanych od wszystkich encji n_1 istniejących w bazie danych i umieszcza je na stosie rezultatów. Przykładowo:

(Katedra.Zatrudnia.Profesor.Prowadzi.Wyklad where Punkty > 5).Temat

[P.2.13]

zostanie przepisane do postaci

(ASR_Katedra_Wyklad where credits > 5).Temat

zakładając, że istnieje ASR nawigująca od obiektów *Katedra* do obiektów *Wyklad*.

Możliwym jest również, aby zamiast P.2.12 użyć takiej oto modyfikacji

$ASR_QRES_n1_nk$

[P.2.14]

Różnica pomiędzy P.2.12 a P.2.14 jest taka, że ta ostatnia nie otrzymuje identyfikatorów encji n_k nawigowanych od wszystkich encji n_1 istniejących w bazie danych, lecz nawigowanych tylko od tych n_1 , których identyfikatory są na szczycie stosu rezultatów. Dodatkowo przed włożeniem identyfikatorów n_k na stos rezultatów P.2.14.

Sugeruje się, aby używać P.2.14 w roli unarnego sufikсового operatora algebraicznego:

$q ASR_QRES_n1_nk$

gdzie q jest podzapytaniem wkładającym na stos rezultatów identyfikatory encji n_1 .

Różnicę pomiędzy tymi dwoma procedurami najlepiej pokazuje poniższy przykład. Jeżeli zapytanie P.2.13 byłoby postaci

```
((Katedra where Nazwa = „fizyka”).Zatrudnia.Profesor.Prowadzi.Wyklad)
where Punkty > 5).Temat
```

nie możliwym byłoby użycie ASR_Wyklad_Katedra, ponieważ pierwsza selekcja wyklucza katedry inne niż fizyka. Natomiast użycie procedury postaci P.2.14 jak najbardziej na to pozwala:

```
((Katedra where Nazwa = „fizyka”).ASR_QRES_Katedra_Wyklad)
where Punkty > 5).Temat
```

2.5.4 Metody optymalizacji zapytań wywołujących perspektywy

Ważnym aspektem optymalizacji zapytań jest optymalizacja zapytań wywołujących perspektywy. Perspektywy znacznie podnoszą poziom abstrakcji (oraz modularność) tworzenia aplikacji, niestety kosztem słabej wydajności. Powodem jest bezpośrednia ewaluacja takich zapytań oraz brak dostępu do ciała perspektywy przez metody optymalizacyjne. W relacyjnych bazach danych poradzono sobie z tym problemem ten sposób, że takie zapytania są optymalizowane specjalną metodą: *techniką modyfikacji zapytania*. Głównym pomysłem jest tutaj połączenie zapytania zawierającegowołanie perspektywy z definicją tej perspektywy. Zapytanie końcowe nie ma żadnych odwołań to wywołań perspektywy (zamiast tego, posiada bezpośredni dostęp do jego definicji) i może zostać zoptymalizowane przez przepisanie.

Zalety tej techniki są dwojakie. Po pierwsze, uniknąć można narzutów wydajności związanych z przetwarzaniem perspektywy. Po drugie, o wiele bardziej ważne, umożliwia zaaplikowanie metod optymalizacyjnych.

2.5.4.1 Usuwanie zbędnych pomocniczych nazw

Nazwy pomocnicze są bardzo przydatnym narzędziem w tworzeniu zapytań. Służą przede wszystkim nazywaniu różnych części zapytania, tak aby później można było się odwołać precyzyjnie do tych części do których chcemy. Dostęp do tych nazwanych części następuje poprzez projekcje do nazwy pomocniczej. Szczegółowa analiza wykonania takiej operacji pokazuje (tj. nazwania podrezultatu pomocniczą nazwą oraz wykonania projekcji do tej nazwy), że często jest ona wykonywana niepotrzebnie (możemy odwołać się do podrezultatu bezpośrednio). Oznacza to, że nazwy pomocnicze mogą zostać zoptymalizowane, w szczególności poprzez ich usunięcie. Zagadnienie nie jest takie proste na jakie brzmi. Usunięcie niektórych nazw z zapytania może zmienić rezultat całego zapytania.

Poniższa perspektywa pobiera profesorów, którzy zarabiają więcej niż dwa tysiące (i nadaje im pomocniczą nazwę *p*):

```
view Psorzy
begin
    return ((Profesor where Zarobek > 2000) as p)
end;
```

Rezultatem tej perspektywy są bindery nazwane *p* a wartościami identyfikatory odpowiednich obiektów *Profesor*. Jeżeli chcemy otrzymać te identyfikatory poprzez perspektywę, trzeba ją wywołać i dokonać projekcji:

Psorzy.p
[P.2.15]

Poprzez wykonanie tej projekcji pozbywamy się binderów zwróconych przez perspektywę, tj. rezultatem zapytania P.2.15 są identyfikatory a nie bindery. Proszę zauważyć, że usunięcie pomocniczej nazwy w P.2.15 jest zabronione, ponieważ taka modyfikacja zmieniałby rezultat zapytania (w szczególności otrzymalibyśmy bindery, nie zaś identyfikatory). Usuwając pomocniczą nazwę *p* musielibyśmy również usunąć jej definicję z ciała perspektywy. Jednakże w zapytaniu P.2.15 nie możemy tego zrobić, gdyż nie mam dostępu do wnętrza perspektywy, gdzie nazwa pomocnicza jest zdefiniowana.

Aby mieć dostęp do definicji nazwy pomocniczej w ciele perspektywy trzeba dokonać wywołania perspektywy poprzez makro. Po tym zapytanie P.2.15 będzie postaci:

$((\text{Profesor where Zarobek} > 2000) \text{ as } p). p$
(1,1) 2 (2,2) 2 (2,2)

[P.2.16]

Jak widać nazwa *p* spełnia tu dwie funkcje:

- nazywa rezultat zapytania (*Profesor where Zarobek > 2000*)
- nawiguje do tego rezultatu

Rezultat P.2.16 nie ulegnie zmianie, jeśli zostanie usunięta pomocnicza nazwa (poprzez usunięcie nazwy rozumie się usunięcie jej wszystkich wystąpień w zapytaniu):

Profesor where Zarobek > 2000

Kolejnym przypadkiem jest przypadek, gdy nazwa pomocnicza jest używana do dalszych projekcji. Przykładowo, używając perspektywy *Psorzy* mamy dostęp nie tylko do obiektów, które zwraca, ale również ich atrybutów:

Psorzy.(p.Nazwisko, p.Wiek)

Pomocnicza zmienna *p* musi zostać użyta w podzapytaniu

p.Nazwisko, p.Wiek

w celu umożliwienia nawigacji „.Nazwisko” i „.Wiek”. Dlatego też usunięcie *p* jest niemożliwe. Jednakże po ponownym wywołaniu perspektywy poprzez makro :

$((\text{Profesor where Zarobek} > 2000) \text{ as } p).(p.Nazwisko, p.Wiek)$

[P.2.17]

można dostać się do atrybutów *Profesora* bezpośrednio, czyli nazwy pomocniczej można się pozbyć:

$((\text{Profesor where Zarobek} > 2000)).(\text{Nazwisko}, \text{Wiek})$

Podsumowując w przypadku ogólnym pomocnicza nazwa może mieć parę zastosowań w jednym zapytaniu:

$((\text{Pracownik as } p) \text{ where } p.Zarobek < 1000).p$

pierwsze *p* pozwala nawigować do atrybutu *Zarobek*, podczas gdy drugie użyte jest w końcowej projekcji. *p* można zredukować:

Pracownik where Zarobek < 1000

We wszystkich powyższych przykładach nazwy pomocnicze były zbędne. Dla odmiany poniższe przykłady pokazują sytuacje kiedy nazwy pomocnicze nie mogą być usunięte.

Student.(Uczęszcza.Wykład.ProwadzonyPrzez.(Profesor as p).(Nazwisko, p.Nazwisko))

(1,1) 2 (2,2) 3 (3,3) 3 (3,3) 3 (3,3) 3 (3,2) (3,3)4 (4,4)
[P.2.18]

Nazwisko jest atrybutem zarówno obiektu *Student* jak i *Profesor*. W podzapytaniu zwracającym *Nazwisko*:

(*Nazwisko*, *p.Nazwisko*)

pierwsza nazwa *Nazwisko* wiązana jest w sekcji otwieranej dla rezultatu podzapytania *Student*, podczas gdy druga wiązana jest w sekcji otwieranej dla rezultatu podzapytania *Profesor* **as** *p*. Jeżeli nazwa *p* zostanie usunięta:

Student.(*Uczęszcza.Wyklad.ProwadzonyPrzez*.(*Profesor*).(*Nazwisko*, *Nazwisko*))

[P.2.19]

obie nazwy *Nazwisko* byłyby wiązane w tej samej sekcji, tj. otwartej dla rezultatu podzapytania *Profesor*. Stąd zapytanie P.2.19 pobiera dla każdego studenta zdublowanie nazwisko swojego profesora, nie zaś nazwisko studenta i nazwisko profesora. W konsekwencji zapytanie P.2.18 i P.2.19 nie są semantycznie równoważne, co oznacza że usunięcie nazwy *p* jest niemożliwe.

Pseudokod algorytmu znajdowania pomocniczych nazw, które można usunąć wygląda tak:

- najpierw tworzona jest kopia drzewo składniowego zapytania
- następnie nazwa usuwana jest z zapytania
- uruchamiana jest statyczna analiza na pierwotnym drzewie składniowym oraz zmienionym w celu sprawdzenia czy wszystkie nazwy wiązane są dokładnie w tym samych sekcjach. Jeżeli tak to oznacza to, że usunięcie pomocniczej nazwy nie zmieniło rezultatu zapytania. Nowa drzewo staje się drzewem pierwotnym.

Usunięcie pomocniczej nazwy *n* możliwe jest tylko wtedy gdy:

- rezultat wykonania *n* użyty jest przez operator projekcji
- bezpośrednim nie-algebraicznym operatorem *n* jest operator projekcji i rezultat wykonania *n* nie jest konsumowany przez żaden inny operator

W przeciwnym wypadku nazwa *n* nie może być usunięta.

2.5.4.2 Zastępowanie złączenia operatorem projekcji

Poza metodami optymalizacyjnymi, których zadaniem jest zmniejszenie czasu wykonania zapytania istnieją takie, które mają za zadanie zmniejszenie ilości pamięci potrzebnej w pewnych fazach wykonania zapytania. Główną ideą jest: jeśli żaden binder z listy podsekcji (która jest częścią sekcji otwieranej przez operator projekcji) nie został użyty, wtedy operator projekcji nie musi tworzyć takiej listy podczas otwierania swojej sekcji. Tym samym potrzeba mniej pamięci na przechowywanie stosu. Dodatkowo mniej sekcji zużywa mniej czasu, co oznacza że metoda ta skraca także czas wykonania zapytania.

Przykład poniżej ilustruje ogólny pomysł. Perspektywa *PracDzial* zwraca pracowników razem z działami, w których pracują:

```
view PracDzial
begin
    return Pracownik join PracujeW.Dzial
```

end;

Perspektywa taka może być użyta do zdefiniowania zapytania zwracającego nazwy działów, w których pracują ich pracownicy:

PracDział.Nazwa

,które po wywołaniu perspektywy poprzez makro wyglądałoby tak:

*(Pracownik **join** PracujeW.Dział).Nazwa*

Operator ostatniej projekcji otwiera sekcję składającą się z dwóch list podsekcji: pierwsza z nich tworzona jest na potrzebę rezultatu podzapytania

Pracownik

druga zaś na potrzebę rezultatu podzapytania

PracujeW.Dział

Ponieważ nazwa *Nazwa* odnosi się do *Dział* (nie ma zaś znaczenia w kontekście *Pracownika*) końcowa projekcja odnosi się tylko do prawej strony operatora zależnego złączenia. Oznacza to, że podczas wykonywania zapytania niepotrzebny jest dostęp do rezultatu lewego operandu tego złączenia. Wynika z tego, że podczas wiązania *Nazwa* wystarczy, aby stos środowiskowy zawierał podsekcje utworzone dla wyników prawego operandu. Aby tego dokonać wystarczy podmienić operatora złączenia z operatorem projekcji:

(Pracownik.PracujeW.Dział).Nazwa

W momencie gdy podzapytanie *Nazwa* jest wykonywane, szczytowa sekcja stosu środowiskowego (otwarta dla rezultatu lewego podzapytanie zewnętrznego operatora projekcji) zawiera tylko jedną listę podsekcji; jest to lista stworzona dla obiektów *Dział*.

Poniższy przykład jest bardziej skomplikowany. Perspektywa *InfoOStudentach* zwraca studentów razem z wykładami, na które chodzą oraz profesorów, którzy prowadzą te wykłady i katedry, w których pracują:

view *InfoOStudentach*

begin

return *Student **join** Uczęszcza.Wykład **join** ProwadzonyPrzez.Profesor **join***

PracujeW.Katedra

end;

Aby otrzymać nazwa tych katedr, na których wykłady uczęszczają studenci trzeba zadać takie zapytanie:

InfoOStudentach.Nazwa

Po wywołaniu perspektywy poprzez makro otrzyma się

*(Student **join** Uczęszcza.Wykład **join** ProwadzonyPrzez.Profesor **join***

PracujeW.Katedra).Nazwa

, które jest równoważne

*((Student **join** Uczęszcza.Wykład) **join** ProwadzonyPrzez.Profesor) **join***

PracujeW.Katedra).Nazwa

Podzapytanie *Nazwa* korzysta jedynie z rezultatu podzapytania *PracujeW.Katedra*, co oznacza że można przepisać całe zapytanie do postaci:

*((Student **join** Uczęszcza.Wykład) **join** ProwadzonyPrzez.Profesor).*

PracujeW.Katedra).Nazwa

Ponieważ podzapytanie *PracujeW* korzysta z rezultatu podzapytania *ProwadzonyPrzez.Profesor*, transformacji można dokonać jeszcze raz:
*((Student join Uczęszcza.Wyklad).ProwadzonyPrzez.Profesor).
PracujeW.Katedra).Nazwa*

I tak dalej. Ostateczną formą zapytanie będzie:

Student.Uczęszcza.Wyklad.ProwadzonyPrzez.Profesor.PracujeW.Katedra.Nazwa

3. Język SBQL (Stack-Based Query Language)

Język SBQL to obiektowy język zapytań podobny do SQL lub OQL (posiada odpowiedniki pewnych konstrukcji językowych). W swej postaci podobny jest do algebry relacji stanowiącej podstawę modelu relacyjnego, ale dodatkowo znacznie bardziej koncepcyjnie przejrzysty i konsekwentny.

SBQL oparty jest na abstrakcyjnej składni i zasadzie kompozycyjności, tzn. brakuje tutaj „cukru syntaktycznego”, zaś operatory są od siebie składniowo oddzielone.

3.1 Składnia SBQL

Składnię SBQL'a można opisać trzema poniższymi stwierdzeniami:

- pojedyncza nazwa lub pojedynczy literal mogą już stanowić zapytanie, np.: *Pracownik*, *Nazwisko*, *Zarobek*, *2500*, „Tomek”
- jeżeli q jest zapytaniem, zaś σ jest operatorem unarnym (np.: *sum*, *count*) to $\sigma(q)$ również jest zapytaniem
- jeżeli q_1 , q_2 to zapytania, zaś Θ jest operatorem binarnym (np.: **where**, **union**, $+$, $-$, $=$, $>$, $<$, $*$, $/$) to $q_1 \Theta q_2$ również jest zapytaniem
- jeżeli q jest zapytaniem, zaś $n \in N$ to q **group as** n jest również zapytaniem; w tym kontekście **as** to operator definiowania pomocniczej nazwy
- jeżeli q jest zapytaniem, zaś $n \in N$ to q **as** n jest również zapytaniem; **as** to unarny operator algebraiczny parametryzowany nazwą n
- jeżeli q jest zapytaniem to (q) jest również zapytaniem; ta reguła definiuje używanie nawiasów w sposób dowolny
- Jeżeli $n \in N$ jest nazwą procedury, funkcji, lub metody posiadającej k parametrów, zaś q_1, q_2, \dots, q_k są zapytaniami, wówczas $n(q_1, q_2, \dots, q_k)$ jest zapytaniem
- Jeżeli $n \in N$ jest nazwą procedury, funkcji, lub metody nie posiadającej parametrów, wówczas $n()$ oraz n są zapytaniami. W większości przypadków będziemy przyjmować, że takie zapytania są równoważne

Poniżej przedstawione są zapytania zgodne z tą składnią:

2500

Zarobek

Pracownik

1800 + 700

Zarobek > 1800

Pracownik where (Zarobek > 2500)

*(Pracownik where (staż() > 3)).(Zarobek * Premia)*

W języku SBQL ortogonalność operatorów jest zapewniona. Np.: *Pracownik*, *Zarobek*, „Tomek” są pełnoprawnymi pojedynczymi zapytaniami i mogą być użyte do zbudowanie bardziej skomplikowanych zapytań:

(*Pracownik* **where** (*Imie* = „Tomek”)).*Zarobek*

Dla porównania to samo zapytanie w SQL wygląda tak:

```
select Zarobek from Pracownik where Imie = „Tomek”
```

zaś w OQL tak:

```
select p.Zarobek from Pracownik as p where p.Imie = „Tomek”
```

3.2 Stosy

3.2.1 Stos środowisk (ENVS)

Pojęcie stosu środowisk ujrzało światło dzienne jeszcze w latach 60-tych wraz z językami Algolo-podobnymi. Od tamtej pory stos stanowi podstawowy element konstrukcji większości znanych języków, takich jak Java, C/C++, Pascal.

ENVS (*ENViromental Stack*) jest strukturą danych istniejącą w pamięci, podzieloną na sekcje. Ich kolejność jest istotna. Nowa sekcja tworzona jest zawsze na wierzchołku stosu, poprzez wykonanie operatora nie-algebraicznego. Tylko operatory nie-algebraiczne modyfikują ENVS.

Stos w języku SBQL zdefiniowany jest w ramach następujących założeń:

- z punktu widzenia programisty maksymalny rozmiar stosu nie jest ograniczony ze strony koncepcyjnej;
- stos jest podzielony na sekcje, przy czym każda sekcja zawiera informacje o środowisku czasu wykonania; rozmiar sekcji nie jest ograniczony koncepcyjnie
- sekcje utworzone przez najbardziej lokalne środowiska umieszczone są na szczycie stosu, zaś sekcja powstała wcześniej odpowiednio niżej
- na samym dole stosu istnieje sekcja globalna, która przechowuje identyfikatory (bindery) obiektów znajdujących się w bazie danych; dzięki temu stos przechowuje informację niezbędną do wiązania dowolnej nazwy jaka może wystąpić w zapytaniu
- stos w jednakowy sposób traktuje dane ulotne (czasu wykonania, np.: dane wywoływanych funkcji, metod i procedur) jak i trwale przechowywane w bazie danych

Jak już wcześniej zostało wspomniane ENVS składa się z sekcji, te z kolei składają się binderów. Formalnie rzecz opisując binder jest parą (n, x) , gdzie n jest zewnętrzną nazwą, zaś x jest referencją do obiektu. Binder zapisujemy w postaci: $n(x)$. Podstawowym zadaniem bindera jest zamiana nazwy występującej w zapytaniu (n) na wartość x , będącą bytem czasu wykonania. Jeżeli dana nazwa nie ma swojego bindera, nie może zostać związana (jest po prostu błędna).

Wartościami binderów może być wszystko co jest *rezultatem* (patrz rozdział 3.2.3). Mówiąc ogólnie binder jest parą $n(x)$, gdzie n dowolną nazwą zewnętrzną definiowaną przez programistę, projektanta bazy danych itp., a x może być dowolnym rezultatem zwracanym przez zapytanie.

3.2.1.1 Wiązanie (ang. *binding*)

Wiązanie to mechanizm, który pozwala na odnalezienie danego bindera na stosie

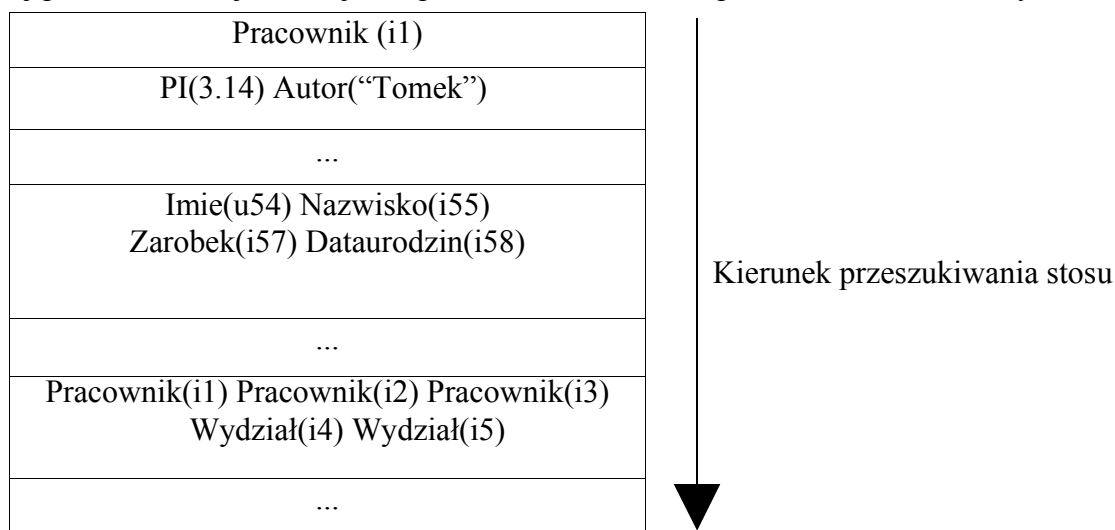
ENVS. Odbywa się ono zgodnie z regułą stosu (czyli szukaj od góry). W celu związania nazwy n mechanizm ten przeszukuje ENVS od góry w poszukiwaniu sekcji zawierającej binder $n(x)$ (który to przechowuje wartość x). Wynikiem wiązania jest właśnie x .

Czasem rezultat wiązania może być wielowartościowy. Jeżeli dana sekcja zawiera kilka binderów o nazwie n : $n(x1)$, $n(x2)$, $n(x3)$, wtedy wszystkie one są wynikiem tego wiązania w postaci kolekcji $\{x1, x2, x3\}$.

Zasady przeszukiwania stosu oraz wyznaczania rezultatu wiązania nazwy są następujące:

- Dla wiązania nazwy n , ENVS jest przeszukiwany aż do znalezienia sekcji, w której znajduje się binder oznaczony nazwą n . Gdy taka sekcja zostanie znaleziona przeszukiwanie zostaje zakończone.
- Wszystkie bindery z tej sekcji oznaczone nazwą n wchodzi do rezultatu wiązania.
- Rezultat wiązania uzyskuje się poprzez odrzucenie ze znalezionych binderów nazwy n i pozostawienie wyłącznie wartości tych binderów.

Poniżej przedstawiona jest kolejność przeszukiwania ENVS podczas wiązania nazwy:



Rys. 3.1 Kolejność przeszukiwania stosu ENVS dla wiązania nazwy

Oto rezultaty wiązania dla stosu z rys. 3.1:

- dla nazwy *Pracownik* rezultatem wiązania jest identyfikator i1
- dla nazwy *PI* rezultatem jest wartość liczbowa 3.14
- dla nazwy *Autor* rezultatem jest wartość stringowa „Tomek”
- dla nazwy *Wydział* rezultatem wiązania jest zbiór identyfikatorów {i4, i5}

W przypadku nazwy *Pracownik* można zauważyć na rys. 3.1, iż bindery o takiej nazwie występują w dwóch sekcjach. Sytuacje takie rozwiązują się same dzięki wspomnianej już regule przeszukiwania stosu, która nakazuje przerwać przeszukiwanie na sekcji zawierającej bindery z nazwą *Pracownik*, która znajduje się najbliżej wierzchołka stosu. Można wtedy powiedzieć, że ten binder znajduje się w bardziej lokalnym środowisku i przesłania inne bindery o tej samej nazwie, które to znajdują się w bardziej globalnym środowisku.

3.2.1.2 Tworzenie nowej sekcji na stosie środowisk

Tworzeniem nowej sekcji (inaczej otwieraniem nowego zakresu) na ENVS zajmują się operatory nie-algebraiczne. Operatory te podobnie działają na stosie podobnie do wywołań bloków programów. Przykładowo w zapytaniu:

Pracownik **where** (*Imie* = „Łazarczyk” **and** *Zarobek* > 2500)

część (*Imie* = „Łazarczyk” **and** *Zarobek* > 2500) ewaluowana jest w nowym środowisku stanowiącym „wnętrze” obiektu *Pracownik* (atrybuty, metody itp.) aktualnie analizowanego poprzez **where**. Na wierzchołku ENVS powstaje nowa sekcja, która zawiera bindery do elementów „wnętrza” obiektu *Pracownik* i w takich warunkach obliczana jest część zapytania występująca po **where**. Tuż po tym omawiana sekcja zostaje usunięta ze stosu, zaś jej miejsce zajmuje następna sekcja zawierająca „wnętrze” kolejnego obiektu *Pracownik*.

3.2.1.3 Funkcja *nested*

Funkcja *nested* zwraca wewnętrzne środowisko obiektu (w szczególności zbiór binderów) dla podanego argumentu jakim jest identyfikator (referencja) do obiektu. To zwracane środowisko stanowi zawartość sekcji, która ma być umieszczona na ENVS.

Jeżeli zaś chodzi o argument funkcji *nested* to został on tak uogólniony, że może nim być dowolny rezultat zapytania.

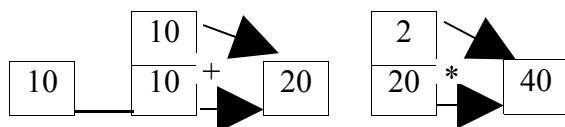
Oto założenia definicyjne dla funkcji *nested*:

- dla wartości atomowych wynik funkcji jest zbiorem pustym
- dla obiektu złożonego $\langle i, n, \{ \langle i_1, n_1, v_1 \rangle, \langle i_2, n_2, v_2 \rangle, \dots, \langle i_k, n_k, v_k \rangle \} \rangle$ wynikiem jest $nested(i) = \{ n_1(i_1), n_2(i_2), \dots, n_k(i_k) \}$
- dla każdego bindera $n(x)$ zachodzi $nested(n(x)) = \{ n(x) \}$
- jeżeli i jest identyfikatorem obiektu pointerowego $\langle i, n, i_1 \rangle$ oraz istnieje obiekt $\langle i_1, n_1, v_1 \rangle$, wówczas $nested(i) = \{ n_1(i_1) \}$
- jeżeli argumentem funkcji *nested* jest struktura, wówczas wynik jest sumą teoriomnogościową rezultatów funkcji *nested* dla pojedynczych elementów struktury:
 $nested(\mathbf{struct}\{ x_1, x_2, x_3, \dots \}) = nested(x_1) \times nested(x_2) \times nested(x_3) \times \dots$

3.2.2 Stos rezultatów (QRES)

Końcowe wyniki zapytań oraz pośrednie ich rezultaty odkładane są na stosie rezultatów (QRES – Query REsult Stack). Najkrócej opisując stos ten jest strukturą danych podobną do stosu arytmetycznego, który można znaleźć w implementacji języków programowania. Rysunek 3.2 pokazuje działanie stosu na krótkiego wyrażenia arytmetycznego:

$$(10 + 10) * 2$$



Rys. 3.2 Kolejno odkładane wyniki na QRES'ie dla zapytania $(10+10) * 2$

Dozwolonymi operacjami na QRES są:

- *push* (włóż nowy element na wierzchołek stosu)
- *pop* (zdejmij element na wierzchołek stosu)
- *top* (odczytaj wierzchołek stosu)
- *empty* (sprawdź czy stos jest pusty)

3.3 Rezultaty zapytań

Na wstępie autor chciałby podkreślić, że w podejściu stosowym zapytania zwracają identyfikatory obiektów, a nie je same. Naturalnie nadal możliwe jest otrzymanie jako rezultat liczby, łańcucha tekstowego, struktury – wystarczy zastosować operator dereferencji.

Rezultat zapytania może być:

- atomowa wartość
- referencja do obiektu (inaczej identyfikator obiektu)
- binder
- strukturę **struct**{ x_1, x_2, x_3, \dots } **struct** jest konstruktorem struktury, czyli pewnym dodatkowym atrybutem (flagą) rezultatu
- kolekcje wielu zbiorów (**bag**) **bag**{ x_1, x_2, x_3, \dots } lub sekwencji (**sequence**) **sequence**{ x_1, x_2, x_3, \dots }, **bag** oraz **sequence** są konstruktorami kolekcji (flagami) podobnie jak konstruktor **struct**

Przykłady rezultatów:

- 24, true, "Tomek", i3, i23
- **struct** {i3, i23}
- **sequence** {i3, i23, i100}
- **bag** {i3, i23, i100}
- **bag** {**struct** {i1, i23}, **struct** {i2, i24}, **struct** {i3, i25}}
- **bag** {i3, **struct** {Imie("Tomek"), Zarobek(2500)}}
- **bag** {**struct** {1500, 2000}, **bag** {**struct** {i1, i23}, **struct** {i3, i25}}, **struct** {**bag** {i2, i24, i30}}}

3.4 Ewaluacja zapytań

Specjalna rekurencyjna funkcja *eval* jest używana do zdefiniowania semantyki operacyjnej SBQL'a. Funkcja ta jako argument przyjmuje zapytanie, zaś zwraca jako wynik rezultat tego zapytania umieszczony na wierzchołku QRES. *Eval* spełnia poniższe założenia:

- *eval* w trakcie wykonywania zapytania q może zmieniać stan *ENVS*, ale po zakończeniu wykonywania q stan ten będzie taki sam, jak na początku
- *eval* w trakcie wykonywania zapytania q nigdy nie zmienia tej części stosu rezultatów, którą zastała w momencie rozpoczęcia wykonywania; może wkładać na ten stos i zdejmować wiele różnych elementów, ale po zakończeniu ewaluacji stos ten będzie

posiadać dokładnie jeden element więcej niż przed ewaluacją (rezultat zapytania q)

Funkcja *eval* jest wspomagana przez parser, który to dokonuje rozbioru gramatycznego zapytania na podzapytania oraz operatory łączące podzapytania. Następnie każde podzapytanie zwraca swój rezultat, zaś rezultat całego zapytania jest obliczony na podstawie rezultatów zwróconych przez podzapytania. Własność taka jest nazywana kompozycyjnością i jest jedna z wielu bardzo mocnych cech **SBQL**. Inne języki zapytań nie posiadają tej cechy co prowadzi do wielu trudności implementacyjnych oraz jest powodem bardzo dużych podręczników użytkownika oraz dokumentacji technicznych.

Funkcja *eval* zbudowana jest z trzech części, każda z nich obsługuje inny przypadek ewaluacji:

- jeżeli zapytanie jest literałem to funkcja *eval* wkłada odpowiadającą mu wartość atomową na wierzchołek QRES
- jeżeli zapytanie jest nazwą to *eval* przeszukuje ENVS od góry do dołu oraz wkłada na QRES do nowej sekcji rezultat wiązania tej nazwy

W przeciwnym wypadku funkcja *eval* rozpoczyna rozbiór gramatyczny w celu wyodrębnienia podstawowych podzapytań oraz łączących ich operatorów. Następnym elementem jest obliczenie rezultatów tych podzapytań oraz ich połączenie w jeden rezultat zgodnie wg operatora. Opisany tu proces jest w swej naturze rekurencyjny, tzn. zapytania są rozkładane na podzapytania itd. aż do uzyskania bądź to literałów bądź nazw (czyli zapytań elementarnych).

Operatory, które łączą zapytania dzielą się *algebraiczne* i *nie-algebraiczne*. Główna różnica między nimi to fakt czy podczas ewaluacji modyfikują ENVS czy też nie. Te pierwsze oprócz tego działają na QRES, odwołują się również do konstrukcji i operacji zachodzących na ENVS. Drugie zaś operują wyłącznie na QRES (najczęściej operacje te dotyczą jednego lub dwóch elementów).

4. Optymalizacja zapytań języka SBQL

4.1 Metoda niezależnych pod-zapytań – pierwszy sposób

Pod-zapytania języka SBQL można nazwać *niezależnymi*, jeżeli mogą być wykonywane na zewnątrz pętli iteracyjnych implikowanych przez operatory nie-algebraiczne. Takie kwerendy są godnymi uwagi, gdyż drzenią w nich największe możliwości optymalizacyjne.

Metodą niezależnych pod-zapytań nazywamy taką metodę, która optymalizują zapytania zawierające w sobie pewne niezależne części. Ogólnie rzecz ujmując polega to na przypisaniu każdej nazwie występującej w zapytaniu numeru sekcji, w której jest wiązana. Działanie takie jest motywowane następującą obserwacją: jeżeli żadna z nazw występujących w pod-zapytaniu nie jest wiązana w sekcji otwartej przez aktualnie ewaluowany operator, to wtedy pod-zapytanie może zostać wykonane wcześniej niż wskazuje na to jego tekstowa pozycja w zapytaniu.

Podsumowując metoda ta oparta jest na przepisywaniu, czyli tak modyfikuje tekstową postać zapytania (bez zmieniania jego końcowego wyniku), aby wszystkie pod-zapytania mogły wykonać się jak najwcześniej, co w rezultacie skraca czas ewaluacji. Opis tej metody można znaleźć w [Subieta 2004] oraz w rozprawie doktorskiej [Płodzień 2000].

4.1.1 Omówienie

Nawiązując do założeń języka SBQL, każdy operator nie-algebraiczny otwiera na stosie środowiskowym swoją sekcję, zaś każda nazwa jest wiązana w pewnej sekcji tego stosu. Określenie, w których sekcjach które nazwy będą wiązane nie jest możliwe, gdyż za każdym razem zapytanie może zostać wywołane przy dowolnym stanie stosu środowiskowego. Dla różnych wywołań sekcje te mogą być umieszczone na innych poziomach stosu, ponieważ liczba i zawartość dolnych sekcji (tj. sekcji, które są na stosie środowiskowym w momencie gdy rozpoczyna się ewaluacja zapytania) dla tego zapytania może być różna dla kolejnych wywołań.

W celu wyznaczenia sekcji, w których nazwy zostaną związane, wystarczy wspomóc się takim oto zabiegiem. Można przyjąć, iż wszystkie sekcje, która są na stosie w momencie rozpoczęcia ewaluacji zapytania należą do sekcji numer 1. Dzięki temu, żądane sekcje zostaną wyznaczone relatywnie w stosunku do sekcji dolnych (tj. sekcji numer 1). Uproszczenie to, nie powoduje żadnych nieścisłości, ponieważ dla wykonującego się zapytania, istotna jest informacja o tym, że niektóre z jego nazw zostaną związane w sekcjach dolnych.

Dzięki powyższemu założeniu, można ustalić dla każdej nazwy sekcję, w której jest wiązana. Dokonuje tego statyczna analiza zapytania wg następujących założeń:

- każdemu operatorowi nie-algebraicznemu przypisywany jest numer sekcji, która otwiera na stosie środowiskowym (odtąd *nos*)
- każdej nazwie przypisywane są dwa numery:
 - ilość sekcji na stosie środowiskowym, w momencie wiązania tej nazwy (odtąd *ss*)
 - numer sekcji, w której wiązana jest ta nazwa (odtąd *bl*)

Jak już zostało wspomniane, numery te są nadawane relatywnie w stosunku do dolnych sekcji stosu. Nadanie tych numerów oraz późniejsza ich analiza (patrz rozdział 4.3.5 – metoda wyznaczania niezależnych pod-zapytań) stanowi podstawę opisywanej w tym rozdziale metody.

4.1.2 Przykład

Jako przykład opisywanej tu metody niech posłuży następujące zapytanie:

[P.4.1]

*Pracownik **where** Zarobek > ((Pracownik **where** Nazwisko = "Kowalski").Zarobek)*

Zapytanie to zwraca pracowników, którzy zarabiają więcej niż pracownik o Kowalski. Należy zwrócić szczególną uwagę na pod-zapytanie:

[P.4.2]

*(Pracownik **where** Nazwisko = "Kowalski").Zarobek*

Jest ono wywoływane dla każdego pracownika istniejącego w bazie danych (w szczególności dla każdego obiektu zwróconego przez operator **where**). Jest to zbyteczne. Wystarczyłoby raz je wykonać, gdyż zwraca ono za każdym razem ten sam rezultat.

Poniższy przykład przedstawia rozkład numerów (nos, ss, bl) nadanych podczas statycznej analizy.

[P.4.3]

*Pracownik **where** Zarobek > ((Pracownik **where** Nazwisko = "Kowalski").Zarobek)*

(1,1) 2 (2,2) (2,1) 3 (3,3) 3 (3,3)

Jak widać nazwy występujące w pod-zapytaniu [P.4.2] wiązane są w sekcji nr 1 lub 3 stosu środowiskowego, zaś nie w sekcji nr 2 otwartej przez zewnętrzny operator **where**. Oznacza to, że pod-zapytanie to jest niezależne od swojego najbliższego operatora nie-algebraicznego. W szczególności żadna nazw występujących w tym pod-zapytaniu nie jest wiązana w sekcji utworzonej przez ten operator.

W związku z tym [P.4.2] można wykonać wcześniej, zanim operator **where** utworzy nową sekcję na stosie. Oczywiście rezultat całego zapytania [P.4.1] pozostanie niezmienny. W tym celu [P.4.2] zostaje wyciągnięte przed operator **where**.

[P.4.4]

*((Pracownik **where** Nazwisko="Kowalski") .Zarobek) **group as i**.(Pracownik **where** Zarobek = i)*

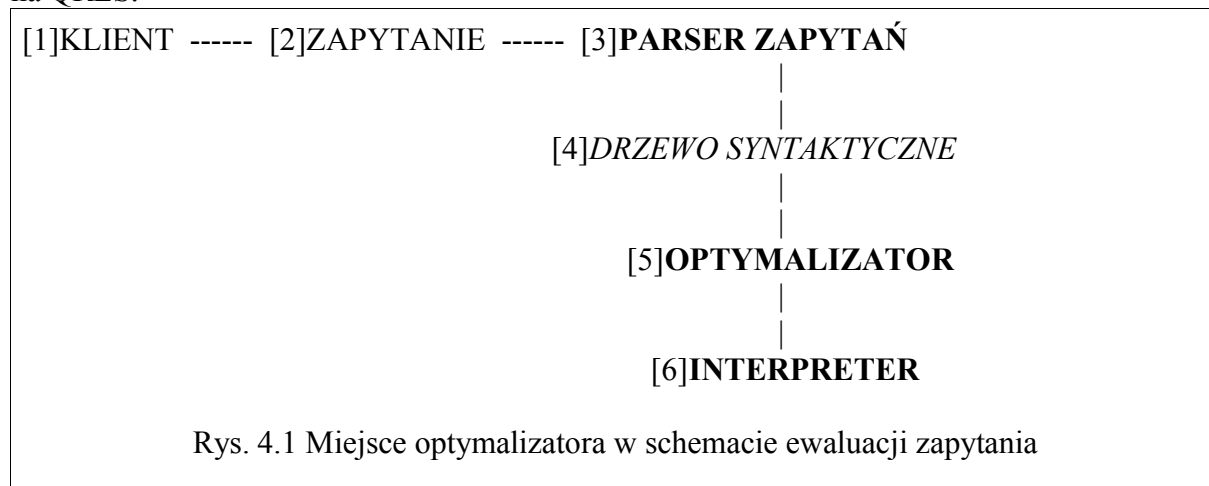
(1,1) 2 (2,2) 2(2,2) 2 (2,1) 3 (3,3) (3,2)

W zapytaniu pojawiła się nowa unikalna nazwa *i*. Pod-zapytanie jest wykonywane na początku, zaś jego wynik jest przechowywany pod nazwą *i*. Tekstowo wygląda to tak, że w stare miejsce pod-zapytania wędruje nazwa *i*, pod-zapytanie przesuwane jest na sam początek, dodawany jest operator **group as** oraz projekcji.

4.2 Budowa optymalizatora

Schematyczna budowa optymalizatora przedstawiono na rys. 4.1. Elementy **pogrubione** (punkty [3], [4], [5]) oznaczają składowe wykonawcze, zaś elementy *pochylone* (punkty [4], oraz rys. 4.2) struktury danych, które one wykorzystują lub tworzą. Klient ([1]) wywołuje

zapytanie ([2]), które jest wejściem dla parsera zapytań ([3]). W wyniku jego działania powstaje drzewo syntaktyczne ([4]). Na nim to działa optymalizator przepisujący ([5]), modyfikując je (czyli przepisując) na równoważne drzewo. Tak zmodyfikowane drzewo otrzymuje na koniec interpreter ([6]), dokonuje wykonania zapytania oraz zwraca jego rezultat na QRES.



Na poniższym rysunku przedstawiono struktury danych, z których korzysta odpowiednio optymalizator i interpreter. Zauważyć można występującą tutaj symetrię. Interpreter korzysta ze składu danych, ENVŚ oraz QRES. Działanie optymalizatora opiera się na meta-bazie (odpowiednik składu danych, ale), statycznym stosie środowiskowym S_ENVŚ (odpowiednik ENVŚ), statycznym stosie wyników S_QRES (odpowiednik QRES).

[5]OPTYMALIZATOR	[6]INTERPRETER
• <i>meta-baza</i>	• <i>skład danych</i>
• <i>statyczny stos S_ENVŚ</i>	• <i>ENVŚ</i>
• <i>statyczny stos S_QRES</i>	• <i>QRES</i>

Rys. 4.2 Porównanie struktur danych dla optymalizatora i interpretera

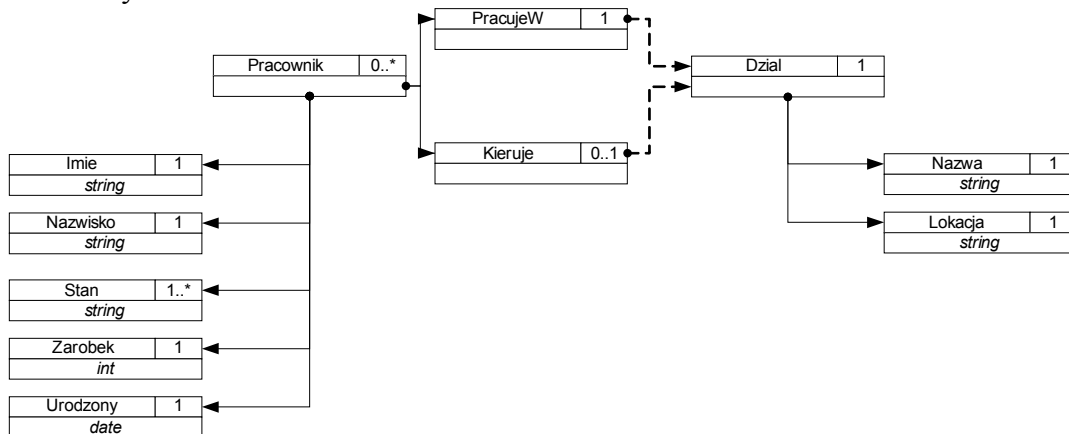
Optymalizator podczas swego działania, korzysta z informacji, które są znane podczas parsingu zapytania (nie odwołuje się do bazy danych). Konsekwencją tego jest zawieranie przez statyczne stosy (S_ENVŚ i S_QRES) sygnatur obiektów oraz operowanie na nich w sposób podobny do działania interpretera na rzeczywistych obiektach bazodanowych. Operacje zachodzące na stosach statycznych i rzeczywistych strukturach są bardzo podobne, jednakże interesuje nas nie konkretny wynik, a wyłącznie sygnatura tego wyniku; to ona jest istotna dla działania optymalizatora.

4.2.1 Meta-baza

Główną różnicą pomiędzy wykonywaniem zapytania a jego optymalizacją jest taka, że podczas optymalizacji nie korzysta się z prawdziwych obiektów (encji), lecz tylko na ich definicjach ze schematu bazy. Schemat jest przedstawiony graficznie jako graf, który modeluje statycznie skład danych. Węzły takiego grafu zawierają definicje encji z bazy danych (obiektów, atrybutów, asocjacji). Krawędzie stanowią relacje pomiędzy poszczególnymi encjami (węzłami). Metoda niezależnych pod-zapytań wyróżnia trzy takie

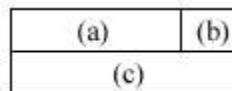
relacje:

- *is_owner_of* (ciągła linia, patrz rys. 4.3), pomiędzy bytem nadrzędnym (np. definicją obiektu) oraz bytem w stosunku do niego podrzędnym (np. definicją atrybutu, pointera lub metody)
- *points_to* (przerywana linia), pomiędzy definicją pointera a definicją obiektu, do którego taki pointer prowadzi
- *inherits_from* (podwójna linia), pomiędzy definicją klasy obiektu oraz definicją jego nad-klasy



Rys. 4.3 Graf schematu metabazy użytego w implementacji

Jak widać na poniższym rysunku pojedynczy węzeł takiego grafu składa się z trzech pól:



Rys. 4.4 Budowa węzła grafu

- (a) nazwa danej encji, np.: Prac, Dział
- (b) licznosc encji, np.: 1, 0..1, 0..*, 1..*, 5..10
- (c) typ, który w przypadku obiektów złożonych jest zestawem referencji do jego pod-obiektów; dla węzłów pointerowych oznacza referencję do definicji obiektu, do którego danych pointer prowadzi

Dodatkowo każdy węzeł grafu (nie jest to zaznaczone na powyższych rysunkach) jest identyfikowany wewnętrznym identyfikatorem (odpowiednik wew. identyfikatora obiektu czasu wykonania). W dalszej części pracy oznaczane będą jako $i_{nazwaWęzła}$, gdzie *nazwaWęzła* jest nazwą przypisaną do węzła (pole (a) na rys. 4.4), na którego wskazuje ten identyfikator.

4.2.2 Stosy statyczne

Głównym zadaniem stosów statycznych jest zasymulowanie działania stosów czasu wykonania (a tym samym zasymulowanie działania ewaluacji zapytań). Do tego celu potrzebne jest wprowadzenie pojęcia sygnatury.

- Każdy typ elementarny należy do zbioru *Sygnatura*.
- Każda referencja do węzła grafu schematu należy do zbioru *Sygnatura*.
- Jeżeli $x \in Sygnatura$, zaś $n \in \mathbb{N}$, wówczas para $n(x) \in Sygnatura$. Taki rezultat

(nazwaną sygnaturę) będziemy nazywać *statyczny binder*.

- Jeżeli $x_1, x_2, \dots \in \text{Sygnatura}$, wówczas $\mathbf{struct}\{x_1, x_2, \dots\} \in \text{Sygnatura}$.
- Jeżeli $x_1, x_2, \dots \in \text{Sygnatura}$, wówczas $\mathbf{variant}\{x_1, x_2, \dots\} \in \text{Sygnatura}$. Opcja ta jest potrzebna w przypadku, gdy opisywany byt czasu wykonania może mieć wiele sygnatur, zaś podczas kompilacji nie jesteśmy w stanie ustalić, która z nich będzie właściwa. Przykładem takiej sytuacji są kolekcje z heterogenicznymi elementami.
- Jeżeli $x \in \text{Sygnatura}$, wówczas $\mathbf{bag}\{x\} \in \text{Sygnatura}$.
- Jeżeli $x \in \text{Sygnatura}$, wówczas $\mathbf{sequence}\{x\} \in \text{Sygnatura}$.
- Zbiór *Sygnatura* nie ma innych elementów.

Sekcje na stosie S_ENVS przechowują statyczne bindery. Bindery te na stosie mogą pojawić się pod następującymi formami:

- $n(\text{typ})$: statyczny binder odpowiada sytuacji, gdy na stosie ENVNS ma pojawić się binder $n(v)$. W tym przypadku *typ* (powstały w wyniku analizy metabazy lub postaci zapytania) jest typem wartości *v*
- $n(\text{ref})$: odpowiada sytuacji, gdy na stosie ENVNS ma pojawić się binder $n(i)$, gdzie *i* jest identyfikatorem pewnego obiektu składu danych. W tym przypadku *ref* jest referencją do metabazy, czyli identyfikatorem tego jej węzła, który w czasie wykonania odpowiada identyfikatorowi *i*
- $n(\text{sygn})$: odpowiada to sytuacji, gdy na stosie ENVNS ma pojawić się binder złożony $n(\text{rezultat})$, gdzie *rezultat* jest wynikiem dowolnego zapytania.

Podobnie na stosie S_QRES: zamiast wartości przechowywany jest typ, zamiast referencji *i* podstawiamy identyfikator węzła metabazy, zaś zamiast kolekcji bag i sequence podstawiamy analogiczne słowa kluczowe z typem elementu lub identyfikatorem węzła metabazy. Poniższy przykład ilustruje te różnice (zakładając oczywiście, że i_1, i_3 są identyfikatorami obiektów Prac, zaś i_2 jest identyfikatorem obiektu Dział):

$\mathbf{struct}\{i_1, i_2\}$	$\mathbf{struct}\{i_{Prac}, i_{Dzial}\}$
$\mathbf{sequence}\{i_1, i_3\}$	$\mathbf{sequence}\{i_{Prac}\}$
$\mathbf{bag}\{\mathbf{struct}\{i_1, i_2\}, \mathbf{struct}\{i_3, i_2\}\}$	$\mathbf{bag}\{\mathbf{struct}\{i_{Prac}, i_{Dzial}\}\}$
$\mathbf{bag}\{\mathbf{struct}\{n(\text{"Tomek"}), \text{Zarobek}(2500), d(i_2)\}\}$	$\mathbf{bag}\{\mathbf{struct}\{n(\text{string}) \text{Zarobek}(\text{int}), d(i_{Dzial})\}\}$

[P.4.5]

4.2.2 Drzewo syntaktyczne

Drzewo syntaktyczne tworzone jest przez parser zapytań. Jest to struktura danych składająca się z węzłów (semantycznie istotne elementy zapytania) i połączeń między nimi. Zanim drzewo zostanie „skonsumowane” na wejściu interpretera powinno zostać zoptymalizowane.

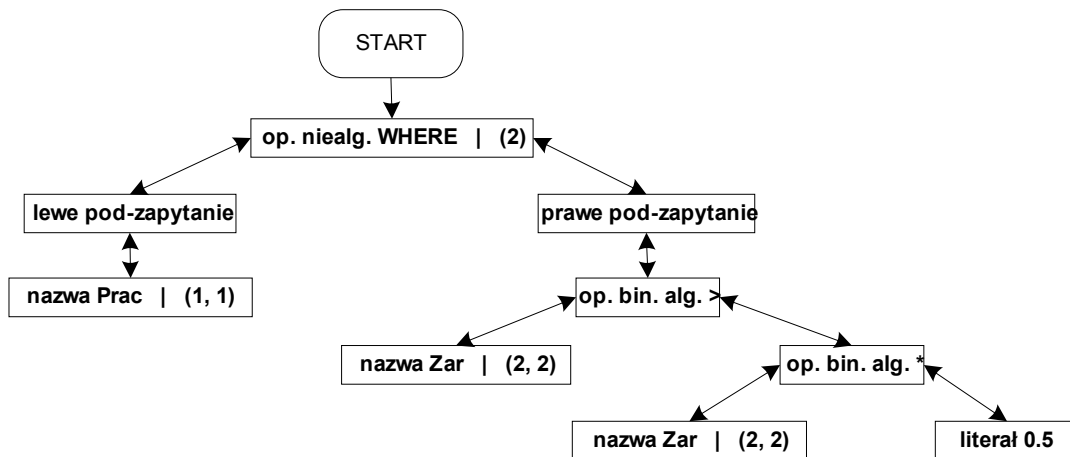
Na potrzeby metod optymalizacyjnych (w szczególności metody opartej na przepisywaniu) drzewo w sensie struktury danych powinno być zaopatrzone w metody

ułatwiającej nawigację po nim oraz modyfikację. Dodatkowo pojedynczy węzeł powinien być połączony wskaźnikiem z węzłem pod- i nad-rzędnym, tak aby odwiedzając dowolny węzeł zawsze można było się odwołać do jego „rodzica” lub „dziecka”. Jest to niezbędne, gdyż przepisywanie polega na przesuwaniu pod-drzewo, usuwaniu go lub modyfikowaniu. W rzeczywistości przekształcenie drzewa to nic innego jak zmiana powiązań wskaźnikowych pomiędzy węzłami. Przeniesienie drzewa o dowolnych rozmiarach polega na przełączeniu jednego połączenia (dwa wskaźniki).

Rysunek 4.5 przedstawia drzewo syntaktyczne dla zapytania z przykładu [P.4.6]. Należy zwrócić uwagę na pojedynczy węzeł, do którego to zostało dołożone pole z numerkami (autor wspomniał już o nich w rozdziale dotyczącym metody niezależnych podzapytań), które przechowują informację o wysokości stosu oraz numerach sekcji wiązania nazw. Dwustronne strzałki pokazują to o czym autor pisał w ostatnim akapicie – nawigowaniu po wskaźnikach.

[P.4.6]

*Pracownik where Zarobek > Zarobek * 0.5*



Rys. 4.5 Drzewo syntaktyczne powstałe dla zapytania P.4.6

4.3 Analiza statyczna zapytania

Analiza statyczna jest podstawą opisywanej statycznej optymalizacji. Wykonywana jest w czasie kompilacji i jej zadaniami są m. in.:

- sprawdzanie typów : każda nazwa sprawdzana jest ze grafem schematu bazy danych; w szczególności jeżeli dana nazwa nie może być związana, uznawana jest za błędną
- tworzenie drzew syntaktycznych zapytań, które następnie są modyfikowane przez metodę przepisywania

Analiza statyczna potrzebuje do swego działania trzech struktur danych: metabazy, statycznego ENVŚ (S_ENVŚ), statycznego QRES (S_QRES). Poniższy rysunek przedstawia odpowiadające sobie struktury czasu wykonania i statycznej analizy:

STATYCZNA ANALIZA		CZAS WYKONANIA
• <i>meta-baza</i>	< ----- >	• <i>skład danych</i>
• <i>statyczny stos S_ENVŚ</i>	< ----- >	• <i>ENVŚ</i>
• <i>statyczny stos S_QRES</i>	< ----- >	• <i>QRES</i>

Rys. 4.6 Statyczne i czasu wykonania struktury danych

Pomimo tego, iż struktury te były już opisywane w rozdziale dotyczącym budowy optymalizatora autor opisze je bardzo ogólnie:

- meta-baza : składa się głównie ze grafu schematu bazy danych; statycznie modeluje skład danych; podczas analizy graf ten nie zmienia się
- statyczny stos środowiskowy (S_ENVS) : statycznie modeluje operacje jakie zachodzą na stosie ENVVS, tj. wiązanie nazwa oraz otwieranie nowych sekcji
- statyczny stos rezultatów (S_QRES) : statycznie modeluje proces obliczania wyników pośrednich i końcowych na stosie QRES

4.3.1 Algorytm

Zanim algorytm zostanie opisany, należy podkreślić, że skład danych jest bliżej nieznanym (np. ilość instancji danego obiektu). Inaczej mówiąc nie wiemy co zrobi funkcja *eval* oraz jakie sekcje otworzy funkcja *nested*. Zatem źródłem, z którego korzysta analiza jest graf schematu bazy danych. Celem analizy jest przypisanie wszystkim operatorom niealgebraicznym numerów sekcji, które otwierają (*nos*), zaś nazwom występującym w zapytaniach par (ilość sekcji na stosie środowiskowym, numer sekcji w której wiązana jest ta nazwa) – w skrócie (*ss*, *bl*). Późniejsza analiza tych numerów pozwoli na stwierdzenie czy dany pod-zapytanie jest niezależne od najbliższego operatora niealgebraicznego i w konsekwencji, wyciągnięcie takiego pod-zapytania „przed nawias”, czyli przed ten operator niealgebraiczny. Zostało to opisane w rozdziale 4.3.5. Zasady działania algorytmu statycznej analizy są następujące:

- wykonywane operacje działają na definicjach (opisach) obiektów, które to dostarcza meta-baza
- wszelkie pod-zapytania (normalnie zwracające wiele wyników) zwracają pojedynczy rezultat jak swój wynik
- dla literału występującego w zapytaniu, sygnaturą jest jego oczekiwany typ, np. $\langle integer \rangle$, $\langle real \rangle$, $\langle boolean \rangle$; sygnatury te wstawiane są na S_QRES
- dla nazwy *n* w zapytaniu, jej sygnatura zwracana jest jako wynik przeszukiwania stosu S_ENVS w poszukiwaniu bindera $n(x)$, czyli ostatecznie $\langle x \rangle$
- jeżeli zapytanie ma postać $q \text{ as } n$, a *q* ma sygnaturę $\langle q_sygn \rangle$, to sygnatura całego zapytania to $\langle n(q_sygn) \rangle$, analogicznie dla $q \text{ group as } n$
- jeżeli zapytanie jest postaci $q_1 \Delta q_2$, gdzie Δ to operator algebraiczny, to wtedy sygnatura zapytania jest połączeniem sygnatur jego pod-zapytań (oczywiście biorąc pod uwagę reguły arytmetyki sygnatur), np. jeżeli q_1 ma sygnaturę $\langle integer \rangle$, $q_2 \langle string \rangle$, a operatorem jest $+$ to sygnaturą całego zapytania jest $\langle string \rangle$
- w przypadku operatorów reguły są analogiczne, z tym że operacje odbywają się na S_ENVS, a ich rezultatem jest otwieranie tam sekcji z rezultatem funkcji *static_nested* (opisanej w kolejnym podrozdziale)
- na początku analizy stos Q_RES jest pusty, zaś S_ENVS zawiera jedną sekcję ze statycznymi binderami do definicji obiektów startowych, np. $\mathbf{bag}\{Prac(i_{Prac})\}$
- analiza odbywa się na drzewie syntaktycznym; węzły operatorów niealgebraicznych tego drzewa powinny posiadać miejsce na zapisanie numerów (otwieranych sekcji), zaś węzły nazw wiązanych powinny posiadać miejsce na dwa numery: aktualny rozmiar stosu, oraz numer sekcji, w której jest wiązana dana nazwa.

4.3.2 Funkcja *static_nested*

W statycznej analizie odpowiednikiem funkcji *nested* jest *static_nested*. Jak już napisano wcześniej *nested* otwiera nowe sekcje na stosie ENVs i wkłada tam bindery. Podobnie działa *static_nested*, z tym że operuje na statycznym ENVs.

Zasadnicza różnica jest taka, że *static_nested* korzysta z metabazy, a nie ze składu obiektów. Dla każdego (pod)obiektu (w szczególności dla każdego węzła grafu schematu metabazy) tworzy jeden statyczny binder niezależnie od jego atrybutu liczności. Sygnatury nowych sekcji odkładanych na S_ENVS są tworzone w oparciu o sygnatury znajdujące się na S_QRES. Przykładowo, jeżeli na S_QRES znajduje się sygnatura $\{i_{Dzial}\}$ to nowo otwarta sekcja na S_ENVS będzie zawierała statyczne bindery (odnoszące się do publicznych właściwości obiektu) takie jak w przykładzie [P.4.6]. Poniżej przedstawiono główne zasady działania funkcji *static_nested*:

- dla sygnatury będącej referencją i_{Nazwa} do węzła grafu schematu opisującego obiekt *Nazwa* rezultat *static_nested* zawiera wszystkie statyczne bindery publicznych właściwości, np.:

$$static_nested(i_{Dzial}) = \{Nazwa(i_{Nazwa}), Lokacja(i_{Lokacja}), Zatrudnia(i_{Zatrudnia}), Szeff(i_{Szeff})\}$$

[P.4.6]

- dla sygnatury będącej referencją i_{Nazwa} do węzła grafu schematu opisującego pointer *Nazwa* rezultat *static_nested* zawiera statyczny binder opisu obiektu do którego prowadzi ten pointer, np.:

$$static_nested(i_{Dzial}) = \{Prac(i_{Dzial})\}$$

[P.4.7]

- dla sygnatury będącej statycznym binderem *static_nested* zwraca zbiór zawierający ten sam binder
- dla statycznych struktur:

$$static_nested(\mathbf{struct}\{s_1, \dots, s_k\}) = static_nested(s_1) \cup \dots \cup static_nested(s_k)$$

4.3.3 Funkcja *static_eval*

W statycznej analizie odpowiednikiem funkcji *eval* jest *static_eval*. To ona dokonuje statycznej analizy zapytań. Jedną z różnic jest to, że operatory nie-algebraiczne nie powodują tutaj iteracji. Poniżej przedstawiono ogólny schemat tej funkcji w postaci pseudokodu:

procedure *static_eval*(*query*)

```
{
  case query is l { ...                               // l jest literałem
}
case query is n {                                     // n jest nazwą
  ... wiązanie nazwy n na stosie S_ENVS ....
  ... odłożenie wyniku na S_QRES ...
  ... ustawienie numerów w węźle drzewa syntaktycznego (ss i bl) ...
}
case query is  $q_1 \Delta q_2$  {                          //  $\Delta$  jest oper. alg. binarnym
  q1result, q2result : TypSygnatura;                   // deklaracja sygnatur pomocniczych
```

```

static_eval(q1); // stat. analiza q1 oraz odłożenie rezultatu q1 na S_QRES
static_eval(q2); // stat. analiza q2 oraz odłożenie rezultatu q2 na S_QRES
q2result := top(S_QRES); // odczytanie i zapamiętanie wyniku q2
pop(S_QRES); // usunięcie rezultatu q2
q1result := top(S_QRES); // odczytanie i zapamiętanie wyniku q1
pop(S_QRES); // usunięcie rezultatu q1
push(S_QRES, q1result Δ q2result); // włożenie końcowego wyniku na S_QRES
}

case query is q1θ q2 { // θ jest oper. niealgebraicznym
  q1result, finalresult : TypSygnatura; // deklaracja sygnatur pomocniczych
  static_eval(q1); // stat. analiza q1 oraz odłożenie rezultatu q1 na S_QRES
  q1result := top(S_QRES);
  push(S_ENVS, static_nested(q1result)); // otwarcie nowego zakresu na S_ENVS
  .... wypełnij numer (nos) otwieranej sekcji w węźle drzewa syntaktycznego...
  static_eval(q2); // stat. analiza q2 oraz odłożenie rezultatu q2 na S_QRES
  final_result := combine(q1result, top(S_QRES)); // obliczenie końcowego wyniku
  pop(S_ENVS); // usunięcie najnowszej sekcji z S_ENVS
  pop(S_QRES); // usunięcie rezultatu q2
  pop(S_QRES); // usunięcie rezultatu q1
  push(S_QRES, final_result); // włożenie końcowego wyniku na S_QRES
}
} // koniec static_eval

```

4.3.4 Przykład

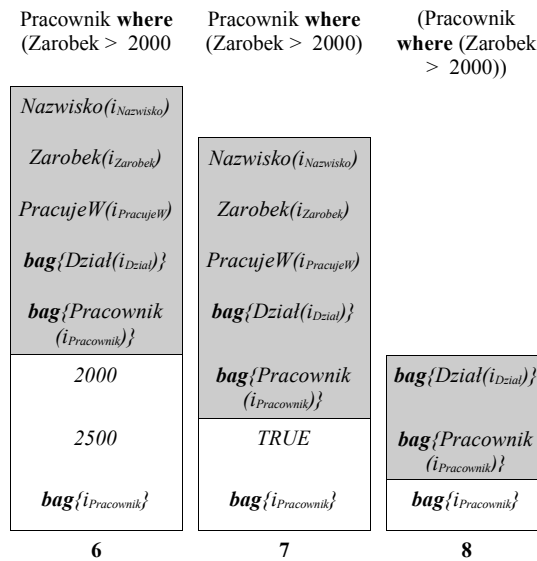
Rys. 4.7 przedstawia przykładową analizę statyczną zapytania ([P.4.8]) rozpisaną na poszczególne kroki (niektóre zostały pominięte) wraz ze stanami stosów.

(Pracownik **where** (Zarobek > 2000))

[P.4.8]

Stos S_ENVS został przedstawiony na szaro, zaś Q_RES na biał.

Stan początkowy	Pracownik	Pracownik where	Pracownik where (Zarobek > 2000)	Pracownik where (Zarobek > 2000)
<div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Dzial(<i>i</i>_{Dzial})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Pracownik(<i>i</i>_{Pracownik})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> pusty </div>	<div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Dzial(<i>i</i>_{Dzial})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Pracownik(<i>i</i>_{Pracownik})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{i_{Pracownik}} </div>	<div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Dzial(<i>i</i>_{Dzial})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Pracownik(<i>i</i>_{Pracownik})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{i_{Pracownik}} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> Nazwisko(<i>i</i>_{Nazwisko}) Zarobek(<i>i</i>_{Zarobek}) PracujeW(<i>i</i>_{PracujeW}) </div>	<div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> Nazwisko(<i>i</i>_{Nazwisko}) Zarobek(<i>i</i>_{Zarobek}) PracujeW(<i>i</i>_{PracujeW}) <i>bag</i>{Dzial(<i>i</i>_{Dzial})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Pracownik(<i>i</i>_{Pracownik})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{i_{Zarobek}} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{i_{Pracownik}} </div>	<div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> Nazwisko(<i>i</i>_{Nazwisko}) Zarobek(<i>i</i>_{Zarobek}) PracujeW(<i>i</i>_{PracujeW}) <i>bag</i>{Dzial(<i>i</i>_{Dzial})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{Pracownik(<i>i</i>_{Pracownik})} </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> 2500 </div> <div style="border: 1px solid black; padding: 5px; background-color: #e0e0e0;"> <i>bag</i>{i_{Pracownik}} </div>
1	2	3	4	5



Rys. 4.7 Statyczna analiza krok po kroku dla zapytania P.4.8

Krok 1: stan początkowy przed rozpoczęciem ewaluacji

Krok 2: wiązana jest nazwa *Pracownik* co powoduje wypełnienie S_QRES oraz przypisanie nazwie *Pracownik* pary numerków <ss: 1, bl: 1>

Krok 3: operator nie-algebraiczny **where** otwiera na S_ENVS sekcję o numerze 2, wobec czego do węzła drzewa syntaktycznego reprezentującego ten operator jest wpisany numer <nos: 2>

Krok 4: wiązana jest (na drugim poziomie) nazwa *Zarobek* co powoduje wypełnienie S_QRES oraz przypisanie nazwie *Zarobek* pary numerków <ss: 2, bl: 2>

Krok 5: operator algebraiczny > powoduje dereferencję przez co na stosie S_QRES pojawia się wartość „2500”

Krok 6: na stosie S_QRES pojawia się literał „2000”

Krok 7: poprzez operator algebraiczny > zachodzi porównanie „2500 > 2000” co powoduje odłożenie na S_QRES wyniku tego porównania jako TRUE

Krok 8: stan po zakończeniu ewaluacji zapytania

4.3.5 Wyznaczanie niezależnych pod-zapytań

Wyznaczanie niezależnych pod-zapytań opiera się na trzech procedurach, które korzystają z „owoców” analizy statycznej w postaci nadanych węzłom drzewa numerów (zostało to opisane w rozdziale 4.1 razem z ogólnym opisem metody). Te procedury to:

- **void** *optymalizujDrzewo* (**węzeł** drzewo)
- **węzeł** *jestNiezależne* (**węzeł** operNiealg)
- **void** *przenieśPoddrzewo* (**węzeł** operNiealg, **węzeł** podDrzewo)

Pierwsza z wymienionych procedur *optymalizujDrzewo* przebiega drzewo syntaktyczne dopóki nie natknie się na operator niealgebraiczny. Po znalezieniu takowego uruchamiana jest druga z procedur *jestNiezależne*, która na wejściu dostaje węzeł tego operatora niealgebraicznego, zaś na wyjściu zwraca węzeł największego pod-drzewa będącego w obszarze działania tego operatora, lecz od niego niezależnego. Jeżeli węzeł takiego pod-drzewa nie zostanie odnaleziony *optymalizujDrzewo* działa dalej dla następnego operatora niealgebraicznego.

W przeciwnym wypadku trzecia z procedur *przenieśPoddzewo* dokonuje modyfikacji drzewa syntaktycznego – fizycznie przenosi znalezione poddrzewo PRZED operator niealgebraiczny (wyłącza przed nawias). Odbywa się to wg następującej reguły

$$q_1 \theta q_2(q_3) \rightarrow (q_3 \mathbf{group\ as\ nowaNazwa}) . (q_1 \theta q_2(nowaNazwa))$$

, gdzie θ jest operatorem niealgebraicznym, $q_2(q_3)$ jest zapytaniem, którego składową jest pod-zapytanie q_3 , przy czym q_3 znajduje się bezpośrednio pod działaniem operatora θ ; *nowaNazwa* jest nową, unikalną nazwą nadaną w sposób automatyczny. Przykłady [P.4.3] oraz [P.4.4] ilustrują tę przemianę.

Procedura *jestNiezależne* w swym działaniu wykorzystuje numery nadane podczas analizy statycznej. Przebiega ona po drzewie zaczynając od największego pod-drzewa z prawej strony operatora niealgebraicznego i następnie schodząc rekurencyjnie w dół drzewa. Procedura ta ignoruje prawe pod-zapytania innych operatorów nie-algebraicznych, ale nie ignoruje ich lewych pod-zapytań. Porównuje ona numery przypisane nazwom z numerami przypisanymi operatorom niealgebraicznym i na tej podstawie wyznacza niezależne pod-zapytania. Jeżeli *nos* różni się od *bl* to dane pod-zapytanie jest niezależne od operatora niealgebraicznego. W takim przypadku procedura zwraca węzeł takiego pod-drzewa. Analizie nie podlegają pod-drzewa składające się z jednego literału lub z jednej nazwy (bo nie można ich zoptymalizować), z wyjątkiem, kiedy nazwa oznacza wywołanie funkcji lub perspektywy.

Gdy procedura *przenieśPoddzewo* zakończy swe działanie drzewo syntaktyczne jest zmienione. Dalsze wywoływanie rekurencyjnych procedur pozbawione jest sensu. Należy wywołać procedury *static_eval* oraz *optymalizujDrzewo* i zacząć optymalizację na zmienionym drzewie. To ponowne wywołanie zapewni, że dane pod-zapytanie zostanie przesunięte przed wszystkie operatory nie-algebraiczne, od których jest niezależne.

Proces ten powtarzamy do momentu gdy po wywołaniu *optymalizujDrzewo* nie nastąpi wywołanie procedury *przenieśPoddzewo*.

5. Implementacja

5.1 Metabaza

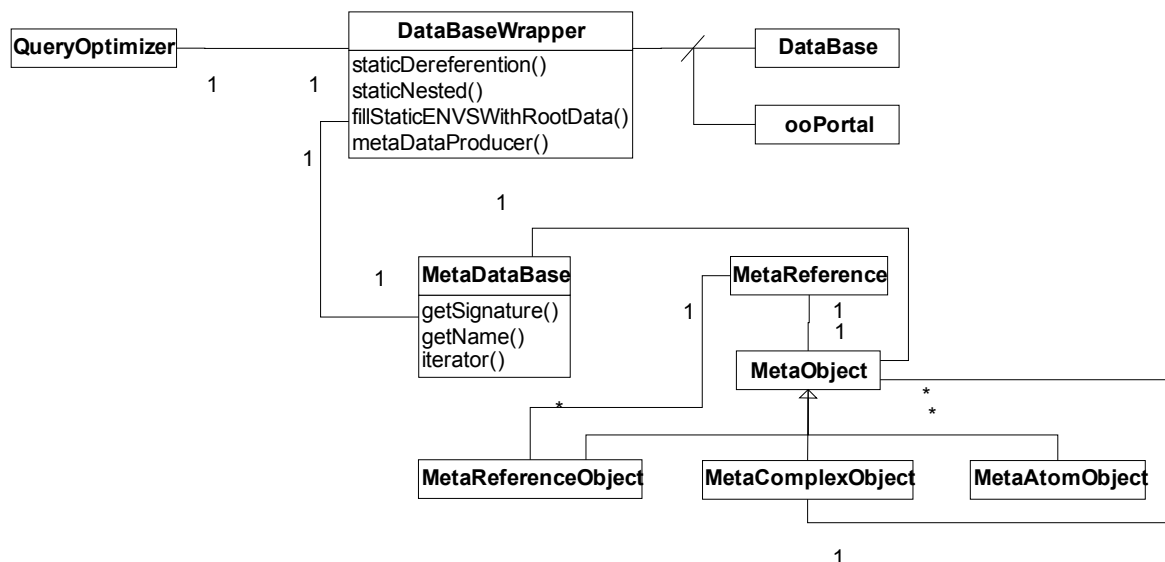
W celu stworzenia prototypu optymalizatora koniecznym było zaimplementowanie metabazy, tak aby optymalizator miał z czego korzystać – schematu metabazy. Potrzeba taka była uwarunkowana budową ooPortal'u, który to nie przekazuje informacji o obiektach (nie można ich uzyskać w prosty sposób); nawet jeżeli takowe są to bardzo ubogie. Interfejs do ich odzyskiwania tych danych jest skomplikowany i sprawia wrażenie archaicznego.

Metabaza została stworzona specjalnie na użytek optymalizatora: stosowanie metod *staticNested*, *staticEval* wymusza specyficzną konstrukcję metadanych, tak aby nie trzeba było przeszukiwać całej bazy danych w poszukiwaniu skomplikowanych zależności (atrybuty dla nazwy). Autor założył, że wszystko ma referencję. Dzięki temu można dostać się do każdego obiektu (złożonego, atomowego, referencyjnego). Tytułem porównania w SQL'u nie ma metadanych. Aby je mieć trzeba tworzyć dodatkowe tabele. Dostęp do metadanych odbywa się poprzez wysyłania zapytań.

Po zaznajomieniu się z prototypem SBQL'a okazało się, że zastałem gotowe struktury do manipulacji (komunikacja) ze składem: *DataBaseWrapper* połączony z *Database* i *ooPortal*'em. Oczywiście trzeba je było przystosować do pracy z optymalizatorem. Owocem tego są cztery dodatkowe metody w klasie *DataBaseWrapper*:

- *staticDereferention*
- *staticNested*
- *fillStaticENVSWithRootData*
- *metaDataProducer*

Rysunek 5.1 przedstawia strukturę metabazy:



Rys. 5.1 Struktury tworzące metabazę

Klasa *MetaDataBase* opisuje metabazę oraz zarządza wszystkimi metaobiettami (np. *MetaAtomObject*, *MetaComplexObject*). Wszystkie one dziedziczą z *MetaObject*, dzięki czemu można nimi w łatwy sposób zarządzać. Ponieważ dla każdego *MetaObject*'u tworzony jest oddzielny obiekt *MetaReference* (unikalna referencja) wszystkie obiekty są jednoznacznie identyfikowalne. *MetaComplexObject* może zawierać dowolną ilość metaobiettów, w

związku z czym zagnieżdżanie dowolnej liczby obiektów jest możliwe (w przeciwieństwie do ooPortal).

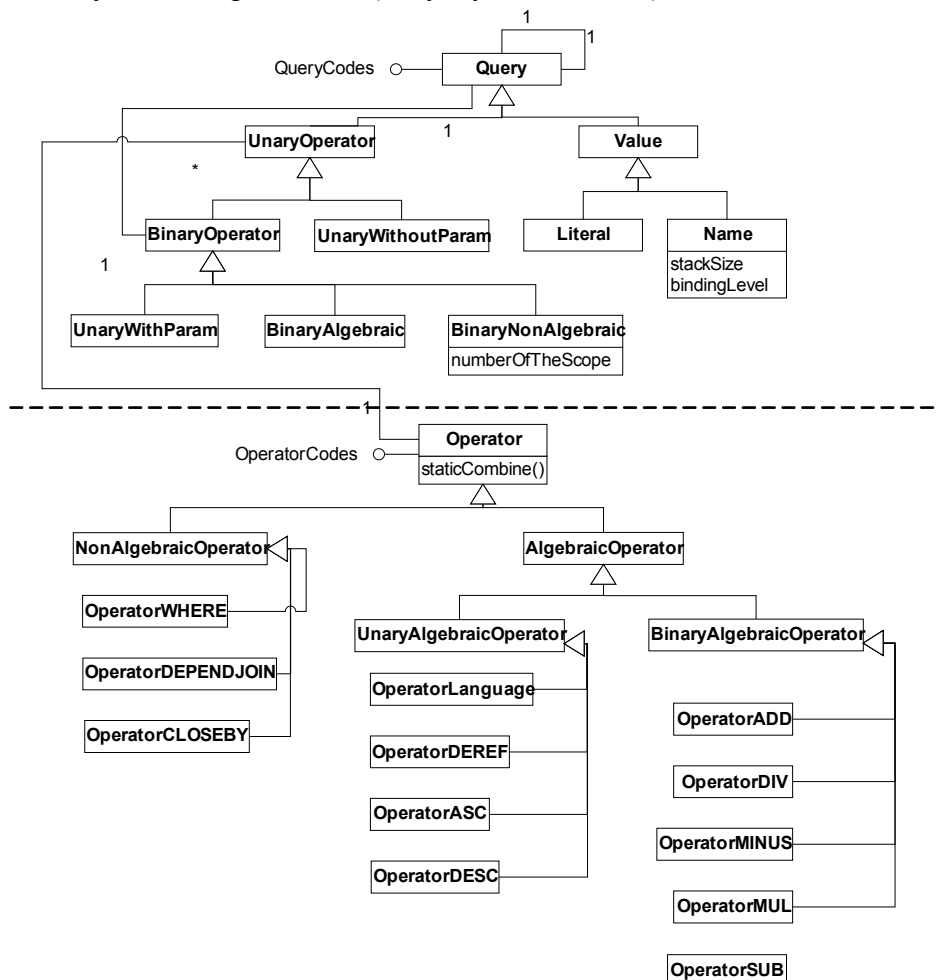
QueryOptimizer łączy się bezpośrednio z *DataBaseWrapper*, który to następnie łączy się z metabazą. Do komunikacji między optymalizatorem a wrapperem służą cztery wspomniane powyżej metody.

Największą ułomnością tego rozwiązania jest wkompilewanie metadanych, które to następuje przez wywołanie metody *metaDataProducer*, w której to tworzone są metadane. Gdy zmieni się skład trzeba przekompilować program z nowymi metadanymi. Ponieważ ooPortal nie udostępnia takich informacji w prosty sposób, w związku z czym metadane muszą być w jakiś sposób tworzone, opisywać stan i skład obiektów w bazie danych.

5.2 Drzewo

Klasy opisujące drzewo syntaktyczne i operatory zaimplementowane w prototypie SBQL'a widoczne są na rys. 5.2. Tutaj podstawową modyfikacją było dodanie dwóch zmiennych do klasy *Name*: *stackSize* oraz *bindingLevel*, zaś do klasy *BinaryNonAlgebraic* zmiennej *numberOfScope*. Wszystkie one są potrzebne algorytmowi statycznej analizy.

Kolejna modyfikacja dokonana się w operatorach. W każdym z operatorów należało zaimplementować metodę zdefiniowaną w klasie *Operator*: *staticCombine*. Odpowiada ona za prawidłowe wyliczenia operatorów (liczy wynik działania).



Rys. 5.2 Struktury definiujące drzewo syntaktyczne

5.3 Optymalizator

QueryOptimizer został zrobiony tak, że operuje na drzewie, przechodzi je (wszystko w sposób rekurencyjny). Zgodnie z budową drzewa dla konkretnych gałęzi wykonuje odpowiednie operatory i działania. Klasa *QueryOptimizer* zawiera metody: <wymienić>.

- *initWithMetadata*
- *optimize*
- *optymalizujDrzewo*
- *jestNiezalezne*
- *przeniesPoddrzewo*
- *staticEval*
- *staticNested*

Pierwsza z nich *initWithMetadata* wypełnia statyczny stos środowiskowy danymi korzeniowymi. Następnie optymalizator uruchamia funkcję *optimize* na drzewie, które zostało wcześniej stworzone przez parser. Przed właściwą optymalizacją następuje faza analizy statycznej (*staticEval*), podczas której nadawane są numerki węzłom drzewa. Trzy metody: *optymalizujDrzewo*, *jestNiezalezne*, *przeniesPoddrzewo* są główną siłą sprawczą optymalizacji. Optymalizator z metabazą komunikuje się poprzez *DataBaseWrapper*.

QueryOptimizer
initWithMetadata()
optimize()
optymalizujDrzewo()
jestNiezalezne()
przeniesPoddrzewo()
staticEval()
staticNested()

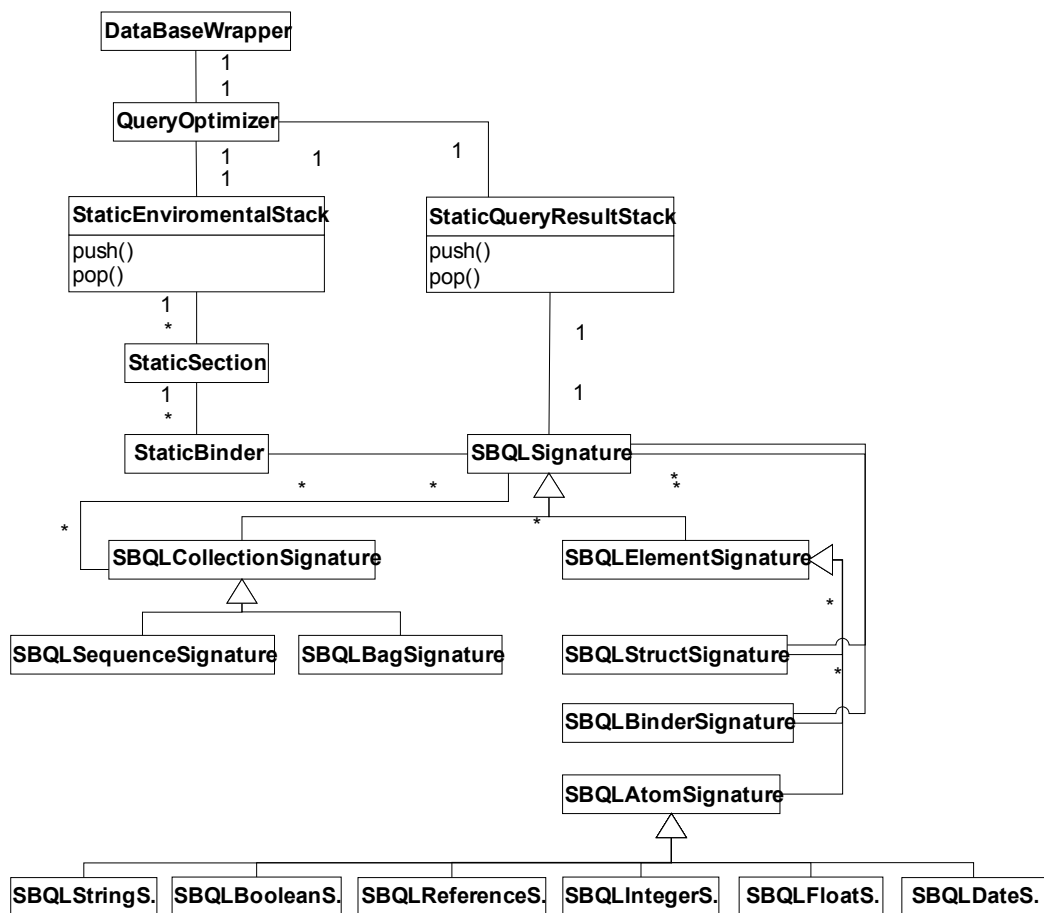
Rys. 5.3 Najważniejsze metody klasy *QueryOptimizer*

5.3.1 Sygnatury

Statyczna analiza wymaga odpowiedników wszystkich rezultatów czasu wykonania. Tą rolę sprawują sygnatury. Opisują one zależności między rezultatami (np.: *int + int = int*; *int + float = float*). Działają całkowicie w oparciu o metabazę, co oznacza że dane, które przechowują są odpowiednikami typów danych w składzie.

5.3.2 Stosy

Stosy są to struktury niezbędne do poprawnej prac optymalizatora. *S_QRES* (statyczny stos rezultatów) przechowuje rezultaty zapytań, natomiast na stosie *S_ENVS* odkładane są sekcje, przechowujące statyczne bindery.



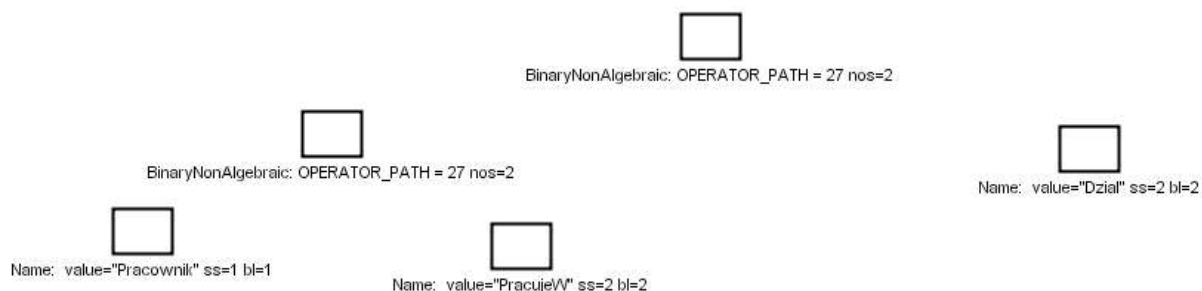
Rys. 5.4 Stosy Q_ENVS, S_QRES oraz struktury powiązane

5.3.4 Przykłady

Poniżej autor przedstawił wyniki działania optymalizatora zapytań. Dla pojedynczego zapytania zwracane są w konsolowym okienku jego dwie postaci: oryginalna oraz zoptymalizowana. Ta druga zawiera numery (w nawiasach []) nadane nazwom i operatorom niealgebraicznym.

Zanim przejdziemy do konkretnych przykładów proszę spojrzeć na poniższy rysunek. Przedstawia on wygląd drzewa po ewaluacji zapytania P.5.1. Daje się zauważyć nadane numerki dla nazw i operatorów algebraicznych.

[P.5.1] *Pracownik.PracujeW.Dzial*



Rys. 5.1 Zrzut ekranu pokazujący wyniki drzewo oraz analizę statyczną zapytania P.5.1

6. Podsumowanie

Autor zrealizował postawiony przed sobą cel. Optymalizator zapytań języka SBQL został zaimplementowany oraz przetestowany. Najważniejszy wniosek wypływający z tej pracy to taki, że optymalizacja zapytań oparta na idei statycznej analizy i przepisywaniu jest w pełni wykonalna i implementowalna. Implementacja prototypu optymalizatora wykazała również, że mechanizm statycznej analizy jest bardzo użyteczny w gromadzeniu informacji, które są niezbędne do zoptymalizowania zapytania. Praca ta pokazuje również, że pomysł aby optymalizację zapytania przeprowadzać poprzez modyfikację jego drzewa syntaktycznego i nie używać żadnych innych pośrednich form jest jak najbardziej poprawny.

W rozdziale 3 zostało zaprezentowane podejście stosowe. Jego uniwersalność polega na tym, że uwzględnia ono bardzo ogólnie model obiektowy, który zawiera np. obiekty złożone, kolekcje atrybutów, klasy, metody oraz dziedziczenie. Ponadto tworzy ono prostą i uniwersalną podstawę semantyczną dla języków zapytań OQL'o podobnych połączoną z pojęciami programistycznymi takimi jak: metody, procedury, perspektywy, przekazywanie parametrów, co ostatecznie pozwała nam na precyzyjne opisanie semantyki SBQL.

Oprócz statycznej analizy (rozdział 4.3) drugim głównym pojęciem, które zostało omówione w tej pracy była metoda niezależnych pod-zapytań (rozdział 4.1). Jest ona najbardziej interesującą metodą spośród metod opartych na przepisywaniu (pozostałe metody Pojedyncze niezależnie pod-zapytanie może być dowolnie złożone – może zawierać funkcje agregacyjne, wołania metod, złączenia, kwantyfikatory. Zewnętrzne zapytanie, które zawiera takie niezależne pod-zapytanie może być również złożone. Głównym powodem jest fakt, że przy podejściu stosowym semantyka wszystkich niealgebraicznych operatorów (where, join, projekcji itp.) jest taka sama oczywiście z wyjątkiem obliczania końcowego wyniku.

Podczas implementacji autor napotkał następujące trudności:

- połączenie optymalizatora z prototypem języka SBQL (klasyczny przykład tworzenia oprogramowania na bazie innego, co zawsze stwarza problemy)
- tworzenie metadanych – konieczność wkompiłowania metabazy w program
- szeroko występująca rekursja, wystarczająco skomplikowana, aby stracić panowanie nad programem

W przyszłości prototyp ten zostanie rozbudowany o pozostałe metody oparte na przepisywaniu połączone razem z modelem kosztów, co stanowić będzie razem pełen algorytm optymalizacyjny. Można by się również pokusić o

- jednoczesną optymalizację wielu zapytań
- dynamiczną optymalizację zamiast statycznej
- indeksy ścieżkowe
- czynniki redukcji dla bardziej złożonych predykatów selekcji
- bardziej realistyczną dystrybucję wartości atrybutów (t.j. inną niż standardowa)

7. Literatura

- [Subieta 2004] K. Subieta “Teoria i konstrukcja obiektowych języków zapytań”
- [Subieta 2002] K. Subieta “Podstawy semantyczne języków zapytań”
- [Płodzień 2000] J. Płodzień “Optimization Methods in Object Query Languages”
- [ICONS_1] Rodan Icons <http://www.rodan.pl/badania/icons/>
- [ICONS_2] Rodan Icons <http://rodan.pl/pdf/icons.pdf>
- [OOPortal] Rodan ooPortal http://rodan.pl/pdf/officeobjects_portal.pdf
- [RR+JG 99] Raghu Ramakrishnan i Johannes Gehrke „Database Management Systems, 2nd edition”

Dodatek A

API (application programming interface)	- interfejs programistyczny dla aplikacji
ASR (access support relation)	- relacja wsparcia dostępu
ENVS (enviromental stack)	- stos środowiskowy
ooPortal	- ooPortal OfficeObjects® Portal
OQL (object query language)	
QRES (query result stack)	- stos rezultatów
S_ENVS (static enviromental stack)	- statyczny stos środowiskowy
S_QRES (static query result stack)	- statyczny stos rezultatów
SBA (stack based approach)	- podejście stosowe
SBQL (stack based query language)	- język oparty na podejściu stosowym
SQL (structured query language)	- strukturalny język zapytań
SZBD	- systemy zarządzania bazami danych
XML (extensible markup language)	