

**Uniwersytet Warszawski**  
Wydział Matematyki, Informatyki i Mechaniki

**Maciej Maciak**

Nr albumu: 201067

# **Aktualizowalne perspektywy w semistrukturalnej bazie danych**

**Praca magisterska  
na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem  
**dra Krzysztofa Stencła**  
Instytut Informatyki

Październik 2006

## **Oświadczenie kierującego pracą**

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

Data

Podpis kierującego pracą

## **Oświadczenie autora (autorów) pracy**

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

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

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

Data

Podpis autora (autorów) pracy

# Spis treści

<b>Wprowadzenie</b>	3
<b>1. Stack Based Query Language</b>	5
1.1. Założenia	5
1.2. Składnia	5
1.3. Model składu obiektów	6
1.4. Przestrzeń wyników zapytań	6
1.5. Konstrukcje imperatywne	9
1.5.1. Create	10
1.5.2. Insert	11
1.5.3. Update	12
1.5.4. Delete	12
1.6. Przykłady zapytań	13
<b>2. Perspektywy w SBQL</b>	15
2.1. Definicja perspektywy	15
2.2. Konstrukcja wirtualnych obiektów	16
2.3. Wartość wirtualnego obiektu	17
2.3.1. Funkcja nested dla wirtualnych obiektów	17
2.3.2. Dereferencja na wirtualnym identyfikatorze	18
2.3.3. Proponowane zmiany w funkcji nested	18
2.4. Operacje na wirtualnych identyfikatorach	19
2.4.1. Aktualizacja wirtualnych obiektów	19
2.4.2. Kasowanie wirtualnych obiektów	20
2.4.3. Wstawianie podobiektów do wirtualnego obiektu	20
2.4.4. Tworzenie wirtualnych obiektów	20
2.5. Materializacja wirtualnych obiektów	20
2.5.1. Wstawianie wirtualnego obiektu do wnętrza innego obiektu	21
2.5.2. Tworzenie wskaźnika na wirtualny obiekt	21
2.6. Perspektywy referencyjne	22
2.7. Przykłady perspektyw	23
<b>3. Prototyp perspektyw dla SBQL</b>	25
3.1. Najważniejsze moduły	25
3.1.1. Interfejs	25
3.1.2. Parser	25
3.1.3. Kontroler typów	26
3.1.4. Optymalizator zapytań	26

3.1.5. Generator kodu . . . . .	26
3.1.6. Interpreter . . . . .	26
3.2. Zakres funkcjonalności . . . . .	26
3.2.1. Perspektywy . . . . .	27
3.3. Przykłady działania systemu ODRA . . . . .	28
<b>4. Podsumowanie . . . . .</b>	<b>31</b>
<b>A. Opis dołączonej płyty . . . . .</b>	<b>33</b>
A.1. Zawartość płyty . . . . .	33
A.2. Instalacja i uruchamianie . . . . .	34
<b>Bibliografia . . . . .</b>	<b>35</b>

## **Streszczenie**

Praca składa się z dwóch części. Poniższe opracowanie stanowi część pierwszą. Przedstawiam w nim teorię dotyczącą perspektyw dla języka zapytań SBQL, oraz staram się rozwiązać część związanych z nimi problemów. Część druga to praca implementacyjna mająca na celu rozszerzenie istniejącego prototypu języka SBQL o aktualizowalne perspektywy, a tym samym praktyczne potwierdzenie przedstawionej teorii.

## **Słowa kluczowe**

SBQL, perspektywa, język zapytań

## **Dziedzina pracy (kody wg programu Socrates-Erasmus)**

Informatyka

## **Klasyfikacja tematyczna**

H. Information Systems

H.2 DATABASE MANAGEMENT (E.5)

H.2.4 Systems

## **Tytuł pracy w języku angielskim**

Updatable views in semistructural database



# Wprowadzenie

Praca ma na celu przedstawienie dotychczasowego podejścia do perspektyw w SBQL. Identyfikuję w niej elementy koncepcyjne, które budzą kontrowersje. Przedstawiam nowy model, który wprowadza zmiany mające na celu uniknięcie przedstawionych problemów. Praca łączy się również z rozbudową projektu programistycznego - rozszerzam istniejący prototyp systemu ODRA tak, aby wspierał najważniejsze konstrukcje imperatywne języka SBQL, oraz wszystkie przedstawione w pracy aspekty dotyczące perspektyw.

Rozdział pierwszy opisuje składnię języka SBQL i sposób wyliczania zapytań. Przedstawiam w nim także konstrukcje imperatywne. W rozdziale drugim przedstawiam dotychczasową teorię dotyczącą perspektyw dla SBQL. Przy tej okazji staram się zidentyfikować i rozwiązać ewentualne problemy, które w niej występują. Rozdział trzeci opisuje prototypową implementację przedstawionej w poprzednich rozdziałach teorii.





# Rozdział 1

## Stack Based Query Language

### 1.1. Założenia

Stack Based Query Language czyli SBQL jest to język zapytań opierający się na środowisku stosowym. Zapytania są w nim traktowane jako wyrażenia języka programowania i posiadają wszystkie cechy, które się z tym łączą. Oznacza to, że mamy tutaj do dyspozycji takie rzeczy jak stałe, zmienne, wszelkiego rodzaju operatory algebraiczne, oraz iteratory niezbędne do obsługi kolekcji.

Jednym z podstawowych zagadnień w języku SBQL, jest kwestia wiązania nazw - każda z nich przyjmuje wartość zgodną z regułami zakresu ustalonymi przez stos środowiskowy. Stos środowiskowy natomiast jest niczym innym jak stosem wywołań znanym dobrze z języków programowania, z tą różnicą, że pozwalamy tutaj na wystąpienie odwołań do jednego obiektu w różnych sekcjach stosu. Ze względu na tą różnicę stos środowiskowy musi być oddzielony od składu obiektów.

Kolejną ważną kwestią jest postać wyników zapytań - w SBQL nie będą to same obiekty, lecz referencje do nich, nazwy, wartości atomowe, bądź złożone struktury składające się z wyżej wymienionych.

### 1.2. Składnia

W języku SBQL występują następujące wyrażenia:

- literały odpowiadające wartościom atomowym - napisom, czy liczbom
- nazwy obiektów, lub zmiennych
- dwa zapytania połączone operatorem binarnym, takim jak  $+$ ,  $-$ ,  $join$ ,  $where...$
- aplikacja operatora unarnego do zapytania - np.  $-a$ ,  $sum(a)...$

W obliczaniu zapytań języka SBQL poruszamy w dwóch różnych, rozłącznych przestrzeniach wartości. Są to: przestrzeń zdefiniowana przez model składu obiektów i przestrzeń wyników zapytań. Dodatkowo używamy stosu środowiskowego, oraz stosu wyników zapytań. Wszystkie te składniki są zdefiniowane poniżej.

### 1.3. Model składu obiektów

Dla SBQL został opracowany szereg modeli składu obiektów. Podstawowy z nich - M0 obejmuje zagnieżdżone struktury danych wraz z powiązaniem między nimi. Kolejne modele rozszerzają go o klasy, dziedziczenie i wielodziedziczenie, dynamiczne role oraz wprowadzają własność hermetyzacji.

Dodatkowe możliwości, które dają rozszerzenia modelu M0 są całkowicie ortogonalne z przedstawionymi w dalszej części perspektywami, z tego powodu zajmę się jedynie modelem podstawowym, który następnie rozbuduje o szereg konstrukcji imperatywnych, oraz o perspektywy.

W modelu M0 każdy obiekt jest trójką  $(i, n, v)$ , gdzie  $i$  jest unikalnym identyfikatorem obiektu,  $n$  jego nazwą, zaś  $v$  - wartością.

Wartością  $v$  obiektu może być:

- wartość atomowa - o takim obiekcie mówimy jako o obiekcie atomowym
- identyfikator innego obiektu - taki obiekt nazywamy referencyjnym
- struktura składająca się z podobiektów - wtedy jest to obiekt złożony

Skład obiektów jest złożony ze zbioru obiektów  $S$ , oraz zbioru identyfikatorów obiektów startowych  $R$ .

Dodatkowo zakładamy, że:

- $S$  zachowuje integralność referencyjną (nie ma wiszących wskaźników)
- każdemu identyfikatorowi z  $R$  odpowiada jakiś obiekt z  $S$

Powyższe reguły stanowią o tym, że możliwe jest budowanie dowolnie zagnieżdżonych struktur danych, w których mogą występować referencje wskazujące na dowolne istniejące obiekty.

### 1.4. Przestrzeń wyników zapytań

Przestrzeń  $P$  wyników zapytań języka SBQL zawiera następujące rodzaje wartości q-value:

- wartości atomowe
- bindery postaci  $n(v)$ , które wiążą z nazwą  $n$  wartość  $v \in P$
- referencje na obiekty należące do składu obiektów
- zbiory wartości oznaczane jako  $bag(v_1, v_2, \dots)$ , gdzie  $v_n \in P$
- struktury wartości oznaczone jako  $struct(v_1, v_2, \dots)$ , gdzie  $v_n \in P$

Tak więc widać, że przestrzeń wyników jest znacząco różna od zbioru obiektów przechowywanych w składzie. Najważniejszą różnicą jest możliwość konstruowania zbiorów dowolnych wartości jako wyników zapytań, a nie tylko struktur składających się z identyfikatorów podobiektów, jak to ma miejsce w składzie.

## Stos środowiskowy

Stos środowiskowy składa się z następujących po sobie sekcji zawierających bindery z przestrzeni wyników zapytań.

W najniższej sekcji stosu znajdują się bindery do referencji na wszystkie obiekty startowe ze składu. Podczas wyliczania zapytania mogą powstać nowe sekcje, w których znaleźć się mogą bindery do dowolnych wyników, zaś po zakończeniu wyliczania zapytania stos powinien wrócić do stanu pierwotnego, czyli zawierać tylko najniższą sekcję.

## Stos wynikowy

Stos wynikowy przechowuje wyniki zapytań. Przed rozpoczęciem wyliczania zapytania jest pusty, w jego trakcie jest rozbudowywany o kolejne sekcje zawierające wyniki podzapytań, które następnie są konsumowane przez operatory budujące zapytania złożone (operatory unarne i binarne), po ewaluacji powinien tam zostać tylko jeden element - zwrócony wynik.

## Ewaluacja zapytania

Do wyliczenia zapytania służy funkcja *eval*. W celu jej przedstawiania oznaczmy przez QRes stos wynikowy, ENVs - stos środowiskowy, Store - skład obiektów.

Dla literalów i nazw funkcja eval ma następującą postać:

```
function eval(q) {  
  if (q jest literałem l) {  
    QRes.push (wartość atomowa(l));  
    return;  
  }  
  if (q jest nazwą n) {  
    QRes.push(ENVs.bind(n));  
    return;  
  }  
}
```

Funkcja *bind(name)* wywoływana na stosie środowiskowym przechodzi przez kolejne jego sekcje, aż natrafi na taką, w której występuje binder *name(v)*. Wtedy przerywa poszukiwania i zwraca zbiór zawierający wartości *v* wszystkich binderów *name(v)*, które należą do znalezionej sekcji stosu. W razie niepowodzenia zwraca zbiór pusty.

Ze względu na sposób wyliczania zapytań operatory służące do ich budowy dzielimy na algebraiczne i niealgebraiczne. Operatory algebraiczne to te, które nie modyfikują stosu środowiskowego, natomiast operatory niealgebraiczne modyfikują go.

Do operatorów algebraicznych należą: +, -, \*, /, mod, i inne.

Funkcja eval dla algebraicznego binarnego operatora  $\delta$  zachowuje się w następujący sposób:

```

function eval(q1 δ q2){
  wynik1, wynik2: q-value;
  eval(q1);
  wynik1 := QRES.pop();
  eval(q2);
  wynik2 := QRES.pop();
  QRES.push(δ'(wynik1, wynik2));
}

```

W podanym schemacie  $\delta'$  oznacza wyliczenia operatora  $\delta$  dla podanych wyników podzapytań. Dla przykładu jeśli za  $\delta$  przyjmiemy  $+$  to odpowiednia funkcja będzie miała postać:

```

function δ'(res1 : q-value, res2 : q-value){
  tmp1, tmp2 : q-value;
  result : q-value;
  result = new bag();
  if (res1 jest bagiem) tmp1 = res1;
  else tmp1 = new bag(res1);
  if (res2 jest bagiem) tmp2 = res2;
  else tmp2 = new bag(res2);
  foreach(single1 : q-value in tmp1) {
    while (single1 jest binderem name(v)) single1 := v;
    if (single1 jest referencją i) single1 := Dereference(i);
    foreach(single2 : q-value in tmp2) {
      while (single2 jest binderem name(v)) single2 := v;
      if (single2 jest referencją i) single2 := Dereference(i);
      if (można zaaplikować dodawanie do single1 i single2)
        result.add(new Result(single1.value + single2.value));
      else throw exception;
    }
  }
  return result;
}

```

Jak widzimy binarne operatory algebraiczne działają na zbiorach, w razie potrzeby automatycznie pozbywają się binderów, oraz referencji. Analogicznie rzecz ma się przy unarnych operatorach algebraicznych:

```

function eval(δ q){
  wynik1: q-value;
  eval(q);
  wynik := QRES.pop();
  QRES.push(δ'(wynik));
}

```

Natomiast przy operatorach niealgebraicznych kod procedury eval przedstawia się następująco:

```
function eval(q1  $\delta$  q2) {
  res1, res2: q_value;
  partialResults: q_value;
  partialResults = new bag();
  eval(q1);
  res1 := QRES.pop();
  if (res1 nie jest bagiem) res1 := new bag(res1);
  foreach(single1 in res1){
    ENV.openScope();
    int opened = nested(single1));
    eval(q2);
    res2 = QRES.pop();
    partialResults.add( $\delta'$ (single1, res2));
    ENV.closeScopes(opened + 1);
  }
  QRES.push( $\delta''$ (partialResults));
}
```

Funkcja  $\delta'$  w podanym schemacie wylicza wyniki częściowe zapytania,  $\delta''$  zaś z ich zbioru wylicza wynik końcowy. Najciekawszym elementem jest jednak funkcja nested(result). Umieszcza ona w najwyższej sekcji stosu środowiskowego wewnątrz wyniku zapytania. Dla poszczególnych rodzajów wyników są to odpowiednio:

- wartość atomowa - zbiór pusty
- binder  $n(v)$  - ten sam binder  $n(v)$
- referencja  $i$  - bindery do podobieństw obiektu o identyfikatorze  $i$ , gdy obiekt ten jest złożony, w przeciwnym wypadku zbiór pusty
- $bag(v_1, v_2, \dots)$  - zbiór pusty
- $struct(v_1, v_2, \dots)$  -  $nested(v_1) + nested(v_2) + \dots$

## 1.5. Konstrukcje imperatywne

Język SBQL z założenia ma łączyć w sobie cechy języka zapytań i języka programowania. W związku z tym potrzebujemy wprowadzić do niego konstrukcje imperatywne. Podstawowymi są te związane z tworzeniem, zmienianiem i kasacją obiektów. Jednak konstrukcja języka nie wyklucza takich rzeczy jak procedury, zmienne lokalne, iteracji i innych. W tym dziale zajmiemy się tylko podstawowymi z nich to jest:

- create
- insert
- update

- delete

### 1.5.1. Create

Procedura *create* służy do tworzenia obiektów. Ma ona następującą składnię:

```
create query;
```

Zapytanie *query* musi zwracać binder  $name(v)$ , lub zbiór binderów. Wynikiem jest zbiór wskaźników na stworzone obiekty.

W najprostszej formie *create* wygląda następująco:

```
create 1024 as liczba;
```

Zapytanie to tworzy obiekt  $(i, \text{liczba}, 1024)$ . Gdzie  $i$  jest identyfikatorem nowo powstałego obiektu.

Mimo iż pozornie wygląda to dość prosto musimy rozważyć parę wariantów.

1. Tworzenie obiektów złożonych.
2. Tworzenie obiektu o wartościach podobnych do innego obiektu.
3. Tworzenie obiektów referencyjnych.

Odpowiedź na pytanie 1) jest prosta, najlepiej ilustruje to przykład:

```
create ("Jan" as Imie, "Kowalski" as Nazwisko) as Osoba;
```

To zapytanie tworzy najpierw obiekt złożony *Osoba* a potem jego podobiektu *Imie* i *Nazwisko*

Odpowiedź na punkt 2) jest nieco bardziej skomplikowana. Chcielibyśmy, aby zapytanie:

```
create (Osoba where Nazwisko="Kowalski").Imie as ImieKowalskiego;
```

Stworzyło obiekt  $(i, \text{ImieKowalskiego}, \text{"Jan"})$ . Jednak gdyby miało być ono wyliczone wprost to otrzymamy obiekt referencyjny  $(i_1, \text{ImieKowalskiego}, i_2)$  gdzie  $i_2$  będzie wskaźnikiem na odpowiedni obiekt. Problem ten rozwiązujemy w ten sposób, że przy tworzeniu nowych obiektów wszystkie wskaźniki podlegają pojedynczej dereferencji.

Takie rozwiązanie ma dodatkowy efekt - dzięki niemu w łatwy i szybki sposób możemy kopiować obiekty:

```
create (Osoba where (Nazwisko="Kowalski" and Imie="Jan"))
as KlonKowalskiego;
```

Zapytanie to stworzy obiekt o wartości odpowiadającej wartości obiektu już istniejącego wraz z wszystkimi jego podobiektami.

Takie rozwiązanie jednak prowadzi do punktu 3) - jak tworzyć obiekty referencyjne? Aby rozwiązać ten problem wprowadzamy do języka nowy operator algebraiczny - *ref* - mówi on, że wartość podzapytania nie podlega automatycznej dereferencji. Czyli:

```
create ref(Osoba where Nazwisko="Kowalski") as Wskaznik;
```

Stworzy obiekt referencyjny  $(i_1, Wskaznik, i_2)$ , w którym  $i_2$  będzie wskazywało na odpowiednią osobę.

Aby zachować spójność wprowadzamy do języka drugi operator - tym razem niealgebraiczny *deref*, który dokonuje dereferencji na wyniku podzapytania.

Uwaga: Konstrukcja *ref* nie zwraca wskaźnika na wynik podzapytania, a jedynie mówi, że jego wynik nie podlega automatycznej dereferencji, w związku z czym zapytania *ref(1024)* nie ma sensu.

Jak większość czytelników pewnie zauważyła gdyby tworzenie obiektu nie robiło automatycznej dereferencji, to problem 3) wcale by nie zaistniał, a wprowadzenie operatora *deref* rozwiązałoby problem 2), jednak wtedy utracilibyśmy możliwość łatwego kopiowania obiektów.

### 1.5.2. Insert

Procedura *insert* służy do przenoszenia już istniejących obiektów do wnętrza obiektów złożonych. Ma ona następującą składnię:

```
insert query;
```

Wynikiem podzapytania *query* powinna być dwuelementowa struktura  $(v_1, v_2)$ , lub zbiór takich struktur. Gdzie  $v_1$  po usunięciu binderów będzie identyfikatorem obiektu, do którego się wstawia, zaś  $v_2$  - po tej samej operacji - identyfikatorem obiektu wstawianego. W wyniku zwracany jest zbiór identyfikatorów obiektów, do których zostały wstawione podobiekty.

Procedura *insert* jedynie wstawia już istniejące obiekty do innych obiektów. Dzięki połączeniu jej z procedurą *create* możemy do wnętrza obiektów wstawiać nowo utworzone obiekty:

```
insert (  
  (Osoba where (Nazwisko="Kowalski" and Imie="Jan")),  
  create 1000 as Pensja  
);
```

Należy tutaj zauważyć, że gdyby podzapytanie *(Osoba where (Nazwisko="Kowalski" and Imie="Jan"))* zwróciło więcej niż jeden identyfikator, *create 1000 as Pensja* nadal zwróci tylko jeden identyfikator, a operator “,” sparuje go z każdym wynikiem pierwszego podzapytania. Następnie *insert* przypisze po kolei każdemu panu Kowalskiemu pensję, zabierając ją poprzedniemu, co ostatecznie da taki efekt, że tylko jeden pan Kowalski ją otrzyma.

Aby uniknąć takich problemów wprowadzamy iterator *for each*  $q_1$  *do*  $q_2$ . Dzięki jego zastosowaniu możemy już dla każdego pana Kowalskiego stworzyć odpowiedni podobiekt:

```
for each (  
  (Osoba where  
    (Nazwisko="Kowalski" and Imie="Jan"))  
  as JKowalski)  
do  
  insert (JKowalski, create 1000 as Pensja);
```

### 1.5.3. Update

Aktualizacja wartości obiektu odbywa się za pomocą operatora  $q_1 := q_2$  po usunięciu binderów  $q_1$  powinno zwrócić pojedynczy identyfikator, zaś  $q_2$  po dodatkowym usunięciu referencji - pojedynczą wartość. W przypadku obiektów atomowych sprawa jest prosta - wystarczy, że typ wartości zgadza się z typem obiektu. Dla obiektów złożonych mamy następujące warianty:

1. Nowa wartość ma tyle elementów co obiekt, na który ją przypisujemy i składa się z podobiektów o identycznych nazwach jak obiekt, na który chcemy ją przypisać
2. Nowa wartość ma tyle elementów co obiekt, na który ją przypisujemy i składa się z podobiektów o identycznych nazwach jak obiekt, na który chcemy ją przypisać i ewentualnie nowych, które tam nie występują
3. Nowa wartość składa się z dowolnych podobiektów

W trzecim przypadku możemy dodatkowo rozważyć, czy wszystkie pierwotne podobiekty należy usunąć, czy tylko te, których nazwy nie występują w nowej wartości.

W mojej implementacji przyjąłem wariant, w którym na obiekt złożony możemy przypisać dowolną wartość złożoną. Podobiekty są odpowiednio:

- usuwane - gdy występują w starej wartości, a w nowej ich nie ma
- tworzone - gdy w starej wartości ich nie ma, a w nowej zaistniały
- zmieniają wartość - gdy występują po obu stronach przypisania

Uwaga: Przy przypisywaniu nowej wartości na obiekty wskaźnikowe można używać operatora ref.

### 1.5.4. Delete

Ostatnią z procedur imperatywnych jaką się tutaj zajmujemy jest *delete query*. Wynikiem podzapytania powinien być zbiór referencji lub pojedyncza referencja. Delete usuwa wskazane obiekty, wszystkie ich podobiekty oraz wszystkie obiekty referencyjne, które na nie wskazują. Dzięki temu unikamy problemu wiszących wskaźników.



## 1.6. Przykłady zapytań

```
(Osoba where (Nazwisko = "Kowalski")).Imie;
```

Zapytanie to znajduje Imiona wszystkich Osób, które na Nazwisko mają "Kowalski".

```
1576 + 154 * 196;
```

To zapytanie nie odwołuje się do składu obiektów, w wyniku dostajemy liczbę będącą wynikiem obliczenia.

```
(Produkt where Kod="1234").NaMagazynie;
```

Zapytanie to zwróci liczbę dostępnych na magazynie produktów, których kod identyfikacyjny ma wartość "1234".

```
Osoba.(Nazwisko,Imie);
```

W tym zapytaniu w wyniku dostaniemy zbiór struktur składających się z identyfikatorów imion i nazwisk wszystkich osób znajdujących się w systemie.

```
(Produkt.(UDostawcy + Marza))as CenaDetaliczna;
```

Tutaj najpierw ze składu wyciągamy ceny wszystkich towarów u dostawcy, oraz nakładaną marżę (indywidualną dla każdego produktu). Po zsumowaniu otrzymujemy ceny detaliczne, które zwracamy jako zbiór binderów *CenaDetaliczna(v)*.



## Rozdział 2

# Perspektywy w SBQL

Najważniejsze założenie dotyczące perspektyw w SBQL jest następujące:

Wirtualne obiekty dla użytkownika powinny być pod każdym względem identyczne jak te przechowywane w składzie obiektów.

W związku z tym powinna istnieć możliwość ich tworzenia, aktualizacji i kasowania. Aby powyższy warunek został spełniony do definicji perspektywy nie wystarczy jedno zapytanie. Ilustruje to dość dobrze przykład:

Chcielibyśmy zdefiniować perspektywę, która określałaby obiekty *Pracownik*, dla których podobiekt *Płaca* zawierałby wartość netto od wartości podanej w obiekcie składowanym w bazie danych. Rozważmy zapytanie:

```
Pracownik.(Płaca := Płaca + 100);
```

Gdyby operacja ta miała się odbywać automatycznie, wtedy wartość brutto składowana w bazie danych zostałaby podniesiona o 100, jednak to niekoniecznie byłoby zgodne z intencjami tej operacji.

Przykład ten ilustruje konieczność umieszczenia w definicji perspektywy opisu zachowania systemu w sytuacji zmiany wartości. Analogiczne przykłady można znaleźć dla kasowania wirtualnych obiektów, oraz ich tworzenia.

### 2.1. Definicja perspektywy

W definicji perspektywy mogą się znaleźć następujące elementy:

- virtual objects - definicja ziaren wirtualnych obiektów
- on.create - procedura opisująca zachowanie w przypadku tworzenia nowych wirtualnych obiektów
- on.delete - procedura opisująca zachowanie w przypadku kasowania wirtualnych obiektów

- `on_retrieve` - procedura opisująca zachowanie w przypadku dereferencji wirtualnych obiektów
- `on_insert` - procedura opisująca zachowanie w przypadku wstawiania podobiektów do wirtualnych obiektów
- `on_store` - procedura opisująca zachowanie w przypadku wstawiania wirtualnych obiektów do innych obiektów
- `on_store_reference` - procedura opisująca zachowanie w przypadku tworzenia referencji na wirtualny obiekt
- definicja podperspektyw
- definicje podobiektów z wnętrza perspektywy

Z tych elementów jedynie *virtual objects* jest obowiązkowy.  
Przykładowa definicja perspektywy zatem wygląda tak:

```
create view PracownikDef {
  virtual objects Pracownik {return Firma.Zatrudnia.Osoba as Os;},
  on_create(new) do {
    insert Firma,
      (create ref (create new as Osoba) as Zatrudnia);
  },
  on_retrieve do {
    licznik := licznik + 1;
    return (Os.Imie,
      Os.Nazwisko,
      Placa);
  },
  on_delete do {delete Os;},
  create view PlacaDef{
    ...
  },
  0 as licznik
}
```

Powstaje pytanie: czym są poszczególne elementy definicji perspektywy i jak są wyliczane?

## 2.2. Konstrukcja wirtualnych obiektów

Wynik procedury *virtual objects* określa zbiór ziaren, które służą do budowy wirtualnych obiektów. Ziarno nie definiuje wartości obiektu, a jedynie służy do jej dalszego wyliczenia. Aby to było możliwe w identyfikatorze obiektu musi się znaleźć także identyfikator perspektywy. Umożliwi to odnalezienie podperspektyw i procedur aktualizujących, a tym samym wyliczenie odpowiednich podobiektów, oraz dokonanie ewentualnych zmian wartości obiektu. Dodatkowo chcielibyśmy aby w procedurach znajdujących się w podperspektywach można było korzystać

z ziaren obiektów nadrzędnych, zatem i one muszą znaleźć się w identyfikatorze obiektu. Ostatecznie taki identyfikator ma postać:

$$\begin{pmatrix} i_1, ziarno_1 \\ i_2, ziarno_2 \\ \dots \\ i_n, ziarno_n \end{pmatrix}$$

$i_n, ziarno_n$  są odpowiednio identyfikatorem perspektywy i ziarnem obiektu, zaś  $i_k, ziarno_k$  - identyfikatorem perspektywy nadrzędnej w stosunku do  $i_{k+1}$  i ziarnem obiektu nadrzędnego w stosunku do  $ziarno_{k+1}$

## 2.3. Wartość wirtualnego obiektu

Wartość obiektów jest wyliczana przez operatory algebraiczne i niealgebraiczne. Te pierwsze dokonują automatycznej dereferencji identyfikatorów obiektów, zaś te drugie korzystają z funkcji *nested* i umieszczają na stosie bindery do podobiektów. Tak więc wartość wirtualnych obiektów powinniśmy rozważyć właśnie w tych dwóch aspektach.

### 2.3.1. Funkcja nested dla wirtualnych obiektów

W dotychczasowym podejściu przedstawionym w [HKoz02] i [Subi04] funkcja *nested*, której argumentem był identyfikator wirtualnego obiektu, tworzyła na stosie środowiskowym następujące sekcje:

1. Dla każdego ziarna  $z$  oddzielna sekcja. Sekcja ta zawiera *nested(z)*
2. Sekcja zawierająca bindery do *virtual objects* wszystkich podperspektyw.

Rozpatrzmy perspektywę:

```
create view DobrzeZarabiaDef{
  virtual objects DobrzeZarabia {
    Osoba where Zarobki > 3000 as Os;  },
  create view ZaraobkiDef{
    virtual objects Zarobki {
      Os.Zarobki as Z;
    },
    on_retrieve do {return deref Z;}
    ...
  },
  on_retrieve do {return (Os.Imie, Os.Nazwisko, Zarobki);}
  ...
}
```

Przykładowe obiekty tworzone przez tą perspektywę to:

$$\begin{aligned}
&1. (Zarobki, \left( \begin{array}{c} Id_{DobrzeZarabiaDef}, Ziarno_{DobrzeZarabia} \\ Id_{ZarobkiDef}, Ziarno_{Zarobki} \end{array} \right)) \\
&2. (DobrzeZarabia, Id_{DobrzeZarabia}, Ziarno_{DobrzeZarabia})
\end{aligned}$$

Dla pierwszego obiektu funkcja *nested* umieści na stosie następujące sekcje:

- $nested(Ziarno_{Zarobki}) = \text{binded } Z(Id_{Zarobki})$
- $nested(Ziarno_{DobrzeZarabia}) = \text{binder } Od(Id_{Osoba})$

W drugim przypadku:

- $nested(Ziarno_{DobrzeZarabia}) = \text{binder } Os(Id_{Osoba})$
- Bindery do *virtual objects* z podperspektyw = binder  $Zarobki(Id_{Zarobki})$

### 2.3.2. Dereferencja na wirtualnym identyfikatorze

W celu przeprowadzenia dereferencji wirtualnego identyfikatora wykonujemy następujące kroki:

1. tworzymy na stosie środowiskowym po jednej sekcji dla każdego ziarna  $z$  zawartego w identyfikatorze. Sekcja ta będzie zawierać  $nested(z)$
2. tworzymy sekcję zawierającą bindery do *virtual objects* podperspektyw.
3. w tak zmodyfikowanym środowisku uruchamiamy procedurę *on\_retrieve*
4. zdejmujemy ze stosu środowiskowego wcześniej utworzone sekcje

Pierwsze dwa kroki są analogiczne jak przy uruchamianiu procedury *nested*, lecz krok trzeci i czwarty są dodatkowe. Nasuwa się pytanie: jak wpływa to na wartość obiektu?

### 2.3.3. Proponowane zmiany w funkcji *nested*

W przedstawionym podejściu procedura *nested* może umieścić na stosie zupełnie inne bindery, niż te które będą częścią wyniku dereferencji. Tak więc wartość wirtualnego obiektu może być różna dla operatorów algebraicznych i niealgebraicznych. Aby temu zaradzić proponuje zmienić funkcję *nested* tak aby wykonywała następujące kroki:

1. Powtórzenie dokładnie wszystkich czterech kroków wykonywanych podczas wywołania dereferencji.
2. Zdjęcie wyniku procedury *on\_retrieve* ze stosu wynikowego.
3. Przetworzenie pary (wirtualny identyfikator, wynik dereferencji) dokładnie w ten sam sposób, w jaki do tej pory przetwarzaliśmy z pomocą funkcji *nested* obiekty rzeczywiste.

Należy tutaj zauważyć, że w przypadku identyfikatora zwykłego obiektu mogliśmy otrzymać:

- wartość atomową
- binder wskazujący na identyfikator obiektu
- strukturę składającą się z identyfikatorów podobieństw

Procedura *on\_retrieve* może zwrócić szereg innych wartości, które powinniśmy obsługiwać:

- binder wskazujący dowolną wartość - na stosie środowiskowym umieszczamy ten sam binder. Taki obiekt wirtualny można traktować jak obiekt wskaźnikowy.
- struktura lub zbiór - dla każdego bindera i każdego identyfikatora obiektu należącego do danej kolekcji na stosie powinien znaleźć się odpowiedni binder. Jest to rozszerzenie obiektu złożonego.
- pozostałe przypadki nie mają wpływu na stos środowiskowy

Rozważmy raz jeszcze wcześniejszy przykład ze strony 18:

Teraz dla pierwszego obiektu funkcja *nested* na stosie środowiskowym nie umieści żadnych sekcji - wartość tego obiektu po dereferencji jest atomowa.

Dla drugiego obiektu na stosie pojawi się sekcja zawierająca:

- bindery Imię i Nazwisko odnoszące się do rzeczywistych obiektów
- binder Zarobki odnoszący do podobiektu wirtualnego

## 2.4. Operacje na wirtualnych identyfikatorach

W pracy [HKoz02] czytamy, że przed wywołaniem procedur z wnętrza perspektywy aktualny stos środowiskowy powinien być wzbogacany o sekcję zawierającą *nested(ziarno<sub>1</sub>)*, *nested(ziarno<sub>2</sub>)* ... *nested(ziarno<sub>n</sub>)*.

W mojej implementacji przyjmuję nieco odmienne podejście. Przy wywoływaniu procedur z wnętrza perspektywy wzbogacam środowisko o szereg sekcji, które kolejno zawierają *nested(ziarno<sub>i</sub>)* zaczynając od ziarna obiektu najbardziej nadrzędnego, a kończąc na ziarnie aktualnego. Prócz tego na stosie umieszczam sekcję, która zawiera bindery do *virtual objects* z podperspektyw. W ten sposób unikam konfliktu nazw, który mógłby wystąpić w przypadku pojawienia się tej samej nazwy na różnych poziomach hierarchii ziaren wirtualnych obiektów i wewnątrz procedur daje dostęp do podperspektyw.

### 2.4.1. Aktualizacja wirtualnych obiektów

Aktualizacja wartości odbywa się poprzez procedurę *on\_update*. Ma ona jeden parametr, którego nazwa jest nadawana przez projektanta perspektywy. Przy wyliczaniu tej procedury za parametr podstawiana jest wartość, którą chcemy przypisać wirtualnemu obiektowi. Odbywa się to standardowo - poprzez umieszczenie odpowiedniego bindera na stosie środowiskowym. W razie braku definicji *on\_update*, przy próbie zmiany wartości obiektu, system zwróci błąd.

Na przykład:

```

create view CenaZVatDef{
  virtual objects CenaZVat{return Cena as C;},
  on_update(New) do {C := New/1.22;},
  on_retrieve do {return 1.22 * C;}
}
(CenaZWat as C where C < 100).C := 100;

```

Przy podanej definicji perspektywy podane zapytanie ustawi wszystkie *CenyZVat* niższe niż 100 na 100. Co za tym idzie wszystkie oryginalne ceny również zostaną zmodyfikowane zgodnie z treścią procedury *on\_update*.

#### 2.4.2. Kasowanie wirtualnych obiektów

Wirtualne obiekty kasujemy poprzez wywołanie procedury *on\_delete*. W razie braku definicji tej procedury przy próbie kasacji wirtualnych obiektów system powinien zwrócić błąd.

#### 2.4.3. Wstawianie podobiektów do wirtualne obiektu

Do wstawiania podobiektów służy procedura *on\_insert*. Procedura ta, analogicznie jak *on\_update*, ma jeden argument, jest nim drugi element struktury będącej argumentem operacji *insert* (domyślnie powinien to być identyfikator obiektu).

#### 2.4.4. Tworzenie wirtualnych obiektów

Tworzenie wirtualnych obiektów odbywa się za pomocą procedury *on\_create*. Jej argumentem jest binder do wartości nowo tworzonego obiektu. Procedura jest znacząco różna od pozostałych procedur definiujących perspektywę. Różnice te są następujące:

- Podczas wyliczania *on\_create* nie mamy dostępu do ziarna, ponieważ obiekt jeszcze nie istnieje.
- Nie dopuszczamy równoczesnego istnienia wirtualnych obiektów o tej samej nazwie co obiekty ze składu, zatem procedura *on\_create* jest uruchamiana przy tworzeniu perspektywy. Dla każdego obiektu, który nosi nazwę zgodną z *virtual objects* nowo tworzonej perspektywy uruchamiamy *on\_create*, a następnie taki obiekt kasujemy, ponieważ został zastąpiony przez obiekt wirtualny.
- Nie dopuszczamy definicji *on\_create* w podperspektywach. Spowodowane jest to przez fakt, że nie mamy możliwości tworzenia podobiektów w inny sposób jak poprzez *insert*.

### 2.5. Materializacja wirtualnych obiektów

W poprzednim dziale rozważaliśmy tworzenie, kasowanie i edycję wirtualnych obiektów. Należy jednak zauważyć, że mogą one być również składowane w bazie danych. Dzieje się tak w dwóch przypadkach:

1. wstawienia wirtualnego obiektu do wnętrza innego obiektu
2. utworzenie obiektu referencyjnego wskazującego na obiekt wirtualny



Ten problem został pominięty w dotychczasowych pracach dotyczących SBQL. W tym dziale spróbuje się nim zająć.

### 2.5.1. Wstawianie wirtualnego obiektu do wnętrza innego obiektu

Procedura wstawiania obiektu składa się z dwóch etapów:

1. Wstawienie identyfikatora obiektu do nowego rodzica
2. Usunięcie identyfikatora obiektu ze starego rodzica

W przypadku identyfikatorów obiektów wirtualnych pierwsza z tych operacji może powodować powstanie problemu wiszących wskaźników - ziarno obiektu wirtualnego może zawierać identyfikatory obiektów, które pomiędzy wstawieniem obiektu, a kolejnym jego wywołaniem zostaną usunięte. Druga operacja natomiast w przypadku obiektów wirtualnych nie może zostać przeprowadzona w sposób automatyczny, ponieważ do końca nie wiadomo, jakich obiektów powinna ona dotyczyć. Przy ograniczeniu jej zakresu do usunięcia obiektów wirtualnych, kolejne wywołanie *virtual objects* zwróci między innymi ten sam obiekt. Jeśli spróbujemy automatycznie przeprowadzić operacje na obiektach rzeczywistych - natrafiamy na pytanie, którymi obiektami powinniśmy się zająć, czy je usuwać, czy tylko wprowadzić flagę, która sprawi, że będą ignorowane przez *virtual objects* i *on\_retrieve*, czy może podjąć jeszcze inne działanie.

Z tych względów wprowadzam kolejną procedurę składającą się na definicję perspektywy - *on\_store*. Procedura ta ma jeden argument - jego wartością jest identyfikator obiektu, do którego próbujemy wstawić obiekt wirtualny.

Przy tej okazji nasuwa się kolejne pytanie: jak powinniśmy się zachować w przypadku próby wstawienia obiektu wirtualnego do innego obiektu wirtualnego. Odpowiedź jest dość prosta - uruchamiamy procedurę *on\_insert* z obiektu, do którego wstawiamy i jeśli zajdzie taka konieczność to podczas jej wyliczania nastąpiwołanie *on\_store* z obiektu wstawianego.

### 2.5.2. Tworzenie wskaźnika na wirtualny obiekt

W przypadku tworzenia obiektu referencyjnego wskazującego na obiekt wirtualny spotykamy się z tym samym problemem co przy wstawianiu identyfikatora obiektu wirtualnego do nowego rodzica, a mianowicie z wiszącymi wskaźnikami. Aby go uniknąć w tym przypadku posłużymy się procedurą *on\_store\_reference*. Jest to procedura bezparametrowa. W wyniku powinna ona zwrócić wartość, którą powinien otrzymać tworzony obiekt wskaźnikowy.

Rozważmy perspektywę:

```
create view BogataOsobaDef {  
  virtual objects BogataOsoba{  
    Osoba where Pensja>20000 as Os;  
  },  
  on_retrieve do {return deref Os;},  
  on_store_reference do {  
    return ref Os;  
  }  
}
```

Utwórzmy obiekty wskaźnikowe i rozważmy zapytania:

```
create ref BogataOsoba as Bogaty;  
(Bogaty.Osoba where Nazwisko = "Kowalski" ).Imie  
(Bogaty where Osoba.Nazwisko= "Kowalski").BogataOsoba.Imie;
```

Przy tworzeniu obiektów referencyjnych otrzymamy obiekty  $(i_1, Bogaty, i_2)$  gdzie  $i_2$  będzie wskazywało na odpowiedni obiekt *Osoba*. W związku z tym pierwsze zapytanie będzie poprawne. Natomiast drugie zadziała z tego względu, że po operatorze kropki na stosie znajdą się bindery *Osoba(i)* iwołanie *virtual objects* z naszej perspektywy będzie miało do nich dostęp, dzięki czemu nie będzie szukało w niższych sekcjach stosu i zwróci ten sam wynik co wcześniejsze zapytanie.

Alternatywą dla przedstawionego pomysłu jest faktyczne przechowywanie w bazie wskaźników na obiekty wirtualne - a więc identyfikatorów obiektów wirtualnych. Aby uniknąć problemu wiszących wskaźników w takim przypadku moglibyśmy, zamiast tworzyć obiekty rzeczywiste zwrócone przez *on\_store\_reference* jedynie zaznaczać w nich, że utworzony obiekt referencyjny na nie wskazuje i w razie ich usunięcia - usuwać również obiekt wskaźnikowy. Takie podejście pozornie wydaje się lepsze, jednak w razie błędnej definicji funkcji *on\_store\_reference* nie rozwiązuje problemu wiszących wskaźników. Dlatego też w mojej implementacji przyjąłem rozwiązanie przedstawione powyżej.

## 2.6. Perspektywy referencyjne

W pracy [HKoz02] oprócz zwykłych perspektyw rozważane są perspektywy referencyjne. Mają one służyć tworzeniu wirtualnych obiektów odpowiadających obiektom referencyjnym ze składu obiektów. Ich definicja jest analogiczna do definicji zwykłej perspektywy, z tym że w celu rozróżnienia początek definicji ma postać *create view NazwaPerspektywy as pointer*. Reszta jest identyczna jak w przypadku zwykłych perspektyw.

Obiekty generowane przez taką perspektywę dodatkowo powinny posiadać flagę "I am virtual pointer". Podczas wywoływania funkcji *nested* ta flaga byłaby po prostu usuwana.

Koncepcja tych perspektyw opiera się na funkcji *nested* przedstawionej w dziale 2.2, która to koncepcja została w mojej pracy zmieniona.

Rozpatrzmy perspektywę:

```
create view PracownikDzialuITRefDef as pointer {  
  virtual objects PracownikDzialuITRef {  
    return Pracownik where (PracujeW.Dzial.Nazwa = "IT");  
  }  
}
```

Jeśli funkcja *nested* działałaby tak jak w pracy [HKoz02] i uwzględniałaby wirtualne wskaźniki, to zapytanie:

```
PracownikDzialuIT.Pracownik.Nazwisko;
```

Zwróciłoby nazwiska wszystkich pracowników działu IT.

W moim podejściu istnienie takich perspektyw jest zupełnie nieuzasadnione, ponieważ analogiczną perspektywę możemy zdefiniować korzystając ze standardowej definicji w następujący sposób:

```
create view PracownikDzialuITRefDef {  
  virtual objects PracownikDzialuITRef {  
    return Pracownik  
      where (Dzial = (Dzial where nazwa = "IT")) as P;  
  },  
  on_retrieve do { return P;}  
}
```

Podane zapytanie zwróci dokładnie ten sam wynik.

W związku z tym zrezygnowałem z implementacji perspektyw referencyjnych.

## 2.7. Przykłady perspektyw

```
create view PracownikITDef {  
  virtual objects PracownikIT {  
    Osoba where PracujeW.Dzial.Nazwa = "IT"  
    as O;  
  },  
  on_retrieve do {return deref O;},  
  on_update(NewValue) do {O := NewValue;},  
  on_insert(Object) do {insert(O, Object);}  
}
```

Ta perspektywa znajduje wszystkich pracowników działu IT.

```

create view DuzyProjektDef {
  virtual objects DuzyProjekt {
    (Projekt where (count(Programista)) > 15) as P;
  },
  on_retrieve do {return deref P;},
  on_update(NewValue) do {P := NewValue;},
  on_insert(Object) do {insert(P, Object);}
  on_delete do {delete P;}
}

```

Dzięki tej perspektywie znajdujemy projekty, przy których pracuje co najmniej 16 programistów.

```

create view StalyKlientDef {
  virtual objects {
    Klient where sum(Faktura.Wartosc) > 500 as K;
  },
  create view ZainteresowanieDef {
    virtual objects Zainteresowanie {
      distinct(K.Faktura.Dotyczy.Produkt.Dzial) as D;
    },
    on_retrieve do { return deref (D.Nazwa);}
  },
  on_retrieve do { return (K.Imie, K.Nazwisko, Zainteresowanie;)}
}

```

W wyniku działania tej perspektywy otrzymujemy Klientów, którzy w sumie wydali ponad 500 złotych. Dodatkowo do wyniku dołączane są zainteresowania danej osoby - czyli nazwy działów, w których kupuje ona produkty.

## Rozdział 3

# Prototyp perspektyw dla SBQL

Prototypowa implementacja przedstawionego podejścia do perspektyw dla SBQL została oparta na projekcie ODRA (Object Database for Rapid Application Development) [Hryń05] - niepełnej implementacji samego SBQL wraz z kontrolą typów. Głównym celem projektu było przetestowanie i praktyczne potwierdzenie przedstawionej teorii. Projekt oparty został na technologii .NET, na której opierała się pierwotna implementacja.

Implementacja opiera się na modelu składu obiektów M0, został on przedstawiony w dziale 1.3. Do języka zostały wprowadzone konstrukcje imperatywne przedstawione w tym samym rozdziale, a także perspektywy (rozdział 2).

Pracę nad projektem podzieliłem na następujące etapy:

- zapoznanie się z otrzymaną implementacją, jej możliwościami i brakami
- rozszerzenie modelu o perspektywy z możliwościami dostosowanymi do zastanej implementacji
- rozszerzenie projektu o brakujące funkcjonalności niezbędne do pełnej implementacji perspektyw
- implementacja pozostałych funkcjonalności perspektyw
- testy i usuwanie błędów

### 3.1. Najważniejsze moduły

#### 3.1.1. Interfejs

W prototypowej implementacji użytkownik zadaje zapytania za pomocą interfejsu tekstowego. Dodatkowo wprowadziłem możliwość wczytania pliku tekstowego, którego treść następnie traktowana jest jako blok zapytań.

#### 3.1.2. Parser

Parser używany w prototypowej implementacji został wygenerowany z pomocą C#CUP i C#LEX, które są wersjami popularnego leksera JLex, oraz generatora parserów CUP. W wyniku parsowania otrzymujemy drzewo syntaktyczne, które następnie jest przetwarzane przez dalsze moduły.

### 3.1.3. Kontroler typów

W obecnej implementacji kontrola typów w znacznej mierze odbywa się na poziomie interpretera podczas wyliczania zapytania. Przeniesienie ciężaru kontroli typów ze statycznej na dynamiczną spowodowana została niepełną implementacją podejścia zaproponowanego w [Hryń05]. W związku z tym obecnie moduł ten sprawdza jedynie zgodność typów dla zapytań, które nie odwołują się do bazy danych. Po rozbudowie modułu o obsługę różnych wariantów typu dla jednej nazwy moduł powinien uzyskać pełną funkcjonalność przewidzianą w wymienionej pracy.

### 3.1.4. Optymalizator zapytań

Moduł ten odpowiada za optymalizację zapytań. W obecnej implementacji jest on odłączony. Spowodowane jest to niepełną implementacją pierwotnego pomysłu przepisywania zapytań. Bezpośredni wpływ na tę decyzję miało także połączenie modułu z modułem odpowiedzialnym za kontrolę typów. Przeniesienie kontroli typów do fazy wykonania spowodowało dalsze błędy w funkcjonowaniu modułu.

### 3.1.5. Generator kodu

W tym module przetwarzane są węzły drzewa syntaktycznego odpowiedzialne za operatory algebraiczne. Identyfikowane są operatory i typy ich argumentów. Następnie węzły takie są zastępowane przez odpowiednie węzły zawierające wszystkie informacje o przeprowadzanej operacji. Ma to na celu przyspieszenie wykonywania operatorów algebraicznych przez kolejne moduły.

### 3.1.6. Interpreter

Drzewo syntaktyczne przedstawiające zapytanie trafia do interpretera, gdzie jest wyliczane przy użyciu stosu środowiskowego, stosu wynikowego, oraz składu obiektów. Cały proces odbywa się zgodnie z przedstawioną w poprzednich rozdziałach teorią.

## 3.2. Zakres funkcjonalności

Pierwotna implementacja zawierała część języka SBQL opisaną następującą gramatyką:

```
zapytanie ::= literal | nazwa
zapytanie ::= (zapytanie)
zapytanie ::= zapytanie op_alg zapytanie
zapytanie ::= zapytanie op_niealg zapytanie
zapytanie ::= zapytanie; zapytanie
zapytanie ::= zapytanie as nazwa
op_niealg ::= where | join | ,
op_alg ::= + | - | = | < | > | <= | >=
```

Podczas prac nad projektem wprowadziłem następujące rozszerzenia:

```

zapytanie ::= foreach zapytanie do zapytanie
zapytanie ::= op_unarny zapytanie
op_unarny ::= -
zapytanie ::= deref zapytanie
zapytanie ::= ref zapytanie
zapytanie ::= insert zapytanie
zapytanie ::= zapytanie := zapytanie
zapytanie ::= delete zapytanie
zapytanie ::= create zapytanie

```

### 3.2.1. Perspektywy

Pierwszym krokiem do wprowadzenia perspektyw było rozszerzenie parsera. Pojawiły się w nim konstrukcje:

```

zapytanie ::= create_view
create_view ::= create view nazwa view_list
view_list ::= view_operation, view_list | create_view, view_list | ε
view_operation ::= virtual object nazwa {zapytanie;}
view_operation ::= on_insert (nazwa) do {zapytanie;}
view_operation ::= on_update (nazwa) do {zapytanie;}
view_operation ::= on_retrieve do {zapytanie;}
view_operation ::= on_create (nazwa) do {zapytanie;}
view_operation ::= on_delete do {zapytanie;}
view_operation ::= on_store (nazwa) do {zapytanie;}
view_operation ::= on_store_reference do {zapytanie;}

```

Do interfejsu wprowadziłem dodatkowo możliwość wczytania zawartości pliku jako bloku komend. Odbywa się to za pomocą polecenia:

```
@nazwa_pliku
```

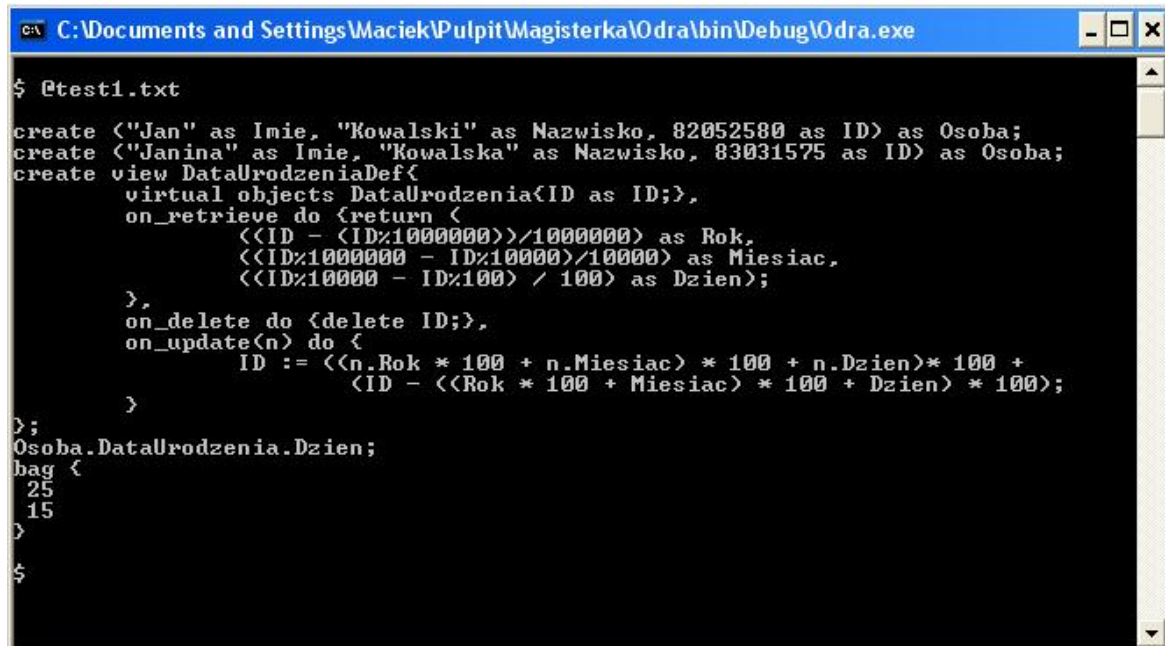
Dalszy etap prac to wprowadzenie do wszystkich modułów obsługi przedstawionych powyżej komend.

- Kontroler typów - w tym module sprawdzam poprawność definicji perspektywy (procedura *create view*). Sprawdzam czy występują w niej definicje niezbędnych procedur i czy żadna z nich nie występuje więcej niż raz.
- Interpreter - obsługuje tu tworzenie perspektyw, oraz wywołuje odpowiednie procedury w razie konieczności.
- Skład obiektów - przechowuje definicje perspektyw jako obiekt złożony z wyróżnionymi podobiektami odpowiedzialnymi za procedury. Procedury przechowują drzewo

syntaktyczne odpowiadające ciału procedury. Tam gdzie jest to niezbędne jest również przechowywana lista parametrów.

- pozostałe moduły - wstępnie przetwarzają ciało procedur z perspektyw.

### 3.3. Przykłady działania systemu ODRA



```
C:\Documents and Settings\Maciek\Pulpit\Magisterka\Odra\bin\Debug\Odra.exe

$ @test1.txt

create <"Jan" as Imie, "Kowalski" as Nazwisko, 82052580 as ID> as Osoba;
create <"Janina" as Imie, "Kowalska" as Nazwisko, 83031575 as ID> as Osoba;
create view DataUrodzeniaDef<
  virtual objects DataUrodzenia{ID as ID;},
  on_retrieve do {return <
    <<ID - <ID%1000000>>/1000000> as Rok,
    <<ID%1000000 - ID%10000>/10000> as Miesiac,
    <<ID%10000 - ID%100> / 100> as Dzień};
  },
  on_delete do {delete ID;},
  on_update(n) do {
    ID := <<n.Rok * 100 + n.Miesiac> * 100 + n.Dzien>* 100 +
    ID - <<Rok * 100 + Miesiac> * 100 + Dzień> * 100>;
  }
>;
Osoba.DataUrodzenia.Dzien;
bag <
  25
  15
>
$
```

Definicja perspektywy, która na podstawie numeru identyfikacyjnego osoby wylicza jej datę urodzenia.



```
C:\Documents and Settings\Maciek\Pulpit\Magisterka\Odra\bin\Debug\Odra.exe
create <"Jan" as Imie, "Kowalski" as Nazwisko, 82052580 as ID> as Osoba;
create <"Janina" as Imie, "Kowalska" as Nazwisko, 83031575 as ID> as Osoba;
create view DataUrodzeniaDef{
  virtual objects DataUrodzenia{ID as ID;},
  create view RokDef{
    virtual objects Rok{ID;},
    on_retrieve do {
      return <(ID - (ID%1000000))/1000000>;
    },
    on_update <y> do {
      ID := y * 1000000 + ID - (ID%1000000);
    }
  },
  create view MiesiacDef{
    virtual objects Miesiac{ID;},
    on_retrieve do {
      return <(ID%1000000 - ID%10000)/10000>;
    },
    on_update <m> do {
      ID := m * 10000 + ID - ((ID%1000000 - ID%10000));
    }
  },
  create view DzieńDef{
    virtual objects Dzień{ID;},
    on_retrieve do {
      return <(ID%10000 - ID%100) / 100>;
    },
    on_update <m> do {
      ID := m * 100 + ID - (ID%10000 - ID%100);
    }
  },
  on_retrieve do {return <Dzień, Miesiac, Rok>;},
  on_update<n> do {
    Rok := n.Rok;
    Miesiac := n.Miesiac;
    Dzień := n.Dzień;
  }
};
```

Definicja podobnej perspektywy, jak powyżej. Tym razem generowane są również wirtualne podobiekty, co daje nam większą kontrolę niż w pierwszym przykładzie.



## Rozdział 4

# Podsumowanie

Teoria dotycząca wirtualnych obiektów dla SBQL, pomimo pozornej prostoty, budzi pewne kontrowersje. Poczynając od tego, że wraz z ich wprowadzeniem rozszerzamy przestrzeń wartości obiektów tak, że pokrywa się ona z przestrzenią wyników zapytań, poprzez konflikt pomiędzy funkcją *nested*, a wartością procedury *on\_retrieve*, aż do materializacji wirtualnych obiektów.

Pierwsze z wymienionych zagadnień możemy raczej potraktować jako dodatkową funkcjonalność niż jako problem i szukać możliwości tworzenia obiektów rzeczywistych odpowiadających obiektom wirtualnym, tak, aby i ich wartości mogły pokrywać całą przestrzeń wyników.

Drugie zagadnienie, mam nadzieję, zostało skutecznie rozwiązane poprzez zmianę funkcji *nested* polegającą na wywołaniu *on\_retrieve* i dalej traktowaniu otrzymanej wartości analogicznie jak wartości obiektu rzeczywistego.

Trzecie zagadnienie zostało częściowo pokryte w tej pracy - poprzez wprowadzenie procedur *on\_store* i *on\_store\_reference*, jednak zdaje sobie sprawę, że w tym zakresie pozostało jeszcze wiele do powiedzenia.

Sam system ODRA - prototypowa implementacja przedstawionego podejścia - okazał się łatwy w rozbudowie, a co za tym idzie może być podstawą kolejnych prac implementacyjnych. Kierunki rozwoju mogą być przeróżne - od rozbudowy modelu składu obiektów, aby obsługiwał klasy, czy role obiektów, poprzez kolejne konstrukcje imperatywne - w kierunku pełnego języka programowania, optymalizację zapytań i wiele innych.



## Dodatek A

# Opis dołączonej płyty

### A.1. Zawartość płyty

Na płycie dołączonej do tego opracowania umieszczone są dwa katalogi:

- *Odra*
- *Praca magisterska*

W katalogu *Odra* umieszczony jest kod źródłowy prototypowej implementacji. Jest on podzielony na podkatalogi:

- *Debug* - zawiera kod źródłowy klas odpowiedzialnych za drukowanie na standardowym wyjściu komunikatów związanych z wyliczaniem zapytań.
- *Exceptions* - znajdują się tu klasy związane z wyjątkami, które mogą się podczas przetwarzania zapytań przez poszczególne moduły
- *SBQL* - ten katalog zawiera:
  - *AbstractSyntax* - klasy związane z drzewem syntaktycznym zapytania
  - *Engine* - poszczególne moduły systemu
  - *QueryResult* - klasy odpowiadające poszczególnym rodzajom wyników zapytań
  - *QuerySignature* - sygnatury jakimi mogą zostać opisane zapytania podczas przetwarzania ich przez moduł kontroli typów
  - *SyntacticAnalyzer* - zawiera parser i programy C#CUP, C#LEX służące do jego budowy
- *Store* - w tym katalogu umieszczone zostały klasy odpowiedzialne za skład obiektów
- *bin* - tutaj znajduje się wersja systemu po ostatniej kompilacji
- *Release* - ostatnia stabilna wersja systemu

W katalogu *Praca Magisterska* znajduje się to opracowanie.

## A.2. Instalacja i uruchamianie

W celu instalacji systemu należy skopiować pliki z katalogu *Odra/Release* do katalogu docelowego.

Po uruchomieniu pliku *Odra.exe* znajdziemy się w środowisku tekstowym, za pośrednictwem którego będziemy mogli korzystać z systemu.

# Bibliografia

- [Subi04] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych, 2004.
- [HKoz02] Hanna Kozankiewicz, *Updatable Object Views*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 2002.
- [Piec04] Kazimierz Subieta, Tomasz Pieciukiewicz, *Recursive Query Processing in SBQL*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 2004.
- [Lesz00] Jacek Leszczyłowski, *Technical Aspects of Updatable Views in Object Bases*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 2000.
- [Subi93] Kazimierz Subieta, Catriel Beeri, Florian Matthes, Joachim W. Schmidt, *A Stack Based Approach to Query Languages*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 1993.
- [Subi90] Kazimierz Subieta, *LOQIS: System programowania z bazą danych*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 1990.
- [Jodł99] Andrzej Jodłowski, Kazimierz Subieta, *Dynamiczne role obiektów w modelowaniu pojęciowym i bazach danych*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 1999.
- [Habe04] Piotr Habela, Krzysztof Kaczmarek, Hanna Kozankiewicz, Michał Lentner, Krzysztof Stencel, Kazimierz Subieta, *Data-intensive grid computing based on updatable views*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 2004.
- [Hryń05] Rafał Hryniów, Michał Lentner, Krzysztof Stencel, Kazimierz Subieta, *Types and Type Checking in Stack-Based Query Language*, Instytut Podstaw Informatyki Polskiej Akademii Nauk, 2005.