Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jakub Matusz
Nr albumu: 201071

Optymalizacja zapytań w środowisku rozproszonym w modelu semistrukturalnym

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra Krzysztofa Stencla
Instytut Informatyki

Sierpień 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

Streszczenie

W niniejszej pracy autor wykazuje, że podejście stosowe [Sub95] wprowadza nową jakość w dziedzinie optymalizacji zapytań w rozproszonych bazach danych. Jako potwierdzenie tej tezy przytacza się, wykonaną przez autora, implementację serwera integracyjnego dla rozproszonej semistrukturalnej bazy danych.

Słowa kluczowe

optymalizacja zapytań, rozproszone bazy danych, semistrukturalne bazy danych, obiektowe bazy danych

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2 Database Management
H.2.4 Systems

Tytuł pracy w języku angielskim

Distributed query optimization in semistructural model

Spis treści

Wprowadzenie . . . 5

1. Przetwarzanie zapytań w środowisku rozproszonym . . . 7
1.1. Wstęp . . . 7
1.2. Składowe procesu . . . 7
1.3. Fragmentacja danych . . . 10
1.4. Dekompozycja zapytań . . . 10
1.4.1. Wysyłka wszystkich danych . . . 10
1.4.2. Dekompozycja statyczna . . . 10
1.4.3. Dekompozycja dynamiczna . . . 11
1.4.4. Dekompozycja dynamiczna z przesyłaniem danych . . . 11
1.4.5. Dekompozycja hybrydowa . . . 11
1.5. Indeksy . . . 11
1.6. Model danych a techniki optymalizacji . . . 12
1.6.1. Model relacyjny . . . 12
1.6.2. Model obiektowy . . . 13
1.6.3. Popularne techniki wspólne dla różnych modeli . . . 13
1.7. Inne istotne zagadnienia . . . 14
1.7.1. Czas przetwarzania zapytań . . . 14
1.7.2. Miejsce przetwarzania zapytań . . . 14
1.7.3. Wyznaczniki kosztu . . . 14
1.7.4. Replikacja danych . . . 15

2. Podejście stosowe . . . 17
2.1. Wprowadzenie . . . 17
2.2. Główne założenia . . . 17
2.3. Model danych . . . 18
2.4. Podstawowe mechanizmy . . . 19
2.5. SBQL . . . 20
2.6. Aktualizowalne perspektywy . . . 21
2.7. Wirtualne repozytorium . . . 21
2.8. Wybrane metody optymalizacji zapytań . . . 22
2.8.1. Statyczna ewaluacja . . . 22
2.8.2. Operatory SBQL a przetwarzanie równoległe . . . 23
2.8.3. Usuwanie martwych podzapytań . . . 24
2.8.4. Metoda niezależnych podzapytań . . . 24
2.9. Podsumowanie . . . 25

3. Implementacja rozproszonej bazy danych w oparciu o SBA 27
3.1. Wprowadzenie 27
3.2. Motywacja 27
3.3. Główne założenia 27
3.4. Zarys architektury 28
3.5. Prezentacja wybranych rozwiązań 31
3.5.1. Wybór technologii 31
3.5.2. Obsługa zdarzeń asynchronicznych 31
3.5.3. Schemat przetwarzania zapytań 32
3.5.4. Metabaza 33
3.5.5. Parser zapytań 37
3.5.6. Komunikacja międzyserwerowa 37
3.5.7. Wykonanie zapytania 38
3.5.8. Analiza semantyczna zapytania 38
3.6. Podsumowanie 41

4. Przykłady przetwarzania zapytań 43
4.1. Wprowadzenie 43
4.2. Przykład nr 1 43
4.3. Przykład nr 2 56
4.4. Przykład nr 3 60

5. Podsumowanie 65

A. Opis załączonego oprogramowania 67

Bibliografia 71

Wprowadzenie

Współczesne komercyjne bazy danych są bardzo złożonymi systemami gromadzącymi często ogromne ilości informacji. Taki stan rzeczy rodzi potrzebę stworzenia interfejsu, który uwalniałby użytkownika od znajomości organizacji danych oraz planowania efektywnej metody ich pozyskania. Rolę wspomnianego interfejsu pełnią języki zapytań.

Jedną z najważniejszych cech współczesnych języków zapytań jest deklaratywność. Oznacza to, że użytkownik wskazuje jakie dane go interesują zamiast przekazywania instrukcji, jak takie dane odnaleźć. To z kolei nakłada na system zarządzania bazą danych wymaganie stworzenia odpowiedniego planu wykonania zapytania.

Nawet dla najprostszych zapytań istnieje wiele równoważnych planów [IK84]. Podstawą oceny jakości planu jest koszt jego wykonania. Dotychczas opracowano wiele różnych modeli obliczania wspomnianego kosztu [SY82, PP85]. Zwykle, by zminimalizować czas potrzebny na uzyskanie odpowiedzi stosuje się miary oparte na użyciu procesora (przetwarzanie danych w pamięci operacyjnej), koszcie operacji wejścia-wyjścia (przesyłanie danych miedzy pamięciami masowymi a pamięcią główną) oraz koszcie transmisji danych poprzez sieć. Koszty potencjalnych planów dla danego zapytania mogą różnić się między sobą wieloma rzędami wielkości. Koszt planu wykonania zapytania przekłada się bezpośrednio na czas oczekiwania użytkownika na końcową odpowiedź. Wynika stad, że naiwny wybór strategii realizacji zapytania może prowadzić do nieakceptowalnie niskiej wydajności systemu i stawiać pod znakiem zapytania przydatność takiej bazy danych.

W praktyce często, by nie dopuścić do tego, żeby koszt optymalizacji przewyższył zyski z niej płynące, rezygnuje się z poszukiwania planu optymalnego na rzecz akceptowalnego. Dzieje się tak dlatego, że potencjalnych strategii wykonania może być zbyt wiele, żeby oszacować koszt każdej z nich z osobna. Ponadto ze względu na różnice między środowiskiem optymalizacji a środowiskiem wykonania oraz wielość czynników wpływających na efektywność realizacji zapytania wyliczony koszt planu jest jedynie bardzo zgrubnym przybliżeniem rzeczywistego wykorzystania zasobów w czasie wyliczania zapytania.

W systemach rozproszonych problem optymalizacji zapytań jest daleko bardziej złożony. Obecność fragmentacji (części kolekcji danych są umieszczone na różnych węzłach sieci) oraz replikacji (te same dane są umieszczone na różnych węzłach) stwarza zarówno nowe możliwości, jak i znacznie utrudnia przetwarzanie i optymalizację zapytań. Wielką zaletą takiego środowiska jest możliwość zrównoleglenia wielu operacji co należy uwzględnić przy wyborze planu wykonania. Koordynacja takiego systemu jest jednak dużo trudniejsza. W przypadku szacowania kosztu dochodzi jeszcze istotny czynnik transmisji danych miedzy węzłami, co w przypadku wolnych sieci może stanowić główny wyznacznik efektywności planu.

Prace nad optymalizacją zapytań w rozproszonych bazach danych trwają już od wielu lat [OV99, Kos00, YM98, Ber95, JR02, LS+99], jednak nadal wiele ważnych zagadnień pozostaje otwartych. W przypadku relacyjnych baz danych istnieją skuteczne metody optymalizacyjne, które obejmują jedynie dość proste przypadki pomijając wiele potrzeb występujących w rzeczywistych systemach. Współcześnie istnieje duże zapotrzebowanie dotyczące integracji

wielu różnorodnych źródeł danych oraz na wygodny, mocny obliczeniowo język zapytań, który pozwoliłby w prosty sposób na takich danych operować. Niestety optymalizacja zapytań w obiektowych i semistrukturalnych rozproszonych bazach danych jest wciąż głównie tematem badawczym, co nie pozwala na upowszechnianie tych obiecujących modeli.

Zdaniem autora prezentowane w dalszej cześci pracy podejście stosowe [Sub95] eliminuje wiele wad występujących w wymienionych wyżej rozwiązaniach. Zastosowany tam prosty i uniwersalny model danych daje wsparcie dla danych obiektowych i semistrukturalnych. Zaprojektowany dla podejścia stosowego język zapytań SBQL jest wysokopoziomowy, sformalizowany, w pełni kompozycjonalny, posiada czystą i jednoznaczną semantykę oraz moc obliczeniową zwykłego języka programowania. Kompozycjonalość i dobrze określona semantyka języka stwarzają bardzo duże możliwości w zakresie metod optymalizacyjnych opartych na przepisywaniu oraz pozwalają na znacznie łatwiejszą dekompozycję zapytania na podzapytania wysyłane do poszczególnych serwerów. Cechy te stanowią o atrakcyjności podejścia stosowego jako odpowiedzi na większość potrzeb współczesnych rozproszonych baz danych.

Dalsza część pracy składa się z pięciu rozdziałów. W rozdziale 1 omówiono podstawowe zagadnienia z zakresu przetwarzania zapytań w rozproszonych bazach danych. W rozdziale 2 przedstawiono podejście stosowe i kilka opartych na nim technik przydatnych w realizacji rozproszonych baz danych, ze szczególnym uwzględnieniem możliwości optymalizacyjnych. W rozdziale 3 zaprezentowano implementację serwera integracyjnego i najciekawsze problemy, które wystąpiły podczas jej realizacji. W rozdziale 4 przedstawiono przykłady obrazujące krok po kroku proces przetwarzania zapytań w systemie opisanym w rozdziale 3. W rozdziale 5 dokonano podsumowania oraz przedłożono propozycje kontynuacji badań opisanych w tej pracy. W dodatku A przedstawiono krótki opis instalacji oraz sposobu użytkowania załączonego do pracy oprogramowania.

Rozdział 1

Przetwarzanie zapytań w środowisku rozproszonym

1.1. Wstęp

Optymalizacja jest jednym z wielu elementów przetwarzania zapytań. Mimo to może ona dotyczyć prawie każdej części tego większego procesu. Ponadto model środowiska wykonania w znaczny sposób wpływa na możliwości i charakter samej optymalizacji.

W tym rozdziale przedstawiono ramową architekturę procesora zapytań dla rozproszonej bazy danych oraz podstawowe zagadnienia dotyczące przetwarzania zapytań w środowisku rozproszonym.

1.2. Składowe procesu

Zwykle wejściem procesu przetwarzania jest zapytanie w postaci tekstu sformułowane przez użytkownika. Na początku należy sprawdzić czy napis wyraża zapytanie zbudowane zgodnie z regułami składniowymi dla danego języka zapytań. Jest to zadanie parsera zapytań. Moduł ten jest również odpowiedzialny za wyodrębnienie składowych zapytania i przedstawienie go w postaci wygodniejszej przy dalszym przetwarzaniu. Zwykle wykorzystuje się tu różne formy drzewa składniowego. Złożoność tego przekształcenia jest uzależniona od budowy języka zapytań. W przypadku języków w pełni kompozycjonalnych, jak omawiany dalej SBQL, jest ono bezpośrednie i jednoznaczne. W przypadku większości popularnych języków zapytań, jak SQL, proces ten jest dużo bardziej złożony.

W celu uniknięcia zbędnego kosztu optymalizacji i wykonania zapytania, dąży się do jak najwcześniejszego wykrycia błędów w nim zawartych. Zgodnie z tą zasadą kolejnym etapem jest sprawdzenie zgodności semantycznej z globalnym, znanym użytkownikowi schematem bazy danych. Schemat globalny wyraża ontologię dla danych dostępnych w bazie z punktu widzenia końcowego użytkownika. Dobrą praktyką jest ukrywanie przed użytkownikiem szczegółów dotyczących organizacji danych. Taka orientacja systemu pozwala na znacznie prostsze jego wykorzystanie oraz na bardziej elastyczną i stabilną architekturę. Komunikacja z tak zbudowaną rozproszoną bazą danych niczym nie różni się od operowania na scentralizowanej bazie danych. Podejście stosowe posiada silne wsparcie dla tego modelu w postaci aktualizowalnych perspektyw [KSS04, KSS05a].

System zarządzania bazą danych musi dysponować informacjami na temat lokalizacji udostępnianych przez siebie danych. W rozbudowanych środowiskach musi więc istnieć wewnętrzny schemat, który utrzymuje informacje na temat danych udostępnianych przez poszczególne

serwery udziałowe. Potrzebne jest również odwzorowanie pomiędzy pojęciami widzianymi przez użytkowników a rzeczywistymi danymi. Istnieją rozmaite sposoby wyrażania takich odwzorowań. Mogą to być wyrażenia w specjalnie zaprojektowanym języku czy też różne warianty perspektyw, jak w przypadku modeli opartych na podejściu stosowym.

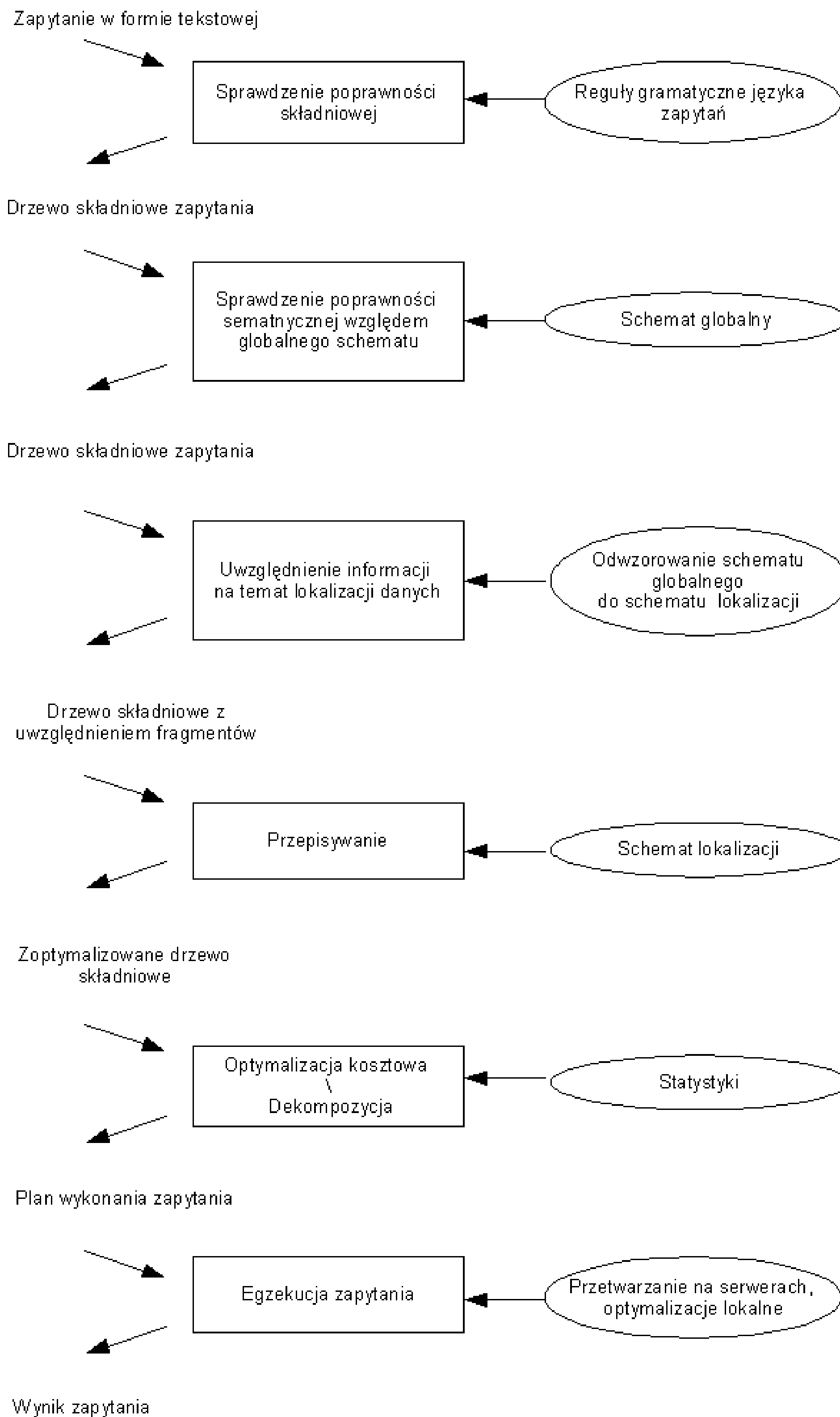
Kolejnym krokiem w procesie przetwarzania zapytania jest rozwinięcie perspektyw i makr, które w nim występują oraz uwzględnienie informacji o lokalizacji danych.

Na tym etapie znajdują zastosowanie optymalizacje oparte na przepisywaniu. Technika ta polega na zamianie zapytania na inne równoważne semantycznie. Żeby w pełni wykorzystać możliwość takich metod język zapytań powinien mieć czystą, jasno określoną semantykę, co osiągnięto w przypadku języka SBQL. Warunek ten nie jest spełniony przez języki zapytań takie jak SQL, czy OQL. Najpopularniejsze techniki polegają na wyodrębnieniu wielokrotnie wyliczanych wyrażeń, tak by ich ewaluacja była jednorazowa, usunięciu fragmentów zapytania, które nic nie wnoszą do końcowego wyniku lub sprowadzeniu zapytania do postaci kanonicznej preferowanej przez optymalizator. Wiele z opisanych przypadków jak martwe czy redundantne podzapytania powstaje w wyniku automatycznego tworzenia treści zapytania, rozwijaniu perspektyw lub zastosowania przepisywania zapytania w trakcie jego przetwarzania. Proces ten ma więc charakter iteracyjny.

Do technik opartych na przepisywaniu należy też wiele metod heurystycznych, w których wykorzystuje się właściwości charakterystyczne dla poszczególnych operatorów.

Dysponując wiedzą o lokalizacji danych można przystąpić do opracowania generalnego planu wykonania zapytania. Najczęściej stosowaną techniką jest ograniczenie się w tym planie do stworzenia harmonogramu wymiany danych między węzłami sieci. W tej fazie znajduje zastosowanie optymalizacja kosztowa. Każdy węzeł jest natomiast odpowiedzialny za lokalną optymalizację sposobu pozyskania danych, które ma dostarczyć. Warto zwrócić uwagę na to, że kompozycjonalość i dobrze zdefiniowana semantyka języka zapytań silnie wspierają ten etap procesu przetwarzania.

Na podstawie wyżej wymienionych faz można nakreślić ramową architekturę procesora zapytań dla rozproszonej bazy danych:



Rysunek 1.1: Ogólna architektura procesora zapytań dla rozproszonej bazy danych

1.3. Fragmentacja danych

Cechą, która ma znaczący wpływ na efektywność procesu optymalizacji jest sposób fragmentacji danych. Jej najpopularniejsze rodzaje wyodrębniono w oparciu o model relacyjny, jednak można w prosty sposób uogólnić te przypadki na model obiektowy.

Z fragmentacją poziomą mamy do czynienia kiedy relacja jest podzielona na zbiory krotek, które znajdują się na różnych serwerach. Może to być na przykład podział względem wartości któregoś z atrybutów relacji. Taki typ fragmentacji jest bardzo popularny i stwarza dość duże możliwości optymalizacyjne.

Innym często spotykanym typem fragmentacji jest fragmentacja pionowa. Polega ona na umieszczeniu wartości grup atrybutów relacji na różnych węzłach. Przykładem może być podział relacji pracownicy na dane personalne oraz te związane z zatrudnieniem.

Fragmentacja hybrydowa jest wynikiem połączenia fragmentacji poziomej i pionowej. Oznacza to, że do podziału relacji stosuje się oba typy kryteriów.

Ciekawym przypadkiem jest tak zwana fragmentacja pochodna (ang. derived fragmentation), która jest wariantem fragmentacji poziomej i dotyczy dwóch relacji. Jeżeli relacja R jest przedmiotem pochodnej fragmentacji poziomej ze względu na relację S, to fragmenty R i S o równych wartościach atrybutu złączenia są umieszczone na tych samych węzłach. Ponadto relacja S może być podzielona zgodnie z fragmentacją poziomą.

1.4. Dekompozycja zapytań

Dekompozycja jest popularną techniką stosowaną do przetwarzania rozproszonych zapytań. Wykorzystuje się ją w modelu z pojedynczym węzłem koordynującym przetwarzanie zapytania. Polega ona na podziale wejściowego zapytania na podzapytania rozsyłane do serwerów z danymi i budowie wyniku na podstawie uzyskanych odpowiedzi. Warunkiem wykorzystania tej metody jest kompozycjonalna budowa języka zapytań. Występuje kilka popularnych rodzajów dekompozycji.

1.4.1. Wysyłka wszystkich danych

Ten wariant polega na pobraniu wszystkich danych na węzeł koordynujący i przetworzenie ich tam w celu uzyskania końcowego wyniku. Takie podejście jest proste koncepcyjnie oraz daje stosunkowo duże możliwości w zakresie optymalizacji opartych na cachingu. Narzut na komunikację sieciową jest jednak ogromny, co powoduje, że stosuje się inne typy dekompozycji wzbogacone ewentualnie przez elementy bazujące na tej technice.

1.4.2. Dekompozycja statyczna

To podejście jest również proste koncepcyjnie i dodatkowo pozwala na bardzo efektywne przetwarzanie. Na podstawie wejściowego zapytania i znajomości rozmieszczenia zasobów w sieci, ustala się listę serwerów dysponujących potrzebnymi danymi. Następnie dla każdego z serwerów komponuje się po jednym zapytaniu w taki sposób, by rozsyłanie zapytań i zbieranie wyników odbywało się równolegle i by odpowiedzi z serwerów wystarczyły do zbudowania końcowego wyniku. Pomimo niekwestionowalnych zalet tego podejścia wiele zapytań nie da się zdekomponować zgodnie z tym schematem. Wariant ten obejmuje głównie przypadki fragmentacji poziomej i zapytań bazujących na operatorach rozłącznych względem sumy (np. selekcja, rzutowanie).

1.4.3. Dekompozycja dynamiczna

Dekompozycja dynamiczna obsługuje szerszą klasę zapytań w porównaniu do dekompozycji statycznej. Dzieje się to jednak kosztem rezygnacji z przetwarzania równoległego. W schemacie tym koordynator na podstawie wejściowego zapytania i danych o lokalizacji zasobów konstruuje zapytanie do pierwszego z serwerów. Otrzymana odpowiedź jest wykorzystywana do stworzenia zapytania do kolejnego serwera. Proces ten jest powtarzany do czasu otrzymania odpowiedzi ze wszystkich serwerów z wymaganymi danymi. Na podstawie cząstkowych wyników konstruowany jest ostateczny rezultat zapytania.

1.4.4. Dekompozycja dynamiczna z przesyłaniem danych

Technika ta bazuje na schemacie dekompozycji dynamicznej, tyle że tutaj zapytania przesyłane na serwery są wspierane dodatkowymi danymi, które mogą posłużyć w celu lokalnych optymalizacji lub ograniczenia wielkości cząstkowego wyniku. Zastosowaniem tej strategii jest jedna z najbardziej popularnych metod stosowanych w celu optymalizacji zapytań w rozproszonych relacyjnych bazach danych - półzłączenia [BG+81] (półzłączenia omówiono dokładniej w podrozdziale dotyczącym optymalizacji).

1.4.5. Dekompozycja hybrydowa

Ten typ dekompozycji zgodnie z nazwą łączy elementy wszystkich omówionych wyżej strategii. Takie połączenie jest w zasadzie dowolne, co powoduje, że jest to najbardziej ogólny schemat.

1.5. Indeksy

Indeksy są narzędziem pozwalającym w niektórych przypadkach zmniejszyć koszt wykonania zapytania o kilka rzędów wielkości. Z tego powodu wiele metod optymalizacyjnych bazuje na ich wykorzystaniu. W przypadku systemów rozproszonych pojawia się wiele wartiantów rozmieszczenia indeksów.

Najprostszym modelem jest indeks scentralizowany. W tym wypadku przechowywany jest on w całości na jednym serwerze. Takie podejście zdecydowanie upraszcza zarówno zarządzanie indeksem, jak i jego wykorzystanie. Ma też swoje wady, ponieważ awaria węzła przechowującego indeks może doprowadzić do zablokowania całej bazy danych. Poza tym taki węzeł jest potencjalnym wąskim węzłem systemu. Mimo to znane są przypadki gdzie wykorzystanie tego rozwiązania okazało się wielkim sukcesem (np. Napster).

Innym skrajnym podejściem jest utrzymywanie lokalnej kopii indeksu na każdej ze stacji klienckich. Zaletą tego rozwiązania jest nieporównywalnie krótszy czas dostępu. Problemem jest natomiast aktualizacja indeksu. Uniknięcie niespójności jest w tym wypadku praktycznie niemożliwe. Dodatkowo w skutek replikacji ilość miejsca zajmowanego przez indeks jest w wymiarze całego systemu bardzo duża.

Za połączenie dwóch powyższych podejść można uznać indeksy rozproszone. Występuje tu jedna kopia indeksu, ale jej części umieszczone są na różnych węzłach. Taki indeks może być zbudowany na zasadzie B+ drzewa lub tablicy haszującej. Przykładami indeksów rozproszonych są Chord [SM+01] przeznaczony do zastosowań P2P oraz SDDS [LNS96] opierający się na działaniu analogicznym do liniowej tablicy haszującej. Rozwiązania tego typu łączą zarówno wady, jak i zalety indeksów scentralizowanych i lokalnie replikowanych. Nie ma tu już pojedynczego wąskiego gardła jak w przypadku centralnego indeksu. Uszkodzenie węzła z

indeksem nie powoduje załamania całego indeksu. Mimo to w takim przypadku dalsza możliwość korzystania z bazy danych nadal pozostaje pod znakiem zapytania. Problem związany z zarządzaniem, choć znacznie mniejszy niż w przypadku indeksów lokalnie replikowanych, nadal pozostaje. Ponadto dochodzi tu istotna kwestia dotycząca usuwania lub dodawania nowych węzłów, kiedy duże części indeksu muszą być przemieszczane.

1.6. Model danych a techniki optymalizacji

Charakter optymalizacji zależy w znacznym stopniu od cech języka zapytań oraz budowy modelu danych.

1.6.1. Model relacyjny

W przypadku baz relacyjnych najkosztowniejszymi operatorami są złączenia. Ponieważ złączenia są łączne i przemienne, a ich koszt zależy bezpośrednio od wielkości argumentów, głównym celem optymalizacji staje się opracowanie odpowiedniej kolejności złączeń.

Do ustalenia takiego harmonogramu najczęściej wykorzystuje się paradygmat programowania dynamicznego [SA+79]. Podejście to polega na stopniowej budowie najlepszego złączenia najpierw dwóch, następnie trzech itd. relacji.

Ze względu na to, że liczba możliwych planów rośnie ekspotencjalnie względem liczby złączanych relacji, stosuje się heurystyki ograniczające przestrzeń możliwych rozwiązań. Jedną z nich jest unikanie tworzenia iloczynów kartezjańskich relacji jeśli to tylko możliwe, ponieważ jest to najbardziej kosztowna postać złączenia. Inną często stosowaną techniką jest ograniczenie rozpatrywanych planów do takich, w których prawy argument złączenia jest zawsze relacją bazową, a nigdy wynikiem pośrednim.

W niektórych wypadkach (np. powyżej 15 złączanych relacji) powyższe restrykcje okazują się niewystarczające. Wykorzystuje się wtedy metody randomizowane jak iteracyjne poprawianie [NSS86, SG88, Swa89], symulowane wyżarzanie [KGV83, IK90, IW87] lub dwufazowa optymalizacja będąca kombinacją dwóch poprzednich [IK90]. Polegają one na wykonywaniu dużej liczby miejscowych ulepszeń planu. Losowość objawia się w wyborze elementu planu, który podlega optymalizacji. W przypadku takich podejść istnieje duże prawdopodobieństwo osiągnięcia nie globalnego a lokalnego optimum. Występuje tu więc kompromis pomiędzy złożonością procesu optymalizacji a jakością generowanego planu.

Inne optymalizacje polegają na zwiększaniu efektywności wykonania pojedynczych złączeń. Do takich technik należy mechanizm półzłączeń [BG+81]. Znajduje on zastosowanie gdy złączane relacje umieszczone są na różnych serwerach a jego głównym założeniem jest przesyłanie tylko tych krotek, które będą brały udział w złączeniu. Zgodnie z tą strategią w przypadku dwóch relacji zamiast pobierać je w całości ze zdalnych serwerów i dokonywać złączenia w węźle koordynującym, stosuje się dodatkowy krok, w którym jeden z serwerów przesyła na drugi krotki składające się z atrybutów złączenia oraz atrybutów występujących w klauzuli select. Na tej podstawie drugi serwer dokonuje złączenia z przechowywaną przez siebie relacją i wysyła wynik do strony koordynującej. Należy zauważyć, że technika ta jest uzasadniona w wypadku, gdy warunek złączenia jest na tyle selektywny, że łączne koszty komunikacji nie przekraczają tych z pierwszego scenariusza. We wczesnych realizacjach rozproszonych baz danych, kiedy sieci były znacznie wolniejsze w stosunku do operacji dyskowych, był to jeden z podstawowych mechanizmów optymalizacyjnych. Obecnie stosuje się go w przypadkach, dla których warunek złączenia w dużym stopniu redukuje wielkość wyniku. Dzieje się tak ze względu na to, że wykorzystanie półzłączeń zwiększa koszty prze-

twarzania po stronie serwerów udziałowych. Opracowano różne warianty tej metody jak filtry Blooma, które wykorzystują wektory bitowe do przybliżenia kolumn złączenia [ML86].

1.6.2. Model obiektowy

W modelu obiektowym ze względu na możliwość bezpośredniej nawigacji użyteczność złączeń jest daleko bardziej ograniczona. Pozwala to postawić nacisk w zakresie optymalizacji na odmienne kwestie. Ponieważ złączenia nie są już głównym czynnikiem decydującym o harmonogramie operacji niezbędnych do wykonania zapytania, można rozważyć wspomniane wcześniej warianty dekompozycji. Należy jednak pamiętać, że warunkiem zastosowania tej techniki jest kompozycjonalna budowa języka zapytań. Język, który jest w pełni kompozycjonalny i ma dobrze zdefiniowaną semantykę stwarza także bardzo duże możliwości w zakresie optymalizacji opartych na przepisywaniu. Przedstawiony w następnym rozdziale SBQL posiada wymienione cechy, dlatego szczegółowe techniki optymalizacyjne dla modelu obiektowego będą omówione podczas prezentacji podejścia stosowego.

1.6.3. Popularne techniki wspólne dla różnych modeli

Poniżej zebrano kilka technik stosowanych niezależnie od modelu danych.

Usuwanie martwych podzapytań

Jeżeli zapytanie w trakcie przetwarzania jest przepisywane do innej postaci (podział relacji na fragmenty, rozwijanie perspektyw, optymalizacje oparte na przepisywaniu) mogą się w nim pojawić części, które nie wnoszą nic do końcowego wyniku. Wykrywanie takich sytuacji eliminuje zbędne operacje oraz błędy w szacowaniu kosztu wykonania zapytania.

Przetwarzanie potokowe

Przetwarzanie potokowe (ang. pipelining) jest techniką znaną z systemów wieloprocesorowych. Złożone procesy można rozbić na łańcuch mniejszych operacji, dla których wyjście jednej jest wejściem następnej. Każdemu elementowi łańcucha przydziela się oddzielny procesor, co w wypadku serii wykonań procesu pozwala zredukować czas jego przetwarzania do czasu wykonania pojedynczej operacji.

Schemat ten można traktować jako alternatywę dla materializacji wyników pośrednich. Dzieje się tak dzięki temu, że wiele operatorów języków zapytań(np. operator selekcji) jest w stanie przetwarzać swoje argumenty element po elemencie.

Środowisko rozproszone jest z natury systemem wieloprocesorowym, co stwarza możliwość wykorzystania pipeliningu jako efektywnej metody optymalizacji przetwarzania zapytań.

Caching

Caching polega na przechowywaniu wyników często występujących zapytań. Wykorzystanie tej techniki pozwala na uniknięcie czasochłonnej konstrukcji rezultatu. W modelach bazujących na cachingu często stosuje się metodę polegającą na budowie planów zapytań, które optymalizują wykorzystanie cache w przyszłości kosztem bieżącej efektywności.

W dziedzinie rozproszonych baz danych efektywnym sposobem implementacji cachingu są materializowalne perspektywy [Hal01]. Optymalizacja zapytań w oparciu o tak zbudowany system polega na dekompozycji zapytania na podzapytania bazujące na dostępnych perspektywach. W przypadku istnienia wielu perspektyw dostarczających te same dane, wybór najlepszego planu jest bardzo skomplikowanym zagadnieniem [ACN00, GM05, MR+01].

Głównym problemem przy zastosowaniu tej formy optymalizacji jest utrzymywanie spójności cache z zawartością składowiska danych.

Globalna optymalizacja

W systemach rozproszonych baz danych w tej samej chwili często wykonuje się jednocześnie wiele zapytań. Wielokrotnie dotyczą one tych samych danych i zwracają podobne wyniki. Bazując na tych obserwacjach, zamiast osobno optymalizować plany wykonania poszczególnych zapytań, można dążyć do osiągnięcia maksymalnej efektywności wykonania wszystkich bieżących zapytań rozpatrywanych łącznie. Podejście to może przyczynić się do znacznego obniżenia średniego kosztu wykonania zapytania kosztem planów dla niektórych zapytań.

Wykorzystanie statystyk

W przypadku optymalizacji kosztowej istotna staje się znajomość rozmiarów przetwarzanych kolekcji czy dystrybucji wartości w poszczególnych atrybutach. Ponieważ obliczanie tych parametrów na bieżąco byłoby skrajnie nieopłacalne, system utrzymuje statystyki dotyczące wspomnianych danych. Trudność w zarządzaniu statystykami stwarza fakt, że należy dążyć do kompromisu pomiędzy jakością dostarczanych przez nie informacji a szybkością dostępu do nich oraz minimalizacją narzutu na ich aktualizację i przechowywanie.

1.7. Inne istotne zagadnienia

1.7.1. Czas przetwarzania zapytań

W rzeczywistych systemach baz danych wiele zapytań jest wykonywanych wielokrotnie. Taka sytuacja ma miejsce najczęściej w przypadku zapytań zawartych w kodzie aplikacji. W takim wypadku bardzo efektywnym rozwiązaniem byłoby jednorazowe ustalenie planu wykonania, czyli tak zwana optymalizacja statyczna. Pomimo zalet takiej techniki ma ona jedno poważne ograniczenie. W czasie optymalizacji nie jest znany dokładny stan środowiska wykonania. Z czasem tak wygenerowany plan może się stać skrajnie nieoptymalny. Żeby uniknąć tej sytuacji można zastosować okresowe przebudowywanie planu lub monitorowanie stanu środowiska i tworzenie nowej strategii obliczeń w momencie, gdy zmiany spowodują nieadekwatność dotychczasowego rozwiązania. Alternatywą dla optymalizacji statycznej jest optymalizacja dynamiczna mająca miejsce w momencie wykonywania zapytania. Traci się tu wszystkie zalety opisanego wyżej rozwiązania, ale w zamian za to uzyskuje dokładne informacje o środowisku wykonania, co pozwala na ustalenie optymalnego planu.

1.7.2. Miejsce przetwarzania zapytań

Środowisko rozproszone daje możliwość zrównoleglenia obliczeń nie tylko podczas wykonania zapytania, ale również podczas jego optymalizacji. Jest to obiecujące podejście, jednak w większości współczesnych systemów ustalanie planu wykonania odbywa się na jednym węźle, głównie ze względu na prostotę takiego rozwiązania.

1.7.3. Wyznaczniki kosztu

Głównymi wyznacznikami kosztu w środowisku rozproszonym są: użycie procesora, operacje wejścia-wyjścia oraz komunikacja sieciowa. Ze względu na możliwość przetwarzania równoległego istnieją jednak dwa główne podejścia do szacowania sumarycznego kosztu.

W pierwszym z nich bierze się pod uwagę sumę kosztów przetwarzania na poszczególnych serwerach. W drugim liczy się czas odpowiedzi czyli okres od momentu wysłania zapytania do momentu zebrania końcowego wyniku.

Nie da się w sposób jednoznaczny wskazać lepszego podejścia. Pierwsze stawia na efektywność systemu jako całości, która może w istotny sposób rzutować na czas wykonania innych zapytań. Z drugiej strony w pewnym sensie dyskredytuje to strategie opierające się na przetwarzaniu równoległym, dzięki którym można uniknąć sytuacji wąskich gardeł i znacznie skrócić czas przetwarzania poszczególnych zapytań.

1.7.4. Replikacja danych

Replikacja polega na przechowywaniu wielu kopii tych samych danych na różnych węzłach sieci. Zaletą takiego rozwiązania jest to, że awaria jednego z serwerów nie grozi utratą danych oraz, że w takiej sytuacji nadal mamy do nich nieprzerwany dostęp. Taka organizacja zasobów stwarza też duże możliwości optymalizacyjne. Można uniknąć w ten sposób sytuacji wąskiego gardła, lepiej wykorzystać topologię sieci oraz uzyskać większe wsparcie dla przetwarzania równoległego. Problemem jest natomiast aktualizacja redundantnych danych. Z tego powodu większość systemów nie wykorzystuje w pełni możliwości stwarzanych przez replikację.

Rozdział 2

Podejście stosowe

2.1. Wprowadzenie

W niniejszym rozdziale przedstawiono podstawy podejścia stosowego (SBA - Stack Based Approach) oraz cechy czyniące je doskonałym fundamentem do implementacji rozproszonych systemów baz danych. Więcej na temat SBA oraz jego zastosowań znaleźć można w [Sub95, Sub04].

2.2. Główne założenia

Główną ideą podejścia stosowego jest potraktowanie języka zapytań jak zwykłego języka programowania ogólnego przeznaczenia. Dąży się tu do wyeliminowania tak zwanej "niezgodności impedancji", polegającej na konflikcie natury koncepcyjnej i realizacyjnej, który występuje podczas łączenia w kodzie aplikacji dwóch wspomnianych wyżej typów języków.

Celem SBA jest stworzenie języka zapytań, który posiadałby następujące cechy:

- wysoki poziom abstrakcji
- prostotę użycia
- przystosowanie do przetwarzania masowych danych
- niezależność od organizacji danych
- moc obliczeniowa języka programowania ogólnego przeznaczenia
- silne wsparcie dla optymalizacji

Na bazie tych wytycznych powstał język SBQL (Stack Based Query Language). Zamierzone założenia osiągnięto dzięki:

- prostemu ale jednocześnie bardzo uniwersalnemu modelowi danych
- sformalizowanej, czystej semantyce operacyjnej wszystkich konstrukcji językowych
- kompozycjonalnej budowie (argumentami operatorów są dowolne poprawnie zbudowane zapytania)
- wykorzystanie stosu środowiskowego (stąd nazwa podejście stosowe) do ustalenia reguł nazywania, zakresu i wiązania (analogicznie jak w językach programowania ogólnego przeznaczenia)

2.3. Model danych

Model danych w SBA pełni rolę podstawy, na której opiera się semantykę operacyjną języka zapytań. Ponieważ każda komplikacja modelu przekłada się na komplikację semantyki języka, zgodnie z zasadą brzytwy Occama dąży się do maksymalnego uproszczenia modelu.

Ze względu na to, że trudno jest pogodzić minimalizm z możliwością wsparcia dla obrazowania skomplikowanych pojęć, stworzono rodzinę modeli różniących się złożonością i funkcjonalnością [Sub04]:

- M0 - pozwala na dowolne zagnieżdżanie obiektów oraz powiązania między nimi
- M1 - wzbogaca M0 o klasy i statyczne dziedziczenie
- M2 - rozszerza M1 o koncepcję dynamicznych ról i dynamicznego dziedziczenia
- M3 - pozwala uzupełnić M1 lub M2 o hermetyzację atrybutów i metod

Warto zauważyć, że już M0 przykrywa struktury występujące w bazach danych opartych o XML i inne modele semistrukturalne oraz podejście relacyjne. Ponieważ rozszerzenie M0 do innych modeli jest stosunkowo proste w dalszej części rozdziału definicję podejścia stosowego oparto na tym właśnie modelu.

Każdy obiekt składowiska danych posiada:

- unikalny wewnętrzny(nie może być jawnie wykorzystywany w zapytaniu) identyfikator
- nazwę używaną do adresowania obiektu w zapytaniach (nie musi być unikatowa)
- wartość (atomowa, pointerowa lub złożona)

Jeśli I to zbiór identyfikatorów, poprzez N oznaczymy zbiór nazw, a zbiór wartości atomowych(liczby, napisy, stałe logiczne itp. oraz abstrakcje programistyczne jak kody procedur, metod lub perspektyw) wyrazimy jako V, to możemy zdefiniować dostępne typy obiektów w następujący sposób:

- obiekt atomowy - $< i, n, v >$, gdzie $i \in I$, $n \in N$ oraz $v \in V$
- obiekt pointerowy - $< i_1, n, i_2 >$, gdzie $i_1, i_2 \in I$, $n \in N$
- obiekt złożony - $< i, n, S >$, gdzie $i \in I$, $n \in N$, a S jest dowolnym zbiorem obiektów

Ponieważ zwykle chcemy, żeby w bazowym zakresie widoczne były tylko niektóre obiekty, składowisko powinno zawierać dodatkowo informację, które z nich będą dostępne bezpośrednio. Takie obiekty nazywamy obiektami korzeniowymi. Do innych danych dostajemy się za pomocą nawigacji poprzez składowe obiektów złożonych oraz poprzez obiekty pointerowe.

Poniżej przedstawiono przykładowe składowisko danych:

Obiekty:

< i_1, Pracownik, {< i_2, identyfikator, 1001 >,
< i_3, nazwisko, "Kowalski">,
< i_4, pensja, 3200>,
< i_5, dział, i_{16} >}>
< i_6, Pracownik, {< i_7, identyfikator, 1002 >,
< i_8, nazwisko, "Nowak">,
< i_9, pensja, 1500>,
< i_{10}, dział, i_{16} >}>
< i_{11}, Pracownik, {< i_{12}, identyfikator, 1002 >,
< i_{13}, nazwisko, "Zieliński">,
< i_{14}, pensja, 2200>,
< i_{15}, dział, i_{19} >}>
< i_{16}, Dział, {< i_{17}, nazwa, "księgowość">,
< i_{18}, lokalizacja, "Warszawa">}>
< i_{19}, Dział, {< i_{20}, nazwa, "produkcja">,
< i_{21}, lokalizacja, "Gliwice">}>

Obiekty korzeniowe:

$\{i_1, i_6, i_{11}, i_{16}, i_{19}\}$

2.4. Podstawowe mechanizmy

Podstawę podejścia stosowego stanowi stos środowisk. Jest to rozwiązanie znane z realizacji popularnych języków programowania. Pozwala ono programiście na opanowanie złożoności problemu poprzez podzielenie go na podzagadnienia zgodnie z zasadą "dziel i rządź" i wyodrębnienie mniejszych, niezależnych części kodu. Kawałki te definiują środowiska w których wybranym nazwom przyporządkowane są poszczególne byty programistyczne jak zmienne czy procedury. Struktura stosu pozwala na odwzorowanie zagnieżdżania takich środowisk.

W SBA stos środowisk składa się z sekcji, w których umieszczane są zbiory obiektów nazwanych binderami. Każdy binder jest uporządkowaną parą, w której pierwszy element stanowi nazwa, drugi natomiast jest identyfikatorem obiektu składowiska danych.

Wiązanie nazwy występującej w zapytaniu polega na znalezieniu zbioru binderów, których pierwszym elementem jest zadana nazwa. Wynikiem operacji jest zbiór obiektów (nazwa nie musi być unikatowa) wskazywanych przez identyfikatory występujące w zwróconych binderach. W tym celu przeszukujemy stos środowisk od góry (zaczynając od najbardziej lokalnego środowiska) i zwracamy bindery znajdujące się w pierwszej sekcji, w której występuje wiązana nazwa.

Strukturą służącą do przechowywania wyników pośrednich jest tak zwany stos rezultatów. Jego rozmiar zmienia sie także zgodnie z wchodzeniem do nowych środowisk oraz ich opuszczaniem.

Sekcje pojawiające się na stosie środowisk tworzone są za pomocą funkcji *nested*, która działa w następujący sposób:

- dla identyfikatora obiektu złożonego zwraca zbiór binderów do jego podobiektów
- dla identyfikatora obiektu pointerowego zwraca binder opisujący wskazywany obiekt
- dla dowolnego bindera jest to przekształcenie torzsamościowe

- dla struktury zwraca zbiór bedacy suma wyników *nested* dla kolejnych jej elementów
- w pozostałych przypadkach wynik przekształcenia jest pusty

2.5. SBQL

SBQL (Stack Based Query Language) jest językiem zapytań zbudowanym na podstawie podejścia stosowego. Jego składnia opiera się na zasadzie kompozycjonalności:

- dowolny literał jest zapytaniem ("Kowalski", 3)
- dowolna nazwa jest zapytaniem ($Pracownik$)
- jeśli q jest zapytaniem, a σ operatorem unarnym, to σ q jest poprawnie zbudowanym zapytaniem (-1, avg($dataUrodzenia$))
- jeśli q_1 i q_2 są zapytaniami, a θ jest operatorem binarnym, to q_1 θ q_2 jest poprawnie zbudowanym zapytaniem

Ze względu na schemat przetwarzania operatory SBQL można podzielić na dwie podstawowe grupy.

Pierwszą z nich są tak zwane operatory algebraiczne, których podstawową cechą jest niezależność ewaluacji ich argumentów. W takim wypadku w pierwszym kroku oblicza się wyniki dla podzapytań-argumentów. Należy zauważyć, że ze względu na wspomnianą niezależność, można obliczać poszczególne rezultaty w dowolnej kolejności, co sprawia, że ta grupa operatorów daje wsparcie dla przetwarzania równoległego. Do operatorów algebraicznych zaliczamy wszystkie operatory unarne oraz wybrane operatory binarne takie jak: porównania, operacje arytmetyczne, działania logiczne, operacje na zbiorach, iloczyn kartezjański itp.

Drugą grupę operatorów stanowią operatory niealgebraiczne. Tutaj kolejność ewaluacji argumentów jest już ściśle określona. Polega ona na wyliczaniu prawego argumentu w środowisku ustalonym przez lewy argument. Dokładniej dla każdego elementu rezultatu lewego podzapytania oblicza się wynik działania funkcji *nested*. Wynik ten jest umieszczany na czubku stosu środowiskowego i stanowi kontekst dla wyliczenia prawego podzapytania. Rezultaty ewaluacji prawego argumentu otrzymane w kolejnych iteracjach są łączone zgodnie z semantyką operatora tworząc w ten sposób ostateczny wynik zapytania. Ważnym wnioskiem płynącym z tego schematu jest to, że operatory algebraiczne implikują pętle, które mogą być czynnikiem decydującym o koszcie wykonania zapytania. Ze względu na tę cechę są one szczególnym przedmiotem optymalizacji. Do operatorów niealgebraicznych należą między innymi: selekcja (where), projekcja i nawigacja (operator kropki), złączenie zależne (join), kwantyfikatory (for all, exists), operator porządkujący (order by) oraz operatory rekurencyjne.

Wyżej zdefiniowany język można w łatwy sposób rozszerzyć o konstrukcje imperatywne, w tym instrukcję przypisania. W SBQL pełni ona rolę uogólnienia klauzuli update znanej z SQL. W obszarze składni traktuje się ją analogicznie do operatorów binarnych, co pozwala na zagnieżdżanie przypisań w większych zapytaniach.

Oparcie się na stosie środowiskowym pozwala też w stosunkowo prosty sposób wzbogacić język o możliwość deklarowania procedur i metod. W składzie danych byty te traktowane są jako obiekty atomowe, gdzie wartością jest kod procedury bądź metody. Pozwala to na wykorzystanie paradygmatu programowania obiektowego do projektowania bazy danych oraz operowania na niej. W SBQL nie występuje teoretyczne ograniczenie na głębokość rekursji.

2.6. Aktualizowalne perspektywy

Perspektywy okazały się bardzo użytecznym narzędziem wykorzystywanym podczas konstrukcji baz danych. Pozwalają one na zwiększenie poziomu abstrakcji zapytań, stwarzają możliwość kontroli dostępu do danych, a także mogą być silnym wsparciem dla optymalizacji zapytań w hurtowniach danych oraz rozproszonych bazach danych.

Ponieważ perspektywy są bytami abstrakcyjnymi, istotnym ograniczeniem ich zastosowania był zawsze problem aktualizacji. O ile łatwo wyobrazić sobie zwiększenie pensji dla perspektywy zwracającej dane pracowników działu "Księgowość", to w przypadku perspektywy podającej średni wiek pracowników jakiekolwiek aktualizacje jej wyniku wydają się być pozbawione sensu.

Do tej pory przedstawiono wiele pomysłów jak radzić sobie z tym problemem. W większości przypadków rozwiązania te można jednak uznać za niewystarczające, ponieważ wprowadzają nadmierną komplikację do semantyki języka oraz implikują zbyt daleko idące ograniczenia.

Aktualizacja perspektyw w podejściu stosowym [KLS02] jest rozwinięciem idei zastosowanej w tzw. tiggerach *instead of* znanych z systemów bazodanowych Oracle i MS SQL Server. Definicja perspektywy składa się z dwóch części. W pierwszej określa się przekształcenie obiektów dostępnych w składowisku danych na obiekty wirtualne. Wynikiem takiego przekształcenia jest zbiór tak zwanych ziaren. Ziarna wykorzystuje się w drugiej części definicji perspektywy, gdzie explicite definiuje się operacje, które są na niej dostępne. Wyodrębniono następujące typy działań na wirtualnych obiektach perspektyw: dereferencję (zwraca wartość wirtualnego obiektu), wstawianie, usuwanie oraz aktualizację. Aby umożliwić daną operację należy zdefiniować procedurę o odpowiedniej nazwie, odpowiednio: on_retrieve, on_insert, on_delete, on_update.

Perspektywy w SBQL mogą być dowolnie zagnieżdżone, co pozwala na konstrukcję złożonych obiektów wirtualnych.

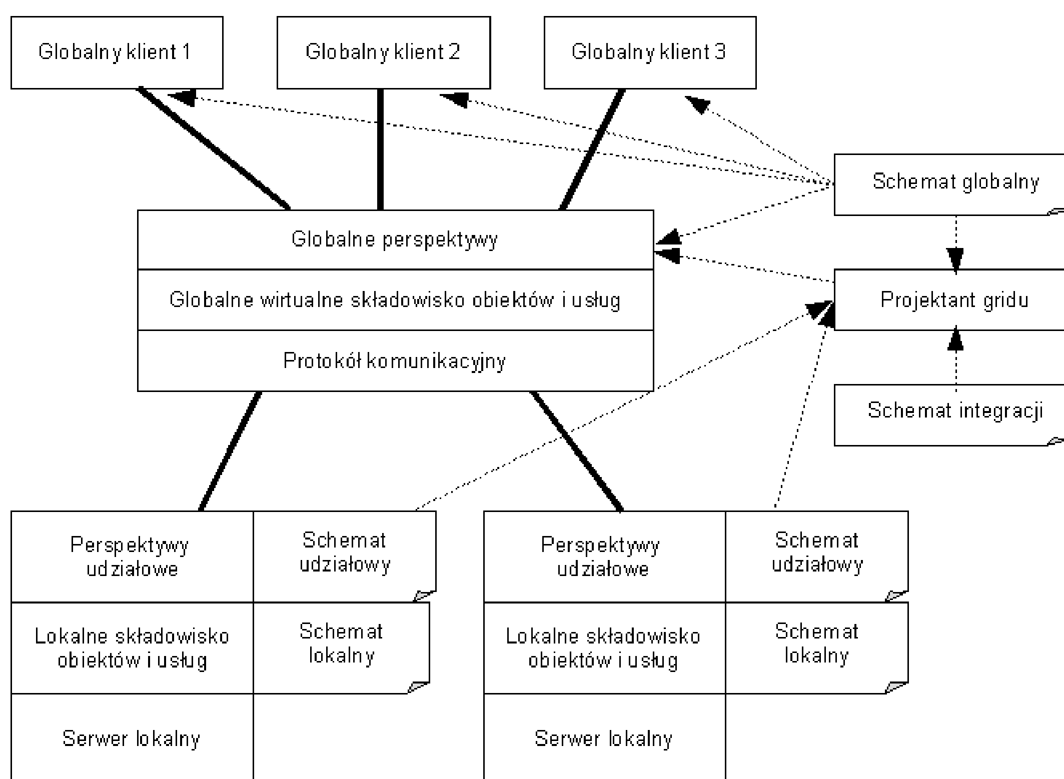
2.7. Wirtualne repozytorium

W oparciu o podejście stosowe powstała ciekawa koncepcja architektoniczna dotycząca rozproszonych baz danych oraz integracji heterogenicznych źródeł danych, tak zwane globalne wirtualne repozytorium danych i usług [KSS04, KSS05a]. Udostępnia ono zasoby zgromadzone na różnych węzłach sieci. Wewnętrzna struktura repozytorium jest całkowicie przezroczysta dla końcowych użytkowników. Z ich punktu widzenia posługiwanie się takim systemem nie różni się od korzystania ze scentralizowanej bazy danych.

Prezentowana architektura opiera się na przedstawionych w poprzednim podrozdziale aktualizowalnych perspektywach. Służą one do konstrukcji interfejsów udostępnianych przez poszczególne serwery uczestniczące w integracji. Mechanizm aktualizowalnych perspektyw jest wykorzystywany także przez wirutalne repozytorium do określenia odwzorowania wspomnianych interfejsów w interfejs globalny.

Należy zauważyć, że dzięki zastosowanym technikom można konstruować w ten sposób bardzo złożone przekształcenia przy zachowaniu dużej elastyczności, prostoty użycia i skalowalności rozwiązania. Jest to zatem bardzo obiecujące podejście biorąc pod uwagę zapotrzebowania współczesnych systemów bazodanowych.

Poniżej przedstawiono ogólną architekturę gridu opartego na idei wirtualnego repozytorium (na podstawie [KSS05b]):



Rysunek 2.1: Architektura gridu opartego na wirtualnym repozytorium

2.8. Wybrane metody optymalizacji zapytań

Obszerne zestawienia technik optymalizacyjnych zaprojektowanych dla scentralizowanych systemów opartych o SBA można znaleźć w [Plo00, Sub04]. Tu przedstawiono w skrócie jedynie wybrane metody, które znajdują szczególne zastosowanie w przypadku środowiska rozproszonego.

2.8.1. Statyczna ewaluacja

Pełne wykorzystanie możliwości optymalizacyjnych jest możliwe dopiero na poziomie semantycznym. W SBQL w celu zbadania semantyki zapytania używa się tak zwanej statycznej ewaluacji. W procesie tym symuluje się prawdziwe środowisko wykonania. Składowisko danych zostaje zastąpione przez schemat bazy, który jest grafem wyrażającym ograniczenia na występowanie i powiązania obiektów w składowisku. Stosom środowisk i rezultatów przyporządkowywane są statyczne stosy, w sekcjach których umieszcza się sygnatury elementów spodziewanych na stosach środowiska wykonania (np. $bag(int)$ dla $bag\{1, 25, -15\}$). Schemat statycznej ewaluacji jest analogiczny do tego, zgodnie którym dokonywane jest rzeczywiste obliczenie wyniku zapytania. Ponieważ w tym przypadku operuje się na sygnaturach, wykonywanie pętli implikowanych przez operatory niealgebraiczne jest całkowicie zbędne. Ta cecha powoduje, że statyczna ewaluacja zwykle przebiega wielokrotnie szybciej od rzeczywistej ewaluacji.

Wynikiem opisywanego procesu jest wzbogacenie drzewa składniowego zapytania o informacje semantyczne wykorzystywane później podczas optymalizacji i dekompozycji zapytania. W węzłach nazw zapisuje się wysokość stosu środowiskowego podczas wiązania nazwy oraz

numer sekcji, w której dokonywane jest to wiązanie. W przypadku środowiska rozproszonego uwzględnia się również identyfikatory serwerów, z których pochodzić będą wiązane z nazwą obiekty. Dla każdego węzła operatora niealgebraicznego zapamiętuje się informację o numerach sekcji otwieranych na stosie środowiskowym przez dany operator.

2.8.2. Operatory SBQL a przetwarzanie równoległe

Algebraiczne operatory binarne doskonale nadają się do przetwarzania równoległego, ponieważ ich argumenty mogą być wyliczane niezależnie. Ponadto istnieje grupa operatorów o bardzo przydatnej w tym względzie właściwości.

Operator θ nazwiemy rozdzielnym względem sumy, jeśli dla dowolnych zapytań q_1, q_2, q_3 zachodzi:

$$(q_1 \cup q_2)\,\theta\, q_3 \;=\; (q_1\,\theta\, q_3) \,\cup\, (q_2\,\theta\, q_3)$$

Równoważnie, jeśli θ jest operatorem rozdzielnym względem sumy, q_1 jest zapytaniem zwracającym zbiór $\{r_1, r_2, \ldots, r_n\}$, a q_2 dowolnym zapytaniem, to mamy:

$$q_1\,\theta\, q_2 \;=\; (r_1\,\theta\, q_2) \,\cup\, (r_2\,\theta\, q_2) \,\cup \ldots \cup\, (r_n\,\theta\, q_2)$$

Do operatorów rozdzielnych względem sumy należą operatory: selekcji (where), nawigacji (kropka), join oraz operacje na zbiorach (różnica, suma i iloczyn) i konstruktor struktur.

Ważną obserwacją jest to, że powyższa własność pozwala na przesuwanie operatorów bliżej serwerów. Cecha ta odnajduje zastosowanie głównie w przypadku poziomej fragmentacji danych, gdzie globalna kolekcja jest sumą fragmentów przechowywanych na poszczególnych węzłach.

Poniższy przykład pomaga zrozumieć tę zależność. Załóżmy, że dane pracowników umieszczone są na dwóch serwerach. Na pierwszym znajdują się dane pracowników oddziału Kieleckiego na drugim zaś dane pracowników oddziału z Bydgoszczy. Użytkownik rozproszonej bazy danych widzi je jako globalną kolekcję obiektów etykietowanych nazwą *Pracownik*. Naiwna ewaluacja zapytania

$$Pracownik\;\textbf{where}\;nazwisko \;=\; \text{"Kowalski"}$$

polegałaby na pobraniu wszystkich danych pracowników na węzeł koordynujący i obliczeniu wyniku selekcji dopiero tam. Dzięki wykorzystaniu własności rozdzielności wystarczy wysłać do każdego serwera wejściowe zapytanie a następnie zsumować wyniki. W przypadku dużej kolekcji pracowników oraz przy wysokiej selektywności warunku zysk, w sensie zmniejszenia kosztu wykonania, z zastosowania takiej metody mógłby być ogromny.

W SBQL istnieje więcej operatorów, dla których można wykorzystać podobną technikę ewaluacji. Należą do nich kwantyfikatory i agregacje. Poniżej przedstawiono reguły, dzięki którym można przesunąć te operatory bliżej serwerów.

$$\forall(q_1 \cup q_2 \cup \ldots \cup q_n)q \;=\; \forall(q_1)q \,\wedge\, \forall(q_2)q \,\wedge \ldots \wedge \forall(q_n)q$$

$$\exists(q_1 \cup q_2 \cup \ldots \cup q_n)q \;=\; \exists(q_1)q \,\vee\, \exists(q_2)q \,\vee \ldots \vee \exists(q_n)q$$

$$\text{sum}(q_1 \cup q_2 \cup \ldots \cup q_n) \;=\; \text{sum}(q_1) \,+\, \text{sum}(q_2) \,+ \ldots + \text{sum}(q_n)$$

$$\text{avg}(q_1 \cup q_2 \cup \ldots \cup q_n) \;=\; \frac{\text{sum}(q_1) \,+\, \text{sum}(q_2) \,+ \ldots + \text{sum}(q_n)}{\text{count}(q_1) \,+\, \text{count}(q_2) \,+ \ldots + \text{count}(q_n)}$$

2.8.3. Usuwanie martwych podzapytań

Martwe podzapytania to te części zapytania, które nie mają wpływu na jego końcowy wynik. Fragmenty takie powstają zwykle podczas automatycznego ustalania treści zapytania, np. zapytania generowane za pomocą interfejsu graficznego, optymalizacje zapytań oparte na przepisywaniu, rozwijanie perspektyw.

Przykładem może być zapytanie:

$$\begin{aligned} &\mathit{Pracownik}\ .\ (\mathit{dzial}\ .\ \mathit{Dział}\ .\ \mathit{lokalizacja}\ \mathbf{as}\ \mathit{oddział},\\ &\qquad \text{pensjaNetto}(\mathit{pensja})\ \mathbf{as}\ \mathit{poboryNetto},\\ &\qquad \mathit{nazwisko}\ \mathbf{as}\ \mathit{nazwisko})\ .\ \mathit{nazwisko} \end{aligned}$$

W tym wypadku składowe *oddział* i *poboryNetto* są wyliczane niepotrzebnie, a całe zapytanie mogłoby być zredukowane do postaci:

$$\mathit{Pracownik}\,.\,\mathit{nazwisko}$$

Nie trudno się domyślić, że w przypadku rozwijania skomplikowanych perspektyw, które mogą okazać się użyteczne podczas realizacji rozproszonych baz danych, zysk z eliminacji martwych podzapytań może być bardzo duży.

W SBQL do powstawania martwych zapytań przyczyniają się operatory zwracające struktury lub kolekcje struktur. Należą do nich konstruktor struktur oraz operator zależnego złączenia. Ponadto operatorami, które mogą pomijać części swoich argumentów przy przetwarzaniu są kwantyfikatory oraz operator nawigacji.

W przypadku zapytania:

$$(\mathit{Pracownik}\ \mathbf{where}\ \mathit{nazwisko} = \text{"Kowalski"})\ .\ \mathit{Dział}$$

lewy argument operatora kropki, będący potencjalnym martwym zapytaniem może wpływać istotnie na wielkość wyniku. W tym przypadku redukcja tego fragmentu nie prowadzi więc do semantycznie równoważnego zapytania. Przyjmuje się więc ograniczenie, że takie podzapytanie można usunąć wyłącznie wtedy, kiedy zwracany przez nie wynik zawiera dokładnie jeden element. Warunek ten daje gwarancję zachowania liczności końcowego rezultatu.

Bazując na mechanizmie będącym rozszerzeniem statycznej ewaluacji można łatwo wykryć, które nazwy w zapytaniu nie uczestniczą w tworzeniu końcowego rezultatu. Na podstawie informacji zawartych w schemacie bazy danych możliwe jest też oszacowanie liczebności wyników poszczególnych podzapytań. Pozwala to na odszukanie i usunięcie poddrzew drzewa składniowego, odpowiadających martwym podzapytaniom.

2.8.4. Metoda niezależnych podzapytań

Nadmiarowe wyliczanie zapytań występuje nie tylko w sytuacjach opisanych w poprzednim punkcie. W poniższym przykładzie:

$$\mathit{Pracownik}\ \mathbf{where}\ \mathit{pensja} > \text{avg}(\mathit{Pracownik}\,.\,\mathit{pensja})$$

prawy argument operatora porównania jest wyliczany w każdym obrocie pętli iterującej po obiektach *Pracownik*. Nie jest to jednak konieczne. Optymalizacja mogłaby polegać w tym wypadku na jednokrotnym wyliczeniu wartości tego argumentu na początku i wykorzystaniu jej przy każdorazowej ewaluacji warunku selekcji. Zapytanie mogłoby być zatem przepisane do postaci:

$$(\text{avg}(\mathit{Pracownik.pensja})\ \mathbf{as}\ \mathit{srednia})\ .\ \mathit{Pracownik}\ \mathbf{where}\ \mathit{pensja} > \mathit{srednia}$$

Algorytm optymalizacji bazuje na porównaniu numerów sekcji stosu środowiskowego otwieranych przez operator niealgebraiczny z numerami sekcji, w których wiązane są nazwy w danym podzapytaniu. Jeśli wszystkie nazwy występujące w podzapytaniu wiązane są w sekcjach poniżej tych otwieranych przez operator, oznacza to, że jest ono niezależne od tego operatora i całe zapytanie można przepisać zgodnie z powyższym schematem. Wyjątkiem jest podzapytanie składające się z pojedynczej nazwy pomocniczej (tu tego typu optymalizacja nie jest już potrzebna).

Należy zauważyć, że przedstawiona metoda jest uogólnieniem bardzo popularnych technik, znanych z relacyjnych baz danych, stosowanych do optymalizacji zagnieżdżonych zapytań oraz polegających na jak najwcześniejszym wykonywaniu operatorów typu selekcja czy złączenie.

Jak się okazuje podobny mechanizm można wykorzystać do zwiększenia efektywności przetwarzania zapytań w środowisku rozproszonym. W wielu wypadkach statyczna dekompozycja zapytania nie jest możliwa ze względu na to, że zapytania wysyłane do serwerów zawierają podzapytania odwołujące się do danych umieszczonych na innych węzłach. Można wyłączyć takie podzapytania za pomocą metody niezależnych podzapytań i obliczyć ich wynik na początku przetwarzania globalnego zapytania. Następnie, jeśli wynik podzapytania składa się z pojedynczej wartości, może on je zastąpić w zapytaniach wysyłanych do poszczególnych serwerów. W przypadku innych, dostatecznie małych wyników, można je przesyłać razem ze wspomnianymi zapytaniami. W pozostałych przypadkach znajdują zastosowanie techniki bazujące na schematach przypominających wspomniane wcześniej półzłączenia.

2.9. Podsumowanie

W zakresie rozproszonych baz danych istnieje bardzo duże zapotrzebowanie na elastyczne i efektywne rozwiązania. Techniki stosowane we współczesnych systemach napotykają jednak wciąż na wiele ograniczeń koncepcyjnych. Ta sytuacja skłania do badań nad nowymi ideami, które pozwolą na przyspieszenie rozwoju dziedziny.

Podejście stosowe jest w pełni sformalizowane, opiera się na uniwersalnym modelu danych oraz w pełni kompozycjonalnym języku zapytań o naturalnej, jednoznacznej semantyce i dużej mocy obliczeniowej. Cechy te pozwalają na uogólnienie i rozszerzenie większości mechanizmów wykorzystywanych we współczesnych systemach bazodanowych. Dotyczy to w szczególności architektury rozproszonych baz danych oraz możliwości optymalizacji zapytań.

Rozdział 3

Implementacja rozproszonej bazy danych w oparciu o SBA

3.1. Wprowadzenie

W tym rozdziale zaprezentowano wykonaną przez autora w oparciu o podejście stosowe implementację serwera integracyjnego dla rozproszonej semistrukturalnej bazy danych. W pierwszych podrozdziałach przedstawiono główne cele oraz założenia systemu. W dalszej części omówiono jego generalną architekturę i opisano wybrane problemy dotyczące jego realizacji.

3.2. Motywacja

We wcześniejszej części tej pracy pokazano, że SBA jest obiecującym podejściem w zakresie realizacji rozproszonych baz danych. Celem jaki postawił sobie autor przed stworzonym przez siebie systemem było zapoczątkowanie budowy platformy umożliwiającej implementację i rozwijanie rozwiązań dotyczących wspomnianej dziedziny ze szczególnym uwzględnieniem optymalizacji.

3.3. Główne założenia

Stworzenie kompleksowego rozwiązania w zakresie rozproszonych baz danych jest zadaniem wymagającym wieloletniego wysiłku dużej grupy osób. Proces ten należy więc podzielić na mniejsze etapy, w których system będzie stopniowo wzbogacany o nowe możliwości. Ze względu na w dużej mierze badawczy charakter prac, wiele z zastosowanych pierwotnie rozwiązań należy z czasem weryfikować i zastępować przez sprawniejsze realizacje poszczególnych zagadnień.

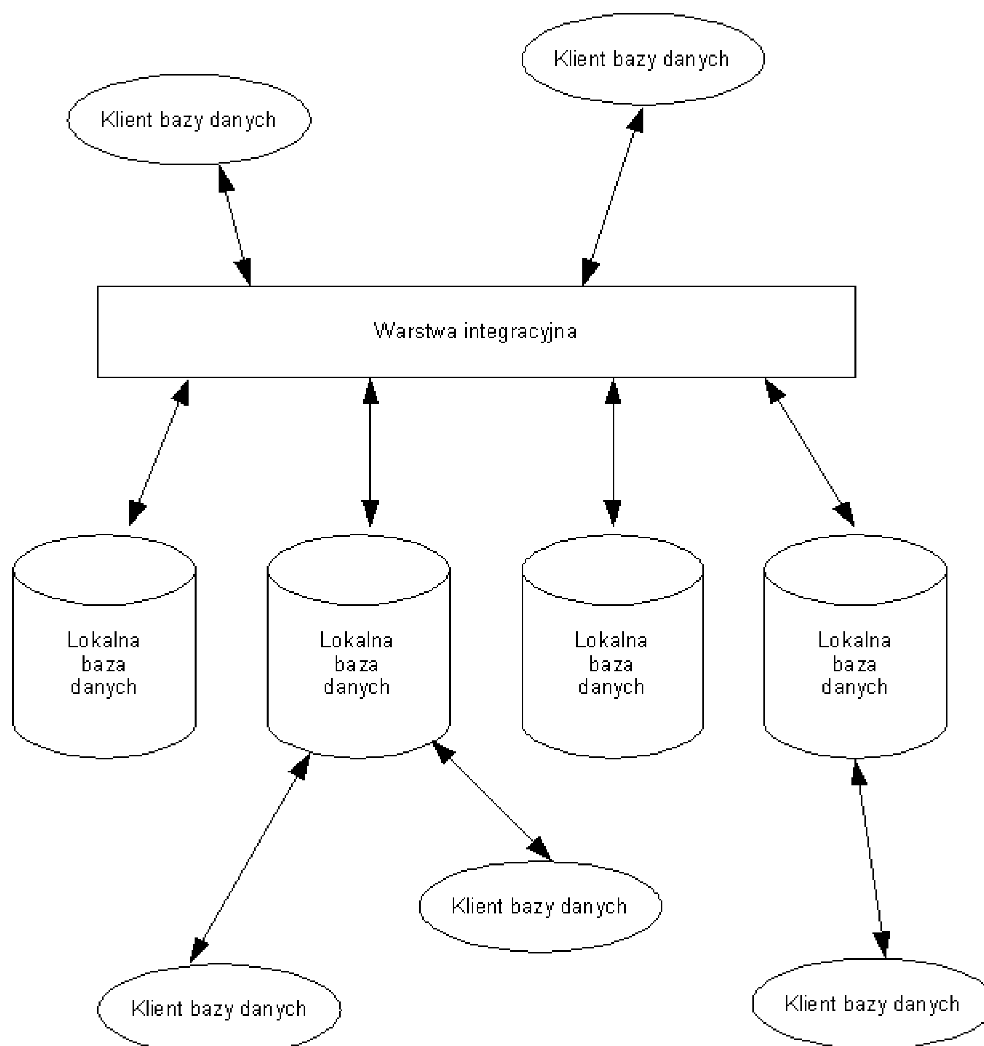
Zgodnie z tym schematem prezentowany system stanowi przede wszystkim wstęp do dalszych zadań koncepcyjnych i implementacyjnych.

Nieco przed rozpoczęciem opisywanych tutaj prac na Uniwersytecie Warszawskim zapoczątkowano budowę bazy danych opartej na podejściu stosowym. Celem platformy tworzonej przez autora stała się więc integracja zasobów rozproszonych w bazach danych tego typu. Ze względu na równoległy przebieg prac nad rozwojem warstwy integracyjnej oraz wspomnianej scentralizowanej bazy danych, postanowiono w maksymalnym stopniu uniezależnić od siebie realizacje tych dwóch systemów. Dzięki takiemu podejściu osiągnięto również bardziej elastyczną budowę całego rozwiązania.

Biorąc pod uwagę złożoność zagadnienia przyjęto ograniczenia dotyczące klasy obsługiwanych zapytań. System ma za zadanie wspierać zapytania podlegające dekompozycji statycznej, w przypadku której jednakowe zapytanie wysyłane jest do grupy serwerów, a końcowy wynik jest sumą otrzymanych odpowiedzi. Podczas dekompozycji zapytania należy wykorzystać własność rozdzielności względem sumy dla operatorów selekcji, nawigacji oraz zależnego złączenia. Operacje modyfikacji danych nie są wspierane.

Wbrew pozorom taki zakres pozwala na realizację dość znacznej funkcjonalności w dziedzinie integracji dla poziomej fragmentacji danych. Dodatkowo zaproponowany tutaj schemat przetwarzania zapytań w sposób bardzo efektywny wykorzystuje budowę systemu oraz możliwość przetwarzania równoległego.

3.4. Zarys architektury



Rysunek 3.1: Ramowy schemat organizacji systemu

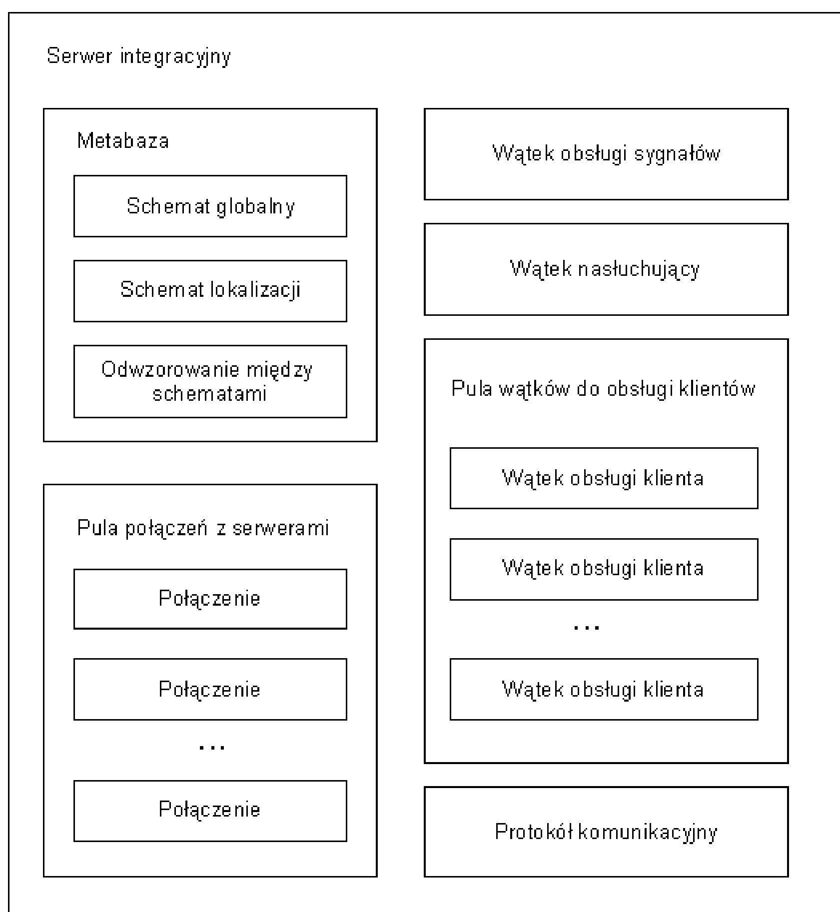
Na powyższym schemacie przedstawiono generalną organizację systemu rozproszonej bazy

danych oraz umiejscowienie warstwy integracyjnej. Podstawę rozwiązania stanowią lokalne bazy danych. Są to samodzielne jednostki udostępniające określone dane i usługi. Dostęp do nich jest możliwy za pomocą aplikacji klienckich obudowujących protokół komunikacyjny.

W proponowanym tutaj modelu z punktu widzenia lokalnych jednostek rozproszona baza danych jest całkowicie niewidoczna. Warstwa integracyjna łączy się bowiem z nimi za pomocą zwykłego protokołu klienckiego. Podejście to pozwala na bardzo znaczące uniezależnienie platformy integracyjnej od implementacji lokalnych baz danych.

Globalne zasoby udostępniane są przez warstwę integracyjną za pomocą tego samego protokołu, który jest wykorzystywany przez lokalne bazy. Wynika stąd, że lokalna aplikacja kliencka może równie dobrze pełnić rolę globalnego klienta. Oznacza to też, że zgodnie z opisywanym schematem rozproszona baza danych tego typu może brać udział w większej integracji jako baza lokalna.

Poniżej przedstawiono główne komponenty serwera integracyjnego:



Rysunek 3.2: Główne komponenty serwera integracyjnego

Zadaniem metabazy jest dostarczenie metainformacji na temat danych zawartych w składowisku. W prezentowanym modelu jej najważniejszymi elementami są schemat globalny, schemat lokalizacji oraz przekształcenie pierwszego schematu w drugi. Schemat globalny opisuje dane i usługi udostępniane przez rozproszoną bazę z punktu widzenia globalnego klienta. Schemat lokalizacji odzwierciedla zasoby dostarczane przez serwery biorące udział w integracji. Ostatni z elementów metabazy umożliwia przejście z poziomu abstrakcji schematu

globalnego, na którym opierają się przesyłane przez klientów zapytania, do operacji na rozproszonych zasobach z wykorzystaniem wiedzy o umiejscowieniu poszczególnych obiektów.

Wątek obsługi sygnałów, jak wskazuje jego nazwa, przejmuje zarządzanie asynchronicznymi zdarzeniami w postaci UNIXowych sygnałów. Scentralizowanie obsługi tego typu sytuacji umożliwia jej elegancką implementację oraz lepszą kontrolę nad systemem w tym zakresie.

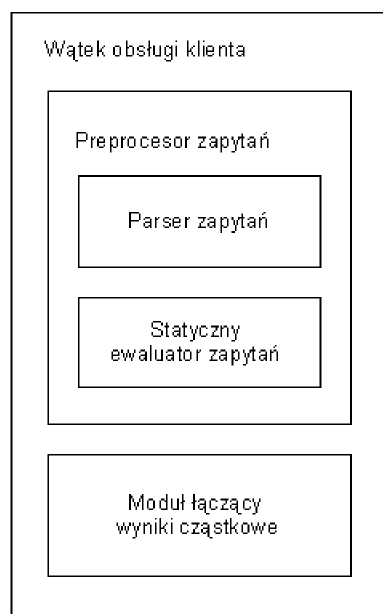
Wątek nasłuchujący jest głównym wątkiem serwera integracyjnego. Służy on przede wszystkim do akceptacji połączeń klientów oraz zarządzania pulą wątków obsługujących połączenia z poszczególnymi klientami.

Rolą protokołu komunikacyjnego jest obudowanie podstawowych operacji służących do wymiany danych przez sieć. Protokół jest wykorzystywany przez inne części systemu, jak: wątek nasłuchujący, wątki do obsługi połączeń z klientami oraz połączenia z serwerami.

Połączenia z serwerami koordynują komunikację między serwerem integracyjnym a serwerami baz danych biorącymi udział w integracji. Udostępniają one wysokopoziomowy interfejs do przesyłania zapytań oraz zbierania ich wyników.

Komponentem zarządzającym połączeniami z serwerami jest pula połączeń. Jej głównym zadaniem jest koordynacja przydziału połączeń do obsługi zapytań globalnych klientów.

Centralnym elementem pod względem przetwarzania zapytań przesyłanych przez globalnego klienta jest wątek obsługi klienta. Każdy klient połączony z serwerem integracyjnym ma przydzielony jeden taki wątek. Główne składowe tego komponentu przedstawione są poniżej:



Rysunek 3.3: Główne składowe wątku obsługi klienta

Zadaniem preprocesora zapytań jest wygenerowanie planu wykonania zapytania na podstawie jego tekstowej postaci. W opisywanym tu modelu preprocesor ma dwie główne składowe. Parser odpowiedzialny jest za analizę syntaktyczną zapytania oraz przedstawienie go w postaci drzewa składniowego. Zadaniem statycznego ewaluatora jest przede wszystkim analiza semantyczna. Ponieważ w prezentowanym systemie ograniczono się do statycznej dekompozycji zapytań, generowanie planu wykonania odbywa się w locie podczas statycznej ewaluacji.

Moduł łączący wyniki cząstkowe służy do scalania wyników zapytań rozesłanych na serwery udostępniające swoje dane i usługi warstwie integracyjnej.

3.5. Prezentacja wybranych rozwiązań

3.5.1. Wybór technologii

Głównym czynnikiem branym pod uwagę podczas wyboru technologii realizacji warstwy integracyjnej była łatwość migracji rozwiązań między implementacją warstwy a wspomnianą wcześniej implementacją lokalnej bazy danych. Co za tym idzie posłużono się tymi samymi technologiami.

Kod systemu został napisany w języku C++. Pozwoliło to na kompromis pomiędzy wydajnością rozwiązania a możliwością programowania na wysokim poziomie abstrakcji. Implementacja została stworzona pod systemem operacyjnym Linux z wykorzystaniem mechanizmów charakterystycznych dla platform UNIXowych, opartych na standardzie POSIX, jak: sygnały, łącza nienazwane, wątki z biblioteki pthread oraz asynchroniczne wejście-wyjście. Komunikacja sieciowa bazuje na protokole TCP/IP i standardowych UNIXowych gniazdach. Do implementacji popularnych struktur danych wykorzystano bibliotekę STL.

3.5.2. Obsługa zdarzeń asynchronicznych

Ze względu na to, że funkcjonowanie bazy danych opiera się na współdziałaniu wielu wątków i procesów, zdarzają się sytuacje, w których podczas przetwarzania danych przez grupę wątków w innej części systemu zachodzi zdarzenie istotne dla tej grupy. Wymaga to odpowiedniego mechanizmu propagacji informacji o zdarzeniu. Ponieważ w przypadku zdarzeń asynchronicznych nie możemy przewidzieć momentu ich wystąpienia, komplikuje to ich obsługę i prowadzi do trudnych do wykrycia błędów implementacyjnych. W proponowanych rozwiązaniach dąży się do maksymalnego uproszczenia kodu odpowiedzialnego za obsługę takich sytuacji przy zachowaniu odpowiedniej szybkości propagacji informacji o zdarzeniu.

Wyłączanie serwera integracyjnego

Wyłączenie serwera integracyjnego odbywa się poprzez wysłanie do jego procesu sygnału INT. Można uzyskać ten efekt poprzez wciśnięcie kombinacji klawiszy $ctrl + c$ na terminalu, na którym został uruchomiony serwer. Zgodnie ze specyfikacją POSIX taki sygnał trafia do dokładnie jednego, nieokreślonego wątku działającego na tym terminalu. Standardowo powoduje on natychmiastowe zakończenie procesu. Zwykle jednak serwer musi wykonać szereg czynności zanim zostanie zamknięty.

W celu uzyskania pełnej kontroli nad tą sytuacją wszystkie wątki blokują dostarczenie tego typu sygnału. Powoływany jest dodatkowy wątek, którego zadaniem jest wychwycenie opisywanego zdarzenia i propagacja informacji o nim. Jako jedyny nie blokuje on dostarczenia sygnału i wykorzystuje funkcję systemową *sigwait* do zawieszenia swego działania do momentu nadejścia sygnału.

Propagacja informacji o tym zdarzeniu odbywa się za pomocą łącz nienazwanych(pipe). Wątek koordynujący obsługę sygnałów zapisuje tam bajt informujący o nadejściu sygnału. Przed tą czynnością łącze pozostaje puste. W kontekście globalnym dostępna jest funkcja, która na podstawie stanu łącza, informuje inne wątki o tym, czy serwer jest zamykany.

Podczas wykonywania blokujących operacji wejścia-wyjścia wątki serwera korzystają z funkcji *poll* pozwalającej na obserwację wielu deskryptorów na raz. Na jej liście argumentów

zawsze umieszczany jest deskryptor wspomnianego łącza, co umożliwia uniknięcie trwałego zablokowania zamknięcia serwera poprzez operacje wejścia-wyjścia.

Błąd komunikacji między serwerami

Przetwarzanie zapytań w opisywanym systemie wymaga często komunikacji warstwy integracyjnej z wieloma serwerami. Błąd podczas takiej komunikacji może doprowadzić do sytuacji, w której niemożliwe będzie skonstruowanie ostatecznego wyniku. W takim wypadku kontynuowanie czasochłonnych operacji związanych z przetwarzaniem tego zapytania prowadzi do marnotrawstwa czasu i zasobów systemowych. W szczególności dotyczy to przesyłania danych pomiędzy serwerami, koordynowanego przez niezależne wątki.

Propagacja informacji o błędzie odbywa się za pomocą mechanizmu analogicznego do stosowanego w przypadku zamykania serwera. Wątek zarządzający scalaniem wyników przesłanych z poszczególnych serwerów posiada łącze nienazwane, którego końcówka służąca do odczytu jest udostępniana wątkom połączeń. W przypadku błędu do łącza zapisywany jest odpowiedni bajt. Tu także wykorzystuje się funkcję *poll* do obudowania blokujących operacji wejścia-wyjścia.

Rozłączenie się klienta

Implementacja protokołu TCP/IP pozwala na łatwe wykrycie rozłączenia się klienta. Jeżeli sytuacja ta zdarzy się podczas przetwarzania zapytania, to ze względu na oszczędność zasobów systemowych, warto byłoby wykonanie takiego zapytania jak najszybciej zakończyć. Wymaga to stałego monitorowania statusu połączenia z klientem.

Opisywana tutaj warstwa integracyjna nie posiada wsparcia dla rozwiązania tego problemu ze względu na to, że w czasie jej implementacji takie rozwiązanie było testowane dla wykorzystywanej w projekcie scentralizowanej bazy danych. Zakłada się włączenie gotowego mechanizmu po zakończeniu jego testów.

3.5.3. Schemat przetwarzania zapytań

Schemat przetwarzania zapytań zastosowany w implementacji nie odbiega od przedstawionego w Rozdziale 1.2. Poniżej przedstawiono jedynie ogólny zarys procesu, a opisy poszczególnych etapów znajdują się w dalszej części pracy.

Wejściem przetwarzania jest zapytanie w formie tekstowej, na podstawie którego parser tworzy drzewo składniowe. Kolejnym krokiem jest analiza semantyczna, która składa się z dwóch głównych części. W pierwszej z nich zapytanie badane jest względem schematu globalnego. Podczas tej fazy następuje także rozwijanie makr, dzięki czemu następuje przepisanie zapytania na bazujące na schemacie lokalizacji. W drugim etapie zapytanie analizowane jest względem schematu lokalizacji. Badana jest możliwość statycznej dekompozycji zapytania z wykorzystaniem własności rozdzielności operatorów względem sumy. Jeśli taka dekompozycja jest możliwa wątek przetwarzający zapytanie przekazuje kontrolę do modułu scalającego wyniki, który zgłasza zapotrzebowanie na niezbędne połączenia z serwerami biorącymi udział w integracji. Po otrzymaniu przydziału połączeń, wątki z nimi związane równolegle wysyłają zapytania do poszczególnych serwerów i zbierają wyniki. Moduł scalający wyniki konstruuje ostateczny rezultat zapytania na podstawie odpowiedzi od zdalnych węzłów. Na końcu procesu połączenia są zwracane do puli połączeń, a ostateczny rezultat zapytania wysyłany do klienta.

3.5.4. Metabaza

Zadaniem metabazy jest umożliwienie dostępu do informacji o zasobach zawartych w bazie danych. W jej skład wchodzą trzy główne elementy: schemat globalny, schemat lokalizacji oraz odwzorowanie pomiędzy tymi schematami. Schemat globalny reprezentuje zasoby dostępne z punktu widzenia globalnego użytkownika. Odzwierciedlane są w nim informacje na temat: nazw etykietujących zasoby, powiązań między danymi, typów przechowywanych wartości oraz liczebności poszczególnych obiektów w danym kontekście. Schemat lokalizacji tworzony jest na wewnętrzne potrzeby warstwy integracyjnej i nie jest widoczny dla użytkowników. Zawiera on informacje o zasobach udostępnianych przez serwery biorące udział w integracji. Sposób opisu tych zasobów przypisanych do poszczególnych serwerów jest taki sam jak w przypadku schematu globalnego. Ponieważ zapytania do bazy danych opierają się na schemacie globalnym, a ich plany wykonania bazują na schemacie lokalizacji, potrzebny jest trzeci element umożliwiający przejście między tymi poziomami abstrakcji, to znaczy odwzorowanie między schematem globalnym a schematem lokalizacji. Co za tym idzie, wyraża on sposób integracji danych w rozproszonej bazie. W SBA mamy doskonałe wsparcie dla realizacji wspomnianego odwzorowania w postaci aktualizowalnych perspektyw [KLS02]. Zastosowanie takiego rozwiązania zwiększyło by jednak bardzo znacząco zakres prac koncepcyjnych i implementacyjnych, dlatego też zdecydowano się na prostsze wyjście w postaci makr integracyjnych, które opisano w dalszej części tego podrozdziału. Kompromis ten okazał sie w pełni wystarczający na tym etapie realizacji systemu.

Z punktu widzenia przetwarzania zapytań istotna jest sama obecność metabazy oraz dostęp do jej zawartości. Moduł administracyjny pozwalający na zarządzanie nią można traktować jako w pełni ortogonalny podsystem bazy danych. Należy zauważyć, że implementacja operacji umożliwiających modyfikacje metabazy stanowi sama w sobie ciekawe i dość złożone zadanie. W opisywanym tu systemie nie ma wsparcia dla tego typu operacji, a cała konfiguracja zapisana jest bezpośrednio w kodzie ładującym metabazę do pamięci serwera.

W implementacji warstwy integracyjnej dostęp do metabazy możliwy jest za pomocą globalnego obiektu inicjalizowanego podczas startu serwera. Ponieważ nie występują operacje modyfikujące metabazę, korzystanie z niej nie wymaga dodatkowej synchronizacji. Zastosowanie pojedynczego obiektu, który zawiera w sobie schematy globalny i lokalny oraz odwzorowanie między nimi czyni architekturę systemu bardziej przejrzystą i elastyczną. Takie podejście ułatwia wszelkie zmiany dotyczące organizacji metadanych oraz zarządzania nimi.

Schemat globalny i schemat lokalizacji

Oba schematy zostały zrealizowane przez tę samą klasę. Bazą schematu jest graf obrazujący powiązania między obiektami składowiska danych. Każdy węzeł takiego grafu reprezentuje grupę obiektów składowiska i posiada następujące atrybuty:

- identyfikator - w unikalny sposób identyfikuje każdy węzeł schematu
- nazwa - służy do odwoływania się do grupy obiektów z poziomu zapytań
- liczebność - wyraża ograniczenie na liczebność grupy obiektów

Różnym typom obiektów składowiska odpowiadają różne typy węzłów schematu. Węzeł reprezentujący grupę obiektów atomowych posiada atrybut informujący o typie wartości przechowywanej w obiektach grupy. Odpowiednik grupy obiektów pointerowych zawiera referencję do węzła odwzorowującego obiekty wskazywane przez rzeczywiste obiekty pointerowe. Węzeł

symbolizujący grupę obiektów złożonych zawiera listę referencji do węzłów reprezentujących odpowiednie podobiekty.

Informacja o lokalizacji danych jest odzwierciedlana w schemacie lokalizacji za pomocą węzłów serwerów, które są zbudowane analogicznie do węzłów obiektów złożonych. W tym przypadku rolę podobiektów pełnią węzły reprezentujące obiekty korzeniowe danego serwera.

Schemat składa się z dwóch głównych elementów. Pierwszy, w przypadku schematu globalnego, jest listą węzłów reprezentujących obiekty korzeniowe, natomiast w przypadku schematu lokalizacji zawarta jest tu lista węzłów serwerów. Drugi element jest tablicą asocjacyjną pozwalającą szybko odnaleźć węzeł schematu na podstawie jego identyfikatora.

Odwzorowanie schematu globalnego na schemat lokalizacji

Jak wspomniano wcześniej, idealnym rozwiązaniem w tym miejscu byłyby aktualizowalne perspektywy, ale ze względu na to, że ich realizacja sama w sobie stwarza wiele dodatkowych wyzwań, zdecydowano się na zastosowanie kompromisu w postaci makr. Makro jest poprawnie zbudowanym zapytaniem zgodnym ze schematem lokalizacji, etykietowanym unikatową, w zakresie zbioru wszystkich makr, nazwą. Każdej nazwie obiektu korzeniowego ze schematu globalnego odpowiada dokładnie jedno makro etykietowane tą nazwą. Podczas przetwarzania zapytania nazwy wiązane w bazowej sekcji stosu zastępowane są przez zapytania występujące w odpowiednich makrach.

Ponieważ, zgodnie z gramatyką języka SBQL, dowolne zapytanie może występować w każdym z kontekstów dopuszczalnych dla nazw, opisana zamiana prowadzi zawsze do zapytania poprawnego składniowo. W skutek braku dodatkowych ograniczeń powstałe w wyniku tego mechanizmu zapytanie może jednak nie być zgodne semantycznie ze schematem lokalizacji. W czasie przetwarzania zapytania ta zgodność jest więc weryfikowana.

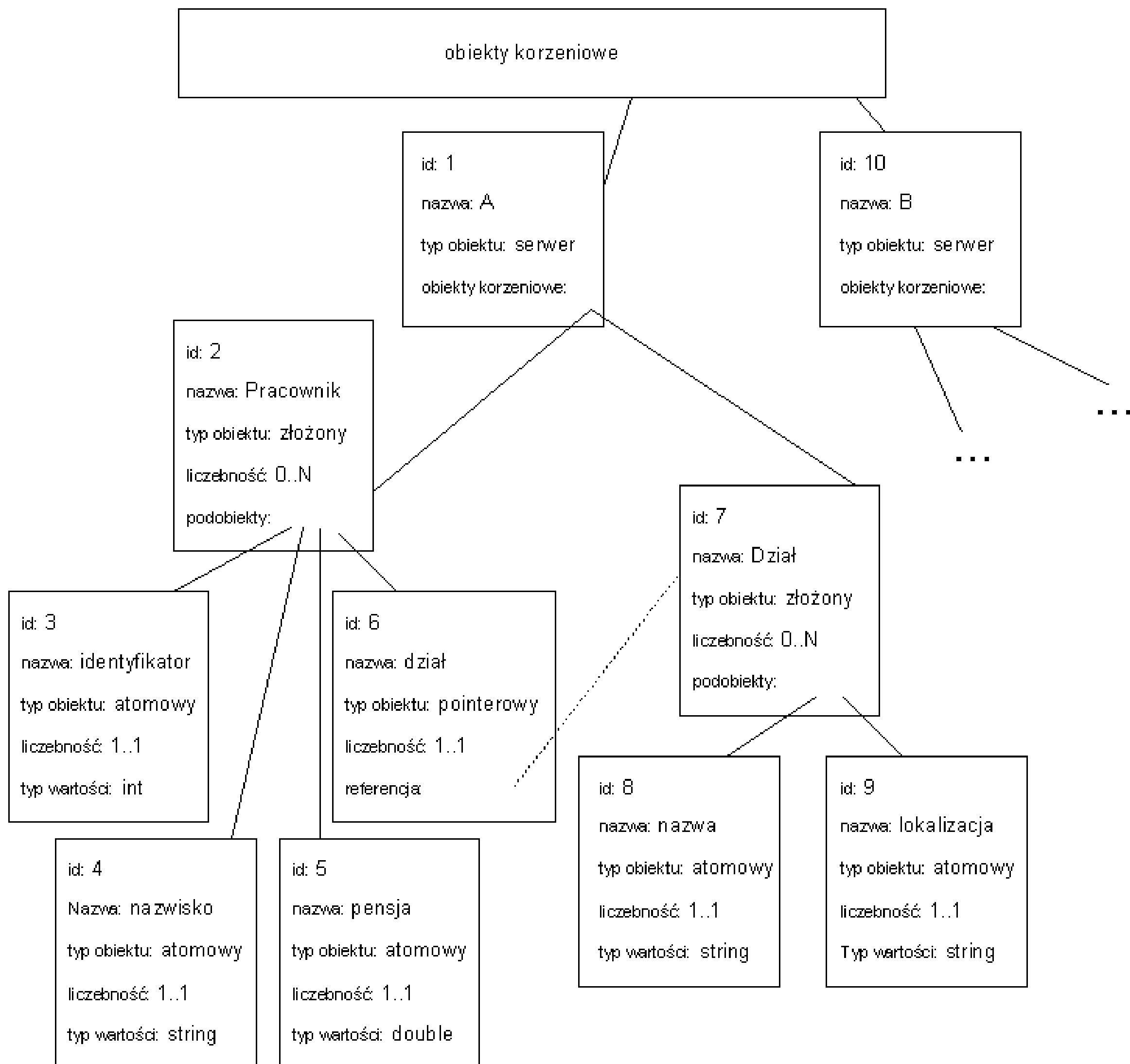
Wszelkie operacje polegające na przepisywaniu zapytania do innej postaci odbywają się na poziomie drzewa składniowego. Dlatego też zapytania w makrach nie są przechowywane w postaci tekstu, ale w formie drzew składniowych. Proces rozwijania makr polega więc na modyfikacji drzewa składniowego zapytania w postaci zamiany węzłów nazw na drzewa makr. Dokonuje się to w locie podczas statycznej ewaluacji zapytania bazującej na schemacie globalnym.

Przykład metabazy

W tym podpunkcie przedstawiono fragmenty przykładowej metabazy. Na rysunku 3.4 znajduje się graf będący podstawą dla schematu lokalizacji. Dla uproszczenia pokazano tylko dane udostępniane przez serwer A. Możemy przyjąć, że dla serwera B organizacja zasobów będzie taka sama. Jak widać w przypadku schematu lokalizacji, rolę zbioru obiektów korzeniowych pełni lista serwerów biorących udział w integracji. Z każdym węzłem reprezentującym serwer związana jest lista referencji do węzłów reprezentujących obiekty korzeniowe tego serwera. W poniższym przypadku serwer A udostępnia obiekty symbolizujące pracowników oraz działy przedsiębiorstwa. Każdy pracownik posiada: identyfikator będący liczbą całkowitą, napis zawierający nazwisko, wysokość pensji reprezentowaną przez liczbę zmiennoprzecinkową oraz referencję do działu, w którym pracuje. Reprezentacja działu w bazie danych składa się z jego nazwy oraz lokalizacji wyrażonych w postaci napisów. Na potrzeby schematu wyróżniono kilka standardowych oznaczeń liczebności:

- 0..1 - element opcjonalny mogący wystąpić co najwyżej raz w danym kontekście
- 1..1 - element obowiązkowy o dokładnie jednym wystąpieniu

- 0..N - element opcjonalny o dowolnej liczbie wystąpień
- 1..N - element obowiązkowy o dowolnej (większej od zera) liczbie wystąpień



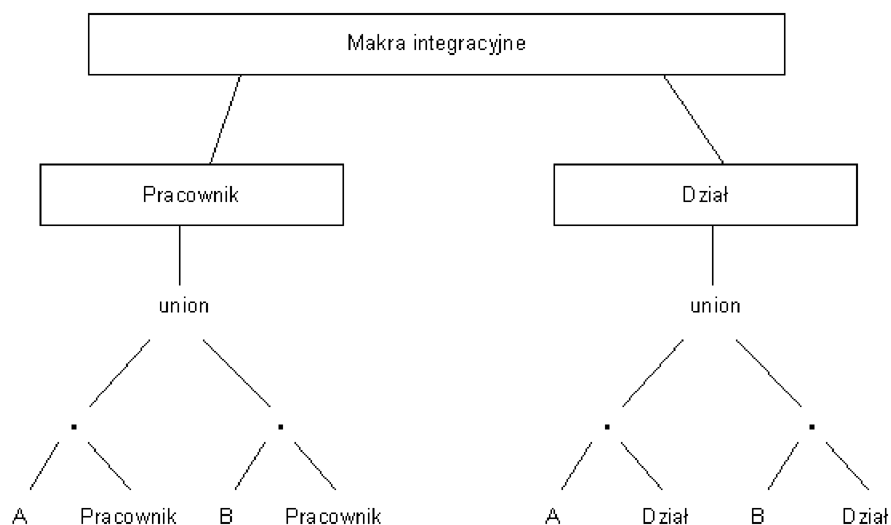
Rysunek 3.4: Schemat lokalizacji

Zgodnie z powyższym liczebność 0..N w węźle schematu o id równym 2 oznacza, że w bazie danych może być dowolna liczba obiektów korzeniowych etykietowanych nazwą *Pracownik*, włączając w to sytuację, w której zbiór takich obiektów będzie pusty. Mamy również, że każdy obiekt reprezentujący pracownika zawiera dokładnie jedno nazwisko.

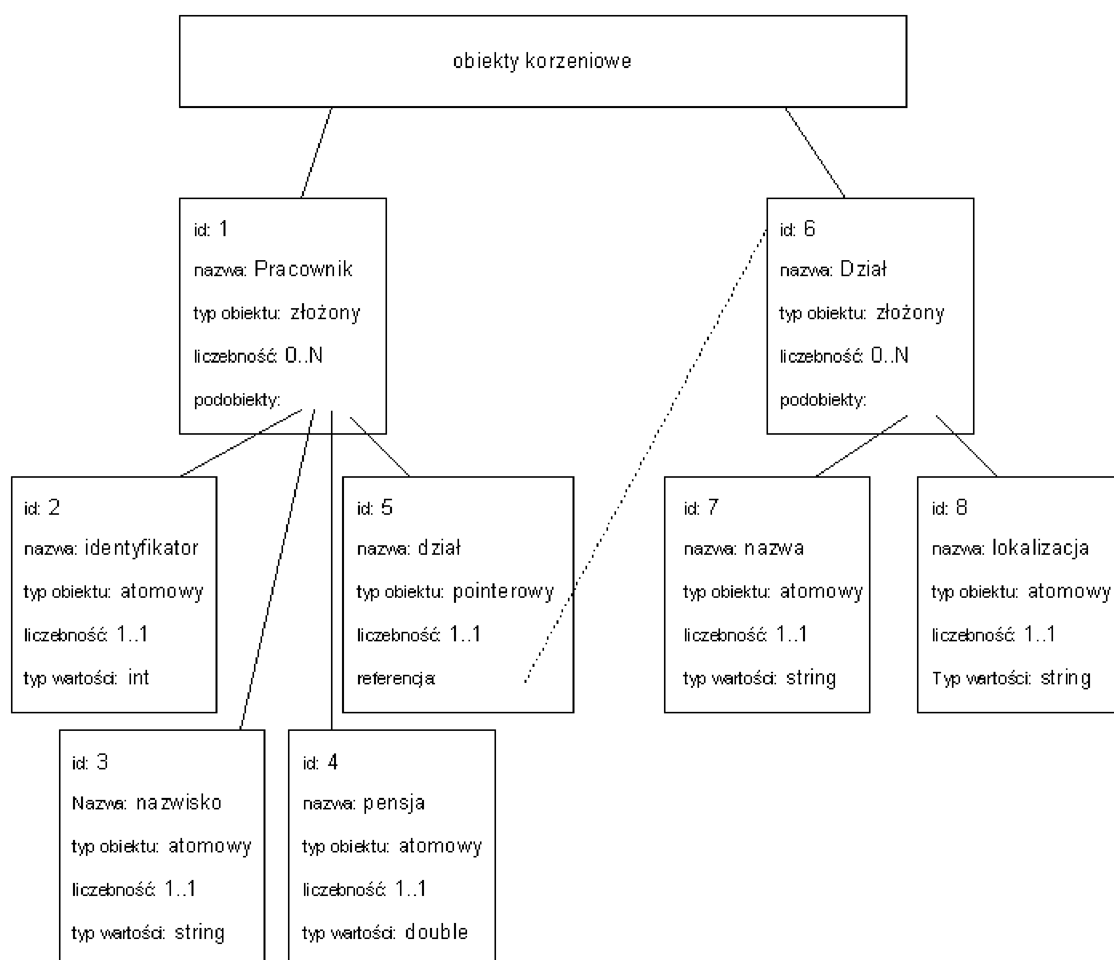
Na rysunku 3.5 przedstawiono makra tworzące na podstawie schematu lokalizacji wirtualne obiekty dostępne w schemacie globalnym. W prezentowanym tu przykładzie mamy do czynienia z poziomą fragmentacją danych zarówno w przypadku kolekcji pracowników, jak i działów. Drzewa makr *Pracownik* i *Dział* reprezentują odpowiednio następujące zapytania:

$$(A\ .\ Pracownik)\ \textbf{union}\ (B\ .\ Pracownik)\ \text{oraz}\ (A\ .\ Dzia\l)\ \textbf{union}\ (B\ .\ Dzia\l).$$

Schemat globalny stworzony na takiej podstawie przedstawiono na rysunku 3.6.



Rysunek 3.5: Makra integracyjne



Rysunek 3.6: Schemat globalny

Schemat globalny jest w tym przypadku taki jak schematy udostępniane przez serwery A i B. Globalne kolekcje powstają w wyniku połączenia za pomocą operatora union odpowiednich

kolekcji zawartych na dwóch wymienionych serwerach.

3.5.5. Parser zapytań

Parsowanie zapytań nie różni się znacząco od parsowania kodu programów komputerowych. Ze względu na stosunkowo niewielki rozmiar pojedynczych zapytań ich analiza składniowa w nieznacznym stopniu wpływa na łączny czas ich przetwarzania. Biorąc pod uwagę powyższe cechy zastosowano gotowe narzędzia do wygenerowania kodu leksera i parsera. Posłużono się tutaj popularnymi programami *flex* i *bison*. Ponieważ uzyskany kod wykorzystywany jest współbieżnie przez wątki obsługujące różnych klientów, istotna była taka konfiguracja tych narzędzi, która pozwoliła na otrzymanie parsera wielowejściowego.

Wynikiem etapu parsowania jest drzewo składniowe. Ze względu na budowę języka SBQL, tekst zapytania w jednoznaczny i bezpośredni sposób może być przekształcony w drzewo składniowe. Warto jednak, by było ono wzbogacone o węzły reprezentujące dodatkowe operacje, jak koercje czy dereferencja, wymuszane przez niektóre z operatorów, jak operatory arytmetyczne, które wymagają argumentów w postaci pojedynczych, numerycznych wartości. Dzięki automatycznemu uzupełnianiu zapytania mogą stać się prostsze i bardziej naturalne. W przypadku zapytania:

$$Pracownik\,\mathbf{where}\,pensja = 3000$$

wiązanie nazwy pensja zwraca potencjalnie zbiór referencji do odpowiednich obiektów. Jeżeli każdy obiekt *Pracownik* zawiera dokładnie jeden numeryczny podobiekt *pensja*, rozsądne wydaje się dopuszczenie takiej formy zapytania. W takiej sytuacji system odpowiedzialny jest za wykonanie dodatkowych operacji i kontrolę zgodności danych z założeniami. By umożliwić wsparcie dla takich zapytań, węzły odpowiednich operacji uzupełniane są w drzewie składniowym podczas jego konstruowania w fazie parsowania.

3.5.6. Komunikacja międzyserwerowa

Zaletą statycznej dekompozycji zapytań jest bardzo efektywne ich przetwarzanie z wykorzystaniem obliczeń równoległych. W warstwie integracyjnej wsparcie dla tego rozwiązania uzyskano dzięki puli niezależnych wątków połączeń.

W stworzonym modelu w warstwie integracyjnej każdemu serwerowi biorącemu w integracji odpowiada osobny wątek. Zadaniem takiego wątku jest przede wszystkim przesyłanie zapytań na serwer, a następnie pobieranie odpowiedzi. Zarządzanie takim wątkiem odbywa się za pomocą specjalnego obiektu-pośrednika. Obiekt ten jest odpowiedzialny za nawiązanie połączenia z serwerem oraz późniejsze zamknięcie połączenia. Do jego zadań należy także stworzenie i likwidacja odpowiedniego wątku oraz przekazywanie mu zapytań do wykonania. Synchronizacja odbywa się za pomocą zmiennych typu pthread_mutex. Komunikacja między wątkiem a jego pośrednikiem jest możliwa dzięki temu, że pośrednik przekazuje siebie jako argument funkcji stanowiącej kod wątku.

W czasie działania rozproszonej bazy danych mogą zdarzyć się sytuacje kiedy połączenia ulegają zerwaniu. Dlatego też w przypadku nieudanego przesłania zapytania na serwer następuje próba wskrzeszenia połączenia.

Dostęp do połączeń możliwy jest za pomocą globalnego obiektu puli połączeń. Pula przydziela połączenia wątkom obsługującym poszczególnych klientów w taki sposób, żeby żadne połączenie nie było użytkowane przez klika takich wątków na raz.

Proces przydziału połączeń wygląda następująco. Na początku wątek przetwarzający zapytanie zgłasza się do kolejki zaimplementowanej za pomocą zmiennej typu pthread_mutex.

Wątek, dla którego ten mutex jest otwarty zamyka go za sobą i otwiera dopiero po rezerwacji wszystkich wymaganych połączeń. Dzięki temu rozwiązaniu proces przydzielania połączeń dotyczy tylko jednego wątku w danej chwili, a tak zwany problem zagłodzenia przerzucany jest na system operacyjny zarządzający mutexami. Ponieważ obiekt puli jest globalny, a rezerwacja połączeń przebiega niezależnie w stosunku do ich zwracania, niezbędna jest synchronizacja operacji polegających na odczycie oraz zmianie stanu puli. Odbywa się to za pomocą innej zmiennej typu pthread_mutex. Wątek, który uzyskał dostęp do rezerwacji połączeń, blokuje mutex synchronizujący dostęp do zmiennych puli, a następnie zleca jej przydział listy połączeń. Jeśli jest to możliwe w danej chwili, uzyskuje on wyłączny dostęp do połączeń, zwalniając na koniec oba wymienione mutexy. Może się jednak zdarzyć, że połączenia zostały pobrane z puli przez inny wątek. W tej sytuacji wątek wstrzymuje swoje działanie z pomocą trzeciego mutexu przeznaczonego specjalnie do tego celu oraz odnotowuje ten fakt w specjalnej zmiennej puli, a następnie zwalnia mutex synchronizujący dostęp do jej zmiennych. W przypadku zwracania połączeń, pula sprawdza czy istnieje wątek oczekujący na dokończenie przydziału. Jeśli w danym momencie udaje się takie żądanie spełnić, pula podnosi mutex, na którym oczekuje wspomniany wątek. Tak jak w poprzednio opisywanym przypadku po rezerwacji połączeń następuje wpuszczenie kolejnego wątku z kolejki.

3.5.7. Wykonanie zapytania

Za wykonanie zapytania odpowiedzialny jest obiekt scalający wyniki cząstkowych podzapytań wysyłanych na zdalne serwery. Jako wejście otrzymuje on zapytanie uzyskane w wyniku dekompozycji oraz listę serwerów na które ma zostać ono przesłane. Pierwszym krokiem jest opisana wyżej rezerwacja odpowiednich połączeń. Następnie moduł ten uruchamia wykonanie zapytania przez pozyskane połączenia i zawiesza swoje działanie w oczekiwaniu na wyniki.

Połączenia dzięki referencji do obiektu scalającego zgłaszają kolejno otrzymane wyniki zapytań. Zgłaszanie cząstkowych rezultatów odbywa się pojedynczo, a synchronizacja bazuje na zmiennej typu pthread_mutex. Połączenie dysponujące gotowym wynikiem, po uzyskaniu mutexu, zapisuje ten wynik w obiekcie scalającym i wznawia działanie tego obiektu.

Po wznowieniu działania obiekt scalający sprawdza kod błędu zapisany przez połączanie. Jeśli operacja była nieudana, informacja o tym jest propagowana do innych połączeń (zgodnie ze schematem opisanym w Rozdziale 3.5.2), co powoduje przerwanie ich pracy. W wypadku kiedy kod wskazuje prawidłowy przebieg operacji, uzyskany wynik jest scalany z cząstkowym rezultatem powstałym w wyniku połączenia wcześniejszych rezultatów.

Na końcu tego procesu formułowany jest ostateczny wynik zapytania gotowy do wysłania do klienta.

3.5.8. Analiza semantyczna zapytania

Zgodnie ze schematem przedstawionym w Rozdziale 3.5.3, analiza semantyczna przebiega dwufazowo. Celem pierwszej fazy jest wykrycie błędów semantycznych na poziomie schematu globalnego, co pozwala możliwie wcześnie wycofać się z obliczania wyniku dla niepoprawnego zapytania. Drugim zadaniem tego etapu jest przepisanie zapytania na postać bazującą na schemacie lokalizacji. Odbywa się to za pomocą rozwijania makr realizujących przekształcenie między schematami. Makra przyporządkowane są obiektom korzeniowym zdefiniowanym w schemacie globalnym. Istotne jest więc wykrycie nazw wiązanych w sekcji stosu środowiskowego, w której znajdują się bindery obiektów korzeniowych. Ponieważ podstawianie makr nie zakłóca przebiegu statycznej ewaluacji, oba procesy zostały połączone. Optymalizacje oparte na przepisywaniu mają zastosowanie głównie na poziomie schematu lokalizacji, dlatego też

dekorowanie drzewa składniowego informacjami semantycznymi ma miejsce dopiero podczas drugiego przebiegu statycznej ewaluacji bazującej na tym schemacie. Tak uzyskane drzewo jest podstawą algorytmu dekompozycji zapytania.

Warto w tym miejscu przypomnieć, że głównym celem analizy składniowej w opisywanej implementacji, poza wykryciem ewentualnych błędów, jest uzyskanie statycznej dekompozycji zapytania. Założonym wynikiem powinno być zapytanie oraz grupa serwerów, do których ma być ono wysłane, pozwalające na skonstruowanie finalnego rezultatu za pomocą prostego połączenia (suma bagów) wyników cząstkowych. Oczywiście może się okazać, że taka dekompozycja nie jest możliwa. Oznacza to, że zapytanie nie należy do obsługiwanej klasy zapytań. W takim przypadku przetwarzanie kończy się sygnalizacją odpowiedniego błędu.

Algorytm dekompozycji wykorzystuje własność rozdzielności operatorów względem sumy. Dzięki temu możliwe jest uzyskanie stosunkowo dobrych wyników w zakresie zasięgu klasy obsługiwanych zapytań w przypadku poziomej fragmentacji danych. Schemat algorytmu jest następujący:

- Algorytm ma charakter rekurencyjny - przed przetworzeniem korzenia drzewa składniowego przetwarzane są jego poddrzewa.
- Wynikiem zastosowania algorytmu dla danego drzewa (poddrzewa) składniowego jest etykietowanie go tekstem zapytania oraz zbiorem serwerów. Suma wyników uzyskanych po wysłaniu tego tekstu zapytania na dane serwery stanowi rezultat wykonania zapytania reprezentowanego przez powyższe drzewo. Dla ułatwienia manipulacji na zapytaniu z etykiety, zapisuje się nie tylko jego tekst, ale także drzewo składniowe.
- Dla węzła reprezentującego literał wynikowe zapytanie składa się z tekstu literału, a zbiór serwerów jest pusty.
- Dla węzła nazwy pomocniczej wynikowe zapytanie jest tekstem postaci:

  (<*zapytanie otrzymane dla argumentu*>) **as** <*nazwa pomocnicza*>

  lub odpowiednio:

  (<*zapytanie otrzymane dla argumentu*>) **group as** <*nazwa pomocnicza*>

  Zbiór serwerów jest taki jak w przypadku argumentu.
- Dla węzła nazwy tekst zapytania stanowi dana nazwa, a zbiór serwerów stanowią serwery, na których znajdują się dane wskazywane przez tę nazwę
- W przypadku operatorów unarnych nie są obsługiwane operacje agregacyjne na danych pochodzących z różnych serwerów. W pozostałych przypadkach uzyskujemy zapytanie w postaci:

  <*nazwa operatora*> (<*zapytanie otrzymane dla argumentu*>)

  Zbiór serwerów jest taki sam jak dla argumentu.
- W opisie algorytmu dla węzłów operatorów binarnych zastosowano pewne uproszczenie. Wszędzie, gdzie pisze się o tych samych zbiorach serwerów, warunek ten obejmuje też wszystkie przypadki, w których jeden ze zbiorów jest pusty. W sytuacji, kiedy zbiór serwerów dla jednego z argumentów jest pusty, sformułowanie: "Zbiór serwerów jest taki sam jak w przypadku argumentów", oznacza przypisanie do wyniku zbioru serwerów otrzymanego dla drugiego z argumentów.

- Ze względu na przyjęte założenia szczególne znaczenie ma operator **union**. Poza tym, że stanowi on podstawę łączenia wyników cząstkowych otrzymanych z różnych serwerów, jest to także operator rozdzielny względem sumy.

  Jeśli zapytania otrzymane dla argumentów są takie same, to wynikiem jest zapytanie takie jak w przypadku argumentów, a zbiór serwerów jest sumą zbiorów serwerów przypisanych do argumentów.

  Jeśli powyższy warunek nie jest spełniony, to zapytania dla argumentów powinny być etykietowane tym samym zbiorem serwerów. Wtedy otrzymujemy zapytanie postaci:

  (<*zapytanie otrzymane dla lewego argumentu*>)

  union

  (<*zapytanie otrzymane dla prawego argumentu*>)

  Zbiór serwerów jest taki sam jak w przypadku argumentów.

  W pozostałych przypadkach zapytanie nie należy do obsługiwanej klasy zapytań.

- Własność rozdzielności względem sumy wykorzystywana jest dla operatorów: **where**, "**.**" oraz **join**. Dla tej grupy warunek jest podobny do drugiego przypadku opisanego dla operatora **union**, ponieważ oba argumenty muszą być etykietowane tym samym zbiorem serwerów. Dodatkowo nazwy występujące w prawym argumencie nie mogą być wiązane poniżej sekcji otwieranej na stosie środowiskowym przez dany operator. W takim przypadku wynikowe zapytanie ma postać:

  (<*zapytanie otrzymane dla lewego argumentu*>)

  <*nazwa operatora*>

  (<*zapytanie otrzymane dla prawego argumentu*>)

  Zbiór serwerów jest taki jak dla argumentów.

  Jeśli nie uda się spełnić powyższych założeń, stosuje się procedury przeznaczone dla innych operatorów binarnych, opisane poniżej.

- Dla niewymienionych wyżej operatorów binarnych stosuje się opisane niżej warunki.

  Wynikowe zapytanie jest postaci:

  (<*zapytanie otrzymane dla lewego argumentu*>)

  <*nazwa operatora*>

  (<*zapytanie otrzymane dla prawego argumentu*>)

  Argumenty muszą być etykietowane tymi samymi serwerami, a zbiór serwerów dla wyniku jest taki sam jak dla argumentów.

  Jeśli listy serwerów dla argumentów nie są puste, a w co najmniej jednym argumencie wiązana jest nazwa serwera, to zapytania mogą być etykietowane co najwyżej jednym serwerem.

Na koniec opisu rozwiązań dotyczących statycznej analizy warto wspomnieć jeszcze o dwóch istotnych cechach implementacji.

System jest przystosowany do pracy na heterogenicznych, semistrukturalnych danych. Możliwe jest to dzięki zastosowaniu sygnatur wariantów podczas statycznej ewaluacji. Sygnatura taka wskazuje typy obiektów, które stanowią alternatywne wyniki dla danego zapytania lub podzapytania. Arytmetyka z udziałem wariantów jest znacznie bardziej złożona niż w przypadku innych typów sygnatur. Pozwala to jednak znacznie poszerzyć możliwości w zakresie integracji danych.

Między innymi dzięki opisanemu wyżej rozwiązaniu w prezentowanym systemie nie ma bezpośredniego założenia dotyczącego sposobu fragmentacji danych. Oznacza to że, zastosowanie innej organizacji danych niż fragmentacja pozioma nie wyklucza działania warstwy integracyjnej. Mimo to, inny typ fragmentacji może w znacznym stopniu ograniczyć funkcjonalność systemu.

3.6. Podsumowanie

Wykorzystanie podejścia stosowego umożliwia realizację rozbudowanych modeli integracji w dość prosty sposób i w stosunkowo krótkim czasie. Prezentowany w tej pracy system mimo, że był tworzony praktycznie od podstaw w silnie ograniczonych ramach czasowych, pozwala na bardzo efektywne przetwarzanie popularnej klasy zapytań ze wsparciem dla danych semistrukturalnych i heterogenicznych. Ponadto istnieje cały szereg dość prostych w realizacji, możliwych rozszerzeń tego oprogramowania, pozwalających zdecydowanie zwiększyć jego funkcjonalność. Stworzona platforma jest również bardzo dobrą bazą dla wszelkich badań dotyczących optymalizacji zapytań, dla której SBA stwarza bardzo szerokie możliwości.

Rozdział 4

Przykłady przetwarzania zapytań

4.1. Wprowadzenie

Poniżej przedstawiono krok po kroku przykłady przetwarzania zapytań w systemie opisanym w poprzednim rozdziale. Położono tu nacisk na fazę, w której za przebieg procesu odpowiedzialny jest moduł preprocesora zapytań, gdyż jest to etap najbardziej złożony koncepcyjnie. Obejmuje on stadia od odebrania zapytania na serwerze do sformułowania planu wykonania tego zapytania. W jego skład wchodzą więc analiza składniowa oraz analiza semantyczna.

W zawartych tu przykładach oparto się na metabazie przedstawionej w rozdziale 3.5.4. Przypomnijmy, że mamy tam do czynienia z poziomą fragmentacją kolekcji danych pracowników i działów przedsiębiorstwa. Dane udostępniane są przez dwa zdalne serwery, na których występuje taka sama organizacja zasobów.

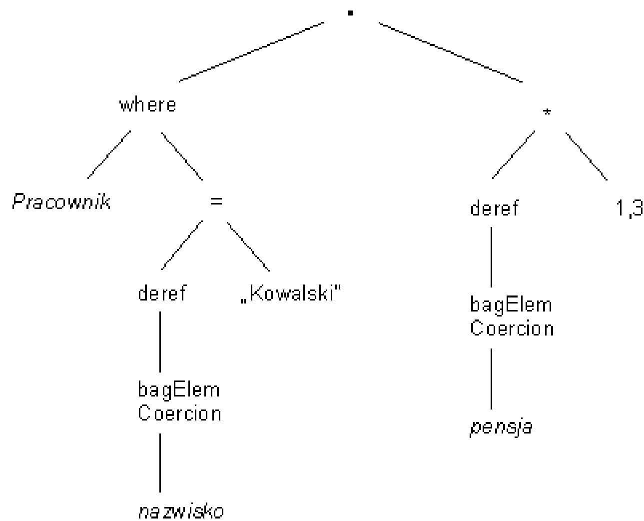
4.2. Przykład nr 1

Załóżmy, że w naszym przedsiębiorstwie pracuje jeden pracownik o nazwisku Kowalski. Ze względu na bardzo dobre wyniki pracy w poprzednim roku, kierownictwo postanowiło przyznać mu premię w wysokości 130% miesięcznej pensji. Poniższe zapytanie ustala wysokość przyznanej premii:

$$(Pracownik\ \mathbf{where}\ nazwisko\ =\ \text{"Kowalski"})\ \mathbf{.}\ (pensja\ *\ 1,3)$$

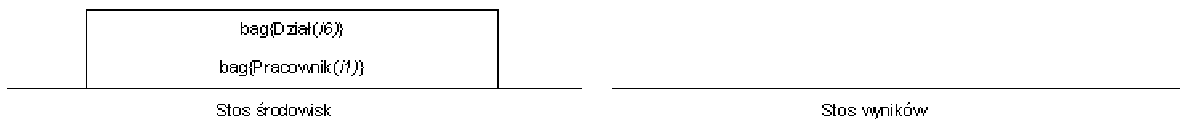
Na rysunku 4.1 przedstawiono drzewo składniowe dla powyższego zapytania stworzone przez parser zapytań. Poza węzłami reprezentującymi elementy występujące w tekście zapytania zawiera ono także węzły ukrytych operatorów, o których była mowa w rozdziale 3.5.5. Operator bagElemCoercion służy do zamiany zbioru na pojedynczy element. Ponieważ nie jest to standardowy operator języka SBQL jego semantyka może się różnić w zależności od konkretnej realizacji modułu wykonującego zapytania. Można jednak przyjąć, że w sytuacjach, kiedy jego argument będzie inny niż zbiór jednoelementowy, zostanie zasygnalizowany błąd typologiczny. Zadaniem operatora deref jest uzyskanie wartości obiektu wskazywanego przez referencję.

Stworzone przez parser drzewo składniowe jest wejściem dla analizy semantycznej. Podstawą tego procesu jest statyczna ewaluacja, która polega na prześledzeniu wyliczania wyniku zapytania w środowisku symulującym prawdziwe środowisko wykonania. Statyczna ewaluacja wykorzystuje do tego celu metabazę oraz statyczne stosy, na których umieszczane są sygnatury elementów oczekiwanych na stosach czasu wykonania.



Rysunek 4.1: Drzewo składniowe po etapie parsowania

W pierwszym etapie statycznej analizy następuje badanie zapytania względem schematu globalnego oraz podstawianie makr integracyjnych. Statyczna ewaluacja zaczyna się więc od umieszczenia na stosie środowiskowym sygnatur obiektów korzeniowych schematu globalnego. Początkowy stan stosów pokazano na rysunku 4.2.



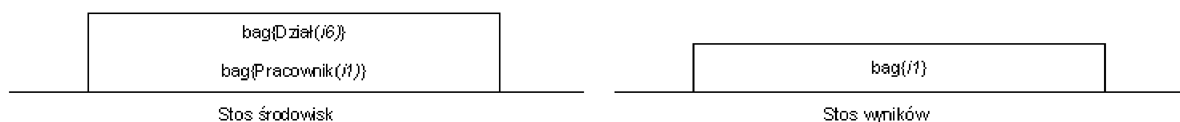
Rysunek 4.2: Statyczna ewaluacja - pierwszy przebieg. Początkowy stan stosów.

Bindery zobrazowano w postaci nazwa(referencja), referencje do obiektów przedstawiono zaś w postaci pochylonej literki i z identyfikatorem odpowiedniego węzła schematu. Na rysunku widać, że stos środowisk zawiera bindery do obiektów korzeniowych, podczas gdy stos wyników na początku pozostaje pusty. W dalszej części pracy podane numery sekcji liczone są od dołu stosu.

Krok 1. Ewaluacja rozpoczyna się od korzenia drzewa. Jest to węzeł reprezentujący niealgebraiczny operator kropki, należy więc najpierw obliczyć jego lewy argument.

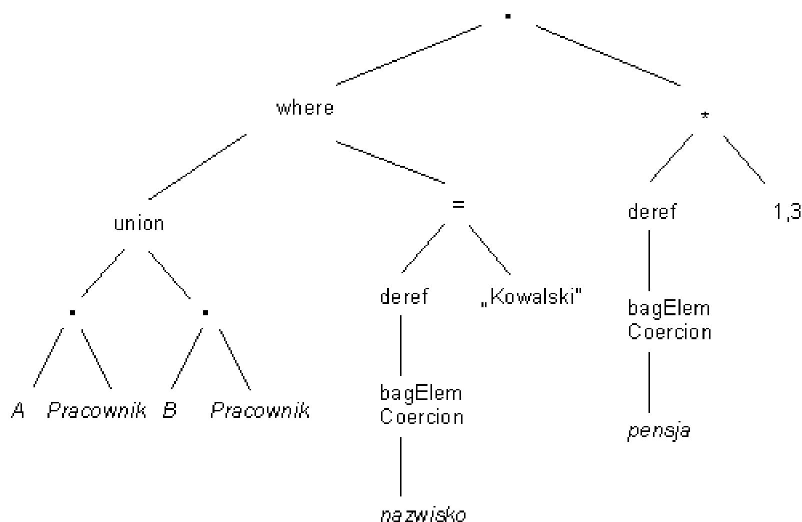
Krok 2. Korzeniem lewego poddrzewa jest operator where, należy więc postąpić jak w kroku 1 i przejść rozpocząć od obliczenia lewego argumentu.

Krok 3. Węzeł nazwy *Pracownik*. Nazwa wiązana jest w pierwszej sekcji stosu środowiskowego. Wynik wiązania umieszczany jest na stosie wyników, co pokazano na rysunku 4.3. Ponieważ wiązanie nastąpiło w bazowej sekcji stosu, oznacza to, że należy dokonać rozwinięcia



Rysunek 4.3: Stan stosów po kroku 3.

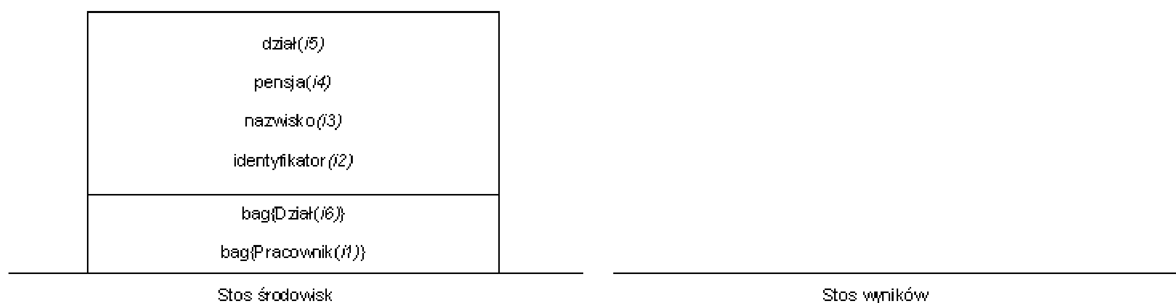
makra. W schemacie poszukiwane jest więc makro etykietowane nazwą *Pracownik*. Wynik podstawienia zaprezentowany został na rysunku 4.4. Nowe poddrzewo nie bierze udziału w



Rysunek 4.4: Drzewo składniowe po podstawieniu makra *Pracownik*.

dalszej części pierwszego przebiegu statycznej ewaluacji.

Krok 4. Wracamy do węzła operatora where napotkanego w kroku 2. Zdejmujemy rezultat z wierzchołka stosu wyników i wykonujemy na nim funkcję *static_nested*, odpowiednik *nested* działający na sygnaturach. Powoduje to umieszczenie na stosie środowisk nowej sekcji z bidnerami do podobiektów obiektu *i1*. Zobrazowano to na rysunku 4.5.



Rysunek 4.5: Stan stosów po kroku 4.

Krok 5. W nowym środowisku dokonywana jest ewaluacja prawego argumentu operatora where. Ponieważ korzeniem prawego poddrzewa jest węzeł operatora algebraicznego =, dokonujemy najpierw ewaluacji jego argumentów.

Krok 6. Operator deref. By dokonać dereferencji przechodzimy do obliczenia sygnatury argumentu.

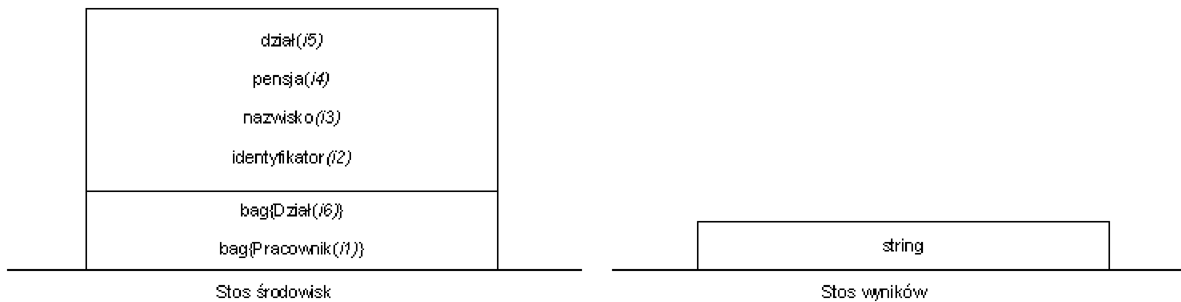
Krok 7. Operator rzutowania bagu na element. Jak wyżej, wyliczamy sygnaturę dla argumentu.

Krok 8. Węzeł nazwy *nazwisko*. Nazwa wiązana jest w drugiej sekcji stosu, co oznacza, że nie następuje rozwijanie makra. Na stos wyników trafia sygnatura *i3*.

Krok 9. Powrót do węzła reprezentującego operator rzutowania. Następuje zdjęcie sekcji

z wierzchołka stosu wyników. Ponieważ zawiera ona sygnaturę referencji wynikiem rzutowania jest ta sama referencja. Rezultat operacji jest wkładany na stos wyników. Stos ten pozostaje taki sam jak po kroku 8.

Krok 10. Powrót do węzła deref. Ponownie wynik działania operatora zostaje obliczony dla sygnatury zdjętej uprzednio ze stosu wyników. Ponieważ węzeł o identyfikatorze $i3$ reprezentuje wartość typu string, taka sygnatura trafia na stos wyników (rysunek 4.6).

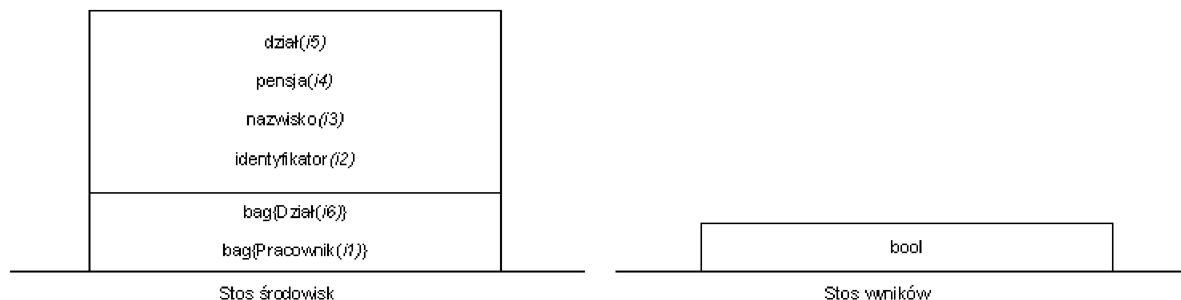


Rysunek 4.6: Stan stosów po kroku 10.

Krok 11. Następuje powrót do węzła operatora =, gdzie sygnatura wyliczona dla jego lewego argumentu zostaje zdjęta ze stosu wyników i zapamiętana. Teraz należy obliczyć sygnaturę dla prawego argumentu.

Krok 12. Napis "Kowalski". Na stos wyników zostaje odłożona sygnatura string.

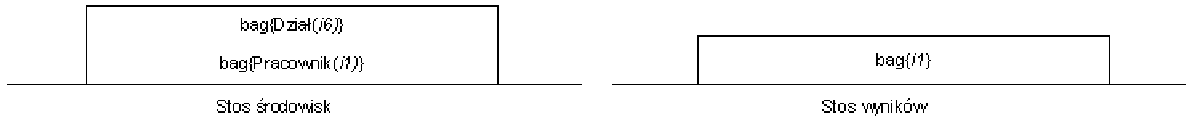
Krok 13. Ponowny powrót do węzła operatora =. Po zdjęciu ze stosu sygnatury prawego argumentu. Następuje kontrola typologiczna argumentów w kontekście operatora. Oba argumenty są napisami, więc operacja ich porównania jest poprawna. Wynik w postaci sygnatury stałej logicznej jest wkładany na stos wyników (rysunek 4.7).



Rysunek 4.7: Stan stosów po kroku 13.

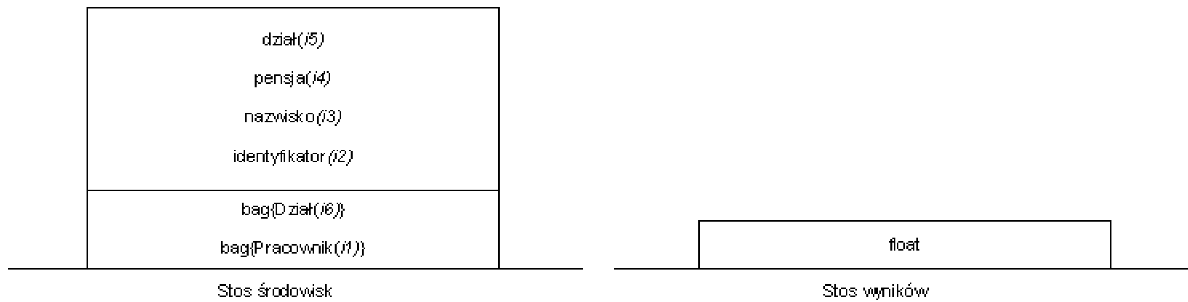
Krok 14. Powrót do węzła operatora where i zdjęcie ze stosu wyników prawego argumentu tego operatora. Następuje sprawdzenie czy rezultatem wyliczenia prawego argumentu jest stała logiczna. Wynik jest zatem poprawny. Ze stosu środowiskowego zdejmowana jest sekcja utworzona przez operator where. Na stos wyników trafia rezultat w postaci bag{*i1*} (rysunek 4.8).

Krok 15. Powrót do operatora kropki. Następuje zdjęcie sygnatury z wierzchołka stosu wyników. Ponieważ jest ona taka sama jak w przypadku operatora where wyliczenie prawego argumentu odbywa się także w takim samym środowisku (ten sam wynik *static_nested*) w analogiczny sposób. W tym przypadku zamiast nazwy *nazwisko*, której przyporządkowany



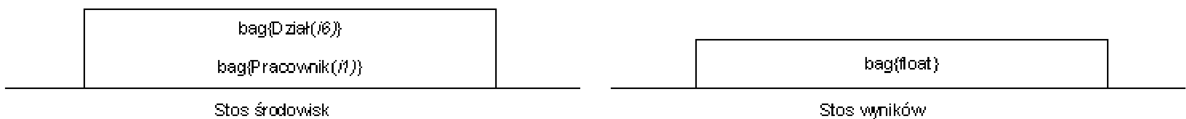
Rysunek 4.8: Stan stosów po kroku 14.

jest napis, mamy nazwę *pensja* reprezentującą wartość zmiennoprzecinkową. W miejscu operatora =, występuje operator *, który symbolizuje mnożenie i w wyniku daje sygnaturę float (rysunek 4.9).



Rysunek 4.9: Stan stosów po kroku 15.

Krok 16. Wynikiem ewaluacji całego zapytania jest bag zawierający elementy reprezentowane przez sygnaturę uzyskaną dla prawego argumentu operatora kropki, czyli liczba zmiennoprzecinkowa. Stan stosów po pierwszym przebiegu statycznej ewaluacji przedstawiono na rysuku 4.10.

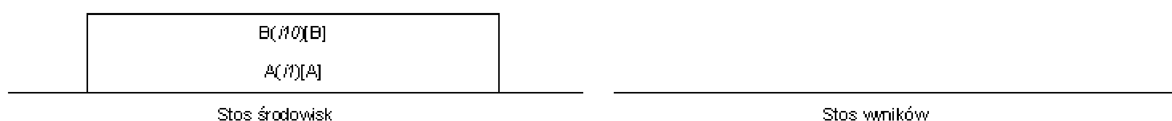


Rysunek 4.10: Stan stosów po pierwszym przebiegu statycznej ewaluacji.

Na tym etapie potwierdzona zostało zgodność zapytania ze schematem globalnym oraz zostało ono przepisane na postać bazującą na schemacie lokalizacji za pomocą rozwinięcia makr integracyjnych. Drzewo składniowe przedstawione na rysunku 4.4 stanowi wejście do drugiego przebiegu statycznej ewaluacji, którego celem jest weryfikacja względem schematu lokalizacji oraz statyczna dekompozycja zapytania. Podobnie jak poprzednio statyczna ewaluacja rozpoczyna się od inicjalizacji stosów. Na stos środowiskowy wkładana jest sekcja z sygnaturami obiektów korzeniowych schematu lokalizacji, co w tym przypadku oznacza sygnatury serwerów. Stos wyników tak jak poprzednio pozostaje pusty. Wejściowy stan stosów przedstawiono na rysunku 4.11. W nawiasach kwadratowych przedstawiono listy serwerów, na których znajdują się dane potrzebne do wyliczenia wartości obiektu etykietowanego nazwą.

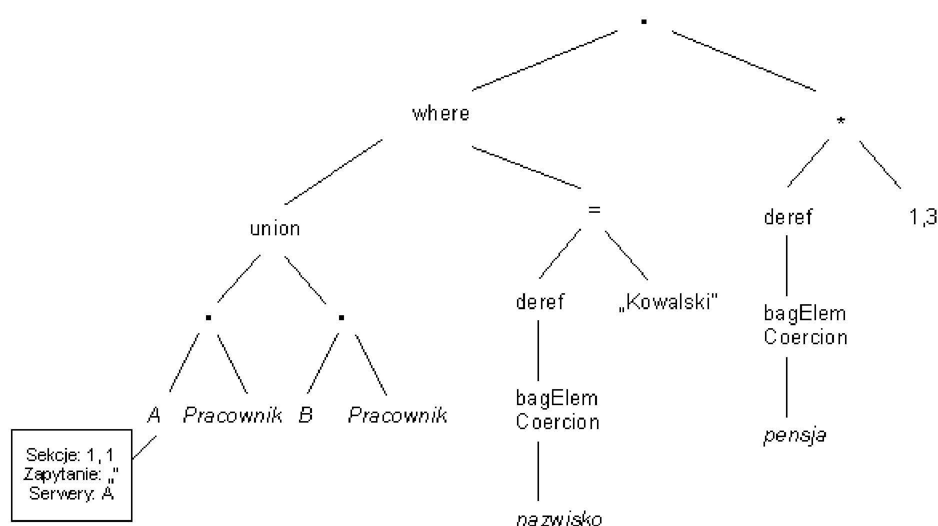
Krok 17. Rekurencyjnie wywoływana jest procedura statycznej ewaluacji lewego argumentu kolejno dla operatorów kropki, where, union i kropki.

Krok 18. Węzeł nazwy A. Etykietujemy go numerem sekcji stosu, w której jest wiązana nazwa, w tym przypadku jest to 1 oraz wysokością stosu w momencie wiązania, też 1. Ze



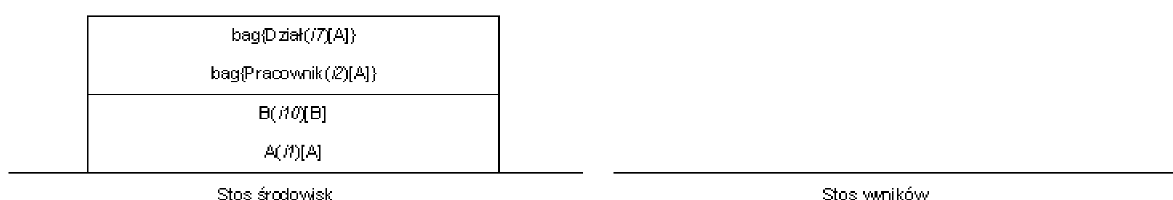
Rysunek 4.11: Stan stosów przed drugim przebiegiem statycznej ewaluacji.

względu na to, że węzeł reprezentuje jedynie operacje na nazwach serwerów, przypisywany do niego tekst zapytania jest pustym napisem. Na listę serwerów trafia serwer A. Wynik etykietowania węzłą przedstawiono na rysunku 4.12. Z binderem zwierającym referencję do reprezentacji serwera postępujemy tak jak w przypadku bindera obiektu złożonego. Na stos wyników trafia więc sygnatura *i10*.



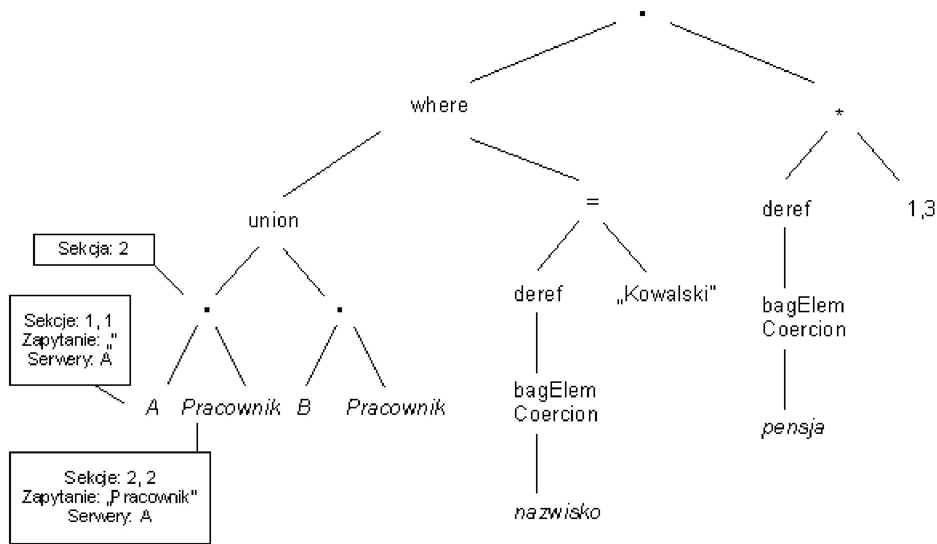
Rysunek 4.12: Wynik etykietowania w kroku 18.

Krok 19. Następuje powrót do węzła kropki oraz włożenie na stos środowisk sekcji zawierającej wynik *static_nested* dla sygnatury zdjętej z czubka stosu wyników. Tu ponownie postępujemy z referencją do węzła reprezentującego serwer w schemacie, tak jakby była to referencja do węzła reprezentującego obiekt złożony (rysunek 4.13). Węzeł zostaje etykietowany numerem sekcji, którą otwiera, czyli 2.



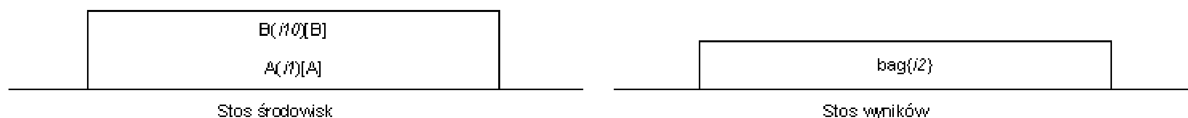
Rysunek 4.13: Stan stosów po kroku 19.

Krok 20. Węzeł nazwy *Pracownik*. Wynik etykietowania przedstawiono na rysunku 4.14. Ponieważ nazwa *Pracownik* jest nazwą pochodzącą ze schematu zdalnego serwera, jest ona uwzględniana w zapytaniu przypisanym do węzła. Na stos wyników trafia sygnatura bag{*i2*}.



Rysunek 4.14: Wynik etykietowania w kroku 20.

Krok 21. Operator kropki, konstrukcja wyniku. Ze stosu środowiskowego zdejmowana jest sekcja włożona przez ten operator. Sygnatura wyniku tworzona jest na podstawie rezultatu otrzymanego w wyniku ewaluacji prawego argumentu. Stan stosów po wykonaniu tych operacji pokazuje rysunek 4.15. Operator kropki jest rozdzielny względem sumy. Aby skorzy-



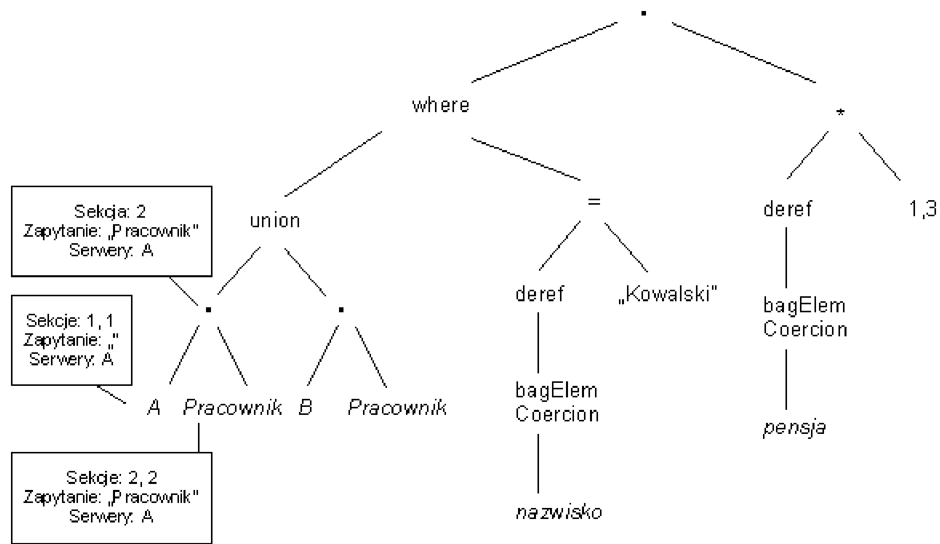
Rysunek 4.15: Stan stosów po kroku 21.

stać z tej własności muszą być spełnione dwa warunki. Po pierwsze, oba argumenty muszą być etykietowane tym samym zbiorem serwerów. To założenie jest więc spełnione. Ponadto, nazwy w prawym argumencie nie mogą być wiązane poniżej sekcji, tworzonej przez dany operator, co też jest w tym wypadku prawdą. Ponieważ zapytanie, którym etykietowany jest lewy argument jest puste (występuje tam tylko nazwa serwera), zapytaniem przypisanym do bieżącego węzła staje się "*Pracownik*" (rysunek 4.16). Etykieta węzła, zgodnie z definicją, oznacza, że by skonstruować wynik zapytania reprezentowanego przez poddrzewo z korzeniem w tym węźle, należy wysłać zapytanie *Pracownik* na serwer A.

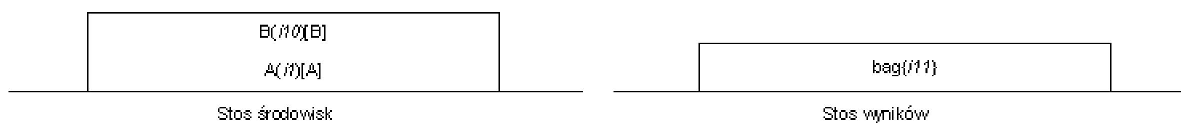
Krok 22. Powrót do węzła operatora union i zapamiętanie wyniku lewego podzapytania. Następnie ewaluowany jest prawy argument operatora.

Krok 23. Prawe poddrzewo wyliczane jest w taki sam sposób jak w przypadku lewego poddrzewa. Stan stosów po jego ewaluacji widoczny jest na rysunku 4.17. Możemy założyć, że numery identyfikatorów węzłów reprezentujących zasoby na serwerze B są o 9 większe od swoich odpowiedników z serwera A. Co za tym idzie, *i11* oznacza kolekcję obiektów *Pracownik* na serwerze B. Wynik etykietowania poddrzewa pokazano na rysunku 4.18.

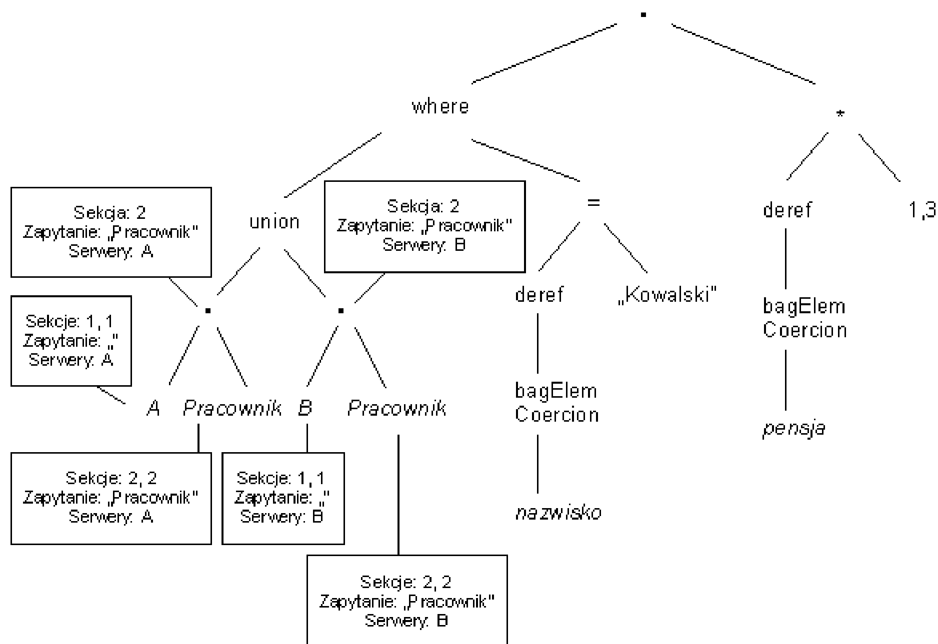
Krok 24. Operator union, konstrukcja wyniku. W tym przykładzie spełniony jest warunek, który mówi, że podzapytania dla argumentów są takie same. Oznacza to, że całe



Rysunek 4.16: Wynik etykietowania w kroku 21.

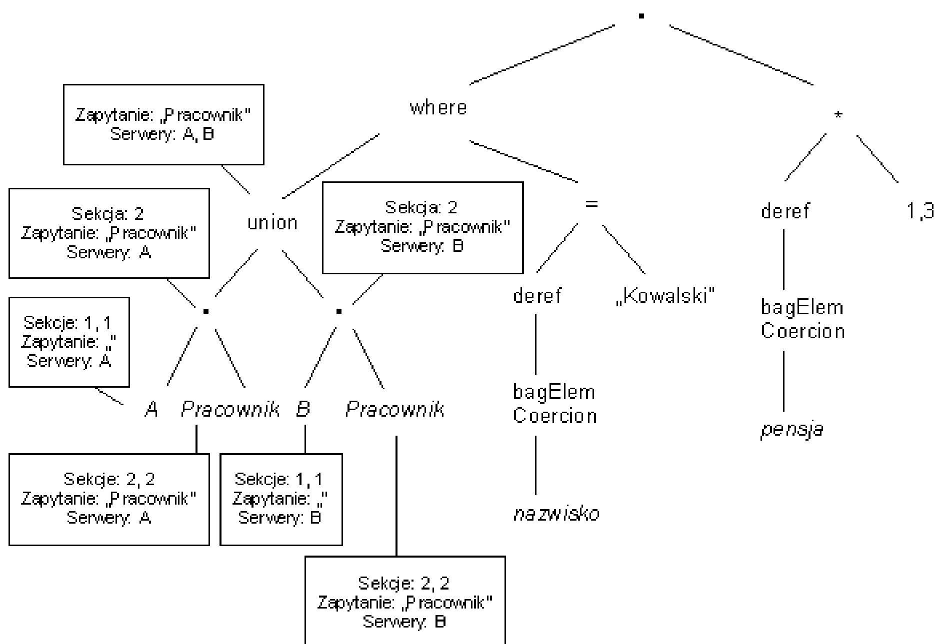


Rysunek 4.17: Stan stosów po kroku 23.



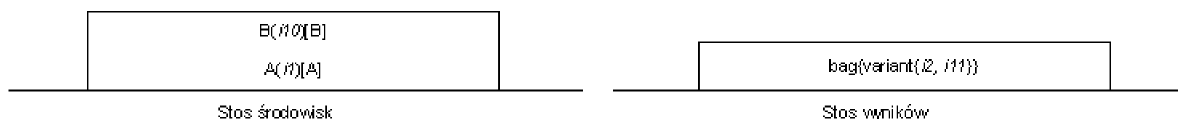
Rysunek 4.18: Wynik etykietowania w kroku 23.

poddrzewo o korzeniu w tym węźle może być etykietowane zapytaniem *Pracownik*, które musi być rozesłane na serwery A i B (rysunek 4.19). Wynikiem działania operatora union



Rysunek 4.19: Wynik etykietowania w kroku 24.

na kolekcjach różnych typów jest sygnatura multizbioru o elementach typu variant (rysunek 4.20). Sygnatura variant oznacza, że w danym miejscu może wystąpić jedna z opcji zawartych w tej sygnaturze, nie jesteśmy jednak w stanie dokładnie określić która z nich.



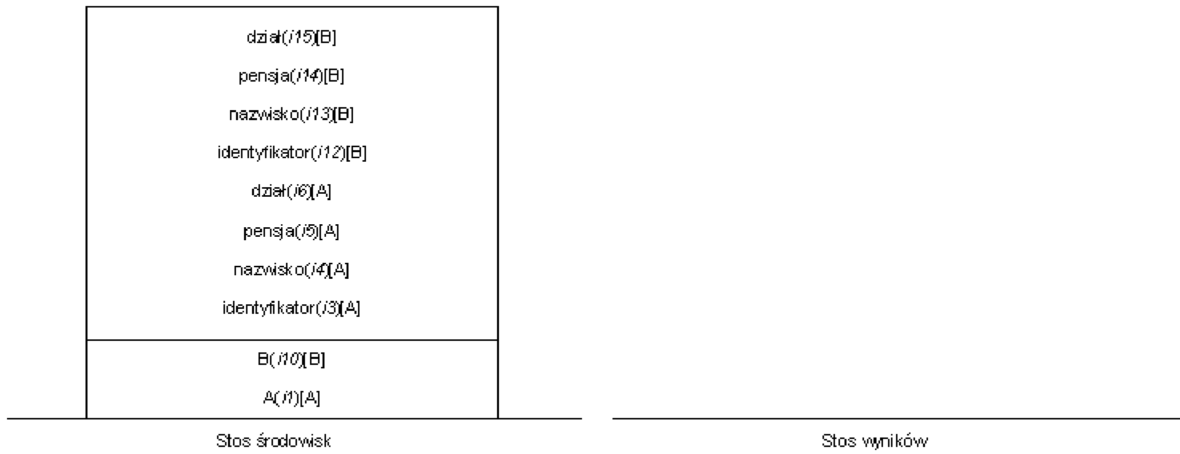
Rysunek 4.20: Stan stosów po kroku 24.

Krok 25. Powrót do operatora where. W wyniku *static_nested* zastosowanej dla sygnatury z wierzchołka stosu wyników, na stosie środowisk zostaje utworzona sekcja z sygnaturami binderów, zarówno podobiektów obiektu Pracownik z serwera A, jak i podobiektów obiektu Pracownik z serwera B (rysunek 4.21). Węzeł operatora etykietowany jest numerem sekcji 2.

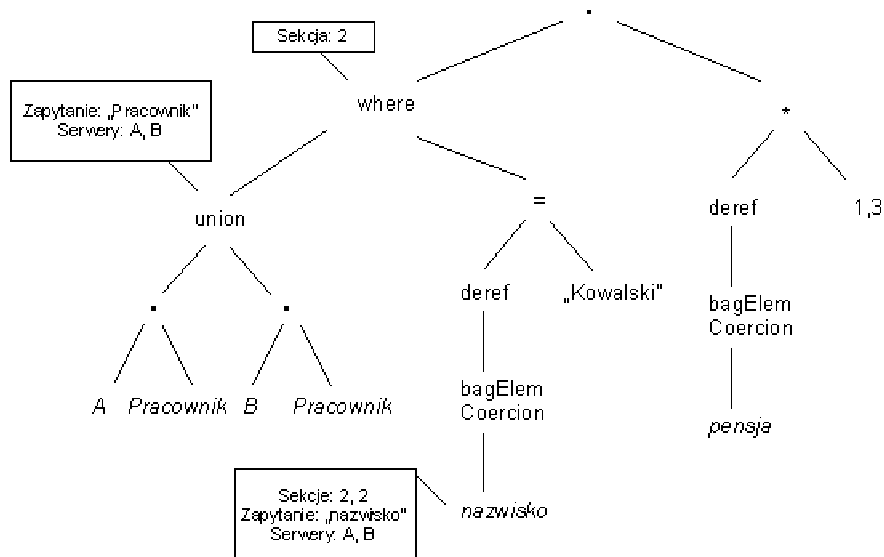
Krok 26. Przechodzimy do ewaluacji prawego argumentu operatora where. Poprzez węzły operatorów =, deref, bagElemCoercion docieramy do węzła nazwy *nazwisko*. Nazwa ta wiązana jest dwukrotnie w drugiej sekcji stosu w kontekście dwóch serwerów. Wynik etykietowania węzła przedstawiony został na rysunku 4.22. Stan stosów po odłożeniu sygnatury wyniku wiązania na stos wyników przedstawiono na rysunku 4.23.

Krok 27. Operator koercji nie zmienia stanu stosów. Jego etykieta jest podobna do tej w przypadku węzła nazwy *nazwisko*, stanowi ją zapytanie "nazwisko" (operator koercji nie jest uwzględniany w tekście zapytania) oraz zbiór serwerów {A, B}.

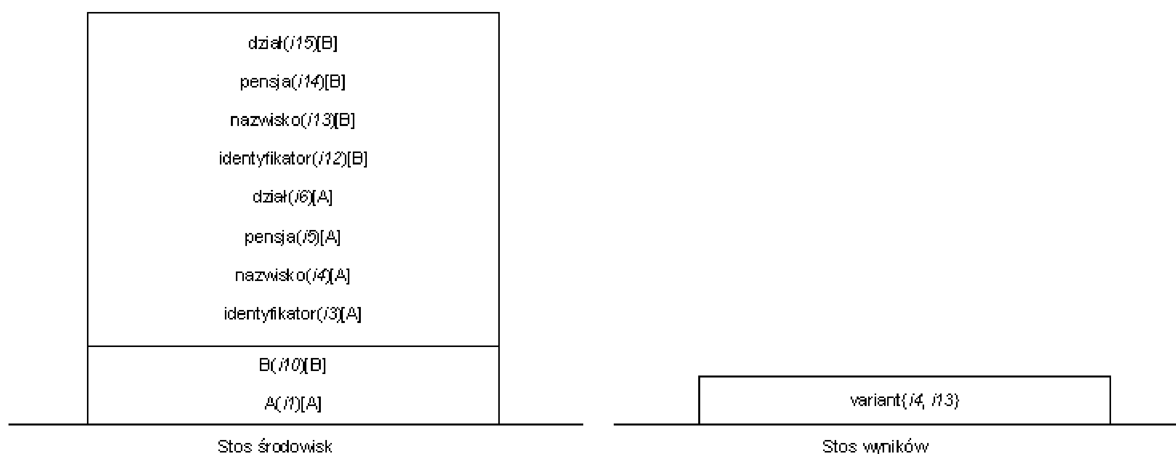
Krok 28. Operator dereferencji. Etykietę stanowi tu zapytanie deref(*nazwisko*) oraz zbiór serwerów {A, B}. Ze względu na to, że węzły *i4* oraz *i13* reprezentują ten sam typ



Rysunek 4.21: Stan stosów po kroku 25.

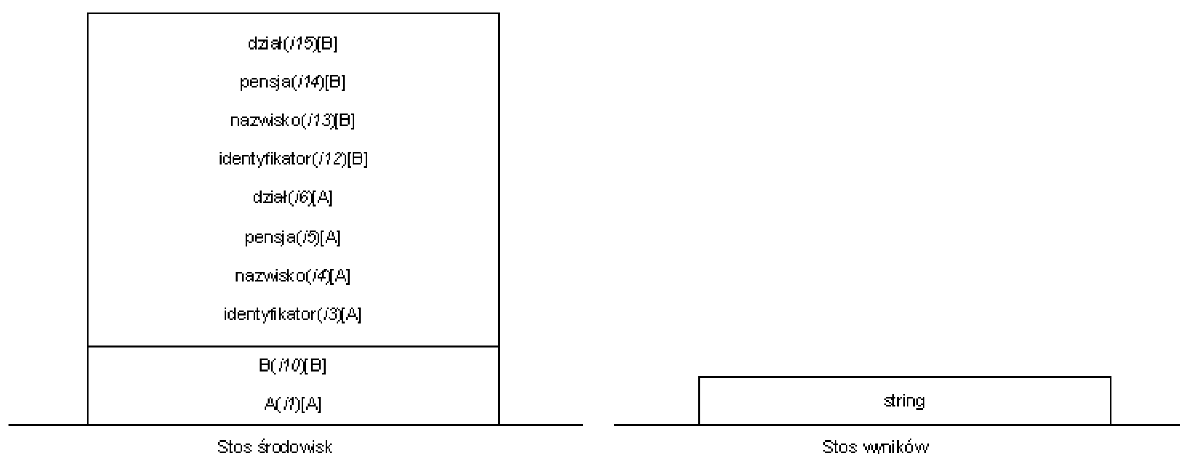


Rysunek 4.22: Wynik etykietowania w kroku 26.



Rysunek 4.23: Stan stosów po kroku 26.

wartości, wynikową sygnaturą jest string (rysunek 4.24).



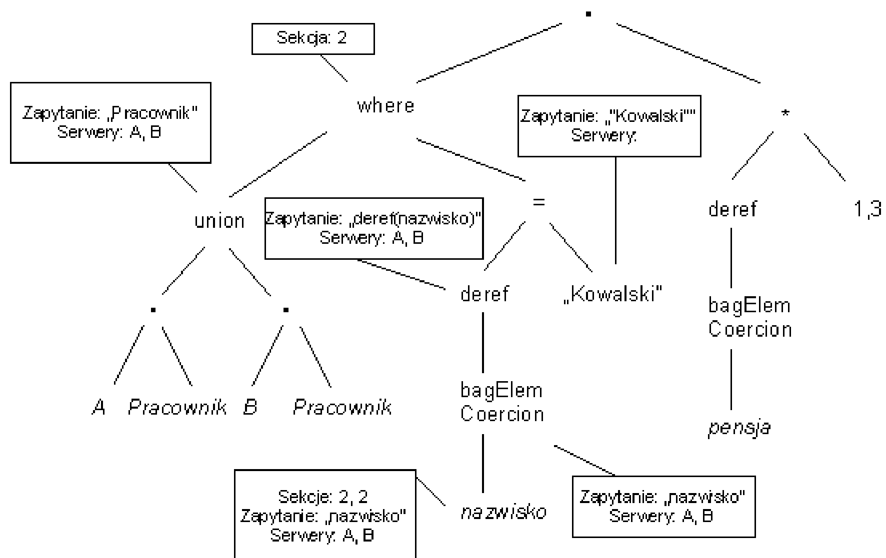
Rysunek 4.24: Stan stosów po kroku 28.

Krok 29. Następuje powrót do operatora = i ewaluacja jego prawego podzapytania, czego wynikiem jest sygnatura string. Rezultat etykietowania przedstawiono na rysunku 4.25. Należy zauważyć, że prawe poddrzewo nie jest etykietowane żadnym z serwerów.

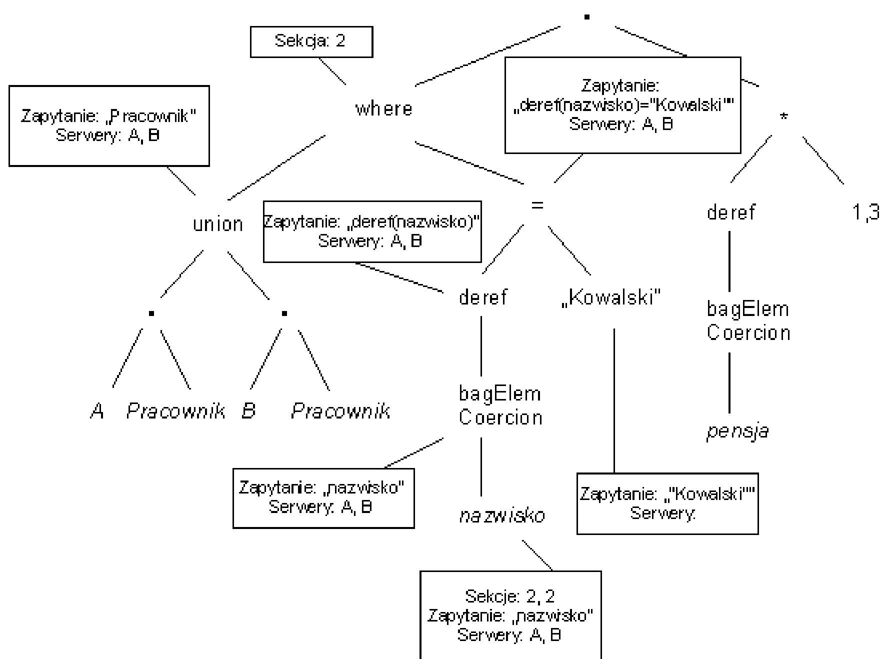
Krok 30. Operator =, konstrukcja wyniku. Ponieważ prawe podzapytanie nie jest etykietowane, żadnym serwerem w wyniku statycznej dekompozycji otrzymujemy zapytanie deref(*nazwisko*) = "Kowalski" z przypisanym do niego zbiorem serwerów {A, B} (rysunek 4.26).

Krok 31. Operator where, konstrukcja wyniku. Oba podzapytania są etykietowane tym samym zbiorem serwerów, a nazwa *nazwisko* wiązana jest w sekcji włożonej na stos przez ten operator. Oznacza to, że możemy ponownie skorzystać z rozdzielności względem sumy (rysunek 4.27). Na stos wyników odkładana jest sygnatura variant{*i2*, *i11*}.

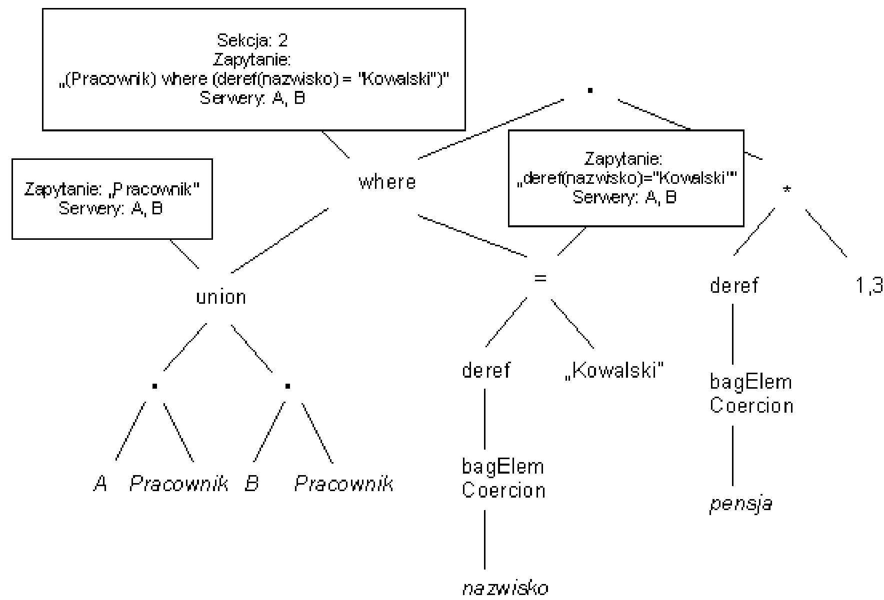
Krok 32. Następuje powrót do korzenia całego drzewa, czyli operatora kropki. Wynik *static_nested* jest taki sam jak na rysunku 4.21. Ze względu na to, że prawe poddrzewo jest bardzo podobne do prawego poddrzewa operatora where, a jego ewaluacja dokonywana jest w tym samym środowisku, jej szczegóły zostały tu pominięte. Etykietowanie prawego



Rysunek 4.25: Wynik etykietowania w kroku 29.

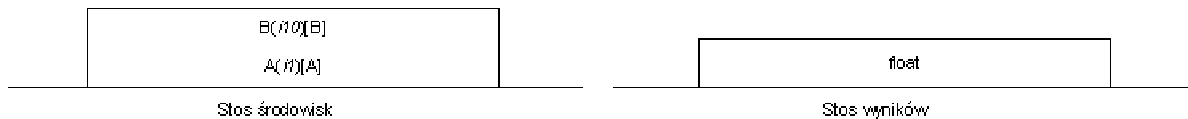


Rysunek 4.26: Wynik etykietowania w kroku 30.



Rysunek 4.27: Wynik etykietowania w kroku 31.

poddrzewa przedstawiono na rysunku 4.29, a stan stosów po zakończeniu jego ewaluacji na rysunku 4.28.



Rysunek 4.28: Stan stosów po kroku 32.

Krok 33. Powrót do węzła operatora kropki, konstrukcja wyniku. Możemy skorzystać z własności rozdzielności operatora względem sumy, na podstawie tego samego warunku co poprzednio. Otrzymujemy, więc końcowy rezultat dekompozycji w postaci zapytania:

$$((Pracownik)\ \mathbf{where}\ (\mathrm{deref}(nazwisko)\ =\ \text{"Kowalski"}))\ \mathbf{.}\ (\mathrm{deref}(pensja)\ *\ 1,3)$$

oraz listy serwerów {A, B}. Wynikiem zapytania będzie natomiast wartość typu zmiennoprzecinkowego (float).

Krok 34. Wątek przetwarzający zapytanie rezerwuje połączenia z serwerami A i B.

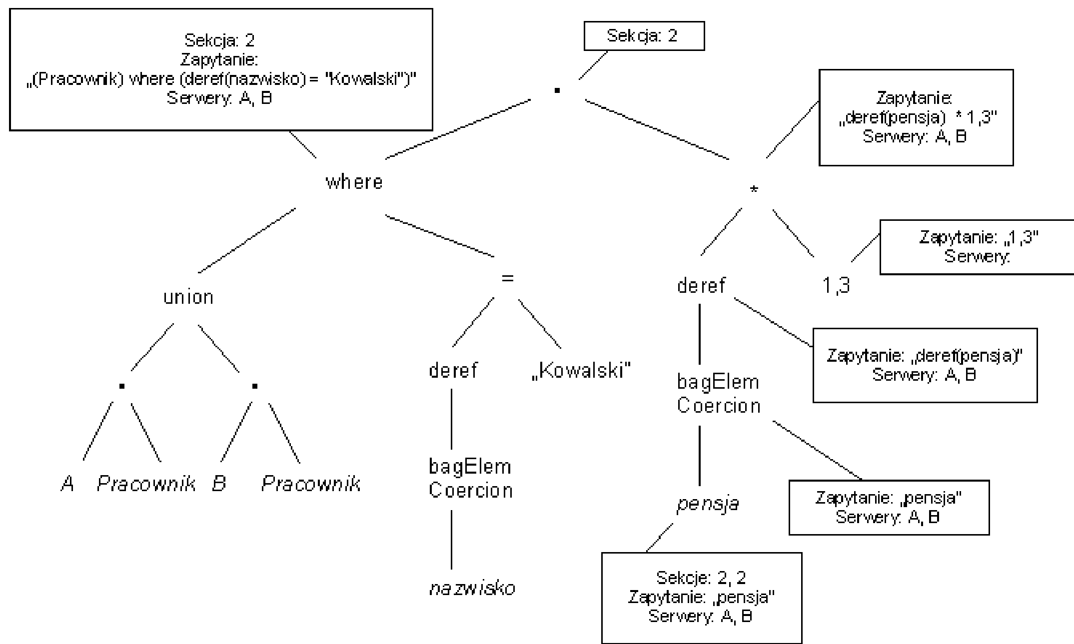
Krok 35. Zapytanie otrzymane w wyniku dekompozycji zostaje równolegle wysłane na serwery A i B.

Krok 36. Również równolegle odbierane są odpowiedzi otrzymane od obu serwerów.

Krok 37. Uzyskane wyniki są sumowane (w sensie sumy multizbiorów) w celu otrzymania ostatecznego wyniku zapytania.

Krok 38. Wynik zapytania jest wysyłany do klienta.

Zauważmy, że w tym przypadku udało się nie tylko zastosować przetwarzanie równoległe, ale także, przy założeniu, że jest tylko jeden pracownik o nazwisku Kowalski, ograniczyć komunikację między serwerami do wysłania dwóch zapytań i odebrania wyników w postaci pustego zbioru oraz zbioru zawierającego pojedynczą wartość zmiennoprzecinkową.



Rysunek 4.29: Wynik etykietowania w kroku 32.

4.3. Przykład nr 2

W przykładzie 2 chcemy uzyskać nazwiska pracowników oraz nazwy i lokalizacje działów, w których pracują, którzy zarabiają poniżej średniej krajowej. Posłużymy się więc zapytaniem:

$$((\textit{Pracownik}\ \textbf{where}\ \textit{pensja} < 2400)\ \textbf{join}\ \textit{dział})\ .\ (\text{deref}(\textit{nazwisko}), \text{deref}(\textit{Dział}\ .\ \textit{nazwa}), \text{deref}(\textit{Dział}\ .\ \textit{lokalizacja}))$$

Drzewo składniowe zapytania stworzone przez parser przedstawiono na rysunku 4.30. Występujący w zapytaniu operator join jest operatorem złączenia zależnego, natomiast "," tworzy struktury będące wynikiem iloczynu kartezjańskiego swoich argumentów.

Ze względu na to, że wiele czynności wykonywanych podczas ewaluacji będzie bardzo podobne do tych z przykładu 1, w opisie przykładu 2 ograniczono się tylko do ciekawszych fragmentów.

Krok 1. Rozpoczyna się pierwszy przebieg statycznej ewaluacji. Początkowy stan stosów jest taki, jak na rysunku 4.2.

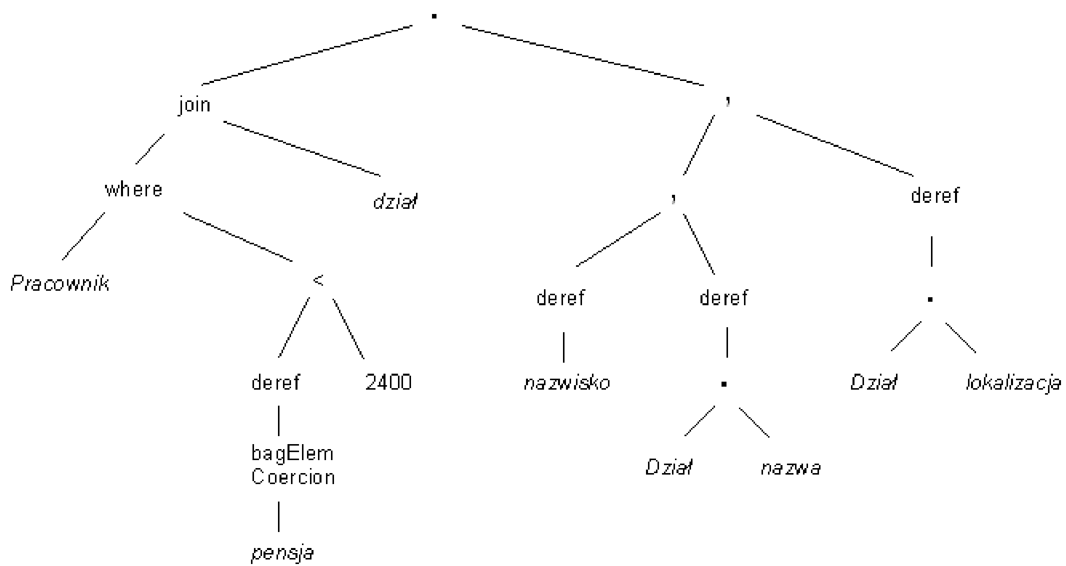
Krok 2. Wiązanie nazwy *Pracownik*. Proces ten odbywa się analogicznie do opisanego w kroku 3 dla przykładu 1. Drzewo po podstawieniu makra przedstawiono na rysunku 4.31.

Krok 3. Ewaluacja lewego poddrzewa drzewa o korzeniu w węźle reprezentującym operator join odbywa się podobnie jak w przykładzie 1 (kroki 1-14). Jej wynikiem jest sygnatura bag{*i1*}. Stan stosów po wykonaniu *static_nested* dla tej sygnatury przedstawiono na rysunku 4.32.

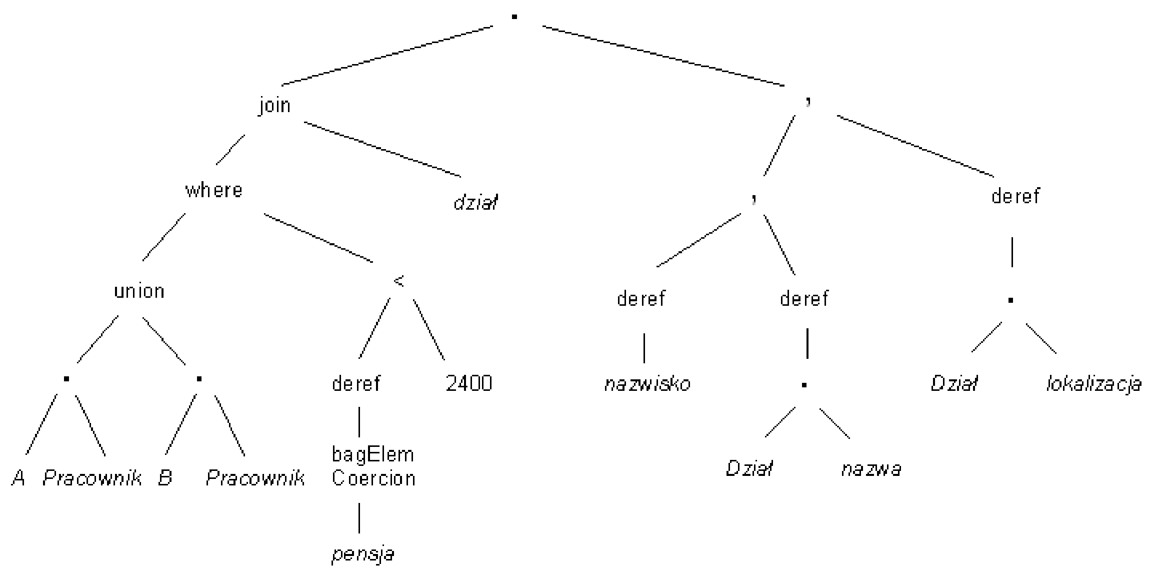
Krok 4. Rezultatem ewaluacji prawego argumentu operatora join jest sygnatura *i5*. W wyniku zastosowania operatora dostajemy więc sygnaturę bag{struct{*i1*, *i2*}}.

Krok 5. *static_nested* dla lewego argumentu operatora kropki (rysunek 4.33). Zauważmy, że w wyniku wcześniejszego działania operatora join, w wierzchołkowej sekcji stosu znajdują się teraz bindery podobiektów obiektu *i1* oraz binder opisujący obiekt wskazywany przez podobiekt *i5*.

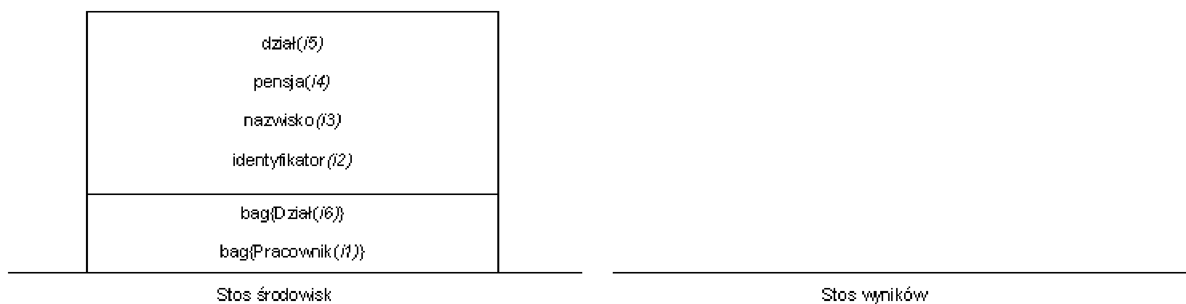
Krok 6. Wiązanie nazwy *nazwisko*. Wynikiem jest sygnatura *i3*.



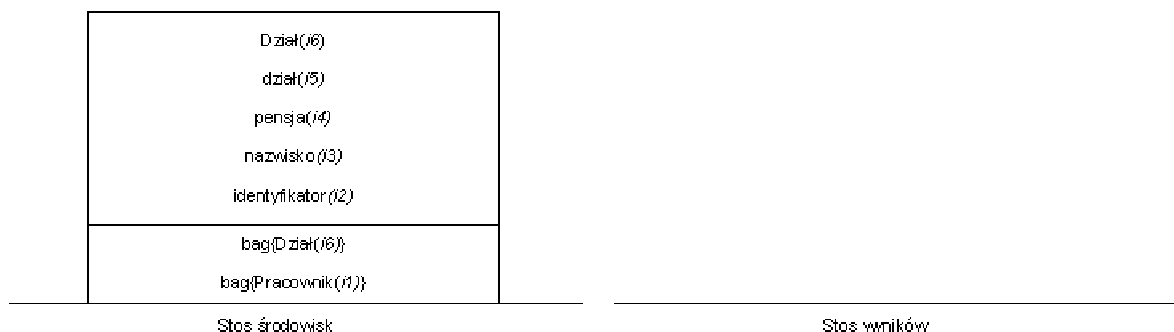
Rysunek 4.30: Drzewo składniowe po etapie parsowania.



Rysunek 4.31: Drzewo składniowe po rozwinięciu makra *Pracownik*.



Rysunek 4.32: Stan stosów po kroku 3.



Rysunek 4.33: Stan stosów po kroku 5.

Krok 7. Zastosowanie operatora deref dla sygnatury otrzymanej w kroku 6. W wyniku dostajemy sygnaturę string.

Krok 8. Wiązanie nazwy *Dział*. Ponieważ stos jest przeszukiwany od góry, oznacza to, że wiązanie następuje w sekcji nr 2 i tylko bindery z tej sekcji są brane pod uwagę. Wynika stąd, że pomimo istnienia makra o nazwie *Dział*, nie zostanie ono tutaj podstawione. Taka podmiana ma miejsce jedynie w przypadku nazw wiązanych w bazowej sekcji stosu. W wyniku dostajemy sygnaturę *i6*.

Krok 9. Operator kropki wkłada na stos środowisk trzecią sekcję z binderami podobiektów obiektu *i6*, gdzie wiązana jest nazwa *nazwa*. W wyniku zastosowania operatora otrzymujemy więc sygnaturę *i7*.

Krok 10. Ponieważ obiekt *i17* reprezentuje wartość typu napis, wynikiem zastosowania dla niego operatora deref jest sygnatura string.

Krok 11. W wyniku zastosowania operatora "," dla argumentów string i string otrzymujemy struct{string, string}.

Krok 12. Ewaluacja poddrzewa dla podzapytania deref(*Dział* . *lokalizacja*) przebiega podobnie do opisanej w krokach 8-10. Tu także wynikiem jest sygnatura string.

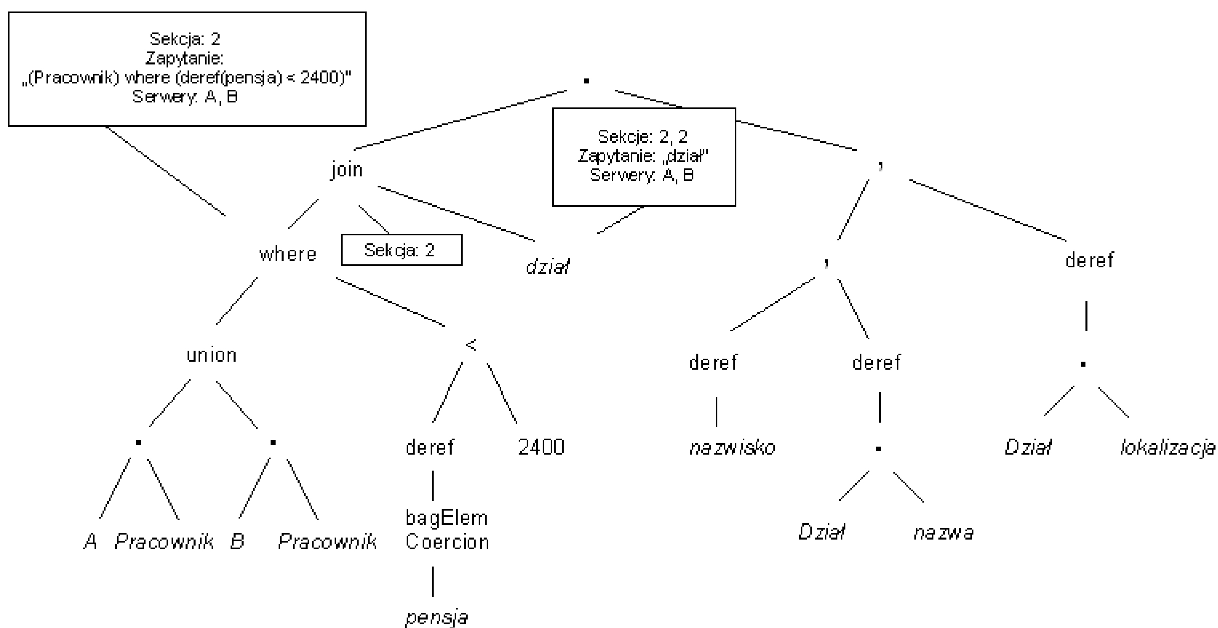
Krok 13. Ponieważ nie zagnieżdżamy struktur, wynikiem zastosowania operatora "," dla argumentów w postaci struct{string, string} oraz string jest sygnatura struct{string, string, string}. Rezultat otrzymany w pierwszym przebiegu statycznej ewaluacji dla całego zapytania jest postaci bag{struct{string, string, string}}. Udało się więc potwierdzić zgodność zapytania ze schematem globalnym.

Krok 14. Ewaluacja drzewa o korzeniu w węźle reprezentującym operator where odbywa sie podobnie do opisanej w krokach 17-31 dla przykładu 1. Nazwa *dział* wiązana jest w drugiej sekcji stosu, włożonej przez operator join i dotyczy obu serwerów. Wynik etykietowania argumentów operatora join przedstawiono na rysunku 4.34.

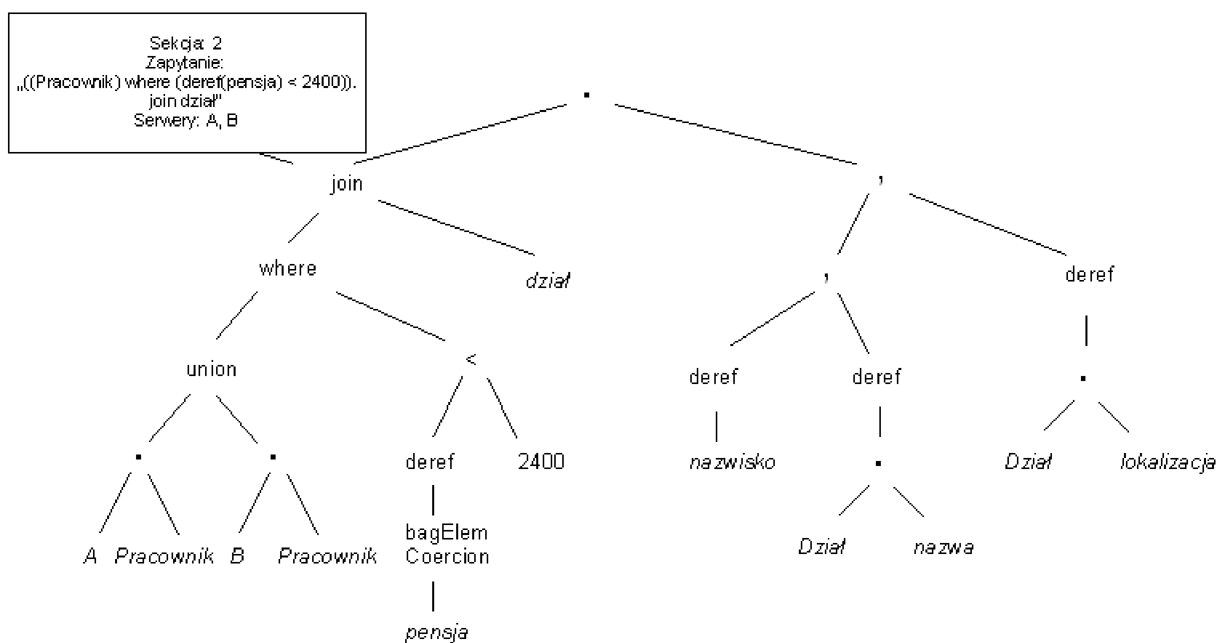
Krok 15. Operator join jest rozdzielny względem sumy, a jego argumenty są etykietowane tym samym zbiorem serwerów oraz nazwy w prawym podzapytaniu nie są wiązane poniżej sekcji tworzonej na stosie środowisk przez ten operator, więc otrzymujemy wynik przedstawiony na rysunku 4.35.

Krok 16. Dla podzapytania deref(*nazwisko*) otrzymujemy etykietę w postaci zapytania *nazwisko* oraz zbioru serwerów {A, B}.

Krok 17. W przypadku podzapytania deref(*Dział* . *nazwa*) dla operatora kropki wykorzystujemy własność rozdzielności względem sumy, co w rezultacie daje nam etykietę w postaci zapytania deref(*Dział* . *nazwa*) i zbioru serwerów {A, B}.

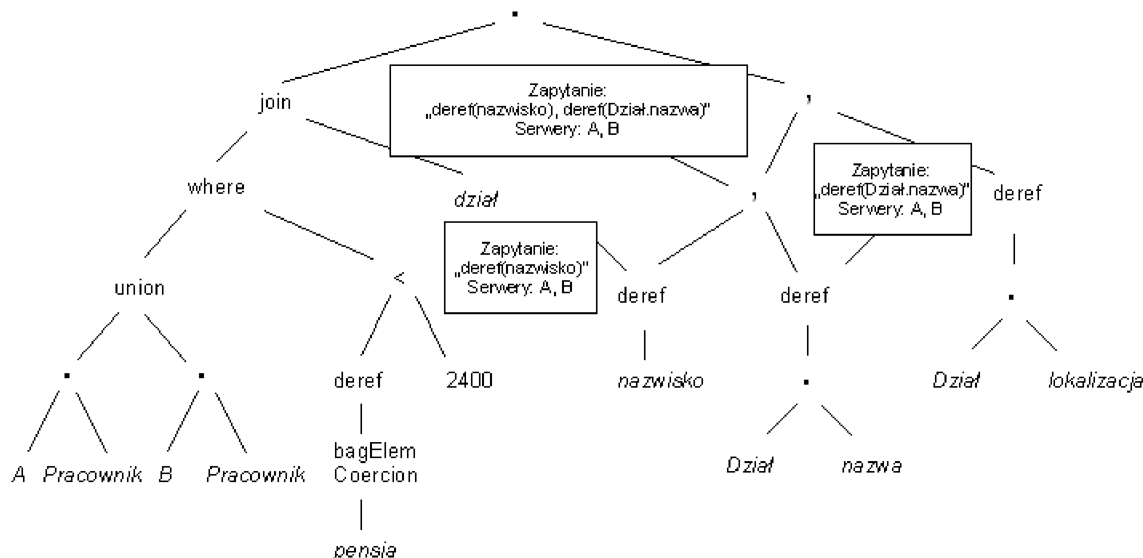


Rysunek 4.34: Wynik etykietowania w kroku 14.



Rysunek 4.35: Wynik etykietowania w kroku 15.

Krok 18. Dla operatora "," mamy argumenty, w których nie występują nazwy serwerów, etykietowane tym samym zbiorem serwerów, co pozwala na zastosowanie statycznej dekompozycji (rysunek 4.36).



Rysunek 4.36: Wynik etykietowania w kroku 18.

Krok 19. Dla operatora ",", którego węzeł jest korzeniem poddrzewa bezpośrednio połączonego z węzłem operatora kropki, ewaluacja przebiega jak w krokach 17-18.

Krok 20. Dla operatora kropki stosujemy własność rozdzielności względem sumy, co prowadzi nas do końcowego wyniku dekompozycji (rysunek 4.37).

Krok 21. Na serwery A i B zostaje wysłane zapytanie:

$$((\textit{Pracownik}\ \textbf{where}\ (\text{deref}(\textit{pensja}) < 2400))\ \textbf{join}\ \textit{dział})\ .\ (\text{deref}(\textit{nazwisko}), \text{deref}(\textit{Dział}\ .\ \textit{nazwa}), \text{deref}(\textit{Dział}\ .\ \textit{lokalizacja}))$$

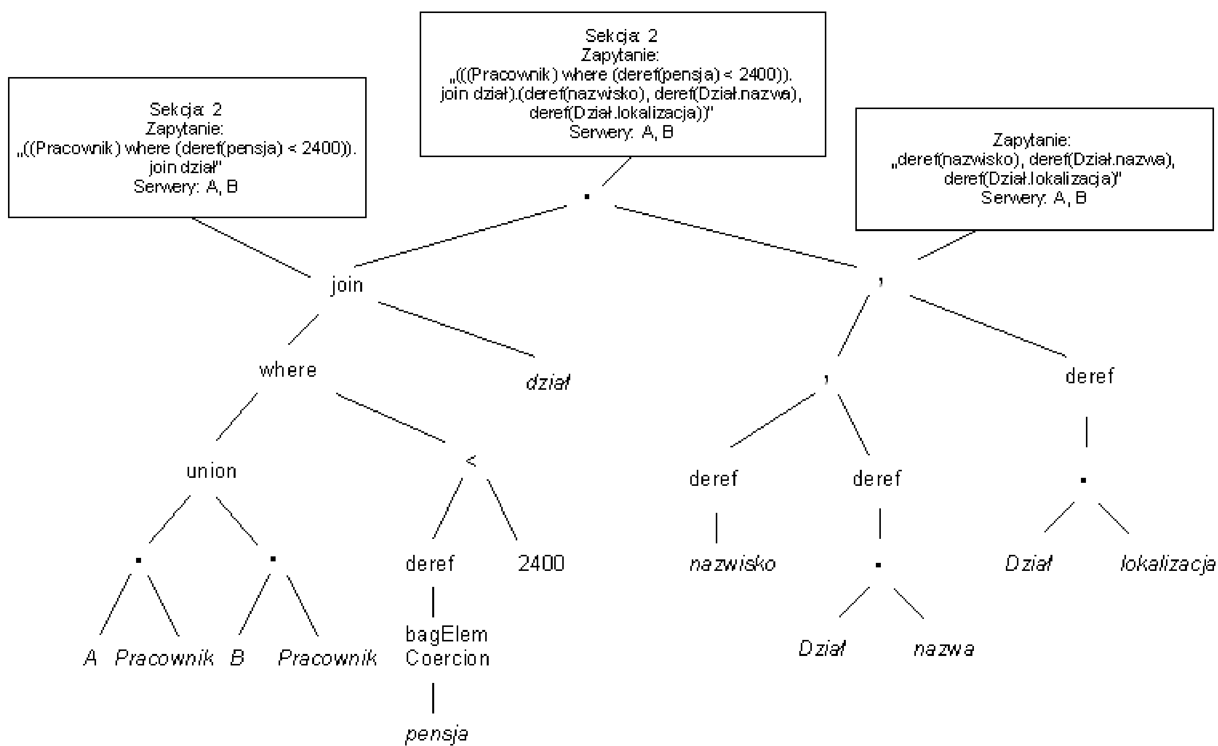
Zebrane wyniki są następnie sumowane, dzięki czemu otrzymujemy ostateczny rezultat wykonania zapytania.

4.4. Przykład nr 3

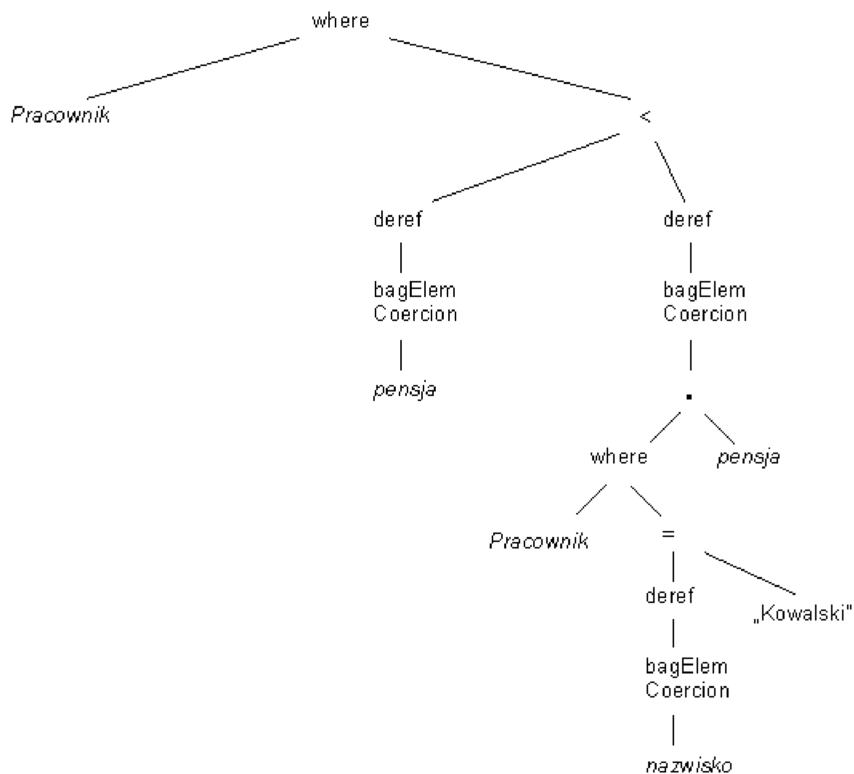
Na zakończenie przedstawiono przykład ewaluacji zapytania nie należącego do wspieranej klasy zapytań. Załóżmy, że chcemy uzyskać referencje do obiektów reprezentujących pracowników, którzy zarabiają mniej niż Kowalski, przy założeniu, że jest tylko jeden pracownik o takim nazwisku. Mamy więc zapytanie:

$$\textit{Pracownik}\ \textbf{where}\ \textit{pensja} < ((\textit{Pracownik}\ \textbf{where}\ \textit{nazwisko} = \text{"Kowalski"})\ .\ \textit{pensja})$$

Intuicyjnie, ponieważ obiekty etykietowane nazwą *Pracownik* pochodzą z różnych serwerów, nie jest możliwa statyczna dekompozycja zapytania, w wyniku której będzie możliwe uzyskanie końcowego rezultatu na podstawie odpowiedzi na tylko jedno, to samo pytanie wysłane do grupy serwerów. Dzieje się tak dlatego, że wysokość pensji "Kowalskiego" jest zapisana tylko na jednym serwerze, co oznacza, że musielibyśmy ją wyliczyć wcześniej i przesłać jako parametr większego zapytania. Niniejszy przykład pokazuje jak system wykrywa taką sytuację.

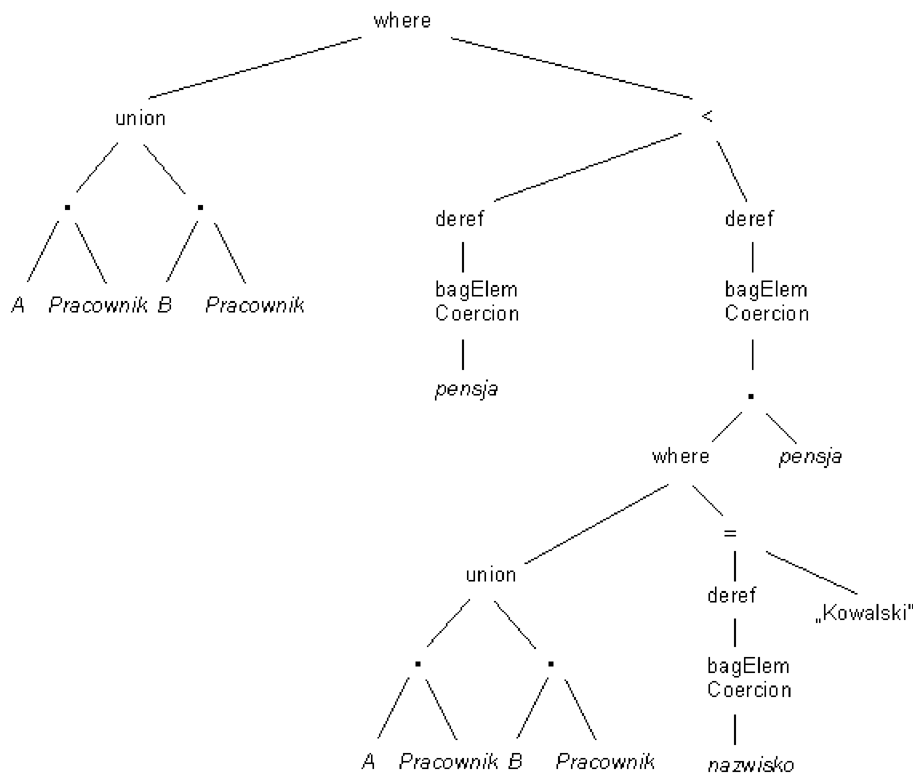


Rysunek 4.37: Wynik etykietowania w kroku 20 (przykład 2).



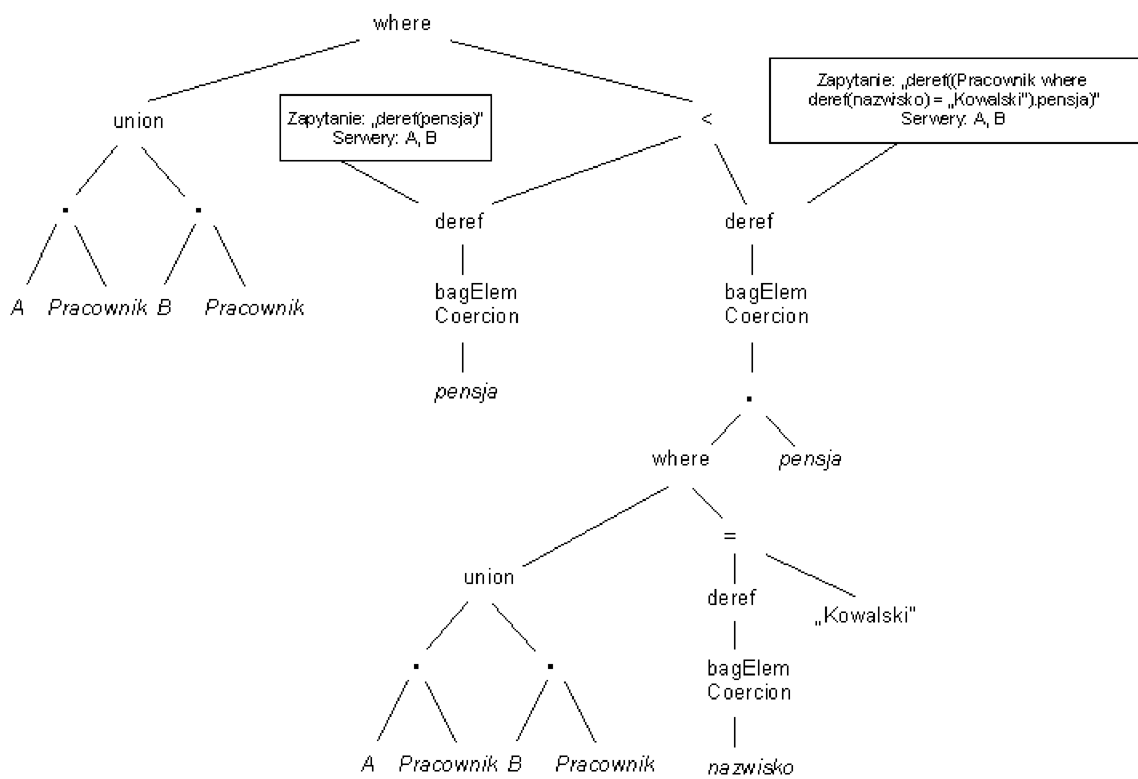
Rysunek 4.38: Drzewo składniowe po etapie parsowania.

Drzewo składniowe otrzymane w wyniku parsowania przedstawiono na rysunku 4.38. Pierwszy przebieg statycznej ewaluacji kończy się pomyślnie, co oznacza zgodność zapytania ze schematem globalnym. Wynikiem ewaluacji jest sygnatura bag{*i1*}, a drzewo składniowe po rozwinięciu makr integracyjnych przedstawiono na rysunku 4.39. Zauważmy, że obie nazwy *Pracownik* wiązane są w bazowej sekcji stosu środowisk.



Rysunek 4.39: Drzewo składniowe po rozwinięciu makr.

Drugi przebieg statycznej ewaluacji zostaje zakłócony przy wyliczaniu wyniku dla operatora "<" (rysunek 4.40). W jego prawym argumencie występują wiązania nazw serwerów. Dla takiego przypadku mamy warunek, że oba podzapytania mogą być etykietowane co najwyżej jednym, tym samym serwerem. Zbiory serwerów są jednak w wypadku obu argumentów dwuelementowe. Dekompozycja zapytania wspierana przez system nie jest więc możliwa, co powoduje natychmiastowe zakończenie przetwarzania zapytania i zgłoszenie błędu.



Rysunek 4.40: Etykietowanie argumentów operatora "<".

Rozdział 5

Podsumowanie

Model relacyjny odegrał bardzo istotną rolę w upowszechnieniu systemów baz danych w ich obecnym kształcie. Wraz z nowymi wyzwaniami coraz bardziej widoczne są jednak ograniczenia tego podejścia, w związku z czym poszukuje się nowych rozwiązań. Odpowiedzią na bieżące problemy miały być obiektowe bazy danych, jednak ze względu na kruche podstawy koncepcyjne poszczególnych propozycji oraz często brak ich praktycznej realizacji nie udało się wypromować systemów bazodanowych opartych na tym modelu danych. W zastępstwie na rynku pojawiły się rozwiązania hybrydowe wzbogacające systemy relacyjne o elementy typowe dla obiektowych baz danych. Ich możliwości są jednak wciąż niewystarczające, a nowe składniki powodują poważne komplikacje języków zapytań oraz samych systemów zarządzania bazami danych.

Taki stan rzeczy nie poprawia sytuacji w dziedzinie optymalizacji zapytań, a w szczególności dotyczy to środowisk rozproszonych, gdzie nowych wyzwań jest dużo więcej. By możliwe było efektywne operowanie na danych zawartych w takich bazach, potrzebne są rozwiązania, które są zarówno proste koncepcyjnie, jak i wystarczająco ogólne, by objąć nawet najbardziej złożone przypadki. Wszystko musi też być oparte na solidnych i spójnych podstawach teoretycznych gwarantujących ponadto naturalny sposób dodawania kolejnych rozszerzeń.

Zdaniem autora tej pracy podejście stosowe, jeśli samo w sobie nie jest odpowiedzią na wymienione wyżej postulaty, stanowi doskonałą bazę wyjściową do zbudowania takiego rozwiązania. Zalety SBA w tym względzie zostały przedstawione w rozdziale 2.

Rozwiązania w zakresie optymalizacji zapytań są wartościowe jedynie, jeśli mogą być w sposób efektywny zrealizowane w rzeczywistych systemach. Często na etapie przejścia między teorią a praktyką pojawia się cały szereg problemów koncepcyjnych, z którymi trzeba się uporać. Niezbędna jest więc platforma systemowa pozwalająca testować nowe propozycje oraz eksperymentować z wariantami ich implementacji. Autor tej pracy ma nadzieję, że stworzone przez niego oprogramowanie będzie przyczynkiem takiego systemu, w oparciu o który powstanie bardzo efektywna, kompleksowa realizacja rozproszonej bazy danych o rozbudowanej funkcjonalności.

Implementacja zaprezentowana w rozdziale 3 umożliwia realizację zapytań podlegających statycznej dekompozycji, której wynikiem jest jedno zapytanie wysyłane do grupy serwerów. Taki typ przetwarzania jest bardzo efektywny, ponieważ pozwala nie tylko oprzeć się na przetwarzaniu równoległym, ale także często znacznie zredukować koszty komunikacji, które mogą być głównym czynnikiem wpływającym na czas odpowiedzi. Zalety te widać między innymi w przykładzie 1 z rozdziału 4.

Pomimo, że przedstawione tu oprogramowanie na obecnym etapie realizacji nie wspiera rozwiązań dla wielu potrzeb implikowanych przez rzeczywiste systemy, istnieją bardzo szero-

kie możliwości jego rozbudowy, które w znaczący sposób mogą poszerzyć jego funkcjonalność. Większość rozszerzeń może być dodawana niezależnie, a wiele z nich wymaga stosunkowo niewielkiego nakładu prac. Poniżej przedstawiono wybrane propozycje:

- rozszerzenie statycznej dekompozycji zapytań o możliwość wysyłania różnych zapytań do różnych serwerów
- dodanie innych niż suma wielozbiorów sposobów łączenia wyników pośrednich
- wykorzystanie w algorytmie dekompozycji pozostałych własności operatorów opisanych w rozdziale 2.8.2
- dodanie metod optymalizacyjnych opartych na przepisywaniu opracowanych dla systemów scentralizowanych
- zastąpienie modułu łączącego wyniki cząstkowe kompletnym egzekutorem zapytań
- dodanie obsługi operacji modyfikujących dane
- dodanie operacji zarządzania metabazą
- stworzenie modułu administracyjnego
- zastosowanie aktualizowalnych perspektyw do integracji zasobów
- wsparcie dla przetwarzania zapytań parametryzowanych pojedynczą wartością (przykład 3 z rozdziału 4)
- dodanie obsługi lokalnego składowiska danych
- implementacja wsparcia dla przetwarzania dowolnych zapytań

Dodatek A

Opis załączonego oprogramowania

Na dołączonej do pracy płycie znajduje się implementacja systemu rozproszonej bazy danych, o którym mowa w rozdziale 3. Poniżej przedstawiono zawartość poszczególnych katalogów. Etykieta LoXiM oznacza, że dany moduł został w całości przeniesiony z systemu LoXiM.

- **LoXiM** - implementacja scentralizowanej bazy danych (stworzona przez studentów Uniwersytetu Warszawskiego - najnowsza wersja dostępna pod adresem: http://stencel.mimuw.edu.pl/bd/szbd/)
- **GInteg** - implementacja warstwy integracyjnej (stworzona przez autora)
  - **Config** - implementacja modułu ładującego ustawienia konfiguracyjne (LoXiM)
  - **Driver** - obudowa protokołu komunikacyjnego (LoXiM)
  - **Errors** - wspomaganie obsługi błędów (LoXiM)
  - **IntraCommunication** - komunikacja międzyserwerowa
    - **Merger.h**, **Merger.cpp** - łączenie wyników cząstkowych
    - **S2SConnection.h**, **S2SConnection.cpp** - połączenie ze zdalnym serwerem
    - **S2SConnectionsPoll.h**, **S2SConnectionsPoll.cpp** - pula połączeń ze zdalnymi serwerami
    - **Makefile**
  - **Metabase** - realizacja metabazy
    - **Metabase.h**, **Metabase.cpp** - implementacja globalnego obiektu metabazy
    - **Schema.h**, **Schema.cpp** - implementacja schematu
    - **Signature.h**, **Signature.cpp** - sygnatury obiektów umieszczanych na stosach
    - **SNode.h**, **SNode.cpp** - węzły schematu
    - **Makefile**
  - **QueryPreprocessor** - preprocesor zapytań
    - **QueryParser.h**, **QueryParser.cpp** - obiekt obudowujący parser zapytań
    - **SBQLLexer.h**, **SBQLLexer.cpp** - klasa bazowa dla leksera
    - **StaticEvaluator.h**, **StaticEvaluator.cpp** - implementacja statycznej ewaluacji
    - **StaticStack.h**, **StaticStack.cpp** - realizacja statycznych stosów
    - **TreeNode.h**, **TreeNode.cpp** - drzewo składniowe
    - **SBQLParser.y** - definicje parsera

      * **scanner.l** - definicje leksera
      * **Makefile**
  - **SBQLCli** - klient bazy danych (LoXiM)
  - **Server** - infrastruktura serwera integracyjnego
      * **Listener.h**, **Listener.cpp** - wątek nasłuchujący
      * **ServerThread.h**, **ServerThread.cpp** - wątek obsługi klienta
      * **ThreadList.h**, **ThreadList.cpp** - implementacja listy wątków obsługi klienta
      * **Server.cpp** - procedura main serwera integracyjnego
      * **Server.conf** - plik konfiguracyjny serwera
      * **Makefile**
  - **TCPProto** - obudowa protokołu TCP/IP (LoXiM)
  - **Utils** - dodatkowe komponenty na użytek innych modułów
      * **Bitmap.h**, **Bitmap.cpp** - implementacja bitmapy
      * **Makefile**
  - **client.sh** - skrypt uruchamiający aplikację kliencką
  - **make.defs** - wspólne definicje dla skryptów Makefile
  - **Makefile** - skrypt służący do kompilacji systemu

- **Dodatki** - materiały dodatkowe różnego typu
  - **inserty1**, **inserty2** - przykładowe pliki z instrukcjami tworzącymi obiekty na lokalnych serwerach

Implementacja przeznaczona jest dla systemu operacyjnego Linux. W celu instalacji warstwy integracyjnej należy skopiować katalog GInteg w docelowe miejsce w systemie plików, a następnie z poziomu katalogu na konsoli uruchomić program *make*. Włączenie serwera integracyjnego odbywa się za pomocą wywołania programu *Server* z podkatalogu GInteg/Server. Do wystartowania globalnej aplikacji klienckiej służy skrypt *client.sh* w katalogu GInteg.

Instalacja lokalnej bazy danych przebiega podobnie. Należy skopiować katalog LoXiM i uruchomić program *make*. Włączanie serwera bazy danych odbywa się za pomocą programu LoXiM/Server/*Listener*, natomiast lokalnej aplikacji klienckiej za pomocą skryptu LoXiM/*client.sh*.

Poniżej przedstawiono, gdzie znajdują się poszczególne ustawienia konfiguracyjne:

- lokalizacja plików składowiska danych oraz inne ustawienia dla lokalnego serwera - LoXiM/Server/Server.conf
- dane serwera w aplikacji klienckiej - LoXiM/SBQLCli/SBQLCli.cpp, main()
- port do nasłuchu w serwerze LoXiM - LoXiM/Server/Listener.cpp, main()
- port do nasłuchu w serwerze GInteg oraz inne ustawienia serwera - GInteg/Server/Server.conf
- makra integracyjne - GInteg/Metabase/Metabse.cpp, loadMacros()
- schematy metabazy - GInteg/Metabase/Schema.cpp, loadSchema()

- adresy i porty zdalnych serwerów - GInteg/IntraCommunication/S2SConnectionsPoll.cpp, buildConnections()

Standardowo warstwa integracyjna na starcie ładuje metabazę przedstawioną w rozdziale 3.5.4. Pliki tworzące odpowiednie obiekty na serwerach A i B zawarte są w katalogu Dodatki (inserty1 i inserty2). Załadowanie zapytań z pliku odbywa się zgodnie ze schematem:

```
./client.sh < inserty1
```

Poniżej znajdują się wybrane informacje na temat formatu zapytań obsługiwanych przez warstwę kliencką:

- system rozróżnia wielkość liter
- napisy należy umieszczać w cudzysłowach (np. "Kowalski")
- do oddzielenia części całkowitej liczby od części dziesiętnej należy używać kropki (np. 4.7)
- każde zapytanie należy zakończyć średnikiem oraz pojedynczym znakiem "/" w następnej linii po średniku
- wszystkie zapytania w systemie LoXiM przetwarzane są w transakcjach. W przypadku wykorzystania lokalnych klientów zadawanie zapytań należy rozpocząć od utworzenia transakcji - zapytanie: *begin*;.
- w warstwie integracyjnej nie ma obsługi transakcji - zapytania zadajemy bezpośrednio po uruchomieniu aplikacji klienckiej

System LoXiM był w momencie tworzenia tej pracy nadal w fazie rozwoju, co oznacza, że nie wykluczone są sytuacje, w których nie będzie on działał stabilnie. Autor tej pracy nie bierze więc za to odpowiedzialności. W przypadku, gdy powstały wcześniej błąd na trwałe zablokuje możliwość korzystania z lokalnej bazy, zaleca się usunięcie wszystkich plików składowiska danych (lokalizacja zapisana w LoXiM/Server/Server.conf). Serwer przy kolejnym uruchomieniu utworzy puste składowisko.

Bibliografia

[ACN00] S. Agrawal, S. Chaudhuri, V.R. Narasayya, *Automated Selection of Materialized Views and Indexes in SQL Databases*, VLDB 2000: 496-505

[Ber95] E. Bertino, *Query Decomposition in an Object-Oriented Database System Distributed on a Local Area Network*, RIDE-DOM 1995: 2-9

[BG+81] P.A. Bernstein, N. Goodman, E. Wong, C.L. Reeve, J. B. Rothnie *Query Processing in a System for Distributed Databases (SDD-1)*, ACM Trans. Database Syst. 6(4): 602-625 (1981)

[GM05] H. Gupta, I.S. Mumick, *Selection of Views to Materialize in a Data Warehouse*, IEEE Trans. Knowl. Data Eng. 17(1): 24-43 (2005)

[Hal01] A.Y. Halevy, *Answering queries using views: A survey*, VLDB J. 10(4): 270-294, 2001

[IK84] T. Ibaraki, T. Kameda, *Optimal nesting for computing n-relational joins.*, ACM Trans. on Database Systems, 9(3):482–502, 1984.

[IK90] Y. Ioannidis, Y. Kang, *Randomized algorithms for optimizing large join queries*, Proc. ACM-SIGMOD Conference on the Management of Data, pages 312-321, Atlantic City, NJ, May 1990

[Ioa96] Y.E. Ioannidis, *Query Optimization*, Computing Surveys 28(1), 121-123, 1996

[IW87] Y. Ioannidis, E. Wong, *Query optimization by simulated annealing*, Proc. ACM-SIGMOD Conference on the Management of Data, pages 9-22, San Francisco, CA, May 1987

[JR02] V. Josifovski, T. Risch, *Query Decomposition for a Distributed Object-Oriented Mediator System*, Distributed and Parallel Databases 11(3): 307-336 (2002)

[KGV83] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, *Optimization by simulated annealing*, Science, 220(4598):671-680, May 1983

[Kos00] D. Kossmann, *The State of the art in distributed query processing*, ACM Comput. Surv. 32(4): 422-469 (2000)

[KLS02] H. Kozankiewicz, J. Leszczydłowski, K. Subieta, *Updateable Object Views*, Institute of Computer Science PAS Report 950, 2002

[KSS04] H. Kozankiewicz, K. Stencel, K. Subieta, *Integration of Heterogeneous Resources through Updatable Views*, Workshop on Emerging Technologies for Next Generation GRID (ETNGRID-2004), June 2004, Proc. published by IEEE

[KSS05a] H. Kozankiewicz, K. Stencel, K. Subieta, *Implementation of Federated Databases through Updatable Views*, Proc. of the European Grid Conference, Amsterdam, The Netherlands, 2005, LNCS

[KSS05b] H. Kozankiewicz, K. Stencel, K. Subieta, *Distributed Query Optimization in the Stack-Based Approach*, High Performance Computing and Communications, First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005, Proceedings, LNCS 3726, Springer 2005, pp.904-909 SBA

[LNS96] W. Litwin, M. Neimat, D.A. Schneider, *LH* - A Scalable, Distributed Data Structure*, ACM Trans. Database Syst. 21(4): 480-525 (1996)

[LS+99] E. Leclercq, M. Savonnet, M.N. Terrasse, K. Yétongnon, *Object Clustering Methods and a Query Decomposition Strategy for Distributed Object-Based Information Systems*, DEXA 1999: 781-790

[ML86] L.F. Mackert, G.M. Lohman, *R* validation and performance evaluation for distributed queries*, Proc. 12th Int. VLDB Conf., pages 149-159, Kyoto, Japan, Aug 1986

[MR+01] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham, *Materialized View Selection and Maintenance Using Multi-Query Optimization*, SIGMOD Conference 2001

[NSS86] S. Nahar, S. Sahini, E. Shragowitz, *Simulated annealing and combinatorial optimization*, Proc. 23rd Design Automatition Conference, pages 293-299, 1986

[OV99] T. Özsu, P. Valduriez, *Principles of Distributed Database Systems, Second Edition*, Prentice-Hall 1999

[Plo00] J. Płodzień, *Optimization Methods in Object Query Languages*, Pd.D. Thesis. Institute of Computer Science, Polish Academy of Sciences, 2000

[PP85] T.W. Page, G.J. Popek, *Distributed Management in Local Area Networks*, PODS 1985: 135-142

[SA+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price, *Access path selection in relational database management system*, Proc. ACM-SIGMOD Conf. on the Management of Data, pages 23-34, Boston, MA, June 1979

[SG88] A. Swami, A. Gupta, *Optimization of large join queries*, Proc. ACM-SIGMOD Conference on the Management of Data, pages 8-17, Chicago, IL, June 1988

[SM+01] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, H. Balakrishnan *Chord: A scalable peer-to-peer lookup service for internet applications*, SIGCOMM 2001: 149-160

[Sub95] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt, *A Stack-Based Approach to Query Languages*, Proc. East-West Database Workshop, 1994, SpringerWorkshops in Computing, 1995 (Również: Institute of Computer Science PAS Report 738, 1993)

[Sub04] K. Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych, 2004

[Swa89] A. Swami, *Optimization of large join queries: Combining heuristics and combinatorial techniques*, Proc. ACM-SIGMOD Conference on the Management of Data, pages 367-376, Portland, OR, June 1989

[SY82] G.M. Sacco, S.B. Yao, *Query Optimization in Distributed Data Base Systems*, Advances in Computers 21: 225-273 (1982)

[YM98] C.Yu, W.Meng, *Principles of Database Query Processing for Advanced Applications*, Morgan Kaufmann Publishers, 1998