Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Dariusz Gryglas

Nr albumu: 203087

Implementacja aktualizowalnych perspektyw w systemie LoXiM

Praca magisterska
na kierunku INFORMATYKA
w zakresie BAZY DANYCH

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Wrzesień 2008

Oświadczenie kierującego pracą

Oświadczam, ze niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego praca

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, ze 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 załączona wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy prezentuję implementację wirtualnych (niezmaterializowanych), aktualizowalnych, obiektowych perspektyw baz danych. Perspektywy zaimplementowałem w systemie LoXiM, prototypowym, obiektowym, semistrukturalnym systemie zarządzania bazą danych z językiem zapytań SBQL. Osoba definiująca perspektywy w systemie LoXiM, może w ich definicji zapisać informacje o dowolnej intencji aktualizacyjnej. Informacja zapisywana jest w postaci procedur przeciążających poszczególne operacje wykonywane na wirtualnych danych. Dla użytkownika wirtualne obiekty generowane przez takie perspektywy są nierozróżnialne od obiektów zapisanych w składzie danych. W pracy przedstawiam swoją koncepcję realizacji wirtualnych obiektów wskaźnikowych. Wirtualne wskaźniki w LoXiM są w pełni przezroczyste, można je aktualizować oraz wykorzystywać do nawigacji tak samo jak zwykłe obiekty wskaźnikowe.

Słowa kluczowe

podejście stosowe SBA, język zapytań SBQL, obiektowe bazy danych, perspektywy, aktualizacja wirtualnych danych

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2. Database Management
H.2.3. Languages
H.2.4. Systems

Tytuł pracy w języku angielskim

Implementation of updateable views in LoXiM system.

Spis treści

Wprowadzenie

W ramach przedmiotu „Systemy zarządzaniami bazami danych" i seminarium magisterskiego „Bazy Danych" pod opieką dr Krzysztofa Stencla stworzona została prototypowa implementacja składu danych zgodnego z podejściem SBA (Stack Based-Approach), roboczo nazwana LoXiM. LoXiM dysponuje wszystkimi niezbędnymi modułami takimi jak: menedżer transakcji czy menedżer dzienników umożliwiający odtworzenie danych po awarii bazy. Powstał także moduł wykonawczy języka zapytań SBQL (Stack Based Query Language) realizujący wszystkie podstawowe operacje zdefiniowane w tym języku.

Moim zadaniem było zaimplementowanie aktualizowalnych perspektyw obiektowych w systemie LoXiM.
Perspektywa w bazie danych jest przydatnym narzędziem znajdującym bardzo wiele zastosowań. Aktualizacja wirtualnych danych generowanych przez perspektywę może powodować komplikacje m.in. złamanie aktualizacyjnych intencji użytkownika. Problem znany jest od ponad trzydziestu lat. Realizacja tego tematu w sposób przezroczysty dla użytkownika (tak by nie musiał być świadomy, że korzysta z wirtualnych danych) i pozbawiony anomalii aktualizacyjnych jest wyzwaniem. Wyzwaniu temu udało się w pełni sprostać dopiero na gruncie podejścia stosowego SBA do języków zapytań [Subieta04]. W swojej pracy magisterskiej zaimplementowałem i przetestowałem główne założenia tego rozwiązania sformułowane w pracy [Kozankiewicz04]. Przy okazji musiałem zaproponować kilka własnych pomysłów na rozwiązanie kwestii w tej pracy niedoprecyzowanych. Zaprojektowałem i zaimplementowałem aktualizowalne wirtualne obiekty wskaźnikowe. Było to największym wyzwaniem. Ponadto opracowałem: fragment składni umożliwiający definiowanie perspektyw, sposób ich przechowywania w składzie danych, wewnętrzną reprezentację wirtualnych obiektów w module wykonawczym oraz mechanizm ich przetwarzania. Moja implementacja umożliwia definiowanie w systemie LoXiM dowolnie złożonych perspektyw obiektowych realizujących dowolne intencje aktualizacyjne. Możliwe jest także definiowanie perspektyw ze stanem.

Celem niniejszej pracy jest szczegółowe przedstawienie mojego rozwiązania. Składa się ona z pięciu rozdziałów. W pierwszym rozdziale przedstawię dokładnie pojęcie perspektywy oraz opowiem o dotychczasowych próbach rozwiązania problemu aktualizacji wirtualnych danych. W rozdziale drugim przybliżę na czym polega podejście stosowe do języków zapytań, opiszę składnię języka SBQL oraz pokażę w jaki sposób udało mi się zaimplementować w systemie LoXiM procedury będące istotnym elementem koncepcji perspektyw w SBA. W trzecim rozdziale przedstawię, w jaki sposób na bazie podejścia stosowego można zdefiniować w pełni przezroczyste aktualizowalne perspektywy obiektowe. W rozdziale czwartym opowiem szczegółowo o tym w jaki sposób zaimplementowałem perspektywy w systemie LoXiM. W piątym rozdziale wskażę możliwe rozszerzenia mojej dotychczasowej implementacji perspektyw oraz opiszę jakich problemów związanych z tematem perspektyw nie udało się rozwiązać.

Rozdział 1
Definicja perspektywy, problem z aktualizacją danych

Perspektywa w bazie danych to wirtualny obraz danych w niej przechowywanych. Dane widoczne przez perspektywę istnieją jedynie na poziomie koncepcyjnym. Za każdym razem, gdy użytkownik korzysta z takich danych, są one na nowo wyliczane na podstawie definicji perspektywy. W temacie baz danych spotykane jest także pojęcie zmaterializowanych perspektyw, będących de facto redundantnymi danymi, pewną kolekcją danych powiązanych z innymi danymi więzami integralności. Zmaterializowane perspektywy mają liczne zastosowania. Przyśpieszają ewaluację zapytań. Zwiększają bezpieczeństwo poprzez replikację danych. Związane są z nimi zupełnie inne problemy niż z perspektywami w pełni wirtualnymi m.in. jak w przypadku każdych redundantnych danych występuje problem zachowania spójności wersji.
Wspominam jednak o nich jedynie dla pełności obrazu. W niniejszej pracy zajmował się będę jedynie w pełni wirtualnymi perspektywami i ilekroć pojawi się pojęcie perspektywy będzie dotyczyło perspektywy wirtualnej.

Głównym wymaganiem jakie powinny spełniać perspektywy jest przezroczystość. Dane wirtualne generowane przez perspektywy powinny być traktowane na identycznych zasadach jak dane rzeczywiste. Nie powinno być żadnych składniowych ani semantycznych różnic w dostępie do takich danych w porównaniu do danych rzeczywistych. Użytkownik z nich korzystający w ogóle nie powinien mieć konieczności zastanawiania się, z jakiego typu danych w konkretnym momencie tak naprawdę korzysta. Każda taka różnica (składniowa czy semantyczna) musiałaby być opisana w podręczniku użytkownika języka. Podręcznik musiałby także opisywać i wyjaśniać wszystkie nałożone na wirtualne obiekty ograniczenia. Pełna przezroczystość perspektyw i brak jakichkolwiek ograniczeń ich stosowalności pozwalają zatem na znaczne zmniejszenie jego objętości. Łatwiejsze jest zatem opanowanie takiego języka. Łatwiej jest tworzyć nowe programy. Dodatkowo, programy już napisane, łatwiej jest dostosowywać do zmieniających się warunków.

Perspektywy pozwalają na dostarczenie użytkownikowi danych przyciętych na miarę jego potrzeb. W literaturze poświęconej tematowi, często podkreślane są następujące własności, czyniące perspektywy tak przydatnym narzędziem:

- Dane dostępne przez perspektywy są dostosowane do potrzeb użytkownika, który rzadko zainteresowany jest pełnym schematem danych zapisanych w bazie. Perspektywa ukazuje użytkownikowi jedynie potrzebne mu dane, w formie którą najłatwiej mu przyswoić. Zwiększa to efektywność pracy z takimi danymi i zmniejsza ryzyko popełnienia błędów.

- Perspektywy umożliwiają zapisanie dowolnie skomplikowanego systemu uprawnień dostępu do danych dla użytkowników. Zwiększone jest dzięki temu bezpieczeństwo. Dla każdego użytkownika można zdefiniować dedykowaną mu perspektywę ograniczającą dostęp jedynie do tych danych jakie są mu niezbędne.

- Dostęp do danych z różnych aplikacji przez dedykowane im perspektywy uniezależnia je od fizycznego modelu składowania tych danych. Schemat danych może dzięki temu być

bardziej elastyczny, można go zmieniać w razie potrzeby. Zmiana taka pociąga za sobą jedynie konieczność zmian w perspektywie. Nie trzeba przepisywać działających z danymi aplikacji.

- Perspektywy umożliwiają także integrację różnych systemów. W każdym z nich definiowany jest poprzez perspektywę taki sam interfejs do danych. Ich fizyczna reprezentacja w różnych systemach może się dowolnie różnić, mimo to użytkownik (lub korzystająca z tych danych aplikacja) widzi je poprzez perspektywę w jednym zunifikowanym schemacie. Perspektywy umożliwiają także nie utrudniające użytkownikom pracy z danymi, fizyczne ich rozproszenie na różnych serwerach. Użytkownik dzięki perspektywie widzi dane tak jakby wszystkie nadal były w jednym miejscu.

Wszystkie te właściwości czynią perspektywy niezastąpionym narzędziem w systemach bazodanowych. Wymaganie pełnej przezroczystości stawiane perspektywom oznacza w szczególności przezroczystość operacji aktualizacyjnych. Użytkownik powinien móc wykonywać operacje aktualizacyjne na wirtualnych danych w identyczny sposób jakby wykonywał je na danych rzeczywistych. To jednak powoduje dużo problemów i mimo iż temat znany jest co najmniej od publikacji Edgara Codda z 1974 roku [Codd74] i bardzo dużo pracy włożono w jego rozwiązanie, prawdopodobnie dopiero twórcom podejścia stosowego SBA udało się spełnić wymaganie pełnej przezroczystości perspektyw.

Z pośród komercyjnych rozwiązań jedynym dostatecznie zaawansowanym jest mechanizm „*instead of triggers*" zaimplementowany m.in. w systemach Oracle i MS SQL. W rozwiązaniu tym z perspektywą związany jest specjalny obiekt – wyzwalacz (trigger). Obiekt ten wykrywa sytuację, gdy aktualizacja ma być dokonana na wirtualnych danych i zamiast wykonywać standardową akcję, wykonuje procedurę zapisaną w jego definicji przez twórcę perspektywy.
Rozwiązanie oparte o podejście stosowe zaimplementowane przeze mnie w LoXiM jest pod pewnymi względami dość podobne do „*instead of triggers*", jednak dzięki mechanice stosowej jest pełniejsze i bardziej uniwersalne. W dalszej części tego rozdziału opiszę główne różnice pomiędzy tymi dwoma podejściami.

1.1 Opis problemu aktualizacji danych poprzez perspektywę

Jak już pisałem, perspektywę można zdefiniować jako wirtualny obraz pewnych danych. Można na nią więc spojrzeć jak na funkcję (odwzorowanie) biorącą jako swój argument dane rzeczywiste i zwracającą w wyniku dane wirtualne. Kiedy jednak użytkownik chce wykonać operację aktualizacyjną na danych wirtualnych (istniejących jedynie w jego wyobraźni), system bazy danych musi przepisać takie zapytanie na sekwencję operacji na rzeczywistych danych. Ta sekwencja operacji musi realizować intencję aktualizacyjną użytkownika. Po wykonaniu tych operacji przez system, użytkownik powinien mieć wrażenie poprawnego wykonania operacji aktualizacyjnej na danych wirtualnych.
By osiągnąć taki efekt, system bazy danych musi wiedzieć które dane i w jaki sposób powinien zmienić niejako odwracając sposób, w jaki wyliczył wpierw wirtualne dane. Aparat wykonawczy musi policzyć funkcję (odwzorowanie) odwrotną do funkcji jaką jest dana perspektywa.

Problem polega jednak na tym, że dla wielu perspektyw nie da się wyliczyć odwrotnego odwzorowania w sposób jednoznaczny. W ogólności może istnieć nieskończenie wiele sposobów na zrealizowanie żądanej przez użytkownika aktualizacji. Przykładem perspektyw, gdzie spotykany jest taki problem są perspektywy, w których użyty został operator agregujący.

Rozważmy perspektywę obliczającą w bazie danych w pewnej dużej sieci handlowej średnią marżę na wszystkich sprzedawanych aktualnie produktach. Wymaganie przezroczystości wymaga, by wirtualną daną jaką jest ta średnia, można było zmodyfikować tak samo jak można zmodyfikować dowolną zapisaną w bazie cenę towaru. Wyobraźmy sobie zatem zapytanie, w którym użytkownik chciałby zwiększyć średnie marże z obecnych 5% na 8%. Istnieje nieskończenie wiele sposobów podniesienia cen poszczególnych towarów realizujących postawiony cel. Można podnieść wszystkie ceny proporcjonalnie. Można podnieść ceny jedynie tych kilku produktów, które się najlepiej sprzedają. System bazy danych sam nie jest w stanie stwierdzić, który ze sposobów ma wybrać, tak by dokładnie spełnić intencje użytkownika.
Możliwa jest jeszcze gorsza sytuacja. Możliwa jest taka aktualizacja wirtualnych danych, której ze względu na więzy integralności nałożone w bazie, system w ogóle nie jest w stanie zrealizować.

Problem aktualizacji danych przez perspektywę zyskał miano bardzo trudnego. Staje się on jeszcze trudniejszy na polu obiektowych baz danych. Model danych obiektowych z klasami, dziedziczeniem i specyficznie rozumianą hermetyzacją jest znacznie bardziej skomplikowany niż model relacyjny. Wiele wysiłku ludzkość włożyła w prace pozwalające na rozstrzyganie, które perspektywy można aktualizować a których aktualizacja jest zabroniona, gdyż powoduje problemy takie jak opisałem powyżej.
W szczególności według jednej z zasad przyjętej niemalże jako aksjomat oczywisty i bezdyskusyjny nie można aktualizować wirtualnych danych, jeśli perspektywa nie zwraca referencji lecz konkretną wartość (tak jak w wypadku powyższego przykładu ze średnią). Podejście SBA nie zakłada takich ograniczeń. W ogóle zakłada zupełny brak ograniczeń co do aktualizowalności perspektyw. Jakiekolwiek tego typu zasady powodują nie spełnienie zasady pełnej przezroczystości, niepotrzebnie komplikują semantykę wirtualnych obiektów. Osoba definiująca perspektywę powinna mieć wszelkie środki, by móc wyrazić swoją dowolną intencję aktualizacyjną w taki sposób, by system bezbłędnie umiał każdą taką aktualizację przeprowadzić. W szczególności, w dalszej części pracy zaprezentuję perspektywę realizującą podstawienie na średnią i niepowodującą żadnych anomalii aktualizacyjnych obalając aksjomat niemożności wykonania takiej operacji aktualizacyjnej.

1.2 Przegląd dotychczasowych podejść do tematu aktualizacji przez perspektywę

W większości znanych podejść do kwestii aktualizacji wirtualnych danych zakłada się aktualizację poprzez efekty uboczne działania perspektywy. Zgodnie z tym założeniem perspektywa powinna zwracać pewnego rodzaju referencje do składowanych danych dla generowanych przez siebie obiektów / krotek wirtualnych. Referencją taką może być wskaźnik do obiektu, identyfikator krotki czy wartość klucza głównego w relacji. Aktualizacja wirtualnych danych odbywa się poprzez aktualizację wskazywanych referencją obiektów w składzie. Może to jednak prowadzić do różnych anomalii. Intencja jaką użytkownik przypisuje aktualizacji wirtualnych danych, może łatwo zostać złamana.

Rozważmy perspektywę *leczony_przez* z atrybutami *pacjent* i *lekarz*, gdzie atrybut *pacjent* zwraca referencję do nazwiska pacjenta, zaś atrybut *lekarz* referencję do nazwiska lekarza prowadzącego danego pacjenta. Ostatecznie perspektywa tworzy wirtualny słownik łączący nazwiska pacjentów z nazwiskami leczących ich lekarzy. Chcielibyśmy związać z tą perspektywą następującą intencję aktualizacyjną. By zmienić lekarza prowadzącego dla danego pacjenta chcielibyśmy napisać następujące polecenie SQL:

```
update leczony_przez set lekarz = „Iksiński”
where pacjent = „Fąfara”
```

Niestety efektem wykonania powyższego polecenia w bazie danych nie będzie zmienienie lekarza opiekującego się panem Fąfarą na innego a jedynie zmiana nazwiska dotychczasowego lekarza na Iksiński. Generalnie udało się osiągnąć wyznaczony cel: Fąfarę leczy obecnie Iksiński. Nie do końca o to jednak chodziło wykonującemu polecenie.

Jak widać aktualizacją poprzez efekty uboczne może prowadzić do tego typu anomalii. Wprowadzone więc zostały różnego rodzaju zasady ustalające, dla jakich perspektyw aktualizacja danych jest bezpieczna (nie powoduje anomalii) a dla jakich powinna być zabroniona. Na ten temat opublikowano bardzo wiele prac. Proponowane ograniczenia aktualizowalności perspektyw likwidują problem anomalii aktualizacyjnych. Są jednak sprzeczne z zasadą przezroczystości perspektyw i za mocną ograniczają ich stosowalność.

Proponowano na przykład podział perspektyw na zachowujące obiekty i generujące obiekty. Perspektywa zachowująca obiekty to taka, która zwraca jedynie referencje do obiektów istniejących w bazie. Jedynie poprzez perspektywy zachowujące obiekty możliwe było by zgodnie z tym podejściem aktualizowanie danych. Perspektywa generująca obiekty to między innymi taka, w której wirtualne atrybuty wyliczane są na podstawie rzeczywistych, lub w której definicji użyto operatora złączenia.

W standardzie SQL [sql92] zdefiniowano pojęcie perspektywy jako wirtualnej tabeli. Perspektywę definiuje się w następującej składni:

```
CREATE VIEW view_name [(column_name1, column_name2, …)]
AS select_query
[WITH CHECK OPTION]
```

Nazwy *column_name1*, *column_name2*, … są opcjonalne i specyfikują aliasy nazw kolumn zwróconych przez zapytanie *select_query*. Opcjonalna jest także klauzula WITH CHECK OPTION. Jeżeli jest użyta, to po wykonaniu aktualizacji system sprawdza czy rekord ten nadal będzie widoczny przez perspektywę. Wirtualne dane wyznaczane są pojedynczym zapytaniem.

Zgodnie ze standardem SQL 92 perspektywa musi spełniać następujące kryteria by być aktualizowana:

- W treści zapytania definiującego perspektywę nie mogą być użyte żadne operatory agregujące (średnia, suma, minimum czy maksimum) ani operatory działające na zbiorach (UNION, INTERSECT, EXCEPT). Nie mogą też w niej występować klauzule DISTINCT, GROUP BY ani HAVING. Ponadto w treści takiego zapytania nie może zostać użyty operator złączenia JOIN.

- Zwracane wirtualne kolumny muszą być wyliczane z istniejących i tylko z jednej źródłowej tabeli.

- Dodatkowo tabela do której odwołuje się klauzula FROM nie może być użyta w klauzuli FROM w podzapytaniu w klauzuli WHERE. Wszystkie kolumny z atrybutem NOT NULL z tabeli źródłowej muszą zostać wybrane w klauzuli SELECT.

Kryteria te zapewniają bezpieczeństwo aktualizowania danych przez taką perspektywę. Są jednak zbyt rygorystyczne i w wielu komercyjnych bazach danych implementujących ten standard zostały one rozluźnione. Dodatkowo jak już wspominałem w kliku komercyjnych systemach m.in. w Oracle i MS SQL zaimplementowano mechanizm „*instead of triggers*" pozwalający na definiowanie aktualizowalnych perspektyw nie spełniających powyższych kryteriów.

Koncepcja podejścia „*instead of triggers*" opiera się na skojarzeniu z definicją perspektywy specjalnego obiektu – wyzwalacza (trigger), który wykrywa sytuację, gdy na wirtualnym obiekcie wykonywana jest operacja aktualizacyjna. Razem z definicją perspektywy definiowane są specjalne procedury wykonywane zamiast poszczególnych instrukcji update, insert czy delete. Procedura wyzwalana przez trigger wykonywana jest w sposób niewidoczny dla użytkownika. Pod tym względem koncepcja „*instead of triggers*" jest bardzo podobna do zrealizowanej w systemie LoXiM koncepcji aktualizowalnych perspektyw. Jest jednak kilka poważnych różnic pomiędzy nimi.

Relacja pomiędzy wirtualnymi obiektami a obiektami na podstawie, których są one generowane, zapisywana jest w podejściu SBA za pomocą specjalnej procedury. Ciało tej procedury może być dowolnie skomplikowane. W przypadku koncepcji „*instead of triggers*" perspektywa definiowana jest poprzez pojedyncze zapytanie SQL. Ponieważ SQL nie jest kompletny algorytmicznie, istnieją perspektywy, których w takim podejściu nie da się zdefiniować.

W przypadku podejścia opartego o „*instead of triggers*", zapytanie definiujące perspektywę oraz procedury przeciążające generyczne operacje zapisywane są osobno. W SBA, procedury takie są zapisane wewnątrz definicji perspektywy co ułatwia zarządzanie perspektywą.

W SBA twórca procedury może przedefiniować sposób w jaki wirtualne obiekty są odczytywane poprzez zdefiniowanie procedury on_retrieve przeciążającej operację dereferencji. W podejściu „*instead of triggers*" można zdefiniować jedynie akcje dla operacji update, insert i delete. Przeciążanie operatora dereferencji może być bardzo przydatne. Dzięki niemu twórca perspektywy w SBA kontroluje nie tylko sposób w jaki wirtualne dane są aktualizowane ale także w jaki sposób są widziane przez użytkownika. Możliwe jest wykonanie dodatkowych operacji, np. zapisywanie w postaci dziennika informacji o tym, kto i kiedy wirtualne dane oglądał.

W podejściu „*instead of triggers*" wewnątrz definicji procedury przeciążającej instrukcję update dostępne są stara i nowa wartość przetwarzanej krotki. Można się do tych wartości odwołać dzięki specjalnym słowom kluczowym *old* i *new*. W operacji usuwania występuje jedynie wartość *old*, zaś w operacji wstawiania jedynie wartość *new*. By odpowiednio zainicjować te wartości, przy każdej z operacji aktualizacyjnej konieczne jest zmaterializowanie całej krotki. Taka pełna materializacja może powodować problemy z wydajnością. W SBA nie jest konieczne obliczanie całego wirtualnego obiektu, by dokonać na nim pewnej operacji. Obliczane jest tylko to co niezbędne.

Perspektywy zbudowane w oparciu o podejście SBA mogą być wykorzystane z obiektowym modelem danych, z relacyjnym modelem danych a także z danymi w postaci XML. Nie stanowi problemu zintegrowanie takich perspektyw z bytami takimi jak klasy czy dynamiczne role oraz z mechanizmami takimi jak dziedziczenie, polimorfizm czy hermetyzacja.

Rozdział 2
Podstawy SBA.

W niniejszym rozdziale przedstawię główne założenia podejścia stosowego SBA (Stack-Based Approach) do języków zapytań. Ograniczę się jedynie do kwestii istotnych dla przedstawienia tematu aktualizowalnych perspektyw. Bardziej szczegółowe informacje na temat SBA, różnych jego możliwości takich jak chociażby: kontrola typów, klasy i dziedziczenie czy dynamiczne role, można znaleźć w publikacjach: [Subieta04], [Humpich08]. W kolejnych podrozdziałach opowiem o modelu danych wykorzystywanym w SBA, jego głównych strukturach danych – stosach środowisk i wyników i głównych mechanizmach działania a także o języku zapytań SBQL (Stack-Based Query Language), jego składni i semantyce. W dalszej części wspomnę o sposobie realizacji procedur i funkcji w LoXiM i o tym jak na bazie tego wszystkiego można było podejść do tematu perspektyw.

2.1 Podejście stosowe, składnia i semantyka języka SBQL.

Głównym założeniem SBA jest traktowanie języka zapytań tak jak każdego innego języka programowania, w którym do regulowania zakresu widoczności obiektów w poszczególnych blokach programu wykorzystywany jest stos środowisk. Stos środowisk umożliwia dynamiczne wiązanie nazw obiektów, przesłanianie nazw oraz poprzez manipulację widocznością poszczególnych sekcji ukrywanie pewnych części środowiska.
Dzięki temu możliwe jest m. in. zaimplementowanie procedur i funkcji wraz z rekurencją, modelu obiektowego z klasami, dziedziczeniem i hermetyzacją a także dynamicznych ról. Na potrzeby SBA zdefiniowano założenia języka zapytań SBQL. SBQL w przeciwieństwie do powszechnie stosowanego SQL nie jest zbudowany w oparciu o zmatematyzowane twory w stylu algebry relacji. Dzięki pełnej ortogonalności operatorów SBQL pozwala na kompozycyjne tworzenie zapytań. W SBQL wszystko jest zapytaniem. Operatory łączą zapytania tworząc większe dowolnie skomplikowane. Piszący zapytanie może składać ze sobą dowolne operatory przestrzegając jedynie pewnych zasad co do zgodności typów jakiej wymagają niektóre z nich. W SBQL identycznie traktowane są dane trwałe jak i lokalne dla danego obliczenia zapytania.
Składnia pozbawiona jest lukru syntaktycznego zaś semantyka w pełni sformalizowana pozwoliła na łatwą implementację prototypu modułu wykonującego zapytania dla tego języka. Ortogonalność operatorów i kompozycyjność pozwalająca rozkładać zapytania na mniejsze części aż do poziomu zapytań atomowych sprawiają, iż możliwe jest optymalizowanie zapytań poprzez ich przepisywanie, wyłączanie pewnych części zapytań przed pewien operator czy odcinanie „martwych” podzapytań.

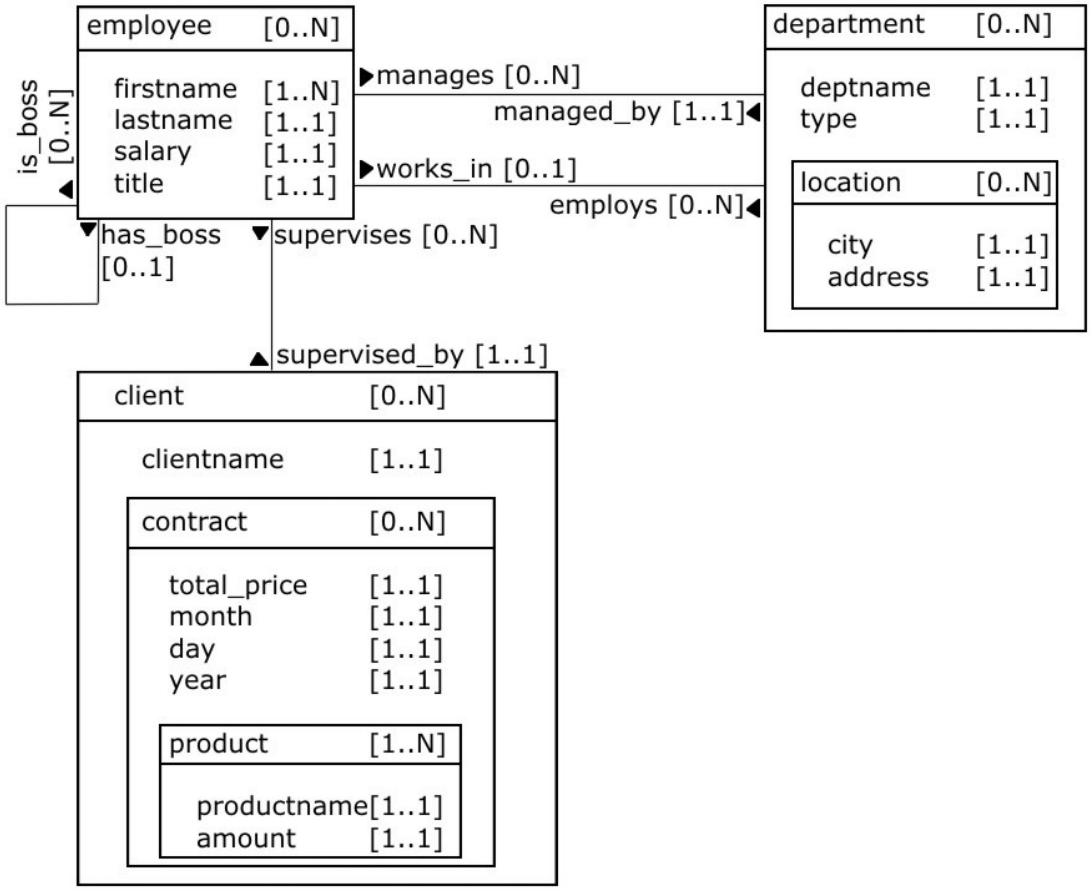
2.1.1. Model danych

Każdy obiekt w podejściu SBA posiada swój unikalny identyfikator nadawany automatycznie przez system, nazwę nadawaną przez projektanta bazy danych oraz wartość. Identyfikatory nie posiadają żadnej semantyki, nie można się do nich odwołać z poziomu języka zapytań. Do rozpoznawania obiektów na tym poziomie służą wyłącznie nazwy obiektów. Ze względu na wartości obiekty w SBA dzielimy na:

- atomowe, ich wartość jest atomowa (liczba lub napis)
- złożone, ich wartością jest zbiór podobiektów

- wskaźnikowe (pointery), ich wartością jest identyfikator wskazywanego przez taki pointer obiektu

Unikalność nazw obiektów na poszczególnych poziomach nie jest wymagana co umożliwia zamodelowanie kolekcji. Obiekt może mieć dowolnie wiele podobiektów na dowolnej liczbie poziomów zagnieżdżenia. Definicja budowy obiektów jest rekurencyjna, podobiekty zbudowane są w taki sam sposób na dowolnym poziomie hierarchii. W model danych SBA wyróżniony został zbiór obiektów korzeniowych czyli takich, które nie są podobiektami żadnych innych obiektów. Tylko obiekty korzeniowe widoczne są bezpośrednio dla piszącego zapytania jako punkty startowe. By dojść do podobiektów, użytkownik musi użyć jednego z operatorów niealgebraicznych.



Rysunek 2.1: Schemat danych użyty w przykładach.

Rysunek 2.1 przedstawia schemat bazy danych wykorzystywanej w niniejszej pracy w większości przykładów. W bazie tej zapisane są informacje o pracownikach firmy ACME, ich imiona, nazwiska, zarobki, stanowiska. Osobno zapisane są informacje o departamentach. Każdy departament jest jakiegoś typu (sprzedaż, wsparcie, finansowy etc.), ma swoją nazwę, zbiór lokalizacji a także swojego szefa i grupę zatrudnionych w tym dziale pracowników. Bezpośrednim przełożonym danego pracownika jest na ogół szef działu, w którym pracuje. Gdy

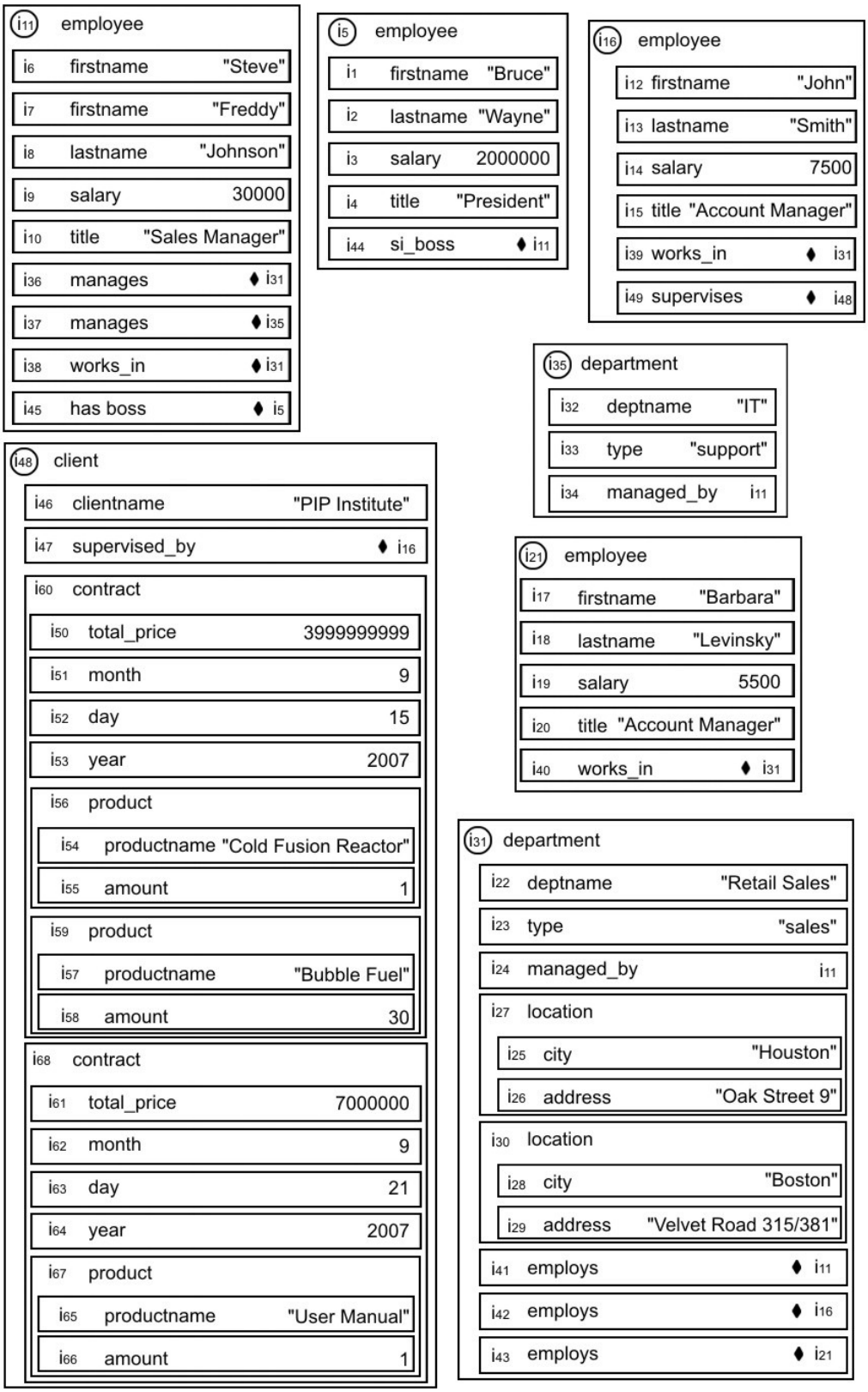
przełożonym jest inna osoba podana jest ona explicite w danych. W bazie zapisano także informacje o klientach firmy ACME. Każdy klient ma swoją nazwę oraz zbiór kontraktów zawartych z firmą ACME. Każdy kontrakt ma swoją kwotę, datę zawarcia oraz zestaw produktów. Klientami opiekuje się grupa pracowników. Każdy klient ma swojego account managera. Account manager opiekuje się wieloma firmami, w zależności od liczby zawartych kontraktów otrzymuje premię bądź nie.

Przykład zbioru danych w SBA spełniający powyższy schemat.

```
<i5, employee, {
      <i1, firstname, ”Bruce”>,
      <i2, lastname, ”Wayne”>,
      <i3, salary, 2000000>,
      <i4, title, ”President”>,
      <i44, is_boss, i11>
      }>,
<i11, employee, {
      <i6, firstname, ”Steve”>,
      <i7, firstname, ”Freddy”>,
      <i8, lastname, ”Johnson”>,
      <i9, salary, 30000>,
      <i10, title, ”Sales Manager”>,
      <i36, manages, i31>,
      <i37, manages, i35>,
      <i38, works_in, i31>,
      <i45, has_boss, i5>
      }>,
<i16, employee, {
      <i12, firstname, ”John”>,
      <i13, lastname, ”Smith”>,
      <i14, salary, 7500>,
      <i15, title, ”Account Manager”>,
      <i39, works_in, i31>,
      <i49, supervises, i48>
      }>,
<i21, employee, {
      <i17, firstname, ”Barbara”>,
      <i18, lastname, ”Levinsky”>,
      <i19, salary, 5500>,
      <i20, title, ”Account Manager”>,
      <i40, works_in, i31>
      }>,
```

```
<i31, department, {
      <i22, deptname, ”Retail sales”>,
      <i23, type, ”sales”>,
      <i24, managed_by, i11>,
      <i27, location, {
            <i25, city, ”Houston”>
            <i26, address, ”Oak Street 9”>
            }>
      <i30, location, {
            <i28, city, ”Boston”>
            <i29, address, ”Velvet Road 315/381”>
            }>,
      <i41, employs, i11>,
      <i42, employs, i16>,
      <i43, employs, i21>
      }>,
<i35, department, {
      <i32, deptname, ”IT”>,
      <i33, type, ”support”>,
      <i34, managed_by, i11>,
      }>,
<i48, client, {
      <i46, clientname, ”PIP Institute”>,
      <i47, supervised_by, i16>,
      <i60, contract, {
            <i50, total_price, 399999999>,
            <i51, month, 9>,
            <i52, day, 15>,
            <i53, year, 2007>,
            <i56, product, {
                  <i54, productname, ”Cold Fusion Reactor”>,
                  <i55, amount, 1>}>,
            <i59, product, {
                  <i57, productname, ”Bubble Fuel”>,
                  <i58, amount, 30>}>,}>
      <i68, contract, {
            <i61, total_price, 7000000>,
            <i62, month, 9>,
            <i63, day, 21>,
            <i64, year, 2007>,
            <i67, product, {
                  <i65, productname,
                        ”Cold Fusion Reactor User Manual”>,
                  <i66, amount, 1>
                  }>
            }>
      }>
```

Przykładowy stan bazy danych zilustrowany jest na rysunku 2.2.



Rysunek 2.2: Przykładowe obiekty zapisane w bazie danych.

2.1.2. Podstawy SBQL

Jak już pisałem w SBQL wszystko jest zapytaniem. Składnię języka można zdefiniować rekurencyjnie.

- Każdy literał jest atomowym zapytaniem. Literały to liczby całkowite, liczby zmiennoprzecinkowe oraz napisy. Np.:

```
8                       - zwróci liczbę 8
"John Smith"            - zwróci napis „John Smith"
3.1415                  - zwróci liczbę 3.1415
```

- Nazwa dowolnego obiektu jest także atomowym zapytaniem. Np:

```
employee                - zwróci wszystkie obiekty employee
department              - zwróci wszystkie obiekty department
```

- Jeżeli *q* jest zapytaniem a **σ** unarnym operatorem to **σ** (*q*) jest także zapytaniem. Np.:

```
count(employee)         - zwróci liczbę obiektów employee
avg(total_price)        - zwróci średnią z wartości obiektów total_price
```

- Jeżeli *p* i *q* są zapytaniami a **φ** jest operatorem binarnym to *p* **φ** *q* jest także zapytaniem. Operatory binarne dzielimy w SBQL na algebraiczne oraz niealgebraiczne Różnice między tymi dwoma typami operatorów binarnych omówię w dalszej części tego rozdziału.

```
2 + 2                   - zwróci liczbę 4
23 <= 56                - zwróci wartość logiczną prawda
employee.salary         - zwróci podobiekty salary wszystkich obiektów
                          employee
```

Wynikiem zapytania w SBQL może być wartość atomowa lub dowolnie złożona struktura. W implementacji egzekutora w LoXiM każde zapytanie zwraca w wyniku obiekt należący do jednej z podklas abstrakcyjnej klasy QueryResult.

QueryIntResult, QueryDoubleResult, QueryStringResult, QueryBoolResult – podklasy dla atomowych wartości zwracanych jako wynik zapytania odpowiednio dla liczb całkowitych, zmiennoprzecinkowych, napisów i dla wartości logicznych. Przechowują one odpowiednie proste typy zmiennych.

QueryReferenceResult – do takiej klasy należy wynik zapytania będący referencją do jakiegoś obiektu. Przechowuje on logiczny identyfikator wskazywanego obiektu (w LoXiM jest to obiekt klasy LogicalID) zapisywany w niniejszej pracy jako i z indeksem.

Przykłady atomowych wyników dla powyższych klas odpowiedzi na zapytania:

- 3
- 1.4142
- "*Account Manager*"
- true
- i23

Wynik z klasy QueryBinderResult to nazwany wynik zapytania. Nazwa może być nazwą obiektu w schemacie danych bądź może być nadana przy użyciu operatorów AS lub GROUP AS w danym zapytaniu. Binder składa się z dowolnego obiektu klasy QueryResult oraz etykietki przechowywanej w postaci napisu. Tego typu wynik zapytania zapisywany jest w niniejszej pracy w konwencji *nazwa*(QueryResult) np.: *a*(3), *b*("*jakiś napis*"), *tmp*(i45).

Wyniki z klasy QueryStructResult są odpowiednikami krotek czy rekordów. Struktury powstają w wyniku zastosowania złączeń i iloczynów kartezjańskich. Ich wartością jest wektor obiektów klasy QueryResult. Np.: struct{i1, i6, „Ala ma kota", i23, 5} to rekord składający się z trzech obiektów wskaźnikowych, napisu i liczby.

QueryBagResult i QuerySequenceResult to odpowiednio wyniki zapytań będące wielozbiorem i sekwencją obiektów klasy QueryResult. Sekwencja różni się od wielozbioru tym, iż w jej przypadku istotna jest kolejność jej elementów. Np.: bag{1, i13, "napis"} sequence{3, 5, 7, 9}.

Wyniki zapytań mogą być dowolnie złożone i dowolnie głęboko zagnieżdżone. Dodatkowo strukturę, wielozbiór i sekwencję zawierające pojedynczą atomową wartość utożsamiamy z tą wartością tzn. bag{4} jest tym samym co 4.

Przykłady złożonych wyników odpowiedzi na zapytania:

- struct{i13, i15}
- bag{struct{*a*(3), *a*(4), *b*(6)}, *op*(i13), *qwe*(bag{i14, i9})}

Aparat wykonawczy języka SBQL używa dwóch struktur danych dla obliczania wyników zapytań. Pierwszą z tych struktur jest stos wyników QRES (Query Result Stack). Przed rozpoczęciem obliczania odpowiedzi na zapytanie stos ten jest pusty. Obliczony wynik każdego zapytania jest odkładany na tym stosie. Gdy wykonywane jest obliczenie dla operatorów, najpierw obliczany jest rekurencyjnie wynik ich podzapytań. Wyniki trafiają na stos wyników skąd są zdejmowane dla obliczenia końcowego wyniku dla konkretnego operatora. Wynik całego obliczenia zapytania zdejmowany jest z tego stosu i przekazywany użytkownikowi. Po zakończeniu obliczeń stos wyników QRES ponownie staje się pusty.

2.1.3 Stos środowisk i funkcja nested

Stos środowisk ENVS (Environment Stack) jest drugim stosem jakiego moduł wykonawczy języka SBQL używa dla obliczania wyników zapytań. Stos ten umożliwia poprawne wiązanie nazw oraz regulację zakresu widoczności obiektów w trakcie przetwarzania poszczególnych części zapytania. Jest to odpowiednik stosu wywołań (Call Stack) znanego z tradycyjnych języków programowania. Dzięki temu programista piszący kod ma pewność iż stworzony przez niego program, procedura czy funkcja zadziała tak samo niezależnie od kontekstu w jakim jego kod zostanie użyty. Stos środowisk umożliwia przekazywanie parametrów do wywołań procedur

i funkcji, regulując zakres widoczności nazw, umożliwia realizację takich mechanizmów jak hermetyzacja, dziedziczenie, polimorfizm czy nadpisywanie metod. Możliwa jest implementacja wzajemnie wywołujących się procedur/funkcji a także rekurencji.

W tradycyjnych językach programowania obiekty żyją na stosie wywołań. Są tam dynamicznie wstawiane, zmieniane są ich wartości. W końcu są one usuwane ze stosu. Na stosie środowisk (ENVS) znajdują się jedynie referencie do obiektów ze składu danych. Składa się on z sekcji. W wielu sekcjach stosu mogą znajdować się referencje do tego samego obiektu przechowywanego w składzie danych. Każda sekcja zawiera zbiór binderów. Formalnie binder jest parą (*n*, *v*) gdzie *n* jest nazwa (etykietą) zaś *v* wartością. Będziemy go zapisywać $n(v)$. Wartość może być dowolnym obiektem klasy QueryResult. Może zatem być wartością atomową w szczególności referencją do obiektu w składzie danych lub może być dowolnie złożonym zagnieżdżeniem wielozbiorów, sekwencji, struktur, binderów i wartości atomowych. Wiązanie nazwy występującej w zapytaniu odbywa się poprzez wyszukiwanie sekcji stosu zawierającej bindery o takiej nazwie. Wyszukiwanie rozpoczyna się od najwyższej sekcji stosu i kontynuowane jest do znalezienia odpowiedniej sekcji. Jeżeli żadna z sekcji nie zawiera binderów o danej nazwie wynik wiązania jest pusty. Jeżeli w danej sekcji znajduje się wiele binderów o szukanej nazwie wynikiem wiązania będzie zbiór wartości wszystkich tych binderów. Przykładowo jeżeli wiązana była nazwa *n* i najwyższa sekcja stosu zawierająca bindery o takiej nazwie zawierała $n(i_1)$, $n(i_2)$ i $n(i_3)$ to w wyniku wiązania nazwy zwrócony zostanie zbiór $bag\{i_1, i_2, i_3\}$.

Na początku obliczenia odpowiedzi na zapytanie stos środowisk zawiera jedynie sekcję bazową. Sekcja bazowa zawiera bindery z referencjami do wszystkich obiektów korzeniowych znajdujących się aktualnie w bazie danych. W trakcie obliczania stos środowisk rośnie i maleje. Stos rośnie, gdy egzekutor przetwarza jeden z operatorów niealgebraicznych lub gdy jest wywoływana procedura lub funkcja. Po zakończeniu przetwarzania danego operatora bądź po zakończeniu wykonywania procedury odpowiednie sekcje są zdejmowane ze stosu. Dodatkowo wywołanie procedur / funkcji powoduje przesłonięcie pewnych sekcji stosu, tak by wewnątrz przetwarzanej procedury widoczne były jedynie te obiekty, których widoczności twórca procedury był świadomy. Innymi słowy by działanie procedur i funkcji nie było zależne od kontekstu w jakim zostały wywołane. Szczegółowiej mechanizm ten opiszę w następnym podrozdziale. Jeżeli w zapytaniu nie występowały operatory imperatywne takie jak create, insert czy delete po zakończeniu obliczeń stan stosu środowisk jest taki sam jak na początku.

Gdy przetwarzany jest jeden z operatorów niealgebraicznych, moduł wykonawczy oblicza zawartość nowej sekcji jaka ma być włożona na stos ENVS przy pomocy funkcji *nested*. Jak dokładnie wygląda przetwarzanie operatorów niealgebraicznych opiszę w kolejnym podrozdziale przy okazji omówienia funkcji *eval*. Na ten moment można myśleć o funkcji *nested* jako o funkcji obliczającej „wnętrze" danego obiektu.

Każda z podklas klasy QueryResult, czyli każdy typ wyniku zapytania w SBQL ma zdefiniowaną funkcję *nested*.

Dla obiektu klasy QueryReferenceResult, czyli wyniku będącego referencją do danych w składzie, funkcja *nested* obliczana jest w następujący sposób.

- Jeżeli referencja wskazuje na obiekt atomowy np.:

$<i_1$, firstname, "Bruce">

to wynik funkcji *nested* jest pusty (atomowy obiekt nie ma „wnętrza").

- Jeżeli wskazuje na obiekt złożony np.:

```
<i30, location, {
      <i28, city, ”Boston”>,
      <i29, address, ”Velvet Road 315/381”>
      }>
```

to funkcja *nested* zwraca zbiór tylu binderów, ile (bezpośrednich) podobiektów zawierał wskazywany tą referencję obiekt. Nazwy binderów odpowiadają nazwom poszczególnych podobiektów zaś wartościami są ich logiczne identyfikatory (referencje do nich). Dla powyższego przykładowego obiektu funkcja *nested* zwróciłaby zbiór:
`bag{`*`city`*`(i34), `*`address`*`(i29)}`.

- Jeżeli zaś referencja wskazuje na obiekt wskaźnikowy (pointer) np.:

```
<i38, works_in, i31>
```

wówczas sprawdzana jest nazwę, jaką ma wskazywany tym pointerem obiekt. W powyższym przykładzie wskazywany obiekt o identyfikatorze `i31` ma nazwę *departament*. Wynikiem funkcji *nested* jest binder ze znalezioną nazwą i logicznym identyfikatorem wskazywanego obiektu. W podanym przykładzie byłby to binder *`departament`*`(i31)`.

Generalnie wynik funkcji *nested* dla typu QueryReferenceResult może być bardziej skomplikowany gdy dołożymy do SBQL takie konstrukcje jak klasy, dziedziczenie czy hermetyzacja. Dla celów mojej pracy nie jest to jednak istotne i ograniczę się jedynie do najprostszej definicji tej funkcji.

Dla obiektów klasy QueryBinderResult funkcja *nested* po prostu zwraca niezmieniony binder to znaczy *nested*(*n*(*v*)) zwraca w wyniku *n*(*v*).

Dla obiektów klasy QueryStructResult, czyli dla struktur (krotek/rekordów) funkcja *nested* obliczana jest rekurencyjnie dla każdego ze składników struktury. Wynikiem *nested* jest suma mnogościowa wyników *nested* dla poszczególnych składników.

Dla wszystkich pozostałych typów wyników zapytań nested zwraca pusty wynik.

2.1.4 Semantyka SBQL, funkcja eval

Formalnie semantykę języka SBQL definiuje się za pomocą specjalnej rekurencyjnej funkcji *eval*. Funkcja ta może być zaimplementowana wprost tak jak ją tu zapiszę, bez żadnych większych zmian jako gotowy prototyp modułu wykonawczego zapytań dla SBQL. Z dokładnością do faktoryzacji kodu, wydzielenia dodatkowych podprocedur, funkcja ta została właśnie w ten sposób zaimplementowana w systemie LoXiM.

Dla literałów czyli liczb i napisów funkcja *eval* tworzy odpowiedni QueryResult i wkłada go na stos wyników QRES. Dla zapytań o nazwę, stos środowisk ENVS jest przeszukiwany przy pomocy funkcji *bindName*. Wynik działania tej funkcji wkładany jest na stos wyników QRES.

Wyniki zapytań mogą być łączone przy pomocy operatorów. Operatory jak już wspominałem w SBQL dzielimy na algebraiczne i niealgebraiczne. Różnicę stanowi to czy działanie danego operatora zmienia stan stosu środowisk ENVS czy nie. Operatory algebraiczne nie zmieniają stanu stosu środowisk.

Binarne operatory algebraiczne obliczane są w następujący sposób. Załóżmy że zapytanie *q* jest postaci *q1* **θ** *q2*. Obliczane jest podzapytanie *q1* poprzez rekurencyjnie wywołanie funkcji *eval* dla tego zapytania. Następnie w ten sam sposób obliczane jest podzapytanie *q2*. Wyniki tych podzapytań zdejmowane są z wierzchu stosu QRES. Wykonywana jest na nich akcja stosowna dla odpowiedniego operatora i wynik całego zapytania trafia na stos wyników QRES. Podzapytania *q1* i *q2* są obliczane niezależnie od siebie. Kolejność tych obliczeń jest nieistotna. Można wręcz spróbować zaimplementować funkcję eval dla takich operatorów tak, by oba podzapytania obliczane były równolegle. Z tego właśnie powodu operatory te nazwano algebraicznymi. Możliwe jest ich formalne zdefiniowanie na bazie algebry relacji.

W przypadku operatorów unarnych funkcja *eval* wygląda podobnie. Zapytanie *q* niech ma postać **σ**(*q1*). Poprzez rekurencyjne wywołanie funkcji *eval* obliczane jest podzapytania *q1*. Jego wynik zdejmowany jest ze stosu QRES. Następnie wykonywana jest akcja właściwa dla danego operatora a jej wynik wkładany jest na stos QRES.

Dla wymienionych do tej pory zapytań algorytm obliczania funkcji *eval* można, by zapisać następująco:

```
procedure eval(q: Query) {
      res: QueryResult;

      case q is literal : {        // q jest literałem
            res := new QueryResult(q);
            QRES.push(res);
      }

      case q is name :{            // q jest nazwą obiektu
            res := ENVS.bindName(q);
            QRES.push(res);
      }

      case q is σ(q1) or σ q1 : { // σ jest operatorem unarnym
            val: QueryResult;
            eval(q1);
            val := QRES.pop();
            res := unary_operation(σ, val);
            QRES.push(res);
      }

      case q is q1 θ q2 : {        // θ jest algebraicznym operatorem binarnym
            val1, val2: QueryResult;
            eval(q1);
            eval(q2);
            val2 := QRES.pop();
            val1 := QRES.pop();
```

```
            res := binary_algebraic_operation(θ, val1, val2);
            QRES.push(res);
        }


        /* dalsza część kodu procedury eval */

}
```

Wśród unarnych operatorów SBQL w LoXiM zaimplementowano m.in.: unarny minus (**-**), logiczne zaprzeczenie (**not**), operatory agregujące: średnia (**avg**), suma (**sum**), liczność (**count**), minimum (**min**), maksimum (**max**) oraz operator zwracający zbiór będący jego argumentem z pominięciem powtórzeń (**distinct**).

Szczególnie istotnym operatorem unarnym jest operator dereferencji (**deref**). Operator ten zastępuje w wyniku obliczonym z podzapytania wszystkie wystąpienia QueryReferenceResult, czyli referencji do danych w składzie ich rzeczywistymi wartościami. Stosowany explicite służy do wypisywania wartości obiektów. Często jest też wywoływany implicite. Wiele operatorów wymaga by ich argumenty były konkretnego typu, np. operatory arytmetyczne wymagają by ich argumenty były liczbami.
Gdy aparat wykonujący zapytanie trafia na taką sytuację, dokonuje automatycznej dereferencji. Automatyczna dereferencja występuje przy operacjach arytmetycznych, porównaniach, operatorach agregujących (**sum**, **avg**, **min**, **max**), sortowaniu i podstawianiu.
Piszący zapytanie może zapobiec automatycznej dereferencji stosując operator unarny **ref()**. Taki operator ustawia odpowiednio flagi w obiektach QueryReferenceResult. Flagi te są sprawdzane przed każdym obliczeniem operacji dereferencji.

Wśród algebraicznych operatorów binarnych SBQL w LoXiM znajdziemy m.in.: operatory arytmetyczne (**+ - * /**), operatory porównania (**= != < > <= >=**), operatory logiczne (**and or**), operatory działające na zbiorach: suma mnogościowa (**union**), przecięcie zbiorów (**itersect**) i różnica (**except**), a także iloczyn kartezjański (**,**).

Dodatkowo wśród algebraicznych operatorów binarnych występują dwa operatory nazywające wyniki zapytań (tworzące bindery) **as** i **group as**. By pokazać różnicę między nimi załóżmy że zapytanie *q* zwraca jako wynik QueryBagResult postaci `bag{i1, i2, i3}`.

Zapytanie *q* **as** *nazwa* zwróciłoby `bag{`*nazwa*`(i1), `*nazwa*`(i2), `*nazwa*`(i3)}`, zaś zapytanie *q* **group as** *nazwa* zwróciłoby: *nazwa*`(bag{i1, i2, i3})`.

Operator **group as** zwraca zawsze pojedynczy binder nazywając cały wynik zapytania wskazaną nazwą. Operator **as** nazywa osobno każdy z elementów zwróconych przez zapytanie *q*.

2.1.5 Operatory niealgebraiczne.

Operatory niealgebraiczne są siłą podejścia SBA. Ich obliczenie wymaga jednak bardziej skomplikowanego algorytmu. Jeżeli *q* jest zapytaniem postaci *q1* **δ** *q2* a **δ** jest operatorem niealgebraicznym, to egzekutor wylicza zapytanie *q2* w kontekście wyznaczonym przez zapytanie *q1*. Kolejność wykonywania tych podzapytań jest zatem bardzo istotna. Właśnie dlatego takie operatory nazywane są niealgebraicznymi. Złamana jest w ich przypadku podstawowa zasada dla operatorów algebraicznych: niezależnego wykonywania podzapytań *q1* i *q2*.

Operatory niealgebraiczne obliczane są w następujący sposób. Najpierw obliczany jest wynik dla zapytania *q1*. Wynikiem jest wielozbiór obiektów QueryResult. Następnie dla każdego obiektu QueryResult z tego zbioru wykonywane są następujące kroki. Obliczana jest funkcja nested na takim obiekcie. Wynik tej funkcji wstawiany jest na stos środowisk ENVS, tworząc jego nową najwyższą sekcję. W tym nowym kontekście obliczane jest zapytanie *q2*. Na otrzymanym wyniku zapytanie *q2* wykonywana jest funkcja *combine*, różna dla poszczególnych operatorów niealgebraicznych. Funkcja ta ma trzy argumenty: typ operatora, aktualnie przetwarzany element podzapytania *q1* i aktualny wynik podzapytania *q2*. Po jej obliczeniu, poprzednio włożona sekcja jest zdejmowana ze stosu. Następnie te same kroki wykonywane są dla następnego elementu wyniku zapytania *q1*. Wyniki funkcji *combine* dla poszczególnych elementów są zapamiętywane. Na koniec, po przetworzeniu wszystkich elementów wyniku zapytania *q1*, cały zbiór tych wyników poddawany jest funkcji *merge*(). Funkcja ta przetwarza wyniki w sposób charakterystyczny dla danego operatora algebraicznego. Wynik funkcji *merge* wkładany jest na stos wyników QRES jako wynik całego zapytania *q*.

Procedura *eval* dla operatorów niealgebraicznych wygląda więc następująco:

```
procedure eval(q: Query) {

      /* wcześniejszy fragment kodu */

      case q is q1 δ q2 : {       // δ jest operatorem niealgebraicznym
            val1, partial: QueryBagResult;
            eval(q1);
            val1 := QRES.pop();
            for each r in val1 do;
                  val2, val3, val4: QueryResult;
                  val2 := r.nested();
                  ENVS.push(val2);

                  eval(q2);        // obliczenie q2 w nowym kontekście
                  val3 := QRES.pop();
                  val4 := combine(δ, r, val3);
                  partial.add(val4);

                  ENVS.pop();
            end;

            res := merge(δ, partial);

            QRES.push(res);
      }

      /* … */
}
```

W systemie LoXiM zaimplementowano w postaci operatorów niealgebraicznych języka SBQL następujące operacje znane z innych języków zapytań:

- Selekcja, (*q1* **where** *q2*)

Dla każdego elementu wyniku zapytania *q1* obliczane jest zapytanie *q2*. Zapytanie *q2* powinno zwracać wartość logiczną. Elementy wyniku zapytania *q1*, dla których *q2* wylicza się jako prawda są zwracane jako wynik zapytania

- Nawigacja / Projekcja, (*q1* **.** *q2*)

Wynikiem zapytania jest suma wyników obliczenia zapytania *q2* dla wszystkich elementów wyniku zapytania *q1*. Jest to sposób na zapisanie wyrażeń ścieżkowych.

```
q1.q2.q3.q4 == ((q1.q2).q3).q4
```

Dzięki temu operatorowi możliwe jest przechodzenie do wnętrza obiektów korzeniowych i poruszanie się po nich. Np.:

```
employee.works_in.department.managed_by.employee
```

- Złączenie, (*q1* **join** *q2*)

Wynikiem zapytania jest zbiór par: dla każdego elementu wyniku zapytania *q1* dołączony jest wynik obliczonego w jego kontekście zapytania *q2*. Ostateczny wynik jest zbiorem tak otrzymanych struktur (krotek / rekordów).

- Kwantyfikatory, (*q1* **for all** *q2*), (*q1* **exists** *q2*)

Podobnie jak dla operatora selekcji zapytanie *q2* powinno się wyliczać jako wartość logiczna dla każdego z elementów wyniku zapytania *q1*. Wynikiem operatora for all jest prawda jeżeli wszystkie wyliczone wartości zapytania *q2* są prawdą. W przeciwnym przypadku wynikiem jest fałsz. Analogicznie dla operatora exists prawda jest wynikiem zapytania jeśli co najmniej jedno z obliczeń zapytania *q2* wyliczy się jako prawda.

- Sortowanie, (*q1* **order by** *q2*)

Wyniki zapytania *q2* są poddawane automatycznej dereferencji, następnie są porównywane ustaloną na wszystkich obiektach QueryResult relacją porządku.
Wynikiem zapytania jest sekwencja (nie wielozbiór) elementów zwróconych jako wynik zapytania *q1* ustawionych w kolejności zgodnej z porządkiem odpowiednich wyników zapytań *q2*.

Ponadto w SBQL występują rekurencyjne operatory niealgebraiczne **close by**, **leaves by** oraz ich wersje z zabezpieczeniem przed zapętleniem **close unique by** i **leaves unique by**.
Operatory te pozwalają na wyliczenie domknięcia przechodniego dla relacji w jakiej znajduje się grupa obiektów. Można to wykorzystać do obliczeń w stylu rachunku materiałowego BOM (Bill of Material). Szczegółowy opis działania oraz możliwe zastosowania tych operatorów, można znaleźć w [Subieta04].

Wszystkie operatory w języku SBQL mają także na poziomie gramatyki języka zdefiniowane moc oraz kierunek wiązania swoich argumentów. Zgodnie z tymi zasadami parser na podstawie

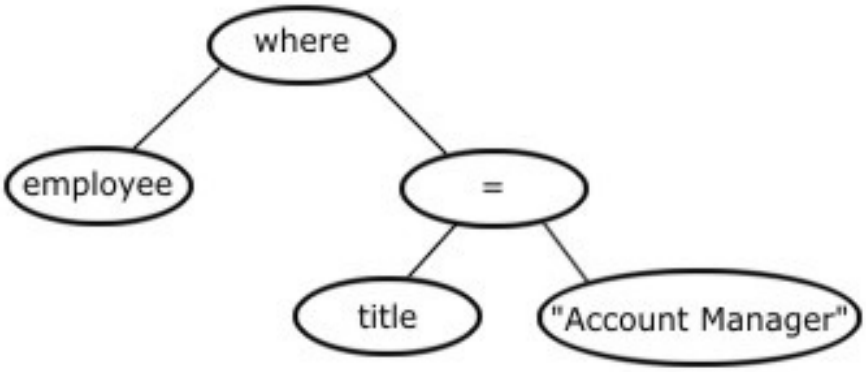
tekstu zapytania buduje odpowiadające mu drzewo jego wykonania. Dla przykładu zależnie od przyjętego kierunku wiązania dla operatora odejmowania (**-**) następujące zapytanie:

```
8 - 4 - 2
```

mogłoby zwrócić 2 albo 6 zależnie czy potraktowane by zostało jako zapytanie `(8 - 4) - 2` czy jako zapytanie `8 - (4 - 2)`. Moc wiązania dla poszczególnych operatorów ustala na nich pewien porządek. Przykładowo dla operatorów arytmetycznych operator mnożenia wiąże mocniej niż operator dodawania. Zapytanie `3 * 2 + 1` obliczone zostanie jako `(3 * 2) + 1` i zwróci 7. Hierarchia mocy wiązania operatorów oraz kierunek tego wiązania są definiowane dla gramatyki konkretnej implementacji języka SBQL. W LoXiM operatorem wiążącym najsilniej jest operator nawigacji czyli kropka. Dalej hierarchia w kierunku coraz słabiej wiążących wygląda następująco: operatory arytmetyczne, operatory porównywania, operatory logiczne, operatory **where**, **join** i **order by**, operatory działające na zbiorach, operatory **as** i **group as**, operatory **exists** i **for all** i operator iloczynu kartezjańskiego czyli przecinek. Piszący zapytanie może jednak sam zdecydować w jakiej kolejności połączone mają być poszczególne części jego zapytania poprzez wstawienie w odpowiednich miejscach zapytania nawiasów.

2.1.6 Przykładowe zapytania w języku SBQL

a) *employee* **where (***title* **=** *"Account Manager"***)**



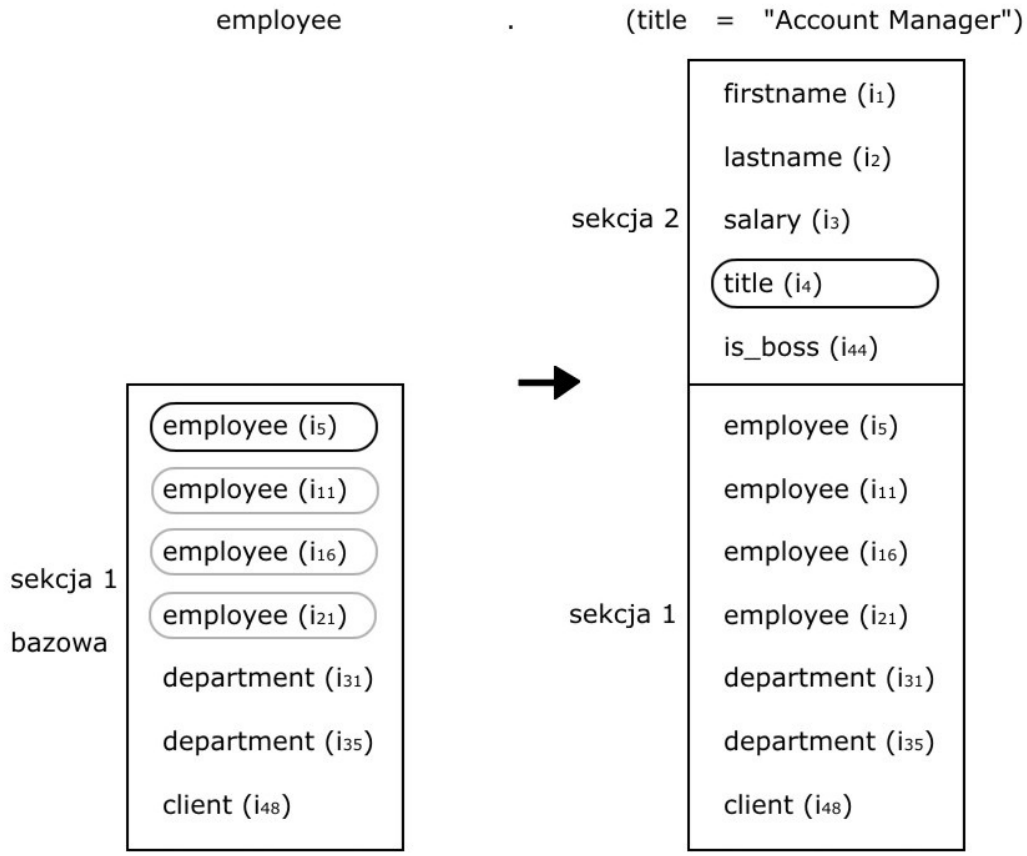
Rysunek 2.3: Drzewo wykonania dla przykładowego zapytania.

Korzeniem drzewa tego zapytania jest operator **where**. Jest to binarny operator niealgebraiczny. Zatem najpierw obliczane jest jego pierwsze podzapytanie. Zapytanie to (*employee*) jest zapytaniem o obiekty o danej nazwie. Na stosie środowisk szukana jest sekcja zawierająca bindery o nazwie employee. W tej fazie obliczenia odpowiedzi na zapytanie stos środowisk zawiera jedynie sekcję bazową a w niej bindery z referencjami do wszystkich obiektów zarejestrowanych w składzie jako korzeniowe. Nazwa *employee* związana będzie w tej sekcji i zwrócone zostaną referencje do wszystkich korzeniowych obiektów o nazwie employee.

Dla każdej z tych referencji obliczana jest funkcja *nested* a jej wynik (wnętrze danego obiektu employee) wkładany jest jako nowa sekcja na stos środowisk. W takim nowym kontekście obliczane jest podzapytanie *title = "Account Manager"*. Operator = jest operatorem algebraicznym. Podzapytania będące jego argumentami są obliczane niezależnie od

siebie. Nazwa *title* powinna zostać związana w nowo dokładanej sekcji stosu środowisk. Jeżeli jednak dany obiekt employee nie posiadał podobiektu *title* to w nowo dołożonej sekcji nie będzie takiego bindera. Kontynuując wyszukiwanie aparat wykonawczy poszukałby takiej nazwy w sekcji bazowej stosu środowisk. Dla przykładowego schematu danych zdefiniowanego w poprzednim rozdziale podzapytanie *title* oblicza się jako QueryReferenceResult czyli referencja. Po drugiej stronie operatora równości literał wylicza się jako QueryStringResult. Moduł wykonujący zapytanie, by móc porównać te dwa różne typy obiektów dokonuje automatycznej dereferencji i zamienia lewy argument z referencji na wskazywaną przez nią wartość. Ostatecznie wynikiem całego zapytania są logiczne identyfikatory obiektów employee reprezentujących pracowników zatrudnionych na stanowisku „*Account Manager*".
Dla przykładowego stanu bazy danych podanego powyżej odpowiedzią na to zapytanie będzie bag{i_{16}, i_{21}}

Na rysunku 2.4 przedstawiono stan stosu środowisk w trakcie przetwarzania podzapytania (*title = "Account Manager"*) dla pierwszej z referencji zwróconych jako wynik podzapytania *employee*.



Rysunek 2.4: Stan stosu środowisk w trakcie wykonywania przykładowego zapytania.

b) **(***employee* **where** *title* **=** *„Account Manager"***).***lastname*

Powyższe zapytanie zwraca referencje do obiektów *lastname* account managerów.
bag{i_{13}, i_{18}} .

By wyświetlić ich wartości użytkownik musi explicite użyć operatora dereferencji.

c) deref((*employee* **where** *title* **=** *„Account Manager”***).***lastname***)**

Dla tak sformułowanego zapytania wynikiem jest: bag{*”Smith”, ”Levinsky”*}.

d) *employee* **for all** *salary* **>** 10000

Zapytanie zwróci prawdę jeśli wszyscy pracownicy zarabiają więcej niż 10000, fałsz w przeciwnym przypadku.

2.1.7 Sterowanie programem w SBQL

SBQL tak jak klasyczny język programowania posiada konstrukcje pozwalające na sterowanie przebiegiem wykonania takie jak instrukcje warunkowe i pętle.

Instrukcję warunkową w LoXiM-owej implementacji SBQL zapisujemy:

if *query1* **then** *query2* **fi**

lub

if *query1* **then** *query2* **else** *query3* **fi**

Pierwsze wyliczane jest zapytanie *query1*. Musi ono zwrócić wartość logiczną. Jeżeli *query1* zwróci prawdę wykonywane jest zapytanie *query2*. W przeciwnym wypadku, jeżeli występuje konstrukcja **else** to wykonywane jest zapytanie *query3*.

W LoXiM instrukcja warunkowa jest funkcyjna tzn. zwraca jako swój wynik, wynik zapytania z odpowiedniej gałęzi.

Kolejne zapytania w ramach jednej sekcji programu rozdziela się w SBQL średnikami.
Jako wynik kilku zapytań rozdzielonych średnikami, zwracany jest wynik ostatniego z zapytań w danej sekcji.

W składni języka SBQL w LoXiM zaimplementowano też dwa rodzaje pętli.
Klasyczna pętla **while** ma składnię:

while *query1* **do** *query2* **od**

Zapytanie *query1* musi zwracać wartość logiczną. W każdym obrocie pętli jest ona obliczana i jeśli jest prawdą wykonywane jest zapytanie *query2*. Instrukcja **while** po wykonaniu nie zwraca żadnego wyniku (w przeciwieństwie do instrukcji **if**)

Drugim rodzajem pętli jest pętla **for each** ze składnią

for each *query1* **do** *query2* **od**

Pętla **for each** działa identycznie jak niealgebraiczny operator nawigacji (kropka). Jedyną różnicą jest to, iż w przeciwieństwie do niego nie zwraca na koniec żadnego wyniku. Wykonanie jest jednak takie samo. Dla każdego elementu wyniku zapytania *query1* obliczana jest funkcja *nested*, jej wynik jest wkładany na stos tworząc nowy kontekst, w którym obliczane jest w pętli zapytanie *query2*.

2.2 Operatory imperatywne w SBQL

By przedstawić temat aktualizowalnych perspektyw muszę jeszcze zaprezentować imperatywne operatory języka SBQL. Operatory te zmieniają stan składu danych. W tym miejscu przedstawię operatory: tworzenia nowych obiektów (**create**), usuwania obiektów ze składu (**delete**), wstawiania obiektów do innych obiektów (**insert :<**) i zmieniania wartości obiektu (**update :=**) tak jak zostały one zaimplementowane w LoXiM. Szczegółowy opis różnych operatorów imperatywnych można znaleźć w [Subieta04].

Tak samo jak dla wcześniej opisywanych operatorów argumentami dla operatorów imperatywnych są podzapytania. Operatory **create** i **delete** wyliczane są podobnie jak inne operatory unarne zaś **update** i **insert** jak algebraiczne operatory binarne. Operator **create** zwraca jako wynik referencje do nowo utworzonego obiektu. Pozostałe operatory nie zwracają nic. Dokładniej w LoXiM zwracają one obiekt klasy QueryNothingResult (kolejna podklasa QueryResult) czyli LoXiM-owy odpowiednik typu *void*. Poniżej szczegółowo opiszę te cztery instrukcje.

- **create** *query*

Podzapytanie *query* powinno się obliczyć jako binder lub worek binderów. Każdy z binderów jest traktowany przez aparat wykonawczy jak przepis na zbudowanie obiektu o nazwie takiej jak etykieta tego bindera. Wynik podzapytania *query* nie jest poddawany automatycznej dereferencji. W szczególności obliczony binder może zawierać referencję jako swoją wartość (QueryReferenceResult). Wówczas nowo tworzony obiekt będzie typu wskaźnikowego. Wartością bindera może być struktura binderów – powstanie wówczas obiekt złożony z podobiektami o nazwach i wartościach takich jak binderów w tej strukturze. Wreszcie binder może mieć wartość atomową – powstanie wówczas atomowy obiekt. Definicja ta jest rekurencyjna.

Przykłady:

```
create 1 as aa
```

Powyższa instrukcja tworzy atomowy obiekt o nazwie *aa* i wartości 1: <i69, aa, 1>

```
create (employee where title = "President") as szef
```

Powyższa instrukcja tworzy wskaźnikowy obiekt o nazwie *szef* wskazujący na obiekt *employee* o logicznym identyfikatorze i5: <i70, szef, i5>

```
create (14 as a, 465 as b, ("coś" as d, 34 as e) as c) as z
```

Powyższa instrukcja tworzy złożony obiekt o nazwie *z*, z atomowymi podobiektami *a* i *b* oraz złożonym podobiektem *c*.

```
<i76, z, {
      <i71, a, 14>,
      <i72, b, 465>,
      <i75, c, {
            <i73, d, ”coś”>,
            <i74, e, 34>
            }>
   }>
```

Nowo utworzone obiekty są rejestrowane jako korzeniowe w składzie danych. Stają się też widoczne w sekcji bazowej stosu środowisk ENVS.

- **delete** *query*

Obliczany jest wynik zapytania *query*. Musi on być typu referencyjnego lub być zbiorem obiektów takiego typu. Obiekty wskazywane przez referencje zwrócone jako wynik podzapytania są trwale usuwane ze składu danych. Jeżeli dany wskazywany obiekt był obiektem korzeniowym, jest on także wyrejestrowywane ze spisu korzeni. Referencje do usuniętych obiektów są usuwane ze stosu środowisk.

Przykłady:

delete *szef*

Powyższa Instrukcja usuwa ze składu danych (także ze zbioru korzeni) wszystkie obiekty o nazwie *szef*.

delete *z.c*

Powyższa instrukcja usuwa podobiekty o nazwie *c* wszystkich obiektów o nazwie *z*.

- *query1* **:=** *query2*

Zapytanie *query1* musi się wyliczyć do pojedynczej referencji wskazującej obiekt, którego wartość ma zostać nadpisana. Zapytanie *query2* może wyliczyć się do pojedynczej referencji wskazującej na jakiś obiekt. Aparat wykonawczy dokonuje wówczas automatycznej dereferencji otrzymując wartość tego obiektu. Obiekt wskazany zapytaniem *query1* ma zmienianą wartość na taką samą jak obiekt wskazany zapytaniem *query2*. Jeżeli w zapytaniu *query2* zostałby użyty w takim wypadku operator **ref()** zabraniający automatycznej dereferencji na swoim argumencie, obiekt wskazany zapytaniem *query1* stał by się w wyniku takiego zapytania obiektem wskaźnikowym i wskazywałby na obiekt zwrócony w zapytaniu *query2*.

Zapytanie *query2* może się także wyliczyć do wartości atomowej. Wówczas z taką wartością będzie nadpisany obiekt wskazany przez zapytanie *query1*. Ponadto *query2* może mieć postać taką jak opisana w przypadku operatora **create** (wielopoziomowego bindera).

Operator podstawienia nie zmienia zawartości stosu środowisk ENVS.

Przykłady:

```
aa := (employee where title = "President").lastname
```

Po wykonaniu powyższej instrukcji obiekt *aa* ma wartość "*Wayne*". Dokonana została automatyczna dereferencja.

```
aa := "Wayne"
```

Powyższa instrukcja także zmienia wartość obiektu *aa* ma wartość "*Wayne*".

```
aa := ref((employee where title = "President").lastname)
```

Po wykonaniu powyższej instrukcji obiekt *aa* staje się obiektem wskaźnikowym i wskazuje na obiekt przechowujący nazwisko prezydenta firmy ACME.

```
aa := (123 as e, „aa" as f) as aa
```

Po wykonaniu powyższej instrukcji obiekt *aa* staje się obiektem złożonym i ma dwa podobiekty o nazwach *e* i *f*.

Drobna uwaga do powyższej instrukcji. Nazwa obiektu w bazie danych raz nadana nie może już być zmieniona. Nawet jeżeli binder będący prawym argumentem takiego zapytania ma inną nazwę niż obiekt, którego wartość jest zmieniana, nie jest to brane pod uwagę.

- `query1 :< query2`

Zapytanie *query1* musi się wyliczyć do pojedynczej referencji wskazującej obiekt złożony, do którego ma być wstawiony nowy podobiekt (wstawianych podobiektów może być wiele naraz). Do obiektów atomowych i wskaźnikowych nie można wstawiać podobiektów.
W implementacji tego operatora w LoXiM zapytanie *query2* może wyliczyć się jako wielozbiór (w szczególności jednoelementowy) zawierający albo binderowe struktury (jak w wypadku **create**) albo referencje do obiektów korzeniowych.
W LoXiM niestety nie da się wstawić do obiektu fragmentu innego obiektu (ze względu na jak się okazało błędne założenia implementacyjne dokonane w module składu danych). Szerzej problem ten opiszę w rozdziale piątym.

Jeżeli *query2* wyliczy się jako referencja do obiektu korzeniowego, obiekt ten wstawiany jest jako nowy podobiekt obiektu wskazanego przez *query1*. Obiekt wskazany przez *query2* zostaje też wyrejestrowany z tablicy obiektów korzeniowych. Referencje do tego obiektu powinny być usunięte ze stosu środowisk ENVS.

Jeżeli *query2* wyliczy się jako binder, tworzony jest na jego podstawie nowy obiekt (dokładnie tak samo jak w przypadku operatora **create**) po czym wstawiany jest on jako nowy podobiekt obiektu wskazanego przez *query1* (różnica w stosunku do **create**, tam nowo utworzony obiekt był wstawiany do składu danych jako korzeń). Można więc myśleć o tym przypadku użycia operatora wstawiania, jakby był operatorem tworzenia podobiektów (podczas gdy **create** pozostaje operatorem tworzenia obiektów korzeniowych). Ani

w przypadku **create** ani w tym wypadku na otrzymanym binderze nie jest wykonywana automatyczna dereferencja. Przy takim użyciu operatora wstawiania nie zmienia się też zawartość stosu ENVS.

Przykłady:

```
z :< 124 as w
```

W wyniku tej instrukcji do wnętrza złożonego obiektu *z* dostawiony zostanie nowy podobiekt o nazwie *w* i wartości 124. Podobny efekt można osiągnąć w dwóch krokach w następujący instrukcją:

```
create 124 as w;
z :< w
```

Różnica jest taka, iż w drugim przypadku do wnętrza obiektu *z* zostaną wstawione wszystkie obiekty korzeniowe *w* znajdujące się w tym momencie w składzie danych a nie tylko ten jeden utworzony w pierwszej z instrukcji. Więc jeśli wcześniej w składzie danych były korzeniowe obiekty o takiej nazwie, efekt będzie inny niż zamierzony.

2.3 Procedury i funkcje w SBA oraz ich realizacja w LoXiM

Dzięki oparciu SBA o podobne mechanizmy jak w tradycyjnych językach programowania, nie ma żadnych problemów koncepcyjnych by dodać do SBQL procedury i funkcje (procedury funkcyjne) o podobnych możliwościach jak te występujące w językach programowania. Procedury w SBQL mogą mieć parametry, własne środowisko ze zmiennymi niewidocznymi na zewnątrz procedury. Procedury mogą (nie muszą) zwracać wynik, są wówczas de facto procedurami funkcyjnymi czy funkcjami. Procedury mogą mieć efekty uboczne, mogą wołać się nawzajem, w tym także rekurencyjnie. Najważniejszą własnością procedur w SBA jest to, iż mają one pełną moc algorytmiczną. Twórca procedury dysponuje wszelkimi konstrukcjami dostępnymi w SBQL.

Wynik zwracany przez procedury funkcyjne jest obiektem tej samej klasy co każda inna odpowiedź na zapytanie, można zatem łączyć wywołanie funkcji operatorami z innymi zapytaniami. SBA nie wprowadza żadnych ograniczeń co do sposobu w jaki mają być przekazywane parametry do procedury. Na jego bazie można zaprojektować dowolny sposób przekazywania zmiennych: przez wartość, przez referencję, przez nazwę (gdy parametr traktowany jest jak swego rodzaju makro i wyliczany jest na nowo za każdym razem gdy ma w ciele procedury być użyty), czy przez potrzebę (leniwa ewaluacja, parametr jest wyliczany tylko raz ale dopiero gdy jest potrzebny).

W implementacji procedur w LoXiM zdecydowałem się na przekazywanie parametrów do procedur ściśle przez wartość (czyli tak jak dzieje się to w języku C). Nie ma zatem żadnego semantycznego ani składniowego rozróżnienia na wołanie przez wartość i przez referencję. Jeżeli do wnętrza procedury przekazywana jest referencja to bez żadnych zmian jest ona dostępna w ciele procedury i możliwe jest trwałe zmienienie wskazywanego przez nią obiektu, możliwe są zatem efekty uboczne wykonania procedury. By wywołać procedurę z przekazaniem parametru przez wartość wystarczy na takim parametrze wykonać dereferencję. Parametr przekazany przez

wartość nie może być w ciele procedury zmieniony. Można taki parametr jedynie przesłonić wkładając na wierzch stosu środowisk element o takiej samej nazwie i innej wartości.

Stosowa semantyka umożliwia rekurencyjne wołanie procedur. Procedury rekurencyjne nie są w żaden specjalny sposób oznaczane w składni. Z ciała dowolnej procedury można wywołać dowolną procedurę zdefiniowaną aktualnie w składzie danych. Procedury SBQL mogą zwracać całe kolekcje tak samo jak pojedyncze wartości. Ciało procedury definiowane jest w tym samym języku zapytań. Parametry procedury mogą być dowolnie skomplikowanymi zapytaniami.

2.3.1 Schemat zapisu procedur w LoXiM

Procedury w LoXiM przechowywane są na tej samej zasadzie jak inne obiekty. Tworzy się je stosując następującą składnię:

```
create procedure nazwa_procedury ([nazwa_parametru]
[,nazwa_parametru]* ) { ciało_procedury }
```

Nazwa procedury będzie tą nazwą, jaką będzie służyć do jej wywoływania. Nazwy parametrów (rozdzielane przecinkami) będą widoczne wewnątrz jej środowiska. Ciało procedury może zawierać dowolny kod zgodny ze składnią SBQL. Nie ma rozróżnienia w składni między procedurami a procedurami funkcyjnymi (funkcjami). Jeżeli procedura ma być funkcyjna, czyli zwracać jakieś wartości, trzeba użyć wewnątrz jej ciała konstrukcję **return** *query*. Instrukcja **return** przerywa wykonywanie procedury (funkcji) i zwraca wartość obliczoną w zapytaniu *query*. Do przerwania działania procedury bez zwracania wartości służy instrukcja **break**. Jeżeli w trakcie wykonania procedury nie zostanie wykonana instrukcja **return** zwrócony (włożony na stos wyników QRES) zostanie obiekt klasy QueryNothingResult. Poszczególne instrukcje (zapytania) w ciele procedury rozdziela się standardowo za pomocą średników.

Po wykonaniu instrukcji **create procedure** w składzie danych dodany jest obiekt złożony o nazwie nazwa_procedury. Obiekt ten rejestrowany jest jako obiekt korzeniowy i staje się widoczny w bazowej sekcji stosu środowisk ENVS. Obiekt ten posiada podobiekt o nazwie *ProcBody* i tyle podobiektów o nazwie *Param* ile nazw parametrów podano w definicji. Nazwy *ProcBody* i *Param* nie są słowami zastrzeżonymi na poziomie składni języka SBQL. Są zdefiniowane jako stałe w kodzie modułu QueryExecutor i mogą zostać zmienione na etapie kompilowania kodu LoXiM-a.

Aparat wykonawczy przetwarzając instrukcję **create procedure** otrzymuje *ciało_procedury* jako sparsowane zapytanie w postaci drzewa. Podobiekt *ProcBody* będzie przechowywał to zapytanie w postaci napisu. Przed zapisaniem nowej procedury w składzie danych, drzewo zapytania będącego ciałem tej procedury jest deparsowane ponownie do postaci tekstu programu. Podobiekty *Param* przechowują nazwy parametrów także w postaci napisów, tak jak zostały podane w definicji procedury. W składzie danych nowo dodany obiekt o nazwie *nazwa_procedury* ma także zaznaczaną flagę, iż jest procedurą. Flaga jest wykorzystywana do sprawdzenia, czy dany obiekt jest rzeczywiście procedurą, czy też tylko mam podobną budowę. Obiekt stworzony zwykłą instrukcją **create** i mający budowę taką jak procedura tzn. posiadający podobiekty *ProcBody* i *Param* nie będzie traktowany jak procedura.

Przykłady:

a) `create procedure pi() {return 3.1415}`

Powyższa instrukcja tworzy w składzie danych obiekt korzeniowy:

```
<i82, pi, {
      <81, ProcBody, "return 3.1415">
      }>
```

b) create procedure *bigger*(*a*, *b*) **{**
if *a* **>=** *b* **then return** *a* **else return** *b* **fi }**

Powyższa instrukcja tworzy w składzie danych obiekt korzeniowy:

```
<i86, bigger, {
      <i83, ProcBody,
            "if (a >= b) then return a else return b fi">
      <i84, Param, „a">
      <i85, Param, „b">
      }>
```

Instrukcji **procedure** *nazwa*([*parametr*][,*parametr*]*) można użyć także w roli prawego argumentu operatorów wstawiania i podstawienia.

z **:< procedure** *p* **(***p1*, *p2***) { return** *p1* **+** *p2* **}**

Po wykonaniu powyższej instrukcji, do złożonego obiektu *z* zostaje dodany nowy podobiekt złożony o nazwie *p*, będący procedurą (w składzie zaznaczona odpowiednia flaga).

z **:= procedure** *p* **(***p1*, *p2***) { return** *p1* **+** *p2* **}**

Po wykonaniu powyższej instrukcji, obiekt *z* staje się procedurą. Ma w składzie zaznaczoną odpowiednią flagę. Jest od tego momentu jak każda procedura złożonym obiektem z podobiektami o nazwach *ProcBody* i *Param*. W tym miejscu podobna uwaga jak przy okazji omawiania operatora **:=** we wcześniejszym rozdziale. Nazwa obiektu nie ulega zmianie w wyniku operacji podstawienia, zatem nie ma znaczenia jaką nazwę nadamy procedurze powyższej instrukcji. W składzie danych nadal widoczna będzie pod nazwą *z*, jaką ten obiekt posiadał przed wykonaniem tej instrukcji.

Ponieważ obiekt będący procedurą dostępny jest tak samo jak każdy inny, można wykonać na nim różne operacje administracyjne. Można usunąć całą procedurę, bądź dowolny z jej podobiektów. Można do tego obiektu jak do każdego innego dokładać nowe elementy. Można wreszcie zmieniać wartości elementów wchodzących w skład procedury. Aparat wykonawczy w chwili wołania procedury sprawdza poprawność jej budowy.
Procedura przestanie działać, gdy nie będzie zawierać obiektu *ProcBody*, gdy takich obiektów będzie mieć więcej niż jeden lub gdy kod zapisany w tym obiekcie nie będzie możliwy do sparsowania. Procedura nie może zawierać obiektów o innych nazwach niż *ProcBody* i *Param*. Ponieważ w trakcie wywołania szukana jest procedura o liczbie parametrów takiej z jaką została wywołana, usunięcie bądź dodanie elementów *Param*, może sprawić, iż programy napisane z użyciem takiej procedury przestaną działać.

Przykłady administracyjnych działań na procedurach:

```
bigger :< "c" as Param;
bigger.ProcBody := "return if a>=b and a>=c then
      a else if b>=c then b else c fi fi"
```

Po wykonaniu dwóch powyższych instrukcji procedura *bigger* zyskuje nowy parametr i zwraca największy spośród trzech obiektów zamiast z dwóch.

Dalsze uwagi:

Kod procedury przechowywany w bazie może się różnić od kodu jaki zdefiniował tworzący procedurę, mimo to będzie ona działać dokładnie w taki sam sposób. Aparat wykonawczy zapisuje ten kod na podstawie zapytania w postaci drzewa dostarczonego przez parser zapytań. Zapytanie takie przy okazji parsowania mogło zostać poddane optymalizacjom. Drzewo jakie zwraca parser zbudowane jest zgodnie z zasadami kolejności wiązania operatorów uwzględniając nawiasy użyte w tekście zapytania. Drzewo to nie zawiera już informacji gdzie znajdowały się nawiasy. Podczas deparsowania konieczne jest zatem dodanie wokół każdego operatora nadmiarowych nawiasów zgodnie z jedną z reguł gramatyki języka SBQL: *query* :== (*query*)

Składnia deklarowania procedur staje się nieco bardziej skomplikowana po dodaniu do parsera zasad półmocnej kontroli typów. Szczegóły w pracy: [Humpich08].

Po dodaniu do modelu danych klas, procedury przechowywane wewnątrz obiektu klasy stają się metodami – inwariantami obiektów należących do danej klasy. Z dokładnością do zasad umieszczania binderów z referencjami do metod na stosie środowisk nie ma żadnych różnic pomiędzy nimi a zwykłymi procedurami. Są identycznie zbudowane i w ten sam sposób są wywoływane.

2.3.2 Wołanie procedur.

Składnia wołania procedury w LoXiM jest następująca:

```
nazwa_procedury ([query][, query]*)
```

Przykładowo, procedura *bigger* zdefiniowana powyżej (w wersji z dwoma parametrami) mogłaby zostać wywołana m.in. na takie sposoby:

```
bigger(3, 6)
```

Wynikiem byłaby liczba 6. Natomiast wynikiem następującego zapytania:

```
bigger((employee where lastname = „Wayne").salary,
      (employee where lastname = „Smith").salary)
```

byłaby referencja do podobiektu salary tego z dwóch pracowników, który więcej zarabia. W tym wypadku odpowiedzią byłaby referencja i_3.

Gdy wołana jest procedura, moduł wykonujący zapytanie szuka jej na stosie środowisk ENVS. Robi to za pomocą funkcji *bindProcedureName*. Funkcja ta działa podobnie do zwykłej funkcji *bindName* wyszukującej na stosie zwykłe obiekty. Wyszukiwana jest sekcja zwierająca binder o nazwie takiej jak nazwa wołanej procedury. Binder musi zawierać jako wartość referencję, wskazującą na obiekt będący procedurą (zaznaczona flaga) o liczbie parametrów zgodnej z liczbą parametrów z wywołania. Jeżeli znaleziony obiekt nie jest procedurą bądź ma inną liczbę parametrów, wyszukiwanie kontynuowane jest w dalszych sekcjach stosu. Gdy po przejrzeniu całego stosu środowisk od góry aż po sekcję bazową nie uda się znaleźć szukanej procedury zwracany jest pusty zbiór. Jeżeli w pewnej sekcji stosu znaleziono więcej niż jedną procedurę o danej nazwie i danej ilości parametrów zwracany jest stosowny błąd wykonania.

Gdy wyszukiwanie zakończy się powodzeniem, funkcja *bindProcedureName* zwraca referencję do obiektu przechowującego szukaną procedurę. Dodatkowym zadaniem funkcji bindProcedureName w porównaniu z *bindName* jest zapamiętanie numeru sekcji stosu środowisk, w której nazwę procedury udało się związać. Ta dodatkowa informacja służy aparatowi wykonawczemu w celu odpowiedniego zasłonięcia tych sekcji stosu, które w trakcie przetwarzania procedury powinny być niewidoczne. Dodatkowo egzekutor zanim zacznie wykonywać procedurę sprawdza poprawność budowy obiektu ją przechowującego i jeśli znajdzie nieprawidłowość kończy przetwarzanie z odpowiednim kodem błędu wykonania.

Z podobiektu *ProcBody* pobierany jest kod procedury, zaś z podobiektów *Param* nazwy parametrów z zachowaniem ich kolejności. Następnie wykonywane są (poprzez rekurencyjnie wywołania funkcji *eval*) zapytania będące aktualnymi parametrami procedury. Wyniki poszczególnych zapytań łączone są z odpowiednimi nazwami parametrów z definicji procedury tworząc bindery: *nazwa_parametru1*(*wynik_zapytania1*) ... . Zbiór tych binderów wkładany jest następnie jako najwyższa sekcja na stos środowisk. W ten sposób w SBA przekazywane są parametry do procedur – przez stos.

Kod procedury zostaje sparsowany i otrzymane drzewo zapytania gotowe jest już do rekurencyjnego wykonania. Przed wykonaniem trzeba jeszcze zadbać o odpowiednie zasłonięcie sekcji stosu środowisk, które nie powinny być widoczne wewnątrz przetwarzanej procedury. Zabieg ten ma na celu nie dopuszczenie do niezamierzonego przez twórcę procedury wiązania nazw.

Twórca procedury w jej kodzie musi mieć możliwość odwołania się do obiektów korzeniowych przechowywanych w składzie danych. Jeżeli procedura sama nie jest obiektem korzeniowym, jest przechowywana wewnątrz jakiegoś obiektu, to sąsiednie podobiekty także powinny być możliwe do wykorzystania w jej kodzie. Ponadto sama procedura musi być widoczna w trakcie jej wykonania po to, by mogła wywołać samą siebie, czyli by możliwa była rekurencja. Przesłaniając sekcje stosu konieczne jest pozostawienie widoczności sekcji bazowej oraz sekcji w której związana została wywoływana procedura (dlatego trzeba w funkcji *bindProcedureName* zapamiętać, która to sekcja). Dla procedury będącej korzeniem sekcje te będą jedną tą samą. Wszystkie pozostałe sekcje na czas wykonywania procedury zostają zasłonięte.

W LoXiM problem odpowiedniego przesłaniania sekcji rozwiązałem kojarząc z każdą sekcją zmienną *es_prior* przechowującą informację o poziomie zagłębienia wywołań procedur, na jakim dana sekcja powinna być widoczna. Pamiętany jest też obecny globalny licznik zagłębienia procedur. Funkcje *bindName* i *bindProcedureName* przeszukując stos środowisk omijają te sekcję, dla których *es_prior* nie jest równy temu globalnemu licznikowi. Przed wykonaniem kodu

procedury licznik ten zwiększany jest o 1. Zmienna *es_prior* dla sekcji bazowej oraz sekcji związania wykonywanej procedury ustawiana jest na równą globalnemu licznikowi. Ich stare wartości są jednak pamiętane, by móc je przywrócić po skończeniu wykonywania procedury. Wszelkie sekcje dokładane na stos środowisk od tego momentu (w tym sekcja z parametrami procedury) będą miały es_prior także równy aktualnej wartości globalnego licznika zagłębienia procedur. Efektem jest ukrycie wszystkich sekcji, które nie powinny być wewnątrz procedury widoczne.

Po tych wszystkich przygotowaniach aparat wykonawczy może wykonać kod procedury rekurencyjnie wołając procedurę eval dla jej drzewa zapytania.

Jeżeli powrót z procedury nastąpił w wyniku instrukcji **return**, zwrócona wartość pozostaje nie zmieniona na stosie wyników QRES. W przeciwnym wypadku na stos QRES wkładany jest obiekt QueryNothingResult.

Po obliczeniu procedury zdejmowane są ze stosu środowisk wszystkie włożone przez nią sekcje. Sekcji bazowej stosu i sekcji gdzie związana była nazwa procedury przywracane są poprzednie wartości zmiennej *es_prior*. Globalny licznik zagłębienia zmniejszany jest o 1.

Przykład działania procedur:

Do pustego składu danych wstawiamy za pomocą następujących instrukcji obiekt złożony *a* oraz obiekty atomowe *x, y, e*:

```
create (((12 as f, 9 as g) as d, 7 as e) as b, 10 as c) as a;
create 4 as x;
create 5 as y;
create 6 as e
```

Następnie wykonujemy instrukcję:

```
a.b :< procedure p(x) {return x + y + e}
```

W wyniku powyższych instrukcji w składzie danych pojawiają się następujący obiekty:

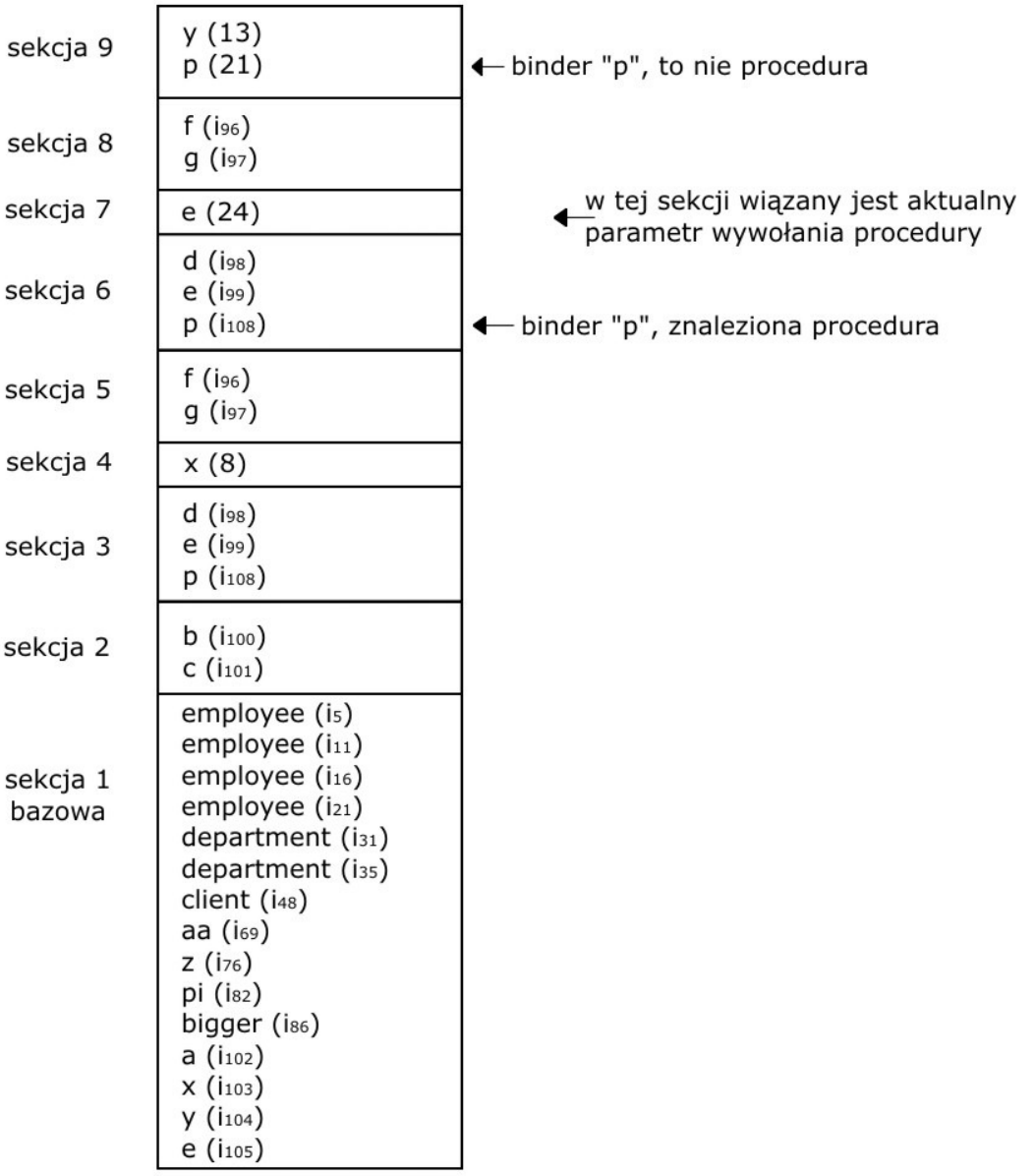
```
<i102, a, {
      <i100, b, {
            <i98, d, {
                  <i96, f, 12>,
                  <i97, g, 9>
                  }>,
            <i99, e, 7>,
            <i108, p, {
                  <i106, ProcBody, "return ((x + y) + e)">
                  <i107, Param, "x">
                  }>
            }>,
      <i101, c, 10>
      }>,
<i103, x, 4>,
```

```
<i104, y, 5>,
<i105, e, 6>
```

Przykładowe zapytanie:

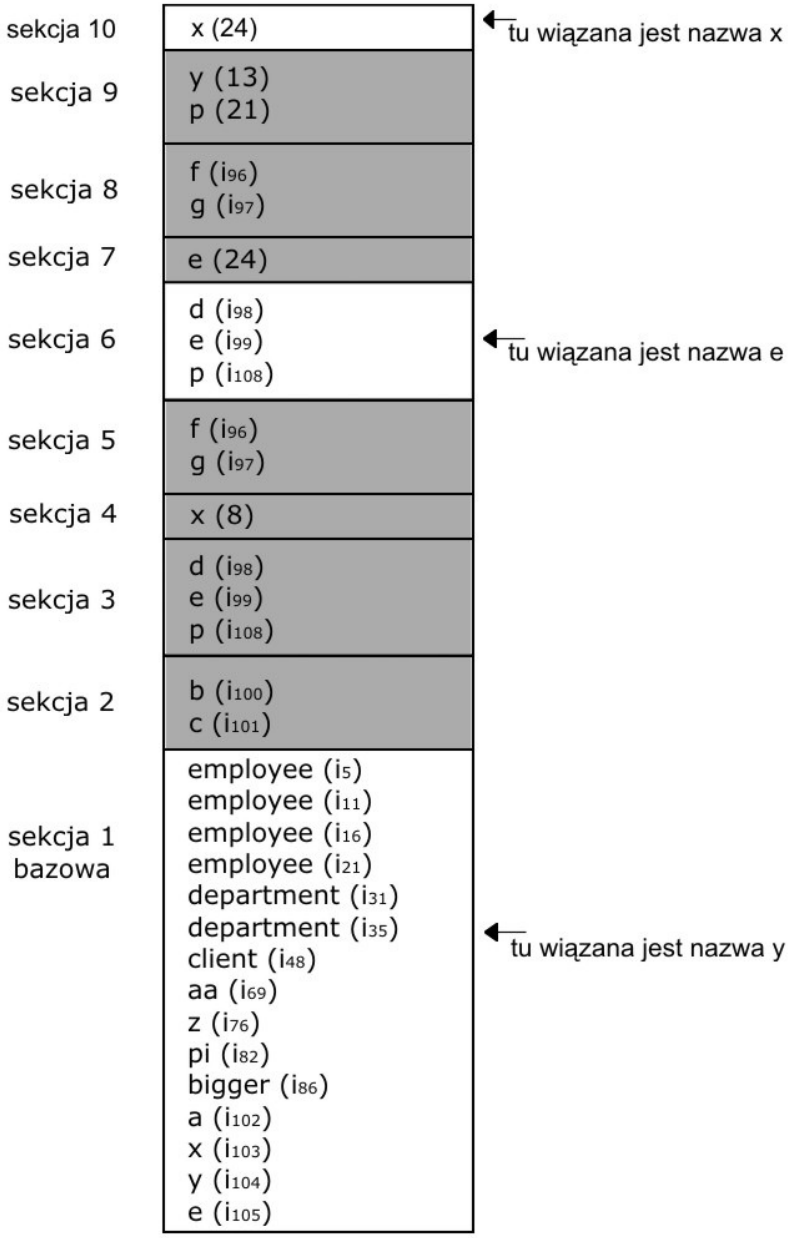
```
a.b.(8 as x).d.b.(24 as e).d.(13 as y, 21 as p).p(e);
```

Stan stosu środowisk w chwili obliczania podzapytania *p*(*e*) wygląda tak jak na rysunku 2.5



Rysunek 2.5: Stan stosu środowisk przed wykonaniem procedury *p*.

Procedura *p* będzie znaleziona w sekcji numer 6. Sekcja numer 9 zawiera binder o nazwie *p* ale nie zawiera on referencji do jednoargumentowej procedury tylko liczbę 21. Obliczany jest parametr z jakim wywołana będzie procedura, wykonywane jest zapytanie *e*. Nazwa *e* zostanie związana w sekcji stosu numer 7 z wartością 24. Przed wykonaniem procedury zostają zasłonięte wszystkie sekcje stosu środowisk po za sekcją bazową (numer 1) i sekcją, gdzie związano nazwę procedury (numer 6). Na wierzchołek wkładana jest nowa sekcja (numer 10) z parametrem wywołania procedury. Zgodnie z definicją procedury parametr ma nazwę x. W takim kontekście wykonane zostanie ciało procedury *p* czyli zapytanie **`return`** *x* **`+`** *y* **`+`** *e*. Stan stosu środowisk w trakcie wykonywania procedury zilustrowany jest rysunkiem 2.6.



Rysunek 2.6 Stan stosu środowisk w trakcie wykonywania procedury *p*.

Nazwy *x*, *y* i *e* w omawianym przykładzie wiązane są odpowiednio w sekcjach numer 10, 1 (bazowa) i 6. Po dokonaniu automatycznej dereferencji na obiektach i104 i i99 obliczony zostaje wynik procedury. Jest nim w tym wypadku liczba 36. (24 + 5 + 7).

2.3.3 Procedury funkcyjne jako perspektywy.

Procedury funkcyjne w SBQL mogą zwracać referencje do obiektów w składzie danych. Przyjrzyjmy się następującej procedurze:

```
create procedure account_manager() {
      return employee where title = "Account Manager"}
```

Zapytanie *account_manager*() zwraca w wyniku referencje do obiektów odpowiadających pracownikom zatrudnionym na stanowisku "*Account Manager*". Funkcja ta działa zatem jak perspektywa. Użytkownik przy jej pomocy widzi wirtualne obiekty o nazwie *account_manager* i może dalej wykorzystując zwrócone referencje obejrzeć ich podobiekty. Może też w szczególności użyć tych referencji w celu wykonania operacji zmieniających stan takich obiektów.

Przykładowe zapytanie:

```
(account_manager() as a).(a.salary := a.salary + 2000)
```

zwiększy wszystkim account managerom zarobki o 2000.
Zapytanie powyższe można też zapisać w następujący sposób:

```
for each account_manager() do salary := salary + 2000 od
```

co jest odpowiednikiem zapytania w SQL:

```
update account_manager set salary = salary + 2000
```

W większości znanych podejść do tematu perspektyw, traktowane są one właśnie jak procedury funkcyjne zaś operacje aktualizacyjne dokonywane są za pomocą zwróconych przez takie procedury referencji. Składnia takich „perspektyw" bywa różna. Różna bywa ich semantyka i związane z nią ograniczenia stosowania. Mimo wszystko sprowadzają się do tego samego co przedstawiono powyżej. Takie podejście powoduje liczne problemy aktualizacyjne co pokazałem w pierwszym rozdziale mojej pracy. Dla ich rozwiązania nakładane są różne ograniczenia na aktualizację obiektów poprzez takie perspektywy. Perspektywy takie przestają spełniać wymóg przezroczystości.

W SBA perspektywy są o wiele bardziej wyrafinowanym tworem niż procedury funkcyjne. Definiujący perspektywę może zapisać w jej definicji dowolną intencję aktualizacyjną. Procedury funkcyjne mogące zwracać referencje do obiektów, pozostawione są jednak do użytku bez żadnych ograniczeń. Twórcy SBA uznali, iż ograniczenia wprowadzane w innych podejściach, przybliżające do siebie pojęcia procedury funkcyjnej i perspektywy, nie prowadzą do żadnego sensownego rozwiązania problemu aktualizacji wirtualnych danych. Komplikują natomiast pojęcie funkcji / procedury funkcyjnej.

Użytkownicy SBQL muszą być świadomi, że procedury funkcyjne to nie perspektywy i aktualizacja poprzez zwrócone z procedury referencje może prowadzić do złamania intencji aktualizacyjnych. Jeżeli twórca procedury chce ją zabezpieczyć przed tym, by użytkownicy nie mogli w nieprawidłowy sposób użyć zwracanych z procedury referencji, powinien przed ich zwróceniem zastosować operator dereferencji. W przeciwnym wypadku bierze on na siebie pełną odpowiedzialność za to co może się stać w wyniku użycia takiej procedury funkcyjnej w roli perspektywy.

Rozdział 3
Koncepcja aktualizowanych perspektyw w SBA

Perspektywy w SBA to coś więcej niż funkcyjne procedury. We wcześniejszych rozdziałach pokazałem, że nie da się stworzyć w ramach aparatu wykonawczego mechanizmu automatycznie rozstrzygającego wszelkie problemy aktualizacyjne. Pokazałem do jakich problemów może prowadzić aktualizacja poprzez referencje zwrócone przez procedury funkcyjne. Wszystkie te pomysły a także cała teoria rozstrzygająca jakie perspektywy można aktualizować a jakich nie, zostały przez twórców podejścia SBA odrzucone. Odrzucona została tez silnie ograniczająca możliwości perspektyw zasada, zgodnie z którą perspektywa definiowana jest pojedynczym zapytaniem.

Perspektywy w SBA zbudowane są na następujących założeniach:

- Perspektywy muszą być w pełni przezroczyste dla użytkowników. Obiekty wirtualne definiowane przez perspektywy podlegają takim samym zasadom jak zwykłe rzeczywiste obiekty. Nie ma żadnych składniowych ani semantycznych różnic w użyciu jednych i drugich. Ideał do jakiego starałem się dążyć to sytuacja, w której użytkownik w żaden sposób nie jest wstanie zgadnąć czy obiekt, do którego ma dostęp jest rzeczywisty czy wirtualny. Z dokładnością do jednej sytuacji udało się to osiągnąć. Jedyny nierozwiązany problem został opisany w rozdziale piątym.

- Pełna informacja o tym, co powinien zrobić w danej sytuacji aparat wykonawczy z wirtualnym obiektem jest zawarta w definicji perspektywy. Procedury jakie twórca perspektywy definiuje dla wirtualnych obiektów przeciążają standardowe metody wykonywania imperatywnych operacji na obiektach. Gdy taka operacja wykonana ma być na wirtualnym obiekcie, zamiast niej wykonywany jest kod odpowiedniej procedury. Ponieważ procedury w SBQL mają pełną moc obliczeniową, twórca perspektywy może przy ich pomocy wyrazić dowolne intencje aktualizacyjne.

- Jeżeli twórca perspektywy nie zdefiniował dla niej jakiejś operacji, operacja ta jest dla generowanych przez nią wirtualnych obiektów zabroniona. Próba wykonania takiej operacji zakończy się stosownym błędem czasu wykonania, obliczenia zapytania zostanie przerwane. Po za tym nie ma żadnych wbudowanych kryteriów ograniczających operacje na wirtualnych obiektach. Na twórcy perspektywy spoczywa zadanie zdefiniowania jej w taki sposób, by dowolna operacja wykonana zgodnie z tą definicją była zgodna z intencją jaką jej przypisuje.

- Zgodnie z zasadami przezroczystości i relatywizmu wirtualne obiekty definiowane przez perspektywę mogą być (tak samo jak zwykłe obiekty) atomowe lub dowolnie złożone. W przypadku złożonych obiektów wirtualnych ich podobiekty definiowane są przez podperspektywy. Nie ma żadnych ograniczeń co do głębokości zagnieżdżania się podperspektyw. Podperspektywy zbudowane są w identyczny sposób jak perspektywy korzeniowe. Wreszcie by w pełni przykryć model danych rzeczywistych, można zdefiniować w SBA perspektywy wskaźnikowe (wirtualne pointery), po których można przechodzić tak jak przechodzi się po wskaźnikowych

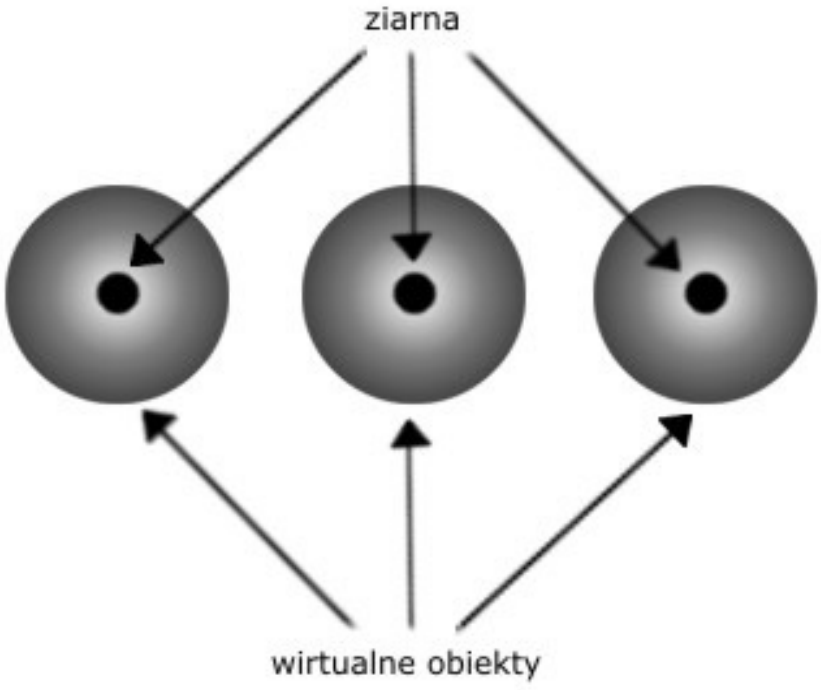
obiektach zapisanych w składzie danych. Wszędzie gdzie w zapytaniu można użyć nazw obiektów rzeczywistych mogą teraz także wystąpić nazwy obiektów wirtualnych.

- Implementacja perspektyw powinna uwzględniać możliwość rozszerzenia modelu o takie byty jak klasy i dynamiczne role. Musi być też zapewniony sposób na optymalizację zapytań wykorzystujących wirtualne dane. Dodanie perspektyw powinno zachować wszelkie stosowane dotąd metody optymalizacji takie jak indeksy, przepisywanie zapytań, wykorzystywanie statystyk zbieranych w składzie danych.

Dzięki tak postawionym założeniom, kosztem zrzucenia większej pracy na definiującego perspektywę, udało się stworzyć mechanizm pozwalający na definiowanie dowolnie skomplikowanych perspektyw, realizujących dowolne intencje twórcy i pozbawione problemów anomalii aktualizacyjnych.

3.1 Budowa perspektyw w SBA

W SBA definicja perspektywy może się składać z bardzo wielu elementów. Jedynym elementem wymaganym w definicji perspektywy jest procedura zwracająca ziarna wirtualnych obiektów. W każdej perspektywie musi być dokładnie jedna taka procedura. Wszelkie inne elementy definicji takie jak: procedury przeciążające poszczególne operacje, inne pomocnicze procedury, definicje podperspektyw, czy dodatkowe obiekty pozwalające na symulację stanu perspektywy są opcjonalne.



Rysunek 3.1: Wirtualne obiekty budowane są na bazie ziaren.

Procedura zwracająca ziarna ma za zadanie zapisanie relacji pomiędzy wirtualnymi obiektami a danymi na bazie których są one zbudowane. Ziarnem nazywany będzie obiekt, na bazie którego zbudowany zostanie obiekt wirtualny. Procedura zwracająca ziarna może być dowolnie skomplikowana. Wirtualnych obiektów generowanych przez nią jest dokładnie tyle, ile ziaren zwraca. W większości przypadków ziarnami będą bindery. Nie jest to jednak wymagane. Wymagane jest natomiast, by funkcja nested obliczana na ziarnach zwracała niepustą wartość.

Przykłady zapytań mogących być treścią procedury zwracającej ziarna:

```
employee where salary > 100000
```

```
(department where type = „sales”) as s
```

```
avg(client.contract.total_price) as p
```

Ziarno, na bazie którego zbudowany jest dany obiekt wirtualny, jest przekazywane jako parametr dla wszelkich operacji na tym obiekcie działających. Procedura zwracająca ziarna będzie wyróżniona w składni języka SBQL specjalnymi słowami kluczowymi *„virtual objects”* zamiast **procedure**. W skład definicji perspektywy może wchodzić wiele różnych procedur. Aparat wykonawczy, dzięki oznaczeniu tej jednej specjalnymi słowami kluczowymi wie, której z nich ma użyć do wyliczenia wirtualnych obiektów. Po za oznaczeniem wspomnianymi słowami kluczowymi procedura ta ma również (jak każda inna) swoją nazwę. Nazwa tej procedury jest jednocześnie nazwą pod jaką widoczne będą przez użytkowników wirtualne obiekty i powinna być inna niż nazwa samej perspektywy. Na potrzeby niniejszej pracy przyjmę następującą konwencję nazewniczą: Perspektywa będzie miała nazwę złożoną z nazwy definiowanych przez nią obiektów i sufiksu *Def*. Np.: perspektywa definiująca wirtualne obiekty o nazwie *bogatyPracownik* sama będzie się nazywać *bogatyPracownikDef*. Rozróżnienie takie jest konieczne w SBQL, gdyż zarówno perspektywa jako definicja jak i generowane przez nią wirtualne obiekty dostępne są na tym samym poziomie schematu danych. W SQL operacje administracyjne na definicji perspektywy mają inną składnię niż operacje na wirtualnych danych w związku z czym można używać tej samej nazwy w obu sytuacjach.

```
drop view bogatyPrac
```

```
delete from bogatyPrac
```

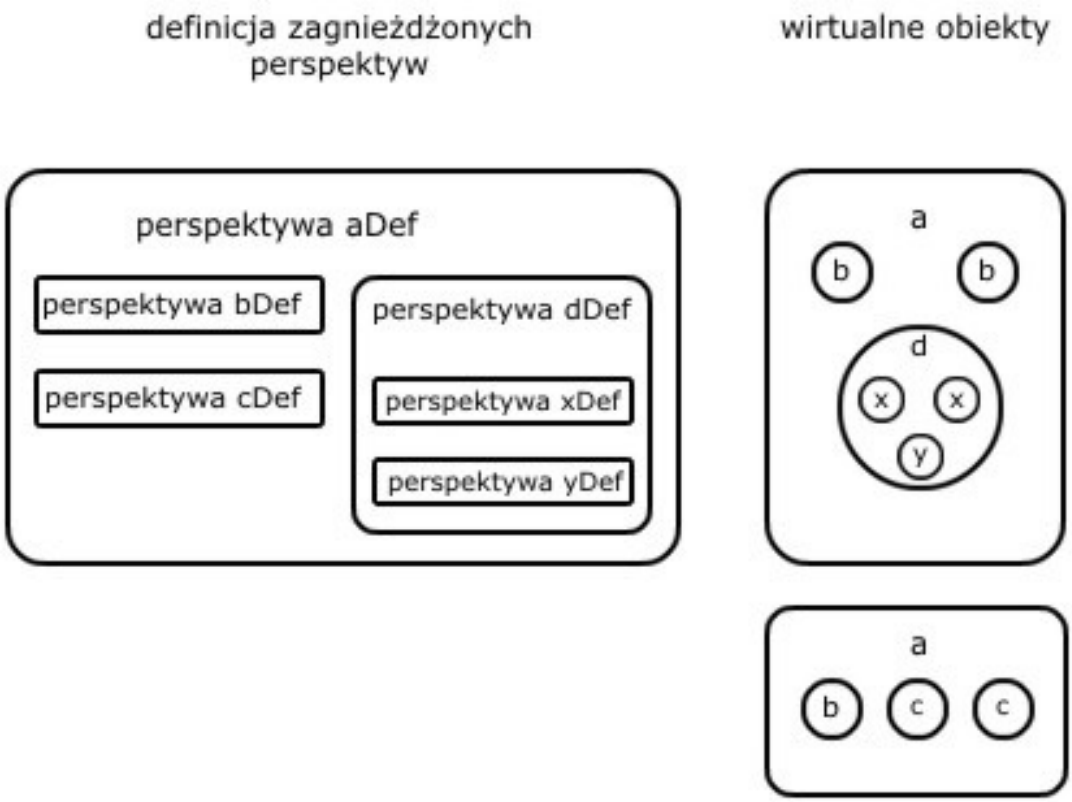
Pierwsze z powyższych poleceń usuwa perspektywę *bogatyPrac*, drugie zaś usuwa ze składu danych generowane przez tę perspektywę obiekty *bogatyPrac*. W SBQL przy przyjętej konwencji nazewniczej te polecenia wyglądałyby odpowiednio:

```
delete bogatyPracDef
```

```
delete bogatyPrac
```

Ziarna na bazie których zbudowane są wirtualne obiekty są niewidoczne dla użytkownika. Użytkownik widzi obiekty wirtualne w taki sam sposób w jaki widzi referencje wskazujące na rzeczywiste obiekty zapisane w składzie danych. Implementacja SBQL w systemie LoXiM dopuszcza możliwość formułowania przez użytkownika zapytań parametryzowanych wynikami zwróconymi na wcześniej zadane przez niego zapytania. Dla każdego wirtualnego obiektu jaki

oddawany będzie jako ostateczna odpowiedź na zapytanie użytkownikowi, aparat wykonawczy generował będzie sztuczny logiczny identyfikator w celu późniejszej identyfikacji. Dzięki temu będzie wiedział, który wirtualny identyfikator ma być podstawiony jako parametr w takim zapytaniu.



Rysunek 3.2: Zagnieżdżanie perspektyw i wirtualnych obiektów.

Wirtualny obiekt może być argumentem dowolnego z dotychczas zdefiniowanych operatorów. Dla wirtualnych obiektów konieczne jest przedefiniowanie operacji imperatywnych czyli tworzenia, usuwania, wstawiania i podstawiania. Przedefiniowania wymaga też operator dereferencji. Gdy któraś z tych operacji wykonywana będzie na obiekcie wirtualnym zamiast wykonać standardowy kod, aparat wykonawczy poszuka procedury przeciążającej dany operator w definicji perspektywy z jakiej dany wirtualny obiekt pochodzi. Pozostałe operatory przetwarzają wirtualne obiekty tak samo jak referencje do składowanych obiektów.

Procedury przeciążające wspomniane operatory są następną częścią definicji perspektywy. Muszą mieć predefiniowane nazwy. Dla operacji dereferencji (**deref**), podstawienia (**:=**), wstawiania (**:<**), usuwania (**delete**), i tworzenia (**create**) są to odpowiednio procedury: *on_retrieve*, *on_update*, *on_insert*, *on_delete* i *on_create*. W skład definicji perspektywy mogą jeszcze wejść procedury: *on_navigate*, *on_virtualize* i *on_store*, także przeciążające pewne operacje wykonywane przez aparat wykonawczy. Wszystkie one będą szczegółowo opisane w następnym rozdziale. Tych osiem nazw procedur „*on_*” jest słowami zastrzeżonymi w składni SBQL w systemie LoXiM i żadna procedura nie wchodząca w skład definicji perspektywy ani żaden inny obiekt nie może się tak nazywać. Występowanie każdej z tych procedur w definicji

danej perspektywy jest opcjonalne. Nie może natomiast żadna z nich wystąpić więcej niż jeden raz.

Po za wymienionymi procedurami perspektywy definiujące złożone obiekty wirtualne zawierają dowolną liczbę podperspektyw. Nie ma żadnych ograniczeń co do liczby poziomów zagnieżdżenia perspektyw w perspektywach. Obiekty generowane przez podperspektywę będą widoczne jako podobiekty (atrybuty) obiektów generowanych przez jej nadrzędną perspektywę.

Perspektywy mogą zawierać ponadto dodatkowe pomocnicze procedury oraz obiekty. Takie procedury i obiekty będą statycznymi elementami wspólnymi dla wszystkich wirtualnych obiektów generowanych przez daną perspektywę co pozwoli m.in.: na zdefiniowanie w tym podejściu perspektyw ze stanem oraz wyłączanie pewnych powtarzających się części procedur „*on_*" w postaci pomocniczych procedur czyli na refaktoryzację kodu definicji perspektywy.

3.2 Działanie perspektyw w SBA

Po dodaniu do składu danych perspektywy, definiowane przez nią wirtualne obiekty stają się widoczne dla następnych zapytań. Jeżeli perspektywa dodawana jest jako nowy korzeń do składu danych, nazwa definiowanych przez nią obiektów rejestrowana jest w strukturze podobnej do tablicy obiektów korzeniowych. Dzięki temu podobnie jak można szybko ustalić czy w składzie są korzenie o szukanej nazwie, można także szybko znaleźć informację czy jakaś perspektywa korzeniowa definiuje obiekty o danej nazwie a jeśli tak, to która. Implementacja perspektyw w LoXiM umożliwia ich dodawanie także jako podobiektów dowolnego obiektu złożonego.

Według założeń SBA na żadnym poziomie hierarchii obiektów nie mogą istnieć jednocześnie obiekty rzeczywiste i zdefiniowane przez jakąś perspektywę obiekty wirtualne o takiej samej nazwie. Jeżeli w chwili wstawiania na jakiś poziom perspektywy aparat wykonawczy wykrywa taką sytuację, próbuje zwirtualizować dotychczas występujące na tym poziomie obiekty rzeczywiste o spornej nazwie wykorzystując procedurę on_create z definicji nowododawanej perspektywy. Jeżeli się to nie udaje zwracany jest błąd wykonania. Perspektywa nie zostanie dodana.

Ilekroć wiązana na stosie środowisk nazwa będzie nazwą wirtualnych obiektów, wykonana zostanie procedura generująca ziarna pobrana z definicji odpowiedniej perspektywy. Na podstawie jej wyników stworzony zostanie zbiór wirtualnych obiektów. Wirtualne obiekty będą w trakcie przetwarzanie w aparacie wykonawczym reprezentowane przez obiekty klasy QueryVirtualResult (nowa podklasa klasy QueryResult). Dla użytkownika obiekty tej klasy będą wyglądać identycznie jak obiekty klasy QueryReferenceResult. Rozróżnienie będzie potrzebne za to samemu aparatowi wykonawczemu, gdy taki obiekt będzie poddany jednej ze wspomnianych operacji imperatywnych. Aparat wykonawczy, by wykonać operację imperatywną na wirtualnym obiekcie musi wiedzieć z jakiej perspektywy taki obiekt pochodzi oraz który to jest obiekt. QueryVirtualResult musi przechowywać zatem informację o tym z jakiego ziarna został zbudowany. Musi także przechować w postaci referencji (logicznego identyfikatora) wskazanie na perspektywę, która go wygenerowała i która zawiera definicje operacji na nim możliwych. W przypadku perspektyw niekorzeniowych wirtualny obiekt dla zachowania pełnej informacji musi przechować referencje nie tylko do perspektywy z której był wygenerowany ale także do wszystkich jej nadperspektyw. Podobnie przechować musi nie tylko ziarno z jakiego bezpośrednio powstał ale także wszystkie nadrzędne ziarna. Umożliwia to definiującemu perspektywy odwołanie się w definicji podperspektyw do ziaren z wyższych poziomów oraz do wszelkich zdefiniowanych na wyższych poziomach obiektów i procedur pomocniczych.

Tak jak już wspominałem, gdy aparat wykonawczy ma wykonać jedną z operacji imperatywnych na wirtualnym obiekcie, szuka on odpowiedniej procedury w definicji perspektywy, która dany obiekt wygenerowała. Gdy uda się znaleźć odpowiednią procedurę, wykonywana jest ona zamiast standardowej akcji dla danego operatora. Jak przy każdym wykonaniu procedury odpowiednio oznaczane są sekcje stosu środowisk ENVS tak by nie doszło do nieprawidłowego związania nazw. Na stos środowisk dokładane są sekcje zawierające parametry dla takiej procedury oraz wszystkie obiekty, które powinny być widoczne w trakcie jej wykonywania. Szczegółowo ten mechanizm będzie opisany w następnym rozdziale. Po zakończeniu obliczania procedury włożone uprzednio sekcje stosu środowisk są zdejmowane, sekcje ukryte na czas jej wykonania znowu stają się widoczne. Cały ten mechanizm jest ukryty przed użytkownikiem, który nie musi i wręcz nie powinien być świadomy, iż przetwarzany właśnie obiekt był obiektem wirtualnym. Gdy w definicji perspektywy generującej dany obiekt odpowiednia procedura nie zostanie odnaleziona, zapytanie zostanie przerwane. Zwrócony będzie błąd wykonania informujący iż dla danego obiektu nie zdefiniowano danej operacji.

Obiekty klasy QueryVirtualResult jak każdy inny rodzaj obiektów QueryResult mogą być argumentem operatorów niealgebraicznych. Konieczne było odpowiednie zdefiniowanie na nich funkcji *nested*. Dla danego obiektu wirtualnego, funkcja ta zwracać będzie jego wirtualne podobiekty. By wyliczyć podobiekty, aparat wykonawczy wykonuje procedury „*virtual objects”* zdefiniowane we wszystkich podperspektywach perspektywy jaka wygenerowała dany wirtualny obiekt. Dla wirtualnych obiektów wskaźnikowych w skład wyniku funkcji *nested* wejdzie z kolei wynik procedury *on_navigate* zdefiniowanej w generującej je perspektywie. Funkcja *nested* zdefiniowana jest na wirtualnych obiektach w taki sposób, aby w trakcie przetwarzania przez operatory niealgebraiczne obiekty te zachowywały się tak jak zwykłe referencje.

Rozdział 4
Implementacja aktualizowalnych perspektyw w LoXiM

We wcześniejszych rozdziałach opisałem czym są a w zasadzie czym powinny być perspektywy. Przedstawiłem problem aktualizowania danych przez perspektywę. Opisałem główne założenia podejścia SBA oraz podstawowe informacje o języku zapytań SBQL. Opowiedziałem o tym jak działają procedury zaimplementowane przeze mnie w systemie LoXiM oraz o tym jak się one mają do tematu perspektyw. W poprzednim rozdziale naszkicowałem koncepcję aktualizowalnych perspektyw zbudowanych na bazie podejścia SBA. W rozdziale niniejszym przedstawię szczegółowo jak tę koncepcję zrealizowałem w systemie LoXiM.

4.1 Definiowanie i składowanie perspektyw

Perspektywy w systemie LoXiM przechowywane są w składzie danych w postaci złożonych obiektów. Odpowiednio ustawiona flaga sygnalizuje, że dany obiekt jest perspektywą. Te same flagi służą też do oznaczanie w składzie procedur czy klas. Perspektywa tworzona jest instrukcją o następującej składni:

```
create view nazwa_perspektywy {

      virtual objects nazwa_wirtualnch_obiektów() {

            /* kod procedury zwracającej ziarna */
      }

      /* pozostałe składniki definicji perspektywy */
}
```

W skład definicji perspektywy może wchodzić wiele elementów. Jedynym wymaganym elementem każdej perspektywy jest procedura obliczająca ziarna, na bazie których budowane są wirtualne obiekty. Procedura ta wyróżniona jest specjalną składnią. Zamiast słowa kluczowego procedure oznaczona jest słowami kluczowymi „*virtual objects*”. Procedura ta musi być zdefiniowana jako pierwszy składnik każdej perspektywy. Może w szczególności być jej jedynym składnikiem. Nazwa tej procedury będzie jednocześnie nazwą, pod jaką widoczne będą wirtualne obiekty. Pozostałe składniki mogą występować w definicji perspektywy w dowolnej kolejności.

Twórca perspektywy może (ale nie musi) zdefiniować procedury przeciążające poszczególne operacje. Żadna z tych procedur nie może jednak wystąpić w definicji więcej niż jeden raz. W następnych podrozdziałach opiszę szczegółowo każdą z ośmiu procedur przeciążających operacje na wirtualnych obiektach. Nazwy tych procedur są słowami zastrzeżonymi na poziomie składni języka w parserze zapytań systemu LoXiM. Żadna procedura nie wchodząca w skład definicji perspektywy nie może się tak nazywać. Procedury *on_create*, *on_update*, *on_insert* i *on_virtualize* mają po jednym parametrze. Procedury *on_retrieve*, *on_delete*, *on_navigate* i *on_store* nie mają żadnego parametru.

Perspektywa może mieć dowolną liczbę podperspektyw. Podperspektywy definiowane są identyczną składnią i same mogą zawierać własne podperspektywy. Składnia definiowania perspektyw jest rekurencyjna. SBA nie zakłada żadnego ograniczenia co do maksymalnej głębokości zanurzenia perspektyw w perspektywach. Ponadto w skład definicji perspektywy mogą wejść dodatkowe pomocnicze procedury oraz dodatkowe obiekty pozwalające na zapisanie stanu perspektywy.

Przykład:

Perspektywa tworzona poniższym poleceniem będzie się nazywała *richEmpDef*. Generować będzie ona wirtualne obiekty o nazwie *richEmp*. Dla obiektów tych zdefiniowane zostały operacje ich tworzenia i usuwania. Wirtualne obiekty *richEmp* będą miały podobiekty o nazwie *sal*. Podobiekty te generowane będą przez podperspektywę o nazwie *salDef*. Dla wirtualnych podobiektów zdefiniowane zostaną operacje dereferencji i zmiany wartości.

```
create view richEmpDef {
      virtual objects richEmp {
            return (employee where salary > 10000) as re
      }

      procedure on_delete() {
            delete re
      }

      view salDef {
            virtual objects sal() {
                  return (re.salary) as s
            }

            procedure on_retrieve() {
                  return deref(s)
            }

            procedure on_update(new_sal) {
                  if new_sal > s then  s := new_sal fi
            }
      }

      procedure on_create(val) {
            if val.sal > 10000
                  then return (create (val.sal as salary) as
                  employee) as re
            fi
      }
}
```

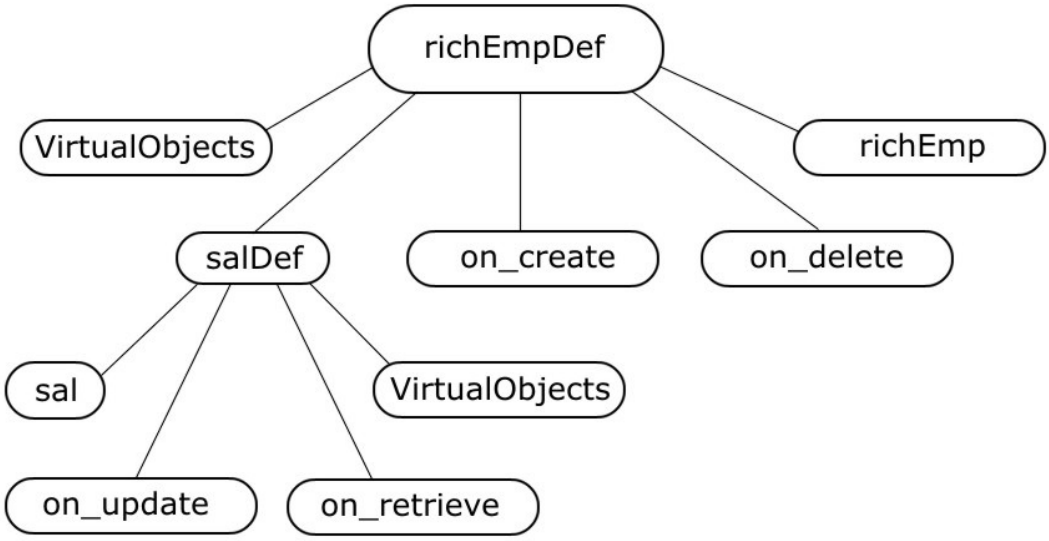
Po wykonaniu powyższego polecenia w składzie danych pojawi się złożony obiekt o następującej budowie:

```
<i128, richEmpDef, {
      <i111, VirtualObjects, ”richEmp”>
      <i113, richEmp, {
            <i112, ProcBody,
                  ”(return ((employee where (salary > 10000))
                  as re))”
                  >
            }>
      <i115, on_delete, {
            <i114, ProcBody, ”(delete re)”>
            }>
      <i118, on_create, {
            <i116, ProcBody,
                  ”if ((val.sal) > 10000) then return (create
                  (((val.sal) as salary) as employee)) as re fi”
                  >
            <i117, Param, ”val”>
            }>
      <i127, salDef, {
            <i119, VirtualObjects, ”sal”>
            <i121, sal, {
                  <i120, ProcBody, ”(return ((re.salary) as s))”>
                  }>
            <i123, on_retrieve, {
                  <i122, ProcBody, ”(return (deref(s)))”>
                  }>
            <i126, on_update, {
                  <i124, ProcBody,
                        ”if (new_sal > s) then  (s := new_sal) fi”
                        >
                  <i125, Param, ”new_sal”>
                  }>
            }>
      }>
```

Obiekty i128 (*richEmpDef*) i i127 (*salDef*) będą miały ustawioną flagę informującą, że są to perspektywy. Obiekty i113 (*richEmp*), i115 (*on_delete*), i118 (*on_create*), i121 (*sal*), i123 (*on_retrieve*) i i126 (*on_update*) oznaczone zostaną jako procedury. Dodatkowo w skład perspektywy *richEmpDef* i jej podperspektywy *salDef* wchodzą obiekty o nazwie *VirtualObjects*. W każdej perspektywie w podobiekcie o takiej nazwie przechowywana jest nazwa procedury zwracającej ziarna wirtualnych obiektów. Wewnątrz perspektywy może być bardzo wiele różnych procedur. Dlatego w jakiś sposób trzeba zapisać, która z nich jest tą, która ma być użyta do wyliczenia ziaren. Nazwa *VirtualObjects* podobnie jak *ProcBody* w budowie procedur jest zdefiniowana jako stała w module QueryExecutor systemu LoXiM.

Po wykonaniu powyższego polecenia obiekt i128 (*richEmpDef*) zostanie zarejestrowany w specjalnej strukturze w składzie danych jako obiekt korzeniowy i stanie się widoczny w sekcji bazowej stosu środowisk. Wspomniana struktura danych przechowuje informacje o wszystkich obiektach korzeniowych. Umożliwia szybkie znalezienie wszystkich korzeni o danej nazwie.

W trakcie implementowania perspektyw dodałem do modułu składu danych analogiczną strukturę przechowującą informację o wirtualnych korzeniach. Gdy wykonywane jest instrukcja tworząca nowy obiekt korzeniowy będący perspektywą, w strukturze tej zapisywana jest informacja o tym iż, dana perspektywa generuje obiekty o wskazanej nazwie. W razie potrzeby możliwe jest ustalenie czy jakaś perspektywa generuje obiekty o szukanej nazwie a jeśli tak to która. W powyższym przykładzie oprócz zarejestrowania jako korzeń o nazwie *richEmpDef* obiektu i128 zapisana zostanie też informacja, że obiekty o nazwie *richEmp* generowane są przez perspektywę o identyfikatorze i128. Wirtualne obiekty *richEmp* staną się widoczne w bazowej sekcji stosu środowisk tak samo jak rzeczywiste korzenie.



Rysunek 4.1: Przykładowy schemat perspektywy.

4.2 QueryVirtualResult, reprezentowanie wirtualnych obiektów

W SBQL zarówno obiekt definiujący perspektywę jak i generowane przez nią wirtualne obiekty dostępne są w taki sam sposób na tym samym poziomie składu danych. Nie ma specjalnej składni na potrzeby administracyjnego zarządzania perspektywą. Jest to jednocześnie powód, dla którego nazwa perspektywy musi być inna niż nazwa wirtualnych obiektów przez nią generowanych. Nazwy perspektyw w przykładach w niniejszej pracy będą nazwami ich wirtualnych obiektów z dodanym sufiksem *Def*.

Zapytanie: `richEmpDef` zwróci referencję i128 wskazującą na obiekt przechowujący definicję perspektywy. Referencję tę można wykorzystać np. w celu dokonania administracyjnych zmian w definicji perspektywy. Możliwe jest przy jej pomocy dodawanie nowych procedur i usuwanie wcześniej zdefiniowanych. W szczególności możliwe jest usunięcie całej definicji perspektywy ze składu danych. Szczegółowo wszelkie możliwe operacje administracyjne omówię w jednym z następnych podrozdziałów.

Zapytanie: `richEmp` zwróci natomiast zbiór wirtualnych obiektów. Wirtualne obiekty reprezentowane są w module wykonawczym jako obiekty klasy QueryVirtualResult. Obiekty

QueryVirtualResult zwane dalej wirtualnymi identyfikatorami przechowują pełną informację niezbędną do wykonywania wszelkich operacji na wirtualnym obiekcie.

By uniknąć niejasności zdefiniuję w tym miejscu co w dalszej części pracy będę nazywał wirtualnym obiektem a co wirtualnym identyfikatorem. Wirtualny obiekt jest pewnym abstrakcyjnym bytem istniejącym jedynie w wyobraźni użytkownika. Wirtualny identyfikator natomiast rzeczywiście istnieje choć nie jest zapisany w składzie danych. Reprezentuje on wygenerowane przez perspektywę wirtualne obiekty w module wykonawczym w trakcie obliczania odpowiedzi na zapytanie.

Z punktu widzenia użytkownika wirtualne identyfikatory mają identyczne właściwości jak obiekty klasy QueryReferenceResult czyli referencje wskazujące na obiekty przechowywane w składzie danych.

Obiekt klasy QueryVirtualResult posiada następujące atrybuty:

- wektor obiektów klasy QueryResult o nazwie *seeds*
- wektor obiektów klasy LogicalID o nazwie *view_defs*
- zmienną typu string o nazwie *vo_name*
- zmienną typu bool o nazwie *refed*

Każdy wirtualny identyfikator reprezentujący pewien wirtualny obiekt musi pamiętać która perspektywa go wygenerowała oraz z którego z jej ziaren powstał. W przypadku zagnieżdżonych perspektyw identyfikator wirtualnego podobiektu pamięta nie tylko perspektywę jaka go bezpośrednio wygenerowała ale także wszystkie jej nadperspektywy aż do perspektywy korzeniowej. Podobnie pamiętana jest pełna hierarchia ziaren. Ziarna przechowywane są w wektorze *seeds*. W wektorze *view_defs* przechowywane są logiczne identyfikatory perspektyw. Gdy na wirtualnym identyfikatorze wykonana ma być jedna z operacji aktualizacyjnych lub dereferencja, stosowna procedura przeciążająca daną operację wyszukiwana jest w definicji perspektywy, której logiczny identyfikator wirtualny identyfikator pamięta. Ziarna oraz wnętrza perspektyw (wynik funkcji nested na referencjach) umieszczane są na stosie środowisk ENVS na czas wykonywania tych procedur. Dzięki temu w definicji poszczególnych procedur można się odwoływać do wszystkich znajdujących się na różnych poziomach definicji perspektywy obiektów.

Na zmiennej *vo_name* wirtualny identyfikator pamięta nazwę pod jaką reprezentowany przez niego obiekt wirtualny widziany jest przez użytkownika. Wykorzystywane jest to między innymi w trakcie wykonywania unarnego operatora **nameof**, zwracającego nazwę obiektu będącego argumentem tego operatora.

Zmienna *refed* to flaga informująca aparat wykonawczy czy na danym obiekcie wirtualnym można wykonywać automatyczną dereferencję, czy też nie. Identyczną flagę posiadają obiekty klasy QueryReferenceResult. Ustawiona domyślnie na false oznacza, iż na danym obiekcie może być wykonywana dereferencja. Po wykonaniu na danym obiekcie operatora **ref()** wartość zmiennej refed zmieniana jest na true. Od tego momentu dereferencji na danym obiekcie już nie będzie można wykonać.

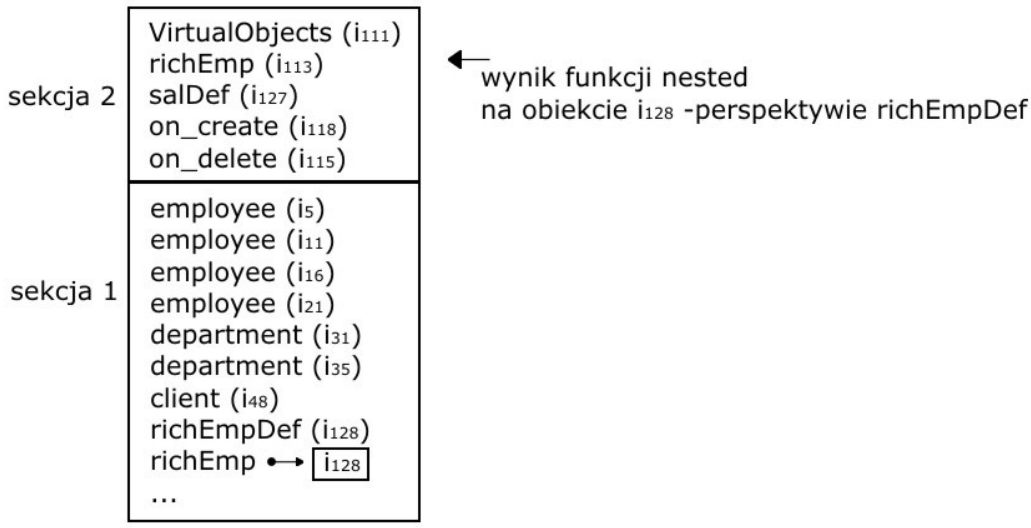
Odpowiedź na zapytanie `richEmp` obliczana jest w następujący sposób. Nazwa *richEmp* wyszukiwana jest na stosie środowisk ENVS. Stos zawiera w tym momencie tylko jedną sekcję, sekcję bazową. Aparat wykonawczy sprawdza czy istnieją zarejestrowane w składzie danych

korzenie o takiej nazwie. Ponadto sprawdza, czy istnieje w składzie danych perspektywa korzeniowa generująca wirtualne obiekty o takiej nazwie. W SBA przyjęta została następująca zasada. Na żadnym poziomie składu danych nie mogą równocześnie być widoczne rzeczywiste i wirtualne obiekty o takiej samej nazwie. Jeżeli w trakcie wyszukiwania na stosie środowisk nazwy *richEmp* znalezione zostały zarówno rzeczywiste jak i wirtualne korzenie o tej nazwie, zapytanie zostałoby przerwane wraz ze stosownym kodem błędu wykonania. Błędem zakończy się także sytuacja, gdy wirtualne obiekty o danej nazwie, są generowane przez więcej niż jedną perspektywę.

W przypadku omawianego zapytania aparat wykonawczy nie znajduje żadnych rzeczywistych korzeni o nazwie *richEmp*. Wirtualne obiekty o takiej nazwie generowane są przez perspektywę o identyfikatorze i_{128}. Aparat wykonawczy sięga do definicji tej perspektywy i szuka kodu procedury „*virtual objects"*. Przy okazji sprawdzana jest poprawność budowy perspektywy. Procedura „*virtual objects"* zwraca ziarna, będące podstawą budowy wirtualnych identyfikatorów. Jest ona wykonywana na podobnych zasadach jak każda zwykła procedura. Na czas jej wykonanie zasłaniane są wszystkie sekcje stosu środowisk za wyjątkiem sekcji bazowej. Na wierzch stosu środowisk wkładany jest jeszcze wynik funkcji nested zastosowanej na referencji do perspektywy. Dzięki temu w ciele procedury zwracającej ziarna widoczne jest całe wnętrze perspektywy i można skorzystać z dowolnych obiektów wchodzących w skład jej definicji. W szczególności można sięgnąć do pomocniczych obiektów składowanych wewnątrz perspektywy. Obiekty takie mogą być wykorzystane do utworzenia perspektyw ze stanem. Przed wykonaniem kodu procedury zwracającej ziarna w omawianym przypadku:

```
(return ((employee where (salary > 10000)) as re))
```

stos środowisk składa się z dwóch sekcji i wygląda tak jak przedstawiono na rysunku 4.2.

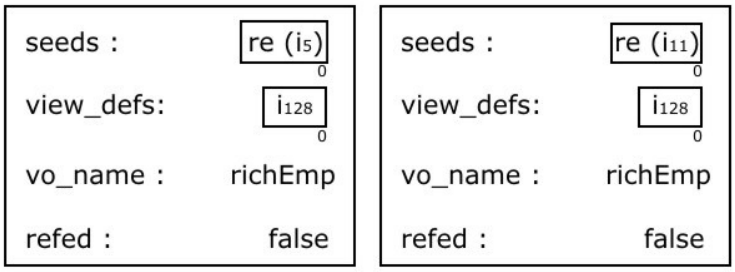


Rysunek 4.2: Stan stosu środowisk przed wykonaniem procedury zwracającej ziarna.

Wynikiem procedury będzie następujący zbiór binderów:

bag{*re*(i_5), *re*(i_{11})}

Każdy ze składników wynikowego zbioru będzie ziarnem jednego wirtualnego obiektu. Ziarna będą przechowane w wirtualnych identyfikatorach reprezentujących wirtualne obiekty jako jedyny element ich wektorów seeds. W wektorach view_defs w tych identyfikatorach zapisana zostanie informacja, z jakiej perspektywy pochodzą reprezentowane obiekty wirtualne, zaś zmienna vo_name przechowa nazwę wirtualnych obiektów. Pozostałe atrybuty nowopowstałych wirtualnych identyfikatorów zostaną zainicjowane wartościami domyślnymi. W omawianym przypadku stworzone zostaną dwa wirtualne identyfikatory. Ich wewnętrzną budowę przedstawia poniższa rysunek 4.3.



Rysunek 4.3: Budowa wirtualnych identyfikatorów.

Na potrzebę niniejszej pracy będę zapisywał wirtualne identyfikatory w konwencji

<VIRT, *vo_name*, seeds{…}, *view*_defs{…}>,

gdzie *vo_name* to nazwa, seeds{...} to wektor z ziarnami a view_*defs*{...} to wektor z identyfikatorami perspektywy (oraz jej nadperspektyw), która dany obiekt wirtualny generuje.

Wirtualne identyfikatory zwrócone w powyższym zapytaniu w niniejszej konwencji można zapisać następująco:

<VIRT, *richEmp*, seeds{*re*(i5)}, view_defs{i128}>

<VIRT, *richEmp*, seeds{*re*(i11)}, view_defs{i128}>

Wirtualne identyfikatory widziane są jak na powyższym rysunku jedynie w trakcie ewaluacji zapytania w module wykonawczym. Użytkownik powinien otrzymać w odpowiedzi obiekt wyglądający tak samo jak referencja czyli identyfikator rzeczywistego obiektu przechowywanego w składzie danych. W tym celu końcowy wynik zapytania przed oddaniem go użytkownikowi poddawany jest operacji dewirtualizacji. W wyniku tej operacji każdy obiekt klasy QueryVirtualResult zostaje zastąpiony w wyniku obiektem klasy QueryReferenceResult ze sztucznym identyfikatorem LogicalID. Identyfikator jest sztuczny, gdyż nie odpowiada żadnemu obiektowi przechowywanemu w składzie. Moduł wykonawczy dysponuje specjalną zastrzeżoną pulą sztucznych identyfikatorów do wykorzystania w takiej sytuacji. Wirtualne identyfikatory oddane w ten sposób użytkownikowi pamiętane są w module wykonawczym w specjalnym słowniku. Użytkownik w LoXiM może zadać zapytanie z parametrem. Parametrem zapytania może być w szczególności wynik jednego z poprzednich zapytań. Moduł wykonawczy otrzymawszy jako parametr referencję o identyfikatorze LogicalID należącym do tej dedykowanej puli szuka wirtualnego identyfikatora odpowiadającego temu LogicalID w słowniku i zastępuje nim tę referencję. To zastąpienie to operacja ponownej wirtualizacji obiektu. Dla każdego wirtualnego obiektu jaki ma być przekazany użytkownikowi obliczana jest

funkcja *toString*. Funkcja ta serializuje identyfikator wirtualny do postaci napisu. Napis ten składa się z nazwy wirtualnego obiektu, identyfikatora generującej go perspektywy oraz zserializowanego ziarna. Dla różnych identyfikatorów wirtualnych funkcja zwróci inny napis. Po obliczeniu tej funkcji aparat wykonawczy sprawdza czy identyczny napis nie znajduje się już w słowniku. Jeśli nie, to pobierany jest następny wolny identyfikator LogicalID. Referencja utworzona z takim identyfikatorem oddawana jest użytkownikowi. Para (uzyskany z funkcji *toString* napis oraz stworzona sztuczna referencja) zapisywana jest w słowniku. Przy ponownym zapytaniu o ten sam obiekt wirtualny użytkownik otrzyma identyczną sztuczną referencję. W omawianym zapytaniu ostateczny wynik zapytania `richEmp` jaki otrzymałby użytkownik wyglądałby tak:

bag{i4261412864, i4261412865}

Wirtualne identyfikatory, tak jak każdy inny rodzaj obliczonego wyniku, mogą być argumentami operatorów niealgebraicznych. Konieczne było odpowiednie zdefiniowanie funkcji nested na wirtualnych identyfikatorach, tak by zachowywały się one w trakcie obliczania niealgebraicznych operatorów tak samo jak referencje. Dla zwykłej referencji funkcja *nested* zwracała wnętrze wskazywanego nią obiektu. Jeżeli wskazywany obiekt był atomowy (nie miał w związku z tym wnętrza) wynikiem funkcji nested był pusty zbiór.

Nested na identyfikatorach wirtualnych reprezentujących wirtualne obiekty atomowe także zwraca w wyniku pusty zbiór. Dla identyfikatorów reprezentujących złożone wirtualne obiekty funkcja *nested* powinna zwracać zbiór identyfikatorów reprezentujących wirtualne podobiekty.

Przyjrzyjmy się zapytaniu:

```
richEmp.sal
```

Zapytanie `richEmp` zwraca dwa wirtualne identyfikatory. Zgodnie z zasadami przetwarzania operatorów niealgebraicznych podzapytanie sal obliczane będzie niezależnie dla każdego z nich. Aparat wykonawczy musi wykonać funkcję *nested* na wirtualnym identyfikatorze, jej wynik włożyć na stos środowisk ENVS i w tak wyznaczonym nowym kontekście obliczyć zapytanie `sal` . Obliczenie funkcji *nested* dla wirtualnych identyfikatorów wygląda następująco. Każdy wirtualny identyfikator ma zapisane w postaci logicznego identyfikatora, z jakiej perspektywy został wygenerowany. Aparat wykonawczy sięga do zapisanej w składzie danych definicji tej perspektywy i szuka jej podperspektyw. Dla każdej ze znalezionych podperspektyw wykonywana jest znajdująca się w jej definicji procedura „*virtual objects”* zwracająca ziarna wirtualnych podobiektów. Na podstawie tych ziaren i wirtualnego identyfikatora, na którym funkcja nested jest obliczana, tworzone są wirtualne identyfikatory dla wirtualnych podobiektów. Wynikiem funkcji *nested* jest suma wyników otrzymanych w trakcie wykonania procedur „*virtual objects”* we wszystkich podperspektywach zdefiniowanych z perspektywą jaka dany obiekt wirtualny generuje.

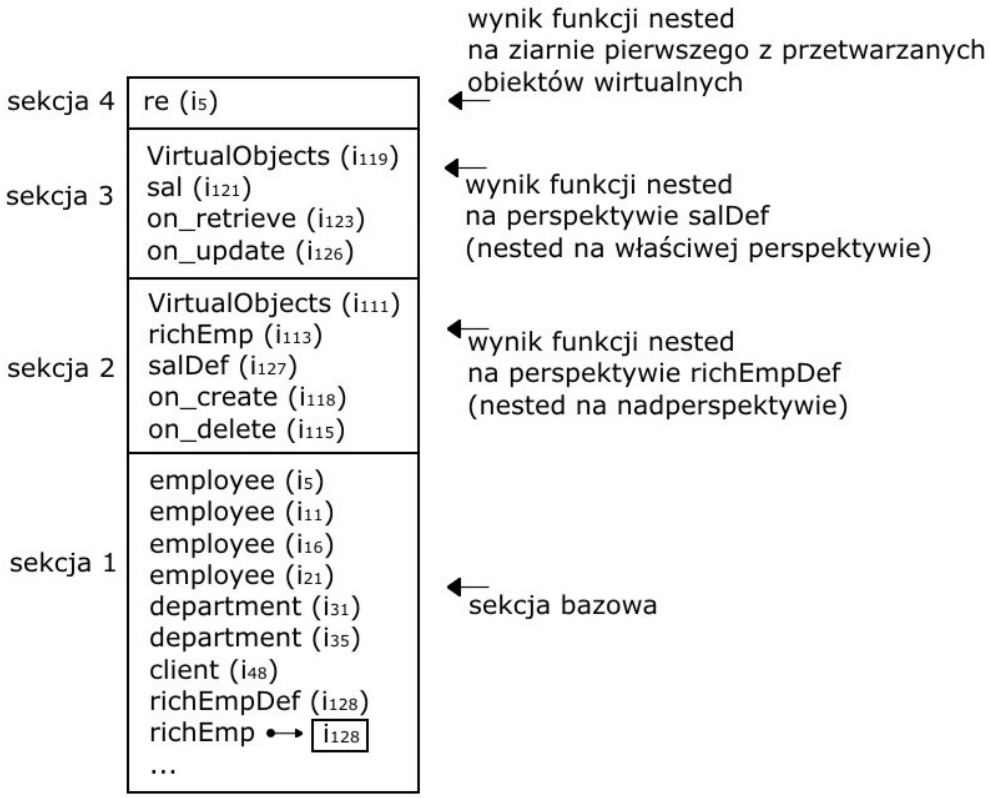
Perspektywa *richEmpDef* zawiera tylko jedną podperspektywę o nazwie *salDef*. Procedura zwracająca ziarna podobiektów w perspektywie *salDef* ma postać:

```
return (re.salary) as s
```

Przed jej obliczeniem aparat wykonawczy standardowo zasłania wszystkie sekcje stosu środowisk po za sekcją bazową. Na wierzch stosu dołożone zostają sekcje zawierające wynik

funkcji *nested* dla kolejnych nadperspektyw. Wynik *nested* dla referencji wskazującej na perspektywę korzeniową trafia na stos jako pierwszy. Dalej wyniki dla kolejnych nadperspektyw. Na sam koniec na stos trafia wynik *nested* dla obecnie przetwarzanej podperspektywy. Następnie wkładane są sekcje zawierające wynik *nested* dla kolejnych ziaren z jakich zbudowany był dany obiekt wirtualny w analogicznej kolejności. Ziarna z perspektywy korzeniowej trafią na niższą sekcję stosu niż bezpośrednie ziarno wirtualnego obiektu, na którym aktualnie obliczana jest funkcja *nested*. W ten sposób komunikowane są do wnętrza procedury *„virtual objects”* wszystkie obiekty jakie w trakcie jej wykonywania powinny być widoczne. Osoba definiująca perspektywę musi pamiętać o tym mechanizmie odpowiednio dobierając poszczególne nazwy obiektów. W szczególności nazwą powinny być rozróżnione poszczególne poziomy hierarchii ziaren. Stos środowisk pozwala aparatowi wykonawczemu na jednoznaczne określenie z jakim obiektem w danym kontekście związana powinna być jakaś nazwa. Nazwa występująca w wyższej sekcji stosu niejako przesłania wszystkie jej wystąpienia w sekcjach niższych. Jeżeli w definicji złożonej perspektywy ziarna jej podperspektywy będą widoczne pod taką samą nazwą jak ziarna perspektywy korzeniowej, pewne operacje staną się niemożliwe do wykonania.

W omawianej przykładowej sytuacji stos środowisk przed wykonaniem procedury *„virtual objects”* dla podperspektywy *salDef* wyglądałby taj jak przedstawiona na rysunku 4.4.



Rysunek 4.4: Stan stosu środowisk przed wykonaniem procedury zwracającej ziarna.

Ostatecznie procedura zwracająca ziarna perspektywy salDef zwraca w omawianym scenariuszu dla pierwszego z przetwarzanych wirtualnych identyfikatorów następujący jednoelementowy zbiór zawierający binder:

`bag{s(i3)}`

Wirtualny identyfikator podobiektu budowany jest następnie w oparciu o wirtualny identyfikator na którym wykonywane jest *nested* oraz otrzymane z procedury „*virtual objects"* ziarno. Nowy obiekt wirtualny ma nazwę taką, jak procedura zwracająca ziarna w definicji danej podperspektywy. Wektor ziaren *seeds* oraz wektor z referencjami perspektyw dziedziczone są z wirtualnego identyfikatora, na którym wykonywane jest *nested*. Na początek (pod indeksem 0) w tych wektorach dopisane są odpowiednio: otrzymane ziarno oraz logiczny identyfikator danej podperspektywy. Pozostałe elementy w tych wektorach przesuwane są jednocześnie o jedną pozycję (zwiększany jest indeks pod jakim się znajdują).

W omawianym przykładzie w wyniku funkcji *nested* na wirtualnym identyfikatorze:

<VIRT, *richEmp*, seeds{*re*(i5)}, view_defs{i128}>

tworzony jest wirtualny identyfikator reprezentujący obiekt o nazwie *sal* :

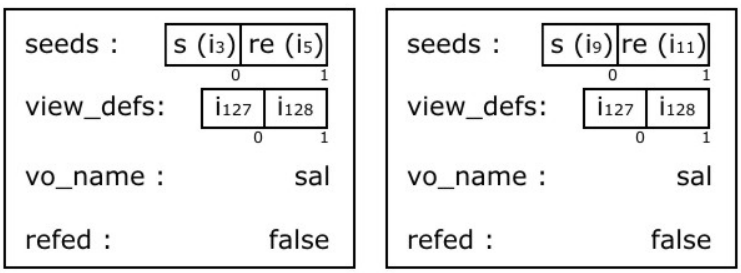
<VIRT, *sal*, seeds{*s*(i3), *re*(i5)}, view_defs{i127, i128}>.

Binder o nazwie *sal* z tym nowoutworzonym wirtualnym identyfikatorem jako wartością jest ostatecznym wynikiem funkcji *nested*. Po obliczeniu *nested* na wirtualnym identyfikatorze włożone w trakcie obliczeń sekcje stosu środowisk są zdejmowane. Wynik funkcji *nested* wkładany jest na stos środowisk jako jego nowa najwyższa sekcja. W tak wyznaczonym kontekście obliczany jest prawy argument niealgebraicznego operatora nawigacji.

W omawianym przykładzie w wyniku podobnych obliczeń dla drugiego z wirtualnych obiektów richEmp tworzony jest wirtualny identyfikator:

<VIRT, *sal*, seeds{*s*(i9), *re*(i11)}, view_defs{i127, i128}> .

Ostatecznie wynikiem zapytania `richEmp.sal` będą dwa obiekty wirtualne. Budowę reprezentujących je wirtualnych identyfikatorów przedstawia rysunek 4.5.



Rysunek 4.5: Budowa wirtualnych identyfikatorów reprezentujących podobiekty wirtualne.

Zanim użytkownik otrzyma odpowiedź wirtualne identyfikatory poddawane są dewirtualizacji. Użytkownik otrzymuje na swoje zapytanie odpowiedź:

bag{i4261412866, i4261412867}

Operowanie na wirtualnych złożonych obiektach przy pomocy operatorów niealgebraicznych wygląda dla użytkownika identycznie jak gdyby korzystał on z rzeczywistych obiektów.

Wymaganie pełnej przezroczystości działania z perspektywami wymaga także, by możliwe było definiowanie wirtualnych obiektów wskaźnikowych. Dla takich obiektów funkcja *nested* będzie dodatkowo rozszerzona, by umożliwić przechodzenie po takim wirtualnym wskaźniku. Mechanizm ten opiszę szczegółowo w jednym z następnych podrozdziałów.

4.3 Mechanizm przeciążania imperatywnych operacji na wirtualnych obiektach

Wirtualne identyfikatory reprezentujące wirtualne obiekty przetwarzane są w trakcie obliczenia zapytania w taki sam sposób jak referencje czyli identyfikatory obiektów zapamiętanych w składzie danych. Mogą być one przekazywane jako parametr wywoływania procedur, mogą być z nich zwracane, mogą być odkładane na stosie wyników QRES, mogą występować wewnątrz bindera na stosie środowisk ENVS. W inny sposób wykonywane są na nich jedynie operacje aktualizacyjne oraz dereferencja. Operacje te są niejako konsumentami wirtualnych identyfikatorów. Wirtualne identyfikatory natomiast można postrzegać jako sposób na przekazywanie informacji o obiekcie wirtualnym od jej źródła czyli od wygenerowania przez daną perspektywę do celu jakim jest procedura przeciążająca daną operację (konsument wirtualnego identyfikatora). Dla każdej z wymienionych w tym rozdziale operacji, twórca perspektywy może (ale nie musi) w definicji perspektywy zapisać procedurę, której kod powinien być wykonany zamiast akcji zwykle wykonywanej dla danej operacji. Jeżeli twórca nie zapisze w definicji perspektywy którejś z procedur, oznacza to iż dana operacja na wirtualnych obiektach jest zabroniona. Próba wykonania takiej operacji na wirtualnym obiekcie zakończy się błędem wykonania przerywającym obliczenia i stosownym komunikatem zwróconym użytkownikowi. Schemat wykonywania wymienionych operacji na wirtualnym identyfikatorze jest dla każdej z nich bardzo podobny. Kod procedury do wykonania oraz ewentualna nazwa parametru formalnego pobierane są z definicji perspektywy. Jeżeli dana operacja posiada parametr jest on obliczany. Przed rozpoczęciem wykonania procedury aparat wykonawczy zasłania wszystkie sekcje stosu środowisk oprócz sekcji bazowej, która powinna być widoczna w każdej procedurze. Na wierzchołek stosu wkładane są sekcje zawierające elementy definicji perspektywy oraz ziarna wirtualnych obiektów. Najwyższą sekcję stanowi sekcja z ewentualnym parametrem procedury. Po zakończeniu wykonywania procedury stan stosu jest przywracany. Włożone dodatkowe sekcje są zdejmowane, sekcje zasłonięte na czas wykonania procedury stają się znów widoczne. W przypadku operacji tworzących nowe wirtualne identyfikatory są one budowane w oparciu o wynik procedury. Operacje dereferencji i tworzenia obiektów zwracają w wyniku wynik z procedur *on_retrieve* i *on_create*. Pozostałe operacje zwracają w wyniku obiekt QueryNothingResult. Nawet jeżeli któraś z procedur *on_update*, *on_insert*, *on_delete* zwraca jakiś wynik jest on ignorowany.

4.3.1 Dereferencja, *on_retrieve*

Operator dereferencji zastępuje w wyniku obliczonym ze swojego podzapytania wszystkie wystąpienia QueryReferenceResult, czyli referencji do danych w składzie ich rzeczywistymi wartościami. Użytkownik może użyć go explicite w zapytaniu, by poznać aktualną wartość danego obiektu. Operacja dereferencji bywa też wykonywana automatycznie, gdy przetwarzany jest operator wymagający by jego argumenty były konkretnymi wartościami lub gdy takie jest domyślne działanie danego operatora. Dla przykładu argumentami operacji arytmetycznych muszą być liczby. Operator podstawienia dokonuje domyślnie automatycznej dereferencji na swoim prawym argumencie.

Dla omawianego przykładowego stanu składu danych, następujące zapytanie:

```
(employee where (salary > 10000)).salary
```

zwraca zbiór referencji bag{i_3, i_9}.

By dowiedzieć się jakie są zarobki w firmie ACME użytkownik musi użyć operatora dereferencji. Następujące zapytanie:

```
deref((employee where (salary > 10000)).salary)
```

zwraca w wyniku zbiór liczb bag{2000000, 30000}.

Automatyczna dereferencja wykonywana jest między innymi w trakcie przetwarzania operatorów agregujących. Dla przykładu wynikiem następującego zapytania:

```
sum(employee.salary)
```

będzie liczba 2043000.

Operacja dereferencji na wirtualnych identyfikatorach powinna z punktu widzenia użytkownika działać identycznie. Aparat wykonawczy stojąc przed zadaniem wykonania dereferencji na wirtualnym identyfikatorze zamiast standardowej dla tego operatora akcji, wykonuje procedurę *on_retrieve* zapisaną w definicji perspektywy jaka wygenerowała ten wirtualny identyfikator. Jeżeli w definicji perspektywy procedura on_retrieve nie zostanie odnaleziona, obliczenia zostaną przerwane z błędem. Dla przykładu następując zapytanie:

```
deref(richEmp)
```

zostanie przerwane. Perspektywa *richEmpDef* nie ma zdefiniowanej procedury *on_retrieve*. Użytkownik otrzyma komunikat:

Error in Query Executor: "This operation is not defined for this object"

Operacja dereferencji zdefiniowana jest natomiast w podperspekywie *salDef*. Zapytanie:

```
deref(richEmp.sal)
```

zadziała dokładnie tak samo jak następujące zapytanie działające na rzeczywistych obiektach:

```
deref((employee where (salary > 10000)).salary)
```

W wyniku w obu zapytaniach zwrócony zostanie zbiór liczb: bag{2000000, 30000}.

Zapytanie `richEmp.sal` zwraca dwa wirtualne identyfikatory:

<VIRT, *sal*, seeds{*s*(i3), *re*(i5)}, view_defs{i127, i128}>,

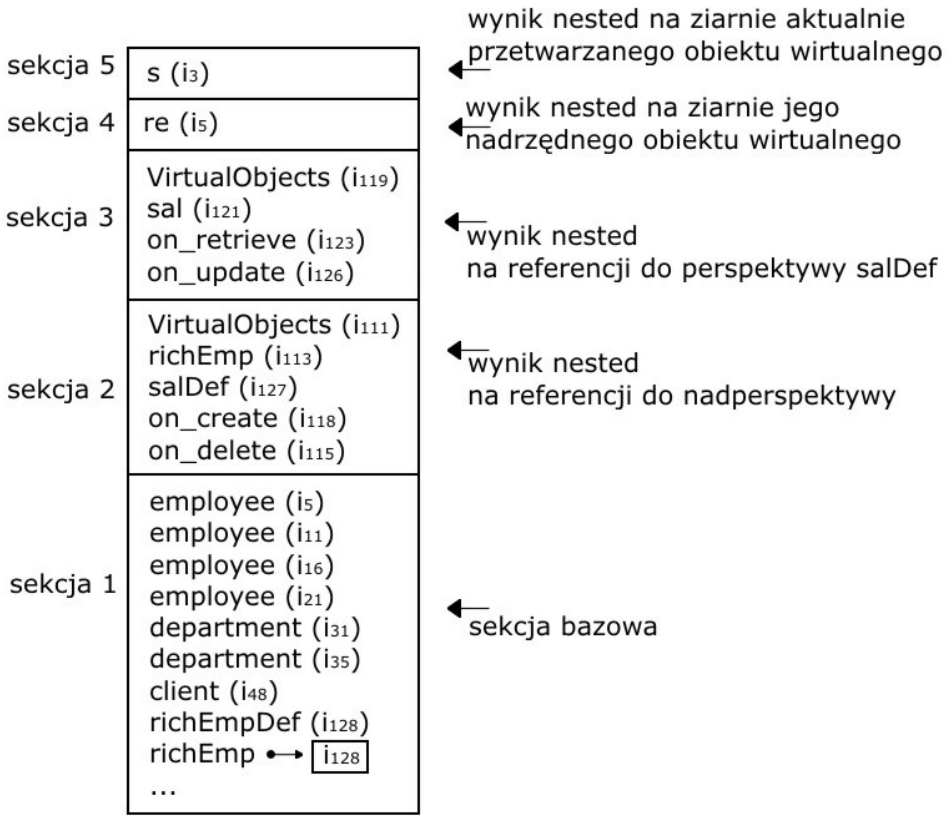
<VIRT, *sal*, seeds{*s*(i9), *re*(i11)}, view_defs{i127, i128}>.

By wykonać dereferencję na tych identyfikatorach, aparat wykonawczy zagląda do definicji perspektywy zapisanej w złożonym obiekcie o identyfikatorze i127 i szuka procedury *on_retrieve*. Identyfikator perspektywy jaka ma być przeszukiwana zapisany jest w wirtualnym identyfikatorze w wektorze *view_defs* pod indeksem 0. Po znalezieniu procedury *on_retrieve* a przed jej wykonaniem na stosie środowisk ENVS, wykonywane są standardowe działania. Sekcje stosu środowisk po za sekcją bazową zostają zasłonięte. Na wierzch stosu dokładane są sekcje zawierające wyniki funkcji *nested* dla przechowywanych w wirtualnym identyfikatorze referencji do definiujących go perspektyw i jej nadperspektyw oraz wyniki *nested* dla ziaren. Stan stosu środowisk przed wykonaniem procedury on_retrieve dla pierwszego z wirtualnych identyfikatorów wyglądałby tak jak na rysunku 4.6.

Procedura on_retrieve zdefiniowana w perspektywie *salDef* ma treść:

```
return(deref(s))
```

Nazwa *s* wiązana jest w piątej sekcji stosu z referencją wskazującą na obiekt *salary* o identyfikatorze i3. Dereferencja zwróci wartość tego obiektu, w tym wypadku liczbę 2000000. Liczba ta będzie jednocześnie wynikiem procedury *on_retrieve* dla pierwszego z wirtualnych identyfikatorów. Z punktu widzenia użytkownika będzie wynikiem zastosowania dereferencji na wirtualnym obiekcie *sal*. Obliczenia dla drugiego z wirtualnych identyfikatorów są analogiczne.



Rysunek 4.6: Stan stosu środowisk przed wykonaniem procedury *on_retrieve*.

Na wirtualnych identyfikatorach może być wykonywana dereferencja automatyczna. Odpowiedzi na poniższe przykładowe zapytania opierają się o wykorzystywany w całej pracy przykładowy stan składu danych, w którym procedura *on_retrieve* dla wirtualnych obiektów *sal* zwraca dereferencję obliczoną na odpowiadających im składowanych obiektach *salary*.

a) `avg(richEmp.sal)`

Powyższe zapytanie zwraca średnie zarobki bogatych pracowników, w przykładzie: 1015000.

b) `richEmp where sal < 100000`

Powyższe zapytanie zwraca wirtualne identyfikatory reprezentujące wirtualne obiekty *richEmp*, których podobiekty *sal* mają wartość mniejszą niż 100000. W omawianym przykładzie będzie to pojedynczy wirtualny identyfikator reprezentujący pracownika o nazwisku „*Johnson*”.

4.3.2 Tworzenie obiektów, *on_create*

Implementacja aktualizowalnych perspektyw w systemie LoXiM pozwala na zdefiniowanie operacji tworzenia nowych wirtualnych obiektów. Tak jak opisywałem w jednym z wcześniejszych rozdziałów, nowe obiekty dodawane są do składu danych w systemie LoXiM za pomocą instrukcji **create**. Przykładowo następująca instrukcja:

create (125000 **as** *salary***) as** *employee*

tworzy obiekt złożony

<i_{130}, employee, {<i_{129}, salary, 125000>}>

W przykładowym modelu danych zdefiniowanym w drugim rozdziale niniejszej pracy zdefiniowałem obiekty *employee* jako złożone obiekty, mające obowiązkowo podobiekty *firstname*, *lastname*, *salary* i *title*. Więzy integralności nie pozwoliłyby na stworzenie obiektu *employee* takiego jak powyżej (jedynie z podobiektem *salary*). W przykładach w niniejszym rozdziale zakładam brak takowych więzów, tak by możliwe było łatwe zobrazowanie działania procedury *on_create* przeciążającej operację tworzenia obiektów. Stworzony powyższą instrukcją nowy obiekt employee spełnia warunek *salary* > 10000. W związku z tym staje się on również widoczny poprzez perspektywę *richEmpDef* jako nowy obiekt *richEmp*. Wymóg pełnej przezroczystości wszelkich operacji aktualizacyjnych na wirtualnych obiektach powoduje, iż użytkownik powinien móc także wykonać instrukcję taką jak poniżej:

create (70000 **as** *sal***) as** *richEmp*

Efektem wykonania takiej instrukcji powinno być pojawienie się kolejnego obiektu *richEmp* z odpowiednio ustawionym atrybutem *sal*.

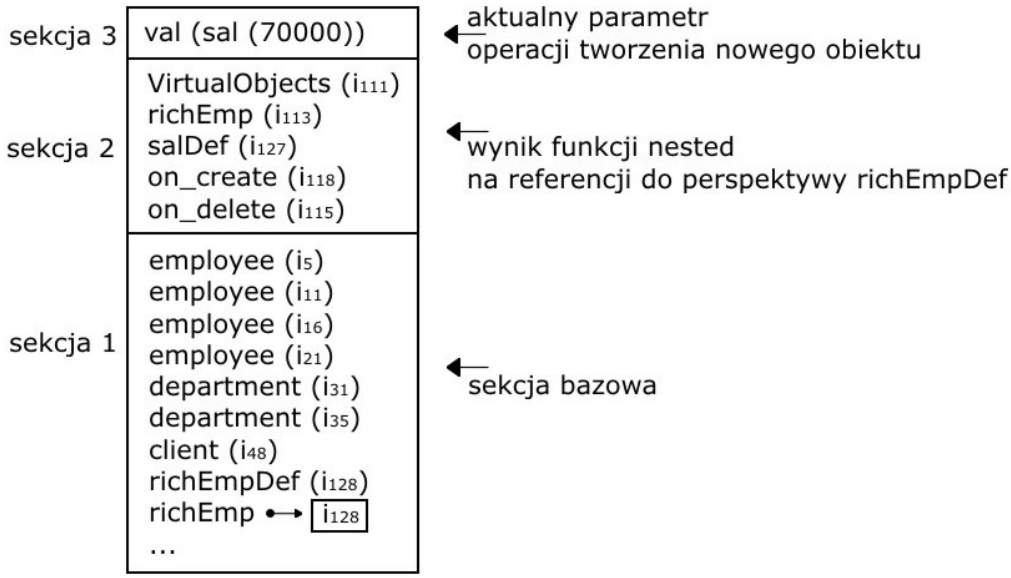
Aparat wykonawczy wykonując instrukcję create, po obliczeniu wartości nowotworzonego obiektu (lewy argument operatora **as**) sprawdza w strukturze wirtualnych korzeni nazwę nowotworzonego obiektu (prawy argument operatora **as**). Jeżeli nazwa ta nie jest zapisana jako nazwa jakichś korzeniowych wirtualnych obiektów, tworzony jest nowy rzeczywisty obiekt korzeniowy zgodnie z normalnymi zasadami. Jeżeli jednak nazwa obiektu jaki ma zostać dodany do składu jest zarejestrowana w strukturze danych wirtualnych korzeni, oznacza to iż tworzony właśnie obiekt ma być obiektem wirtualnym. Aparat wykonawczy sprawdza, która perspektywa generuje obiekty o takiej nazwie. Jeżeli takich perspektyw jest więcej niż jedna, instrukcja zostaje przerwana z błędem. W definicji perspektywy generującej obiekty o wskazanej nazwie poszukiwana jest procedura *on_create*. Gdy nie uda się jej odnaleźć instrukcja jest przerywana. Zwracany jest błąd wykonania oznaczający, iż operacja tworzenia wirtualnych obiektów o takiej nazwie nie została zdefiniowana.

Procedura *on_create* posiada jeden parametr. Parametrem tym jest wartość jaką ma mieć nowotworzony obiekt. Przed wykonaniem procedury *on_create* na stosie środowisk ENVS zasłanianie są wszystkie sekcje poza sekcją bazową. Procedura *on_create* nie jest wykonywana w tym wypadku w kontekście konkretnego wirtualnego identyfikatora. Wirtualny identyfikator dopiero powstanie w wyniku wykonania tej procedury czyli w wyniku stworzenia nowego wirtualnego obiektu. Ponieważ procedura *on_create* nie jest wykonywana w kontekście wirtualnego identyfikatora na stos środowisk przed jej wykonaniem wkładana jest jedynie sekcja z wynikiem funkcji *nested* na referencji do perspektywy generującej obiekty o danej nazwie oraz sekcja z parametrem wywołania procedury.

W rozważanym przykładzie procedura *on_create* zdefiniowana w perspektywie *richEmpDef* ma parametr o nazwie *val* oraz kod:

if (*val***.***sal* **>** 10000) **then return (create ((***val***.***sal* **as** *salary*) **as** *employee*)) **as** *re* **fi**

Przed wykonaniem kodu stan stosu środowisk jest taki jak na rysunku 4.7:



Rysunek 4.7: Stan stosu środowisk przed wykonaniem procedury *on_create*.

W przykładzie w procedurze zastosowano warunkowe sprawdzenie. Nowy obiekt zostanie utworzony tylko wtedy, gdy podana wartość atrybutu *sal* będzie większa niż 10000. Dzięki temu procedura uniemożliwia dodanie do składu obiektów, których nie będzie można zobaczyć przez tę perspektywę. Nie jest to jednak wymóg nałożony przez podejście SBA. Twórca perspektywy ma pełne środki algorytmiczne, by wyrazić swoje intencje aktualizacyjne za pomocą definiowanych procedur. Dopuszczalne jest zdefiniowanie procedury *on_create* w pewnej perspektywie w taki sposób, iż możliwe byłoby dodawanie obiektów przez nią niewidocznych. W ogólności dowolna procedura przeciążająca jedną z omawianych w tym rozdziale operacji może mieć dowolne efekty uboczne.

Następująca instrukcja

create (70000 **as** sal) **as** richEmp

obliczana jest jako:

if (70000 **>** 10000) **then return** (**create** ((70000 **as** *salary*) **as** *employee*)) **as** *re* **fi**

W rezultacie do składu danych dodany zostaje obiekt

<i132, employee, {<i131, salary, 70000>}>

Obiekt ten widoczny jest jako nowy obiekt wirtualny *richEmp* z atrybutem *sal* o wartości 70000.

Instrukcja **create** jako jedyna z operacji aktualizacyjnych zwraca swój wynik. Pozostałe operacje usuwania, podstawiania i wstawiania zwracają obiekt QueryNothingResult. Wynikiem zwracanym przez instrukcję **create** jest referencja do nowoutworzonego obiektu. Operacja

tworzenie obiektów wirtualnych powinna zachowywać się podobnie, powinna zwracać wirtualny identyfikator reprezentujący utworzony obiekt wirtualny. Identyfikator tworzony jest na podstawie ziarna. Ziarnem w tym wypadku jest wynik zwrócony przez procedurę *on_create* (binder *re*(i132)). W wirtualnym identyfikatorze zapisywane jest też z jakiej perspektywy pochodzi dany obiekt wirtualny i pod jaką nazwą jest widoczny. W analizowanej sytuacji byłby to następujący identyfikator

<VIRT, richEmp, seeds{*re*(i132)}, view_defs{i128}>.

Użytkownik otrzyma wynik zdewirtualizowany bag{i4261412867}.

Jedna uwaga odnośnie umieszczania pewnej logiki wewnątrz perspektyw a konkretnie odnośnie warunków, zgodnie z którymi jakaś operacje jest wykonywana lub nie.

Następująca instrukcja:

```
create (1300 as sal) as richEmp
```

zostałaby poprawnie obliczona. W jej wyniku nie powstałby jednak żaden nowy obiekt. W jej wyniku również nic nie zostałoby zwrócone. 1300 to za mały zarobek jak na bogatego pracownika. Definicja procedury *on_create* sprawia iż nie można dodać przy jej użyciu nowych obiektów *employee* mających mieć *salary* mniejsze niż 10000. Użytkownik mógłby jednak mieć problem z interpretacją co się stało. Instrukcja została wykonana. Nie wystąpił żaden błąd a jednak obiekt nie został utworzony. By nie wprowadzać tego typu nieporozumień dla użytkownika warto w tego typu procedurach dodać konstrukcję przerywającą obliczenie zapytania i zwracającą błąd zdefiniowany przez twórcę perspektywy. W systemie LoXiM zaimplementowano podstawową funkcjonalność wyjątków. Treść procedury *on_create* w formie najlepiej zrozumiałej dla użytkowników powinna mieć postać:

```
if (val.sal > 10000)
then return (create ((val.sal as salary) as employee)) as re
else throw "WrongParameterException" fi
```

Wyjątek `WrongParameterException` informuje użytkownika, iż zapytanie nie zostało wykonane z powodu podania niepoprawnej wartości parametru.

Możliwość zdefiniowania wewnątrz procedury, warunku sprawdzającego czy nowotworzony obiekt będzie widoczny przez daną perspektywę pozwala na zasymulowanie znanej z podejścia SQL klauzuli WITH CHECK OPTION. Podobny warunek może być użyty w przypadku operacji aktualizującej wartość wirtualnego obiektu.

4.3.3 Usuwanie obiektów, on_delete

Wirtualne obiekty mogą być przez użytkownika usuwane tak samo jak obiekty rzeczywiste składowane w bazie danych. Aparat wykonawczy w celu zrealizowania żądania usunięcia obiektu wirtualnego wykonuje zdefiniowaną w definicji perspektywy procedurę *on_delete*. Mechanizm jest w tym wypadku bardzo podobny jak w przypadku operacji dereferencji. Jedyną różnicą jest fakt, iż operator usuwania nie zwraca nic jako swój wynik. Dokładniej zwraca on obiekt klasy QueryNothingResult czyli LoXiM-owe nic. Nawet jeśli procedura *on_delete* została zdefiniowana tak, iż zwraca jakiś wynik, jest on ignorowany. Argumentem operacji usuwania

może być zbiór referencji wskazujących obiekty do usunięcia. Zamiast referencji mogą się pojawić w tym zbiorze wirtualne identyfikatory. Wskazane obiekty usuwane są po kolei niezależnie od siebie. Na przemian mogą być usuwane rzeczywiste i wirtualne obiekty, każde we właściwy dla danego typu sposób. Operacja usuwania wirtualnego obiektu przebiega podobnie jak operacja dereferencji. W definicji perspektywy generującej dany obiekt szukana jest procedura *on_delete*. Jeśli nie zostanie znaleziona, oznacza to iż danego obiektu wirtualnego nie można usunąć. W przeciwnym wypadku znaleziona procedura jest wykonywana. Wcześniej, przed jej wykonaniem wszystkie sekcje stosu poza sekcją bazową zostają zasłonięte. Dołożone zostają sekcje zawierające wyniki *nested* kolejno dla referencji perspektyw przechowywanych w wektorze *view_defs* i dla ziaren przechowywanych w wektorze *seeds* w usuwanym wirtualnym identyfikatorze. Po obliczeniu w kontekście tak przygotowanego stosu środowisk procedury *on_delete*, włożone na stos sekcje zostają zdjęte. Ze stosu wyników QRES zdejmowany jest wynik tej procedury. Jako wynik działania operatora delete na stos QRES trafia nowy obiekt klasy QueryNothingResult.

Przykład:

```
delete (richEmp where sal = 125000)
```

Powyższa instrukcja zleca aparatowi wykonawczemu usunięcie wirtualnego obiektu reprezentowanego przez wirtualny identyfikator:

<VIRT, *richEmp*, seeds{*re*(i130)}, view_defs{i128}>

Zlecenie to interpretowane jest poprzez wykonanie procedury *on_delete* z definicji pespektywy *richEmpDef*. Kod tej procedury to: **delete** *re*. Nazwa *re* wiąże się w najwyższej sekcji stosu środowisk z ziarnem na bazie którego zbudowany jest dany wirtualny identyfikator. W rezultacie ze składu danych usuwany jest na trwałe obiekt o identyfikatorze i130. Obiekt złożony usuwany jest wraz ze wszystkimi swoimi podobiektami. Gdy usuwany jest obiekt korzeniowy, jest on także wyrejestrowywany ze struktury korzeni.

Po fizycznym usunięciu obiektu o identyfikatorze i130 ze składu danych, przestaje on być widoczny poprzez perspektywę richEmpDef. Mechanizm SBA oraz odpowiednio zdefiniowana przez twórcę perspektywy procedura *on_delete*, pozwalają na przezroczyste usunięcie wirtualnego obiektu. Mimo iż usuwany obiekt istniał jedynie na poziomie koncepcyjnym, operacja jego usuwania przebiegła w taki sposób, iż użytkownik ma złudzenie jakby faktycznie obiekt ten został usunięty. Identyczny efekt użytkownik osiągnąłby korzystając z rzeczywistej referencji i wykonując następującą instrukcji:

```
delete (employee where salary = 125000)
```

Zgodnie z przyjętą zasadą pełnej przezroczystości nie ma żadnych semantycznych ani składniowych różnic przy wykonywaniu poszczególnych operacji na rzeczywistych i wirtualnych obiektach.

4.3.4 Podstawienie nowej wartości, *on_update*

Zmiana wartości obiektu wirtualnego możliwa jest tylko wówczas, gdy twórca perspektywy generującej dany obiekt zdefiniował w niej procedurę *on_update*. Rozważmy przykładowo następujące zapytanie:

```
(richEmp where sal = 70000).(sal := 71000)
```

Na wirtualny identyfikator:

<VIRT, *sal*, seeds{*s*(i131), *re*(i132)}, view_defs{i127, i128}>

podstawiana jest wartość 71000. Aparat wykonawczy pobiera z definicji perspektywy *salDef* procedurę *on_update*. Procedura ta ma jeden parametr w tym wypadku nazwany *new_sal*. Treść przykładowej procedury *on_update* jest następująca:

```
if (new_sal > s) then  (s := new_sal) fi
```

Przed wykonaniem procedury aparat wykonawczy zasłania wszystkie sekcje stosu środowisk oprócz sekcji bazowej. Tak jak w przypadku omawianych wcześniej operacji poprzez dołożenie na stos nowych sekcji przekazywane są do wnętrza procedury wyniki funkcji *nested* na referencjach perspektyw i ziaren pobrane z wektorów *view_defs* i *seeds* aktualizowanego wirtualnego identyfikatora. Na wierzchołek stosu wkładana jest jeszcze sekcja zawierająca aktualny parametr procedury. Parametrem tym jest binder o nazwie takiej jak nazwa formalnego parametru procedury (w omawianym przypadku – *new_sal*) i wartości równej obliczonemu prawemu argumentowi operatora podstawienia (71000). Wartość obliczona jako prawy argument operatora podstawienia poddawana jest domyślnie automatycznej dereferencji. W tak utworzonym kontekście wykonywana jest treść procedury *on_update*. Po jej wykonaniu włożone sekcje stosu są zdejmowane, zaś zasłonięte wcześniej sekcje są odsłaniane. Operacja podstawienia zwraca obiekt QueryNothingResult. Jeżeli procedura *on_update* zwróci jakiś wynik, będzie on zignorowany.

W wyniku wykonanego przykładowego podstawienia na obiekt wirtualny, wartość trzymanego w składzie danych obiektu i131 zostanie zwiększona o 1000. Obiekt ten widoczny jest nadal przez perspektywę, od tej pory już ze zwiększoną wartością wirtualnego atrybutu *sal*. Dla użytkownika operacja podstawienia jest przezroczysta. Jego polecenie zmieniło wartość wirtualnego obiektu.

4.3.5 Wstawianie podobiektów, *on_insert*, *on_create*

W systemie LoXiM operator wstawiania obiektów **:<** może być użyty w dwóch różnych trybach. Lewy argument operatora wstawiania musi być pojedynczą referencją lub pojedynczym wirtualnym identyfikatorem. Prawy argument musi być zbiorem (w szczególności jednoelementowym) referencji lub binderów. Sytuację gdy lewym argumentem operacji wstawiania jest referencja omówiłem w rozdziale 2.2.4. Podobnie powinna być wykonywana ta operacja na wirtualnym identyfikatorze.

Jeżeli prawym argumentem operatora wstawiania jest referencja, wówczas wskazywany tą referencją obiekt zostanie wstawiony jako nowy podobiekt obiektu będącego lewym argumentem tej operacji. Wstawiany obiekt musi być obiektem korzeniowym. Ze względów technicznych (błąd koncepcyjny w sposobie zapisu obiektów w składzie danych) nie możliwe jest wycięcie poddrzewa z jednego ze złożonych obiektów i wstawienie go do innego obiektu. Wstawiany obiekt jest wyrejestrowywany ze struktury przechowującej wszystkie korzenie. Efektem takiego zastosowania operatora wstawiania jest fizyczne przeniesienie pewnego obiektu korzeniowego do wnętrza innego obiektu.

Prawym argumentem operatora wstawiania może być również binder. Na podstawie takiego bindera tworzony jest nowy obiekt, który będzie wstawiany do obiektu będącego lewym argumentem tej operacji. Operacja taka przypomina składniowo i semantycznie operację tworzenia nowych obiektów instrukcją create. Jedyną różnica jest to, że create tworzy obiekty korzeniowe, zaś instrukcja insert z binderem jako argumentem tworzy nowe podobiekty złożonych obiektów istniejących w składzie. Operacja wstawiania do obiektów wirtualnych możliwa jest w obu przedstawionych trybach.

Jeżeli lewym argumentem operatora wstawiania będzie wirtualny identyfikator a prawym referencja, wówczas wykonywana będzie procedura *on_insert* zdefiniowana w perspektywie, która dany obiekt wirtualny generuje. Schemat działania aparatu wykonawczego w takiej sytuacji jest następujący:

Wirtualny obiekt do, którego wstawiany jest nowy podobiekt będę nazywał wirtualnym rodzicem. Aparat wykonawczy pobiera z definicji perspektywy, która wygenerowała wirtualnego rodzica kod procedury *on_insert* oraz nazwę jej formalnego parametru. Jeśli procedury tej nie uda się znaleźć, wówczas zwracany jest błąd wykonania informujący iż nie można na danym obiekcie przeprowadzić tego typu operacji. Przed wykonaniem procedury na stosie środowisk zasłaniane są sekcje nie bazowe. Na stos wkładane są sekcje z wynikami *nested* dla perspektyw i ziaren wirtualnego identyfikatora reprezentującego wirtualnego rodzica. Na sam wierch stosu wkładana jest sekcja z aktualnym parametrem wywołania procedury *on_insert*. Parametrem jest binder, którego nazwa pochodzi z definicji procedury *on_insert* zaś wartością jest referencja będąca lewym argumentem operacji wstawiania. Po wykonaniu procedury stos środowisk przywracany jest do stanu sprzed jej wykonania.

Aparat wykonawczy nie sprawdza przed wykonaniem procedury *on_insert*, czy referencja będąca prawym argumentem operacji wstawiania jest korzeniem. Obiekt ten nie jest też automatycznie usuwany w tym momencie ze zbioru korzeni. Zadania te wykonywane są w trakcje wykonywania samej procedury. Procedura może być zdefiniowana w taki sposób, iż jej argument (czyli wcześniejszy prawy argument operacji wstawiania) jest wstawiany do ziarna obiektu wirtualnego (czyli do rzeczywistego obiektu, na bazie którego dany obiekt powstał). Wówczas sprawdzenie korzeniowości tego argumentu oraz ewentualne usunięcie ze zbioru korzeni dokonywane są w trakcie wyliczania procedury. Twórca procedury *on_insert* może jednak zapisać w niej dowolnie skomplikowane warunki pozwalające na wstawianie obiektów konkretnego typu, o konkretnej nazwie. Możliwe jest zapisywanie jakie operacje zostały zlecone do wykonania, nawet jeśli nie można ich wykonać, gdyż argument nie był korzeniem. Takie podejście zwiększa elastyczność oraz poszerza możliwości wykorzystania operacji *on_insert* na wirtualnych identyfikatorach.

Do wirtualnego rodzica może być także wstawiany binder. Akcja wykonywana wówczas jest inna niż dla wstawianej referencji. Operacja wstawiania bindera do wirtualnego identyfikatora jest interpretowana jako zlecenie utworzenie nowego wirtualnego podobiektu o wskazanej wartości i nazwie.

W perspektywie z jakiej wygenerowany został wirtualny rodzic wyszukiwane są wszystkie zdefiniowane podperspektywy. Szukana jest ta z podperspektyw, która generuje wirtualne podobiekty o nazwie takiej jak w binderze będącym prawym argumentem operacji wstawiania. Znalezienie więcej niż jednej perspektywy generującej obiekty o wskazanej nazwie zakończy przetwarzanie zapytania z błędem. W znalezionej podperspektywie szukana jest definicja

procedury *on_create*. Procedura ta zostanie wykonana zamiast normalnych akcji związanych z tworzeniem rzeczywistych podobiektów. Jak w przypadku wszystkich omawianych operacji, nie znalezienie procedury *on_create* powoduje przerwanie wykonania operacji wstawiania oraz poinformowanie użytkownika iż dana operacja nie została zdefiniowana.

Przed wykonaniem procedury standardowo zasłaniane są wszystkie sekcje poza bazową. Wstawiane są sekcje zawierające wyniki *nested* dla perspektyw definiujących dany obiekt wirtualny. Wynik *nested* na perspektywie z najwyższego poziomu (perspektywy korzeniowej) wstawiany jest przed innymi. Następne wstawiane są w kolejności aż do perspektywy bezpośrednio generującej wirtualnego rodzica. Następnie wkładana jest jeszcze sekcja z wynikiem *nested* dla podperspektywy, z której definicji pobrana była procedura *on_create*. Następnie na stos wkładane są sekcje zawierające wynik funkcji *nested* na kolejnych ziarnach wirtualnego identyfikatora. Na sam wierzch stosu trafia sekcja z binderem będącym aktualnym parametrem wywołania procedury *on_create*. Nazwa parametru pobrana została z definicji procedury. Wartością jest natomiast wartość bindera, będącego prawym argumentem operacji wstawiania.

Przykład:

```
(richEmp where sal = 70000) :< 12345 as sal
```

Powyższa instrukcja byłaby interpretowana jako próba wstawienia dodatkowego wirtualnego obiektu *sal* do jednego z obiektów *richEmp*. W omawianym przykładzie powyższa instrukcja zakończyłaby się błędem, gdyż w perspektywie *salDef* nie zdefiniowano procedury *on_create*.

4.3.6 Operacja *on_store*

Obiekty w modelu danych w SBA mogą mieć wartości atomowe, złożone albo wskaźnikowe. Obiekt wskaźnikowy (pointer) jako swoją wartość przetrzymuje logiczny identyfikator wskazywanego obiektu. Referencje mogą wystąpić wewnątrz bindera będącego argumentem operacji tworzenia nowego obiektu. Mogą wystąpić również jako prawy argument operacji aktualizacji wartości obiektu. W każdej z tych sytuacji w składzie danych tworzone są obiekty wskaźnikowe.

Przykłady:

```
(employee where lastname = "Johnson").works_in :=
      ref(department where deptname = "Retail sales")

create (employee where lastname = "Wayne") as main_boss
```

Ponadto referencje mogą wystąpić jako prawy argument operacji wstawiania nowego podobiektu do wnętrza innego obiektu. Referencja taka musi wówczas wskazywać na obiekt korzeniowy. Obiekt taki w wyniku operacji wstawiania przestaje być korzeniem.

By wirtualny identyfikator mógł być wykorzystywany w takich sytuacjach tak samo jak referencja, konieczne jest zdefiniowanie w generującej go perspektywie procedury *on_store*. Procedura *on_store* wywoływana będzie za każdym razem, gdy identyfikator wirtualny będzie zapisywany jako wartość składowanego obiektu. Procedura ta w większości przypadków zwracać będzie ziarno wirtualnego obiektu. Ziarno przechowuje informację o relacji pomiędzy

wirtualnym obiektem a obiektem na bazie którego został zbudowany. Właśnie ta informacja wykorzystywana jest w trakcie wyżej wspomnianych operacji.

Gdyby perspektywa *richEmpDef* zawierała procedurę *on_store* zdefiniowaną w następujący sposób:

```
procedure on_store() {
      return re
}
```

możliwe było by wykorzystanie wirtualnego obiektu *richEmp* tak, jak w przykładach podanych powyżej dla rzeczywistych referencji.

```
create (richEmp where sal = 2000000) as main_boss
```

Po wykonaniu powyższej instrukcji obiekt *main_boss* będzie obiektem wskaźnikowym. Jego wartość uzyskana jest z wirtualnego identyfikatora zwróconego jako wynik zapytania: `richEmp where sal = 2000000` . Procedura *on_store* zwraca ziarno na podstawie którego zbudowany jest konkretny obiekt wirtualny. Ziarno zawiera referencję, która ostatecznie będzie zapisana jako wartość obiektu *main_boss*. Mechanizm wykonania takiej operacji w aparacie wykonawczym jest taki sam jak w przypadku wcześniej omawianych operacji aktualizacyjnych. Procedura *on_store* wyszukiwana jest w definicji generującej obiekt perspektywy. Przed jej wykonaniem zasłaniane są sekcje stosu środowisk poza sekcją bazową. Dokładane są nowe sekcje, tak by wewnątrz procedury widoczne były ziarna oraz wnętrza perspektyw tworzących dany obiekt wirtualny. Po wykonaniu procedury stan stosu środowisk jest przywracany. Zwrócona z procedury *on_store* wartość wykorzystywana jest w aktualnie wykonywanej operacji. Jeżeli zwrócona wartość jest następnym wirtualnym identyfikatorem, także na nim wykonywana jest właściwa mu procedura *on_store*. Obliczanie kolejnych procedur *on_store* kontynuowane jest aż do uzyskania innego typu wyniku niż wirtualny identyfikator lub do momentu, gdy dla pewnego z wirtualnych identyfikatorów nie zdefiniowano procedury *on_store*. Opisana sytuacja może się zdarzyć, gdy perspektywa definiuje wirtualne obiekty na bazie innych wirtualnych obiektów. Przykład takiej perspektywy „wyższego rzędu" opisany będzie w jednym z następnych podrozdziałów.

W opisywanej sytuacji obiekt *main_boss* tworzony był na podstawie wirtualnego obiektu *richEmp*. Zawiera jednak wskaźnik na rzeczywisty obiekt *employee*. Następujące zapytanie:

```
main_boss.richEmp
```

zwróci jako wynik zbiór wszystkich obiektów *richEmp*. Funkcja *nested* na obiekcie *main_boss* zwraca jedynie binder o nazwie *employee*. Nazwa *richEmp* w powyższym zapytaniu zostanie zatem związana w sekcji bazowej stosu środowisk. Jest to ewidentne złamanie zasady przezroczystości.

Rzeczywiste obiekty wskaźnikowe nie mogą wskazywać na wirtualne obiekty. W takim rzeczywistym wskaźniku nie ma miejsca na zapisanie koniecznych dla takiego działania informacji. Jest to na szczęście jednak jedyna sytuacja z jaką podejście SBA zaimplementowane w systemie LoXiM sobie nie radzi. W następnym rozdziale przedstawię pomysł na poradzenie sobie z tym problemem, tak by przetwarzanie wirtualnych danych w systemie LoXiM stało się w stu procentach przezroczyste.

Wskazywanie na wirtualne obiekty jest jednak możliwe do zapisania w innych wirtualnych obiektach. W następnym podrozdziale opiszę jak wirtualne wskaźniki zostały zaimplementowane w systemie LoXiM.

4.4 Wirtualne wskaźniki

Zaimplementowanie wirtualnych wskaźników wraz z możliwością ich aktualizacji było najtrudniejszą częścią mojej pracy. Rozpoczynając pracę nad tym problemem nie miałem pewności, czy jest on w ogóle możliwy do rozwiązania. Wirtualne wskaźniki powinny umożliwiać nawigację pomiędzy obiektami w taki sam sposób w jaki możliwe jest poruszanie się z wykorzystaniem składowanych obiektów wskaźnikowych. Twórca perspektywy powinien mieć możliwość zdefiniowania wirtualnych wskaźników zarówno wskazujących na rzeczywiste obiekty jak i na inne obiekty wirtualne. Wreszcie powinna istnieć możliwość aktualizacji takich wirtualnych wskaźników tak by korzystanie z nich było w pełni przezroczyste dla użytkownika. Wszystkie te wymagania udało się w systemie LoXiM spełnić.

Koncepcja aktualizowalnych perspektyw na bazie SBA została opracowana w pracy doktorskiej Hanny Kozankiewicz [Kozankiewicz04]. Większa część mojej implementacji aktualizowalnych perspektyw w systemie LoXiM opiera się na teorii w tej pracy przedstawionej. Temat wirtualnych wskaźników jest wspomniany na kilku stronach tej pracy. Główny problem z wirtualnymi wskaźnikami nie został w niej jednak rozwiązany. W przedstawionej koncepcji nie możliwa jest aktualizacja wirtualnego wskaźnika. Cytując fragment tej pracy:

"*The only problem with virtual pointers concerns their updates. Since virtual pointers have no identity, they cannot be updated. This is a slight violation of transparency requirement, which however cannot be avoided."* [Kozankiewicz04], strona 95

W implementacji wirtualnych wskaźników w systemie LoXiM podszedłem do tematu w nieco inny sposób. Jak się okazało wirtualne wskaźniki przy takim podejściu można aktualizować. Jedyny problem z ich przezroczystością znikł. Wirtualne wskaźniki szczegółowo opisane w niniejszym podrozdziale są w pełni przezroczyste dla użytkownika.

Wirtualny wskaźnik w systemie LoXiM definiowany jest bardzo podobnie jak wirtualny obiekt atomowy. Perspektywa generująca wirtualny wskaźnik zbudowana jest identycznie jak inne omawiane w tej pracy perspektywy. Zawiera procedurę „*virtual objects"* zwracającą ziarno, na bazie którego wirtualny obiekt wskaźnikowy jest tworzony. W perspektywie takiej twórca może zdefiniować operacje jakie na danym wirtualnym obiekcie mogą być wykonywane. Możliwe staje się dzięki temu wykonywanie operacji dereferencji czy aktualizacji na takim wirtualnym wskaźniku. Jedyne co odróżnia takie perspektywy od innych to specjalna procedura *on_navigate*. Procedura ta wykorzystywana jest w trakcie obliczania funkcji *nested* na wirtualnym wskaźniku. Dzięki temu możliwe jest przejście po takim wirtualnym wskaźniku w trakcie przetwarzania niealgebraicznych operatorów, w szczególności podczas nawigacji.

Procedura *on_navigate* pozwala na wyjście z wirtualnego obiektu. Przy jej użyciu można z wnętrza obiektu wirtualnego wskazać konkretny obiekt rzeczywisty i przejść po takim wirtualnym wskaźniku do niego. By jednak móc przejść po wirtualnym wskaźniku do innego wirtualnego obiektu nie wystarczy zdefiniowanie procedury on_navigate. W procedurze tej osoba definiująca perspektywę może wskazać do jakiego wirtualnego obiektu wskaźnik będzie

prowadził. Można wskazać perspektywę generującą wirtualne obiekty takie jak ten będący celem wskaźnika. Wewnątrz procedury *on_navigate* można także wykorzystując ziarno wskazać, który spośród wirtualnych obiektów generowanych przez tę drugą perspektywę ma być celem wskaźnika. Nie można jednak tego wskazywanego procedurą *on_navigate* wirtualnego obiektu zbudować. Przepis na budowanie wirtualnych obiektów zapisany jest tylko i wyłącznie w generującej je perspektywie. Konieczne jest założenie pewnej modularności. Osoba definiująca w swojej perspektywie wirtualny wskaźnik może nie mieć dostępu do definicji perspektywy generującej obiekty jakie mają być danym wskaźnikiem wskazywane.

Sytuacja jest zatem taka. Wewnątrz procedury *on_navigate* wiadome jest jaki typ wirtualnego obiektu (przez jaką perspektywę generowany) ma być nią wskazywany. Dostępne jest ziarno, jednoznacznie określające, na który konkretnie obiekt dany wskaźnik ma wskazywać. Przepis na zbudowanie wirtualnego obiektu na podstawie tego konkretnego ziarna zapisany jest w tamtej perspektywie. Trzeba zatem zwrócić się do niej, by w oparciu o przekazane ziarno wygenerowała ten jeden konkretny wirtualny obiekt. Taką operację będę nazywał wirtualizowaniem ziarna. Mechanizm wirtualizowania ziaren, czyli generowania wirtualnych obiektów na podstawie konkretnego ziarna wygląda podobnie do zwykłego mechanizmu wyliczania wirtualnych obiektów. Twórca perspektywy może zdefiniować, czy chce by jego perspektywa taką „usługę" pełniła. Ostatnia z ośmiu procedur „*on_*" nie opisana jeszcze w niniejszej pracy, procedura *on_virtualize* będzie spełniać właśnie taką rolę.

Gdy wykonywane będzie przejście po wirtualnym wskaźniku do innego wirtualnego obiektu, w definicji perspektywy generującej wskazywany obiekt wirtualny wyszukiwana będzie procedura *on_virtualize*. Jeżeli takowa nie zostanie znaleziona w definicji perspektywy oznaczać to będzie, iż operacja przejścia wirtualnym wskaźnikiem do takiego wirtualnego obiektu jako niezdefiniowana jest zabroniona. Jeżeli jednak uda się taką procedurę odnaleźć jest ona wykonywana. Parametrem dla procedury *on_virtualize* jest ziarno pochodzące z procedury *on_navigate* zdefiniowanej dla wirtualnego wskaźnika po jakim przejście się odbywa.

Jedyne co pozostało jeszcze do opisanie to sposób w jaki taką procedurę wirtualizującą swój parametr wywoływać z wnętrza procedur *on_navigate*. Procedura *on_virtualize* wywoływana będzie na skutek zastosowania nowego operatora dodanego do systemu LoXiM. Operator **virtualize as** został zaimplementowany w stylu binarnych operatorów niealgebraicznych. Jego składnia jest następująca:

```
query virtualize as virtual_object_name
```

gdzie *query* jest zapytaniem zwracającym obiekty, które mają zostać zwirtualizowane zgodnie z definicją z perspektywy generującą obiekty o nazwie *virtual_object_name*. Operator **virtualize as** wyliczany jest podobnie jak operatory niealgebraiczne. Najpierw obliczane jest lewe podzapytanie. Wynik tego zapytania jest zapamiętywany. Następnie aparat wykonawczy sprawdza, która z perspektyw definiuje obiekty o nazwie takiej jak prawy argument operatora **virtualize as**. W definicji perspektywy generującej obiekty o takiej nazwie wyszukiwana jest procedura *on_virtualize*. Po znalezieniu procedury jest ona wykonywana w pętli dla każdego ze składników wyniku dla lewego podzapytania. Składniki są komunikowane do procedury *on_virtualize* jako jej parametr.

Przykład:

```
create view v_empDef {
      virtual objects v_emp() {return employee as e}

      view v_nameDef {
            virtual objects v_name() {return e.lastname as n}
            procedure on_retrieve() {return deref(n)}
      }

      view v_works_inDef {
            virtual objects v_works_in() {e.works_in as w}
            procedure on_update (new_val) {w := new_val}
            procedure on_navigate() {
                  return (w.department) virtualize as v_dept
            }
      }
}


create view v_deptDef {
      virtual objects v_dept() {return department as d}
      procedure on_store() {return d}
      procedure on_virtualize(to_virt) {return to_virt as d}

      view v_deptnameDef {
            virtual objects v_deptname() {return d.deptname as dn}
            procedure on_retrieve() {return deref(dn)}
      }
}
```

Powyższe instrukcje tworzą dwie perspektywy *v_empDef* i *v_deptDef* generujące wirtualne odpowiedniki obiektów *employee* i *department*. Obiekty *v_emp* mają podobiekty *v_name* i *v_works_in*. Podobiekty *v_works_in* są wirtualnymi wskaźnikami.

Przeanalizujmy zapytanie:

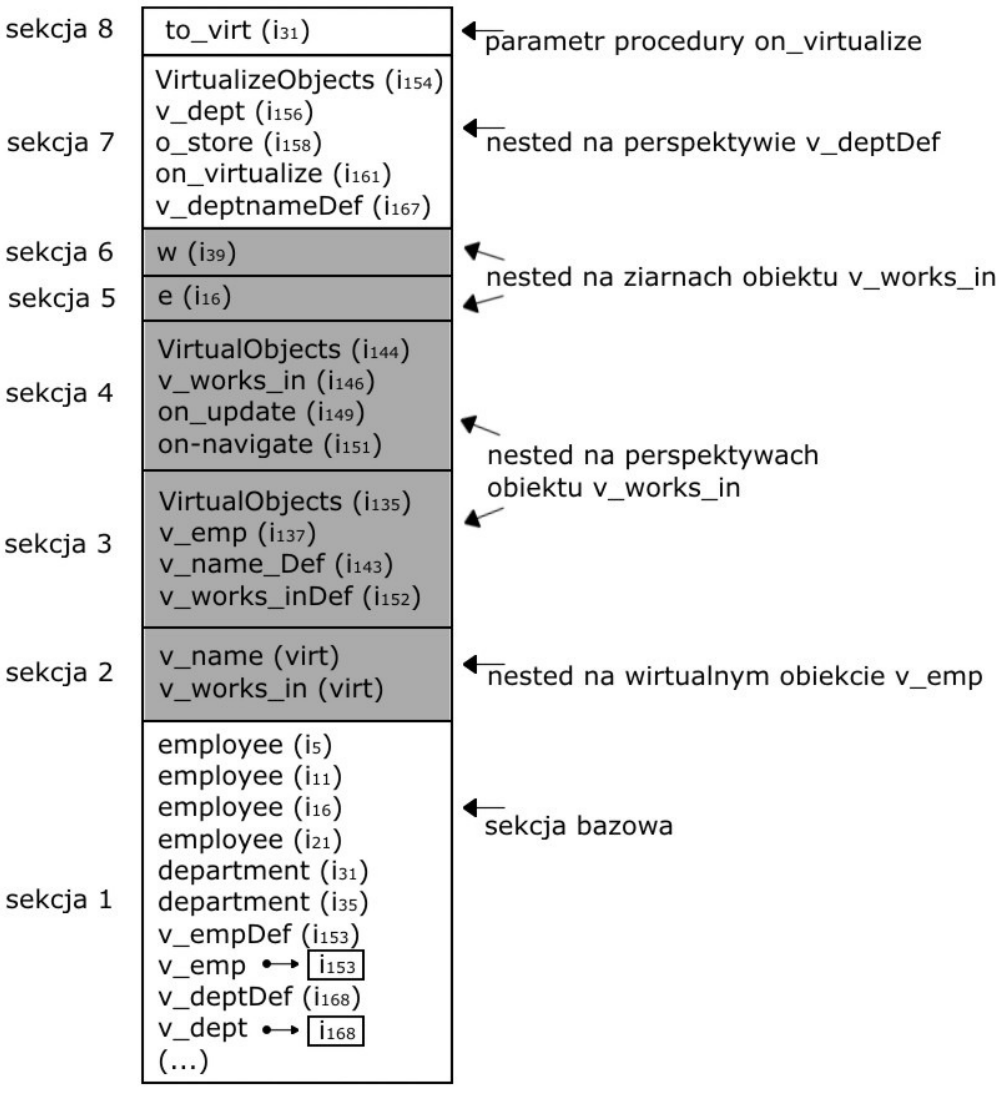
```
deref((v_emp where v_name ="Smith").v_works_in.v_dept.v_deptname)
```

Zapytanie takie powinno zwrócić nazwę działu w jakim pracuje pan Smith. Zapytanie `(v_emp where v_name ="Smith").v_works_in` zwróci wirtualny identyfikator reprezentujący wirtualny obiekt wskaźnikowy. Na obiekcie tym wykonywana jest dalsza część nawigacji, próba przejścia do obiektu *v_dept*. Nawigacja czyli operator niealgebraiczny powoduje obliczenie na wirtualnym identyfikatorze funkcji *nested*. We wcześniejszym rozdziale opisywałem jak obliczana jest funkcja *nested* dla wirtualnych identyfikatorów reprezentujących wirtualne obiekty złożone i atomowe. Teraz wprowadzę drobną modyfikację do przedstawionej definicji. W trakcie obliczania *nested* na wirtualnym identyfikatorze aparat wykonawczy w pierwszym kroku szuka w definicji perspektywy generującej dany obiekt procedury *on_navigate*. Jeżeli taka procedura zostanie znaleziona oznacza to iż obecnie przetwarzany wirtualny identyfikator reprezentuje wirtualny wskaźnik. Gdy procedura *on_navigate* nie zostanie odnaleziona szukane są wszystkie podperspektywy i wykonywane są znajdujące się w ich

definicjach procedury *„virtual objects”* tak jak to opisałem we wcześniejszym rozdziale. W omawianym przypadku procedura *on_navigate* zostaje odnaleziona. Standardowo sekcje stosu środowisk są przesłaniane, na wierzchołek stosu trafiają sekcje z zawartością perspektyw *v_empDef* i *v_works_inDef* oraz ziaren *e*(i16) i *w*(i39). W tym kontekście wykonywana jest treść procedury *on_navigate*:

```
return (w.department) virtualize as v_dept
```

Zapytanie `w.department` zwraca pojedynczą referencję i31. Aparat wykonawczy ustala, iż wirtualne obiekty o nazwie *v_dept* generowane są przez perspektywę *v_deptDef*. W definicji tej perspektywy odszukuje procedurę *on_virtualize*. Przed jej wykonaniem wszystkie sekcje stosu środowisk (także te włożone na potrzebę obliczenia procedury *on_navigate*) zostają zasłonięte.



Rysunek 4.8: Stan stosu środowisk przed wykonaniem procedury *on_virtualize*.

Jak zwykle widoczna pozostaje jedynie sekcja bazowa. Na wierzch stosu wkładana jest sekcja zawierająca wnętrze perspektywy *v_deptDef*. W najwyższej sekcji stosu pojawia się

parametr wywołania procedury *on_virtualize*. Stan stosu środowisk w trakcie wykonywania procedury *on_virtualize* przedstawiony jest na rysunku 4.8.

Kod procedury *on_virtualize* w omawianym przykładzie to: **return** *to_virt* **as** *d*. Nazwa *to_virt* wiązana jest w najwyższej sekcji stosu z parametrem przekazanym do procedury czyli z referencją i31. Po zakończeniu wykonywania procedury *on_virtualize* dwie dołożone sekcje zdejmowane są ze stosu środowisk, sekcje włożone na potrzeby procedury *on_navigate* są odsłaniane. Jak widać procedura *on_virtualize* wykonywana jest podobnie jak procedura „*virtual objects"*. Różnica jest taka, że zamiast szukać w składzie danych wszystkich obiektów spełniających daną definicję, sprawdzane jest jedynie czy definicję perspektywy spełnia obiekt przekazany jako parametr procedury *on_virtualize*.

Po zakończeniu procedury *on_virtualize* przetwarzanie wraca do procedury *on_navigate*, konkretnie do obliczania operatora **virtualize as**. Wynik procedury *on_virtualize* jest jednocześnie ziarnem tworzonego wirtualnego obiektu. W wirtualnym identyfikatorze zostaje zapisane ziarno *d*(i31), identyfikator perspektywy *v_deptDef* oraz nazwa wirtualnego obiektu *v_dept*. Zbudowany wirtualny identyfikator jest wynikiem operacji **virtualize as** i jednocześnie wynikiem procedury *on_navigate*. Obliczenie procedury *on_navigate* kończy się, sekcje stosu włożone w trakcie jej obliczenia są zdejmowane. Wirtualny identyfikator opakowywany jest następnie w binder z nazwą taką samą jak jego. Binder ten jest ostatecznym wynikiem funkcji *nested* na wirtualnym identyfikatorze *v_works_in*. Binder trafia na stos środowisk i wykonywane jest w tym kontekście zapytanie:

```
v_dept.v_deptname
```

W wyniku zwracany jest wirtualny identyfikator reprezentujący obiekt *v_deptname*. Na tym identyfikatorze wykonywana jest dereferencja. Wykonywana jest procedura *on_retrieve* zdefiniowana w perspektywie *v_deptnameDef*. Ostatecznym wynikiem całego zapytania jest napis „*Retail sales*" czyli nazwa działu, w którym pracuje pan Smith. Zapytanie operowało jedynie na wirtualnych obiektach. W przykładzie pokazałem, że możliwe jest w systemie LoXiM zdefiniowanie wirtualnych wskaźników, po których można przechodzić z jednego wirtualnego obiektu do innego dokładnie w taki sam sposób w jaki nawigować można po rzeczywistych obiektach zapisanych w składzie danych.

Operatora **virtualize as** można w systemie LoXiM używać także poza definicją procedury *on_navigate*. W szczególności można użyć go do testowania, które z obiektów spełniają warunek zdefiniowany w perspektywie. Dla przykładu wykorzystam jeszcze raz zdefiniowaną na początku rozdziału perspektywę *richEmpDef*. Do definicji perspektywy dodana zostaje procedura:

```
procedure on_virtualize(test) {
      return ((test where (salary > 10000)) as re) }
```

Po dodaniu tej procedury można zadawać zapytania:

```
(employee where name = "Smith") virtualize as richEmpDef

(employee where name = "Levinsky") virtualize as richEmpDef
```

itd. i sprawdzać, którzy z pracowników spełniają warunek dochodowy i są bogatymi pracownikami. Dla pracowników nie spełniających definicji bogatego pracownika zapytanie takie zwróci pusty zbiór.

Operator **virtualize as** został rozszerzony by można było wskazywać nim nie tylko wirtualne obiekty korzeniowe, ale także konkretne wirtualne podobiekty. Uogólniona składnia tego operatora wygląda następująco:

```
VIRTUALIZE_AS_QUERY :: = QUERY virtualize as NAME |
      :: = QUERY virtualize as (VIRTUALIZE_AS_QUERY).NAME
```

Dla zobrazowania jak działa w takiej sytuacji zdefiniuję następującą perspektywę:

```
create view v_clientDef {
      virtual objects v_client() {return client as c}
      procedure on_virtualize(t) {return t as c}

      view v_contractDef {
            virtual objects v_contract() {return c.contract as o}
            procedure on_virtualize(t) {return t.contract as o}
      }
}
```

Do perspektywy *v_empDef* dołożona zostaje podperspektywa *most_exp_sold_contractDef* generująca dla każdego wirtualnego obiektu *v_emp* wskaźnik do wirtualnego obiektu *v_contract*, będącego najdroższym kontraktem sprzedanym przez danego pracownika.

Operator *virtualize as* może być w takiej perspektywie użyty w następujący sposób:

```
view most_exp_sold_contractDef {
      virtual objects most_exp_sold_contract() {
            return ((max (e.supervises.client.contract) as best).
            ((e.supervises.client.contract) as x) where x = best)
            as m
      }

      procedure on_navigate() {
            return m virtualize as
            (parent(m) virtualize as v_client).v_product
      }
}
```

Zapytanie takie jak zapisane w powyższej procedurze *on_navigate* pozwala zbudować wirtualny podobiekt, generując na odpowiednich poziomach zagnieżdżenie operatora **virtualize as** potrzebne ziarna. By jednak zapytanie takie zadziałało konieczne jest zdefiniowanie w systemie LoXiM operatora **parent**(*query*), zwracającego obiekt będący rodzicem danego obiektu. Przy obecnej implementacji modułu składu danych, operatora takiego nie można niestety zdefiniować.

Na zakończenie opowieści o wirtualnych wskaźnikach winny jestem jeszcze przykład pokazujący, iż w systemie LoXiM wirtualne wskaźniki mogą być aktualizowane.

```
(v_emp where v_name = "Smith").v_works_in :=
      ref(v_dept where v_deptname = "IT")
```

Lewa strona operatora podstawienia zostanie obliczona jako wirtualny identyfikator reprezentujący wirtualny wskaźnik. Definicja generującej go perspektywy zawiera procedurę *on_update*. Obliczana jest zatem prawa strona operatora podstawienia. Wynikiem jest wirtualny identyfikator reprezentujący obiekt *v_dept*. Operator podstawienia wykonuje na swoim prawym argumencie automatyczną dereferencję. W powyższym przypadku nie jest ona jednak wykonana, gdyż użyty został zabezpieczający przed nią operator **ref()**. Podstawienie na obiekt jako nowej wartości wirtualnego identyfikatora nie jest technicznie możliwe. W definicji perspektywy *v_deptDef* odnajdywana jest więc procedura *on_store*. Procedura ta zwraca ziarno, z jakiego zbudowany był wirtualny obiekt będący prawym argumentem operacji podstawienia. Ziarno to przekazywane jest następnie jako parametr do procedury *on_update*. Wykonanie tej procedury powoduję zmianę wartości składowanego obiektu wskaźnikowego o identyfikatorze i_{39} z wartości i_{31} na i_{35}. Tym samym według zapisu w schemacie danych pan Smith zmienia dział zatrudnienia z „*Retail sales*" na „*IT*". Zmiana ta jest automatycznie widoczna na poziomie danych wirtualnych. Użytkownik ma wrażenie poprawnej aktualizacji wirtualnego wskaźnika.

4.5 Perspektywa ze stanem, pomocnicze procedury w definicji perspektywy

W skład definicji perspektyw w systemie LoXiM oprócz wymaganej procedury zwracającej ziarna wirtualnych obiektów, opcjonalnych procedur przeciążających poszczególne operacje wykonywane na wirtualnych obiektach oraz definicji podperspektyw mogą wchodzić dodatkowe obiekty i pomocnicze procedury. Takie dodatkowe obiekty trzymane wewnątrz definicji perspektywy mogą posłużyć do zapisania stanu perspektywy. Jako przykład niech posłuży perspektywa zwracająca informacje o klientach i zawartych z nimi kontraktach. Załóżmy, iż raz na dobę wykonywane są na bazie danych administracyjne operacje aktualizujące informacje o kontraktach. By nie przeszkadzać w tych obliczeniach przez jakiś czas perspektywa przestaje zwracać jakiekolwiek informacje.

```
create view v_client2Def {
      virtual objects v_client2() {
            if blocked = "no" then return client as c fi
      }

      create "no" as blocked

      /* pozostałe składniki definicji perspektywy */
}
```

Przez większą część doby zapytanie *v_client2* zwraca informację o klientach. Jednak po wykonaniu następującej instrukcji:

```
v_client2Def.blocked := "yes"
```

perspektywa przestaje generować wirtualne obiekty. Zmiana wartości podobiektu *blocked* z powrotem na wartość „*no*" spowoduje, iż wirtualne obiekty staną się ponownie widoczne. Obiekt *blocked* widoczny jest w każdej definiowanej procedurze, gdyż przed wykonywaniem każdej z nich na stos środowisk wkładane jest wnętrze perspektywy. Dzięki temu w szczególności w trakcie wykonywania procedur nazwa blocked może być poprawnie związana na stosie środowisk.

W perspektywie *richEmpDef* warunek definiujący jak dużo musi zarabiać pracownik by był bogatym pracownikiem zapisany był w procedurze zwracającej ziarna. Ta sama kwota wymieniona była literalnie także w procedurach aktualizujących wirtualne dane. Definicję perspektywy *richEmpDef* można zapisać w następujący sposób:

```
create view richEmpDef {
      virtual objects richEmp {
            return (employee where salary > sal_level) as re}

      procedure on_create(val) {
            if val.sal > sal_level then
                  return (create (val.sal as salary) as employee)
                  as re
            fi
      }

      create 10000 as sal_level

      /* pozostałe składniki definicji perspektywy */
}
```

Pomocniczy obiekt *sal_level* jest teraz punktem odniesienia. Pracownicy zarabiający więcej niż *sal_level* są bogatymi pracownikami. Zmiana wartości tego obiektu powoduje zwiększenie bądź zmniejszenie grona bogatych pracowników. Dzięki temu, że we wszystkich miejscach definicji perspektywy warunki odwołują się do tego samego obiektu, łatwiej jest zachować spójność definicji, łatwiej jest zarządzać taką perspektywą.

Pomocnicze procedury przechowywane w definicji perspektywy także są widoczne podczas wykonywania wszelkich operacji na wirtualnych obiektach generowanych przez daną perspektywę. Taka pomocnicza procedura może na przykład przeliczać zarobek pracownika na inną walutę. Można dzięki temu udostępnić wirtualny obiekt z wartością zarobku pracownika w innej walucie. Wszystkie operacje wykonywane na takim obiekcie wykorzystują pomocniczą procedurę w celu poprawnego przeliczania wartości. Procedury pomocnicze definiuje się w taki sam sposób jak procedury „*on_*". Mogą być one dowolnie złożone, z wieloma parametrami. Mogą zwracać wartości (są wtedy procedurami funkcyjnymi) bądź jedynie zmieniać w jakiś sposób pewne dane, nie zwracając żadnego wyniku.

Jak widać przechowywane w definicji perspektywy dodatkowe obiekty i procedury, będące de facto statycznymi składnikami każdego wirtualnego obiektu, można wykorzystać na wiele praktycznych sposobów. Funkcjonalność taka osiągnięta została w zasadzie za darmo, dzięki zastosowaniu mechanizmów SBA. Po zaimplementowaniu perspektyw bez stanu okazało się,

iż nie muszę wykonywać żadnych zmian w algorytmie przetwarzania wirtualnych obiektów, by pojęcie stanu można było wprowadzać do definicji perspektyw w systemie LoXiM.

4.6 Polecenia administracyjne, ewolucja schematu perspektywy

Definicja perspektywy w systemie LoXiM przechowywana jest w składzie danych w postaci złożonego obiektu. Obiekt ten może być przetwarzany jak każdy inny obiekt w składzie. W szczególności może być poddany operacjom aktualizacyjnym. Operacje aktualizacyjne pozwalają na dynamiczne zmiany w definicji perspektywy.

W dowolnym momencie można do definicji perspektywy dodać którąś z procedur „*on_*" lub procedurę pomocniczą. Przykładowo w wyniku polecenia:

```
richEmpDef :< procedure on_retrieve() {return deref(re)}
```

do definicji perspektywy *richEmpDef* dołożona zostanie procedura *on_retrieve*. Od tego momentu na generowanych przez tę perspektywę obiektach *richEmp* możliwe będzie wykonywanie dereferencji.

Możliwe jest dodefiniowywanie nowych podperspektyw. Po wykonaniu następującego polecenia:

```
richEmpDef :< view v_nameDef {
      virtual objects v_name() {return re.lastname as vn}
      procedure on_retrieve() {return deref(vn)}
      procedure on_update(new_val) {vn := new_val}
      }
```

obiekty *richEmp* zyskają nowy podobiekt *v_name*, na którym można będzie wykonywać dereferencję oraz aktualizację wartości. Dodatkowe obiekty reprezentujące stan perspektywy także można w dowolnym momencie dodawać do jej definicji.

Każdy ze składników definicji perspektywy może mieć zmienioną wartość.

```
richEmpDef.richEmp := procedure richEmp() {
      return (employee where count(salary) = 1 and salary > 6000)
      as re}
```

Powyższe polecenie zmieni definicję procedury tworzącej ziarna wirtualnych obiektów *richEmp*.

Każdy ze składników definicji perspektywy może być także z niej usunięty. Usunięcie którejś z procedur „*on_*" spowoduje, iż dana operacja na wirtualnych obiektach przestanie być dostępna. Usunięcie definicji jednej z podperspektyw sprawi, iż wirtualne podobiekty stracą jeden ze swoich podobiektów. W dowolnym momencie możliwe jest też całkowite usunięcie definicji perspektywy ze składu danych. Gdy tego typu instrukcja jest wykonywana, nazwa wirtualnych obiektów generowanych przez usuwaną perspektywę zostanie wyrejestrowana

ze spisu korzeniowych obiektów wirtualnych. Wraz z usunięciem definicji perspektywy znikają generowane przez nią wirtualne obiekty.

Możliwość dokonywania zmian w definicji perspektywy czyli dynamiczna ewolucja jej schematu jest kolejną cechą odróżniającą perspektywy w SBA od perspektyw znanych z SQL.

Niektóre z operacji administracyjnych mogą zepsuć definicję perspektywy. Usunięcie procedury definiującej ziarna albo obiektu *VirtualObjects*, na którym przechowywana jest nazwa takiej procedury, sprawi iż nie będzie możliwe wyliczenie wirtualnych obiektów. Definicja perspektywy musi zawierać dokładnie jeden obiekt *VirtualObjects*. Musi zawierać procedurę o nazwie takiej jak nazwa przechowywana na obiekcie *VirtualObjects*. Żadna z procedur „*on_*" nie może w definicji perspektywy wystąpić więcej niż jeden raz. Nazwy poszczególnych wirtualnych podobiektów muszą być różne. Liczba parametrów procedur „*on_*" nie może zostać zmieniona. Procedury *on_retrieve*, *on_delete*, *on_navigate* i *on_store* nie posiadają żadnych parametrów. Procedury *on_create*, *on_update*, *on_insert* i *on_virtualize* muszą mieć po jednym parametrze. Złamanie którejkolwiek z wymienionych zasad spowoduje wygenerowanie błędu podczas przetwarzania wirtualnych obiektów przez daną perspektywę generowanych.

4.7 Pozostałe funkcjonalności perspektyw w LoXiM

Wszystkie perspektywy jakie do tej pory tworzyłem w przykładach były perspektywami korzeniowymi. W systemie LoXiM możliwe jest też zdefiniowanie perspektywy wewnątrz jakiegoś obiektu. Obiekt do którego dodana będzie definicja perspektywy zyska jednocześnie wirtualne podobiekty.

Przykład:

```
create (12 as a) as outer;

outer :< view vaDef {
      virtual objects va() {return a as aa}
      procedure on_retrieve() {return deref(aa)}
      procedure on_update(new_a) {aa := new_a}
}
```

W powyższym przykładzie tworzony jest obiekt złożony o nazwie *outer*. Do tego obiektu dostawiana jest perspektywa *vaDef*. Obiekt *outer* zyskuje w wyniku takiego polecenia wirtualny podobiekt *va*, którego wartość może być zmieniana i który może być poddany operacji dereferencji. Następujące zapytanie:

```
deref(outer.va)
```

zwróci w wyniku liczbę 12.

Możliwość zagnieżdżania perspektyw wewnątrz dowolnych obiektów złożonych spowodowała konieczność wprowadzenia kilku zmian w mechanizmie przetwarzania zapytań w systemie LoXiM.

Poprawiona została funkcja *nested* działająca na referencjach do złożonych obiektów. Funkcja *nested* na takim obiekcie zwraca zbiór binderów z nazwami i referencjami do wszystkich jego podobiektów. Ponadto w trakcie obliczania tej funkcji aparat wykonawczy sprawdza czy dany obiekt nie zawiera definicji perspektyw. Dla każdej znalezionej perspektywy wykonywana jest jej procedura „*virtual objects"* i tworzone są wirtualne obiekty. Obiekty wygenerowane przez perspektywy uzupełniają wynikowy zbiór funkcji *nested* dla danego obiektu. Dla obiektów złożonych będących perspektywami funkcja nested obliczana jest jak dotąd.

Druga zmiana związana jest z zasadą widoczności poszczególnych sekcji wewnątrz procedur. Wewnątrz każdej procedury widoczna jest sekcja bazowa stosu środowisk. W procedurach składowanych jako podobiekty, widoczna jest także sekcja stosu środowisk, w której w trakcie wywołania procedura została znaleziona. Widoczne w takiej procedurze są zatem wszystkie podobiekty obiektu ją przechowującego. Podobnie w przypadku perspektyw zanurzonych w obiektach złożonych, wnętrze tych obiektów powinno być widoczne wewnątrz każdej z procedur zawartych w definicji takich perspektyw. W tym celu do klasy QueryVirtualResult dodałem nowy atrybut *view_parent*. Wirtualny identyfikator, będzie na tym atrybucie trzymał logiczny identyfikator obiektu przechowującego definicję generującej go perspektywy. Wartością parametru *view_parent* dla wirtualnych identyfikatorów tworzonych w perspektywach korzeniowych będzie *null*. Parametr *view_parent* będzie dziedziczony przez wirtualne podobiekty w trakcie obliczania funkcji *nested* na wirtualnym identyfikatorze. Wirtualne podobiekty tworzone w trakcie obliczanie tej funkcji będą miały taką samą wartość atrybutu *view_parent* jak wirtualny identyfikator, na którym wykonywana jest funkcja *nested*. Atrybut *view_parent* będzie wykorzystywany w trakcie wykonywania dowolnej z procedur zdefiniowanych dla danego wirtualnego obiektu. Na jego podstawie tworzona będzie jedna z sekcji stosu środowisk wkładanych na niego przed wykonaniem każdej z procedur zdefiniowanych na danym wirtualnym obiekcie. Sekcja ta wkładana będzie przed włożeniem sekcji tworzonych na podstawie wyników funkcji *nested* na samej definicji perspektywy i ziarnie wirtualnego obiektu. W sekcji tej znajdą się bindery z nazwami i referencjami do podobiektów rodzica perspektywy oraz binder o nazwie *ViewParent* z referencją do samego rodzica. Dzięki temu w definicji dowolnej z procedur operujących na wirtualnym obiekcie możliwe jest odwołanie się do dowolnego z podobiektów rodzica perspektywy.

Możliwość definiowania perspektyw niekorzeniowych zmienia też operację tworzenia nowych podobiektów. W trakcie dodawania nowego podobiektu sprawdzana jest jego nazwa. Jeżeli lewym argumentem operacji wstawiania jest referencja złożonego obiektu, aparat wykonawczy sprawdza, czy wewnątrz tego obiektu nie zdefiniowano perspektywy generującej wirtualne obiekty o nazwie takiej jak dodawanego właśnie podobiektu. Jeżeli taka perspektywa zostanie znaleziona, to zamiast zwyczajnie dodać nowy podobiekt wykonywana jest procedura *on_create* z definicji tej perspektywy. Brak procedury *on_create* w perspektywie oznacza brak możliwości dodania obiektu o takiej nazwie.

Przykład: Do definicji perspektywy *vaDef* przechowywanej w obiekcie *outer*, dołożona zostaje procedura *on_create*:

```
outer.vaDef :< procedure on_create(val) {ViewParent :< val as a}
```

Można teraz wykonać następującą instrukcję:

```
outer :< 54 as va
```

Aparat wykonawczy obliczając powyższą instrukcję ustali, iż nazwa *va* jest nazwą wirtualnych podobiektów obiektu *outer*. Wywołana zostanie procedura *on_create* z perspektywy *vaDef*. Ostatecznie do rodzica perspektywy zostanie dodany nowy podobiekt o nazwie *a* i wartości 54. Jednocześnie przez perspektywę staje się widoczny wirtualny obiekt *va* z identyczną wartością. W ten sposób można tworzyć wirtualne podobiekty obiektów trzymanych w składzie danych. By dodać nowy podobiekt wewnątrz procedury *on_create* nastąpiło odwołanie do nazwy *ViewParent*, która wiązana jest na stosie z referencją obiektu, w którym trzymana jest definicja perspektywy.

W powyższym przykładzie obiekt *outer* posiada rzeczywisty podobiekt o nazwie *a* i wirtualny podobiekt o nazwie *va*. Możliwość swobodnego mieszania wirtualnych i niewirtualnych obiektów jest kolejną cechą odróżniającą perspektywy oparte o SBA od perspektyw SQL.

Wspominałem już w niniejszej pracy o tym, iż niedozwolona jest sytuacja, by w składzie danych zapisane były korzenie o takiej samej nazwie jak wirtualnych obiektów generowanych przez jakąś korzeniową perspektywę. Zasada ta dotyczy również perspektyw umieszczonych wewnątrz obiektów. Na dowolnym poziomie, dowolna nazwa musi wiązać się albo tylko z rzeczywistymi albo tylko z wirtualnymi obiektami. Wstawienie na dany poziom perspektywy generującej obiekty o nazwie takiej samej, jak obiektów generowanych już przez wcześniej na tym poziomie znajdującą się perspektywę, zakończy się przerwaniem wykonania i zgłoszeniem stosownego błędu. W przypadku, gdy podczas wstawiania perspektywy wykryty zostanie konflikt nazwy wirtualnych obiektów z nazwą rzeczywistych obiektów znajdujących się na tym poziomie, aparat wykonawczy będzie próbował zwirtualizować rzeczywiste obiekty w oparciu o definicję wstawianej perspektywy. We wstawianej perspektywie aparat wykonawczy poszuka procedury *on_create*. Jeżeli jej nie znajdzie, to przerwie operację wstawiania perspektywy i zgłosi błąd. Jeżeli natomiast procedura *on_create* zostanie znaleziona, wówczas każdy z zapisanych obiektów zostanie przy jej pomocy zmieniony w obiekt wirtualny. W pętli dla każdego z tych obiektów, wywoływana będzie procedura *on_create*. Jej parametrem będzie aktualnie wirtualizowany obiekt. Po zakończeniu procedury obiekt ten zostanie usunięty ze składu danych.

Perspektywy tworzone w dotychczasowych przykładach budowane były w oparciu o dane zapisane w składzie. Wirtualne obiekty były przez nie generowane bezpośrednio na podstawie danych rzeczywistych. Nie jest to jednak żadne ograniczenie. Tak samo jak w przypadku perspektyw znanych z SQL, perspektywy w systemie LoXiM mogą być definiowane na podstawie wirtualnych danych generowanych przez inne perspektywy. Ziarna takich wirtualnych obiektów, same są wirtualnymi identyfikatorami. Obiekty takie przetwarzane są na takich samych zasadach.

Przykład:

```
create view double_virt_empDef {
      virtual objects double_virt_emp() {return v_emp as e}
      procedure on_retrieve() {return deref(e)}
}
```

Następujące zapytanie:

```
deref(double_virt_emp)
```

obliczone zostanie w następujący sposób. Aparat wykonawczy w celu wyliczenia obiektów *double_virt_emp* wykona procedurę zwracającą ziarna w perspektywie *double_virt_empDef*. W trakcie wykonywania tej procedury aparat wykonawczy będzie musiał wyliczyć obiekty *v_emp*, wywołując procedurę zwracającą ziarna pobraną z definicji perspektywy *v_empDef*. Obiekty *v_emp* budowane są na podstawie rzeczywistych obiektów *employee*, zaś obiekty *double_virt_emp* na podstawie wirtualnych obiektów *v_emp*. Operator dereferencji na obiektach *double_virt_emp* powoduje obliczenie procedury *on_retrieve* z definicji perspektywy *double_virt_empDef*. W procedurze tej wykonywane jest operator dereferencji na ziarnie, będącym wirtualnym identyfikatorem. To powoduje rekurencyjne wywołanie procedury *on_retrieve*, tym razem z definicji perspektywy *v_empDef*. W taki sposób, poprzez rekurencyjne wołanie się procedur z kolejnych poziomów perspektyw, wykonywane są wszelkie operacje na takich wirtualnych obiektach. Może to prowadzić do zapętlenia obliczeń, w sytuacji gdy wirtualne obiekty w dwóch perspektywach budowane są w oparciu o siebie nawzajem. Aparat wykonawczy nie wykrywa sytuacji rekurencyjnego odwoływania się perspektyw.

Rozdział 5
Dalszy rozwój aktualizowalnych perspektyw w LoXiM

5.1 Nierozwiązane problemy

Podczas implementacji perspektyw w systemie LoXiM natrafiłem na cztery problemy, których nie udało mi się rozwiązać. Trzy z nich spowodowane są pewnymi brakami w funkcjonalności obecnego modułu składu danych. Czwarty problem, jest poważniejszy i powoduje złamanie zasady pełnej przezroczystości wirtualnych danych.

W sytuacjach gdy instrukcją **create** tworzona jest nowa perspektywa korzeniowa oraz gdy perspektywa wstawiana jest do jakiegoś obiektu, aparat wykonawczy sprawdza czy po dodaniu perspektywy nie dojdzie do konfliktu nazw rzeczywistych i wirtualnych obiektów. Gdy konflikt taki jest wykryty rzeczywiste obiekty konwertowane są na wirtualne, a gdy jest to nie możliwe operacja jest przerywana z błędem.

Kolejną sytuacją kiedy na danym poziomie może pojawić się perspektywa powodująca konflikt nazw jest operacja podstawienia. Dowolny istniejący obiekt może stać się w wyniku podstawienia perspektywą. Jeżeli taki obiekt nie jest obiektem korzeniowym, aparat wykonawczy powinien sprawdzić nazwy pozostałych obiektów będących wewnątrz tego samego obiektu. Obiekty zapisane w obecnym składzie nie mają zapisanej informacji, częścią jakiego są obiektu. Relacja zapisana jest tylko w jedną stronę. Złożony obiekt ma zapisaną informację o tym jakie obiekty wchodzą w jego skład. Z tego powodu nie ma możliwości szybkiego odnalezienia rodzica danego obiektu. Fakt ten ma kilka nieprzyjemnych konsekwencji. Jedną z nich jest problem w sytuacji opisanej powyżej. Niekorzeniowy obiekt staje się perspektywą. Aparat wykonawczy nie może jednak znaleźć jego rodzica i dalej jego rodzeństwa. Niemożliwe staje się sprawdzenie czy taka operacja podstawienia nie spowoduje konfliktu nazw.

Brak funkcji zwracającej rodzica danego obiektu spowodował również, że nie można było zaimplementować operacji wycięcia obiektu z jednego miejsca w składzie i wstawienia jako podobiekt innego obiektu. By usunąć obiekt z miejsca, gdzie dotąd był zapisany konieczne byłoby odnalezienie jego dotychczasowego rodzica i zaktualizowanie jego informacji o posiadanych podobiektach. Z tego powodu obecnie w systemie LoXiM do obiektów można wstawiać jedynie obiekty korzeniowe oraz obiekty nowotworzone.

Następny nierozwiązany problem, wynikający z tej samej przyczyny, pojawił się w trakcie implementacji operatora **virtualize as**. Operator ten umożliwia zbudowanie wirtualnego obiektu na podstawie wskazanego ziarna. W swej rozszerzonej wersji umożliwia wygenerowanie wskazywanego wirtualnym wskaźnikiem wirtualnego podobiektu innego wirtualnego obiektu. Jednoznaczne określenie, który wirtualny podobiekt jest wskazywany, wymaga zbudowania wszystkich jego ziaren. W tym celu konieczne może być odnalezienie bezpośredniego rodzica rzeczywistego obiektu i rodzica tego rodzica itd. Bez wsparcia dla funkcji odnajdującej rodzica obiektu w składzie danych nie da się w pełni zrealizować przedstawionej przeze mnie koncepcji operatora **virtualize as**.

Następny problem związany jest z efektywnością przetwarzania wirtualnych obiektów. Na poziomie korzeni dodałem strukturę danych pozwalającą na szybkie stwierdzenie, czy istnieje perspektywa generująca obiekty o danej nazwie a jeśli tak to która. W sytuacji gdy dodawany jest do danego obiektu nowy składnik o jakiejś nazwie, konieczne jest przejrzenie zawartości tego obiektu w celu odszukania ewentualnej perspektywy generującej wirtualne obiekty o tej nazwie. Proces może być czasochłonny i spowolnia operację wstawiania podobiektów nawet w sytuacji, gdy w składzie nie ma żadnych perspektyw. Możliwe byłoby jednak znaczne poprawienie efektywności, gdyby w danym obiekcie w składzie danych można było zapisać taką sama informacje jak na poziomie korzeni. Informacja o tym, jakie nazwy generowane są przez jakie perspektywy w obrębie podobiektów danego obiektu byłaby w nim zapisywana w chwili dodawania do niego perspektywy. Usunięcie perspektywy z tego miejsca usuwałoby też odpowiedni wpis. Dzięki temu łatwe byłoby ustalenie czy dany obiekt zawiera perspektywę generującą obiekty o danej nazwie. Podobnie jednak jak z operacją odnajdywania rodzica tak i ten problem wymagałby bardzo dużych zmian w module składu danych.

Powyższe trzy problemy są stricte technicznej natury i wynikają jedynie z ograniczeń obecnej implementacji modułu składu danych. Problem czwarty natomiast jest rysą lekko psującą przezroczystość perspektyw w zaproponowanej przeze mnie implementacji. Problem ten nie był sygnalizowany w pracach teoretycznych dotyczących perspektyw w SBA, na których opierałem swą implementację. Wynika z faktu, iż w rzeczywistym, przechowywanym w składzie danych obiekcie wskaźnikowym nie ma możliwości zapisania informacji, iż wskazuje on na obiekt wirtualny. Rzeczywiste wskaźniki mogą wskazywać jedynie na rzeczywiste obiekty, podczas gdy wirtualne wskaźniki mogą wskazywać zarówno wirtualne jak i rzeczywiste obiekty. Sytuacja, w której problem ten występuje była już wspomniana w niniejszej pracy.

```
create (employee where salary = 2000000) as main_boss
```

W wyniku powyższej instrukcji tworzony jest obiekt wskaźnikowy *main_boss* wskazujący na obiekt *employee*. Obiekt wskaźnikowy można użyć w następującym zapytaniu (wynikiem zapytania jest liczba 2000000):

```
deref(main_boss.employee.salary)
```

Celem do jakiego dążyłem implementując perspektywy w LoXiM była stuprocentowa przezroczystość. Pełna przezroczystość oznacza, iż podobna operacja powinna być możliwa do wykonania z wykorzystaniem wirtualnych obiektów. Efekt powinien być identyczny. Analogiczne polecenie z wykorzystaniem wirtualnego obiektu:

```
create (richEmp where sal = 2000000) as main_boss
```

W wyniku powyższego polecenia także tworzony jest wskaźnikowy obiekt o nazwie main_boss. Zapytanie `richEmp where sal = 2000000` zwraca wirtualny identyfikator. Przed zapisaniem wartości obiektu *main_boss* wirtualny identyfikator zamienia się w logiczny identyfikator za pomocą procedury *on_store* zwracającej ziarno wirtualnego obiektu. Gdyby perspektywy były w pełni przezroczyste wynikiem zapytania:

```
deref(main_boss.richEmp.sal)
```

także powinna być liczba 2000000. Tak jednak nie jest. Obiekt *main_boss* wskazuje na rzeczywisty obiekt *employee*. Funkcja *nested* obliczona na obiekcie *main_boss* zwraca binder o nazwie *employee* a nie o nazwie *richEmp*. W rezultacie występująca w zapytaniu nazwa *richEmp* wiązana jest w bazowej sekcji stosu środowisk. Wynikiem powyższego zapytania jest zbiór wszystkich wartości podobiektów *sal* wszystkich obiektów *richEmp*.

Sytuacja ta pozwala rozróżnić wirtualne obiekty od rzeczywistych. Zaimplementowane przeze mnie perspektywy w związku z tym nie są niestety stuprocentowo przezroczyste.

Być może problem wynika z błędnej koncepcji. Być może pomyłką jest w ogóle procedura *on_store*. Być może funkcjonalność uzyskiwaną dzięki tej procedurze można osiągnąć w inny sposób. W sposób nie powodujący problemu takiego jak powyżej.

Nadzieją na rozwiązanie tego problemu mogą być role. Model danych w SBA opisuje taką funkcjonalność. W modelu z dynamicznymi rolami dany obiekt może być jednocześnie chłopem, wójtem i plebanem. Zależnie w jakiej roli jest oglądany, takie swe właściwości pokazuje. Być może wykorzystując ten model, dałoby się zapisać w obiekcie wskaźnikowym, że oprócz bycia wskaźnikiem na rzeczywisty obiekt *employee*, występuje również w drugiej roli. Wskazuje na pewien wirtualny obiekt *richEmp* i że wskazywany obiekt generowany jest przez perspektywę *richEmpDef*. Wówczas funkcja *nested* na takim wskaźniku zwróciłaby dwa bindery, jeden o nazwie *employee*, drugi o nazwie *richEmp*. Zależnie od użytej w zapytaniu nazwy obiekt wskazywałby raz obiekt wirtualny raz na niewirtualny. Są to jednak tylko moje gdybania a problem pozostaje otwarty.

Możliwe jest także, że problem jest szerszy. Może się okazać, że błąd leży w samej koncepcji nawigacji po wskaźnikach w SBA. Podobne problemy mogą zostać zidentyfikowane w trakcie implementacji klas i innych obiektów z modeli wyższych niż model M0 tego podejścia [Subieta04].

Problem nie jest na szczęście bardzo poważny. W porównaniu ze wszystkimi możliwościami aktualizowalnych perspektyw zaimplementowanych w systemie LoXiM, to jedno ograniczenie wydaje się być do zaakceptowania. Rysa ta powoduje jednak, iż nie można uczciwie nazwać takich perspektyw w pełni przezroczystymi.

5.2 Możliwe rozszerzenia

Celem mojej pracy było zaimplementowanie pełnej funkcjonalności aktualizowalnych perspektyw i przetestowanie czy teoria przedstawiona w pracy [Kozankiewicz04] zadziała w praktyce. Kwestie związane z wydajnością były mniej ważne. W związku z tym najważniejszym rozszerzeniem mojej pracy będzie optymalizacja zaimplementowanych mechanizmów. Obecna implementacja z dokładnością do jednego problemu przedstawionego we wcześniejszym podrozdziale spełnia kryterium przezroczystości. Użytkownik nie odczuwa różnic składniowych ani semantycznych pomiędzy tym jak korzysta z wirtualnych a jak z rzeczywistych, zapisanych w składzie obiektów. Na tej podstawie nie jest w stanie rozróżnić z jakiego typu obiektów w danej chwili korzysta. Nieprzezroczysta jest niestety prędkość wykonywania zapytań w zależności od tego jakie obiekty są przetwarzane. Obecnie czas obliczenia odpowiedzi na zapytanie korzystające z wirtualnych obiektów jest znacznie (co najmniej o rząd wielkości) większy, niż czas wykonania analogicznego zapytania wykorzystującego jedynie składowane obiekty. Nie przeszkadza to testować możliwości aktualizacyjnych perspektyw. Stanowi jednak poważny problem w kontekście ich produkcyjnego

wykorzystania. Bardzo dużo jest w tym temacie do poprawienia. Optymalizacji wymaga sama implementacja poszczególnych mechanizmów. Przed wykonaniem każdej z operacji na wirtualnym obiekcie w definicji perspektywy poszukiwana jest odpowiednia procedura. Wymaga to przejrzenia wszystkich podobiektów tej perspektywy. Każdy z nich trzeba pobrać ze składu, sprawdzić jaką ma nazwę. Po znalezieniu odpowiedniej procedury trzeba sprawdzić czy ma ona poprawną budowę tzn. czy posiada podobiekt *ProcBody* i odpowiednią liczbę podobiektów *Param*. Podobnie gdy wykonywana jest na wirtualnym obiekcie funkcja *nested*, w definicji perspektywy trzeba odnaleźć wszystkie podperspektywy. Następnie w każdej z nich odszukać procedurę generującą ziarna. W obecnej implementacji za każdym razem obliczane są wszystkie procedury generujące ziarna we wszystkich podperspektywach.

Jak widać, wiele jest do poprawienia. Perspektywa mogłaby zawierać zestaw flag pozwalający na szybkie ustalenie, które z procedur „*on_*" są w jej definicji, a których nie ma. Jeszcze lepszym rozwiązaniem byłoby trzymanie w obiekcie procedury słownika opartego o strukturę w rodzaju mapy haszującej. Poszukiwanie podobiektu o danej nazwie wymagałoby jedynie zajrzenia do słownika i sprawdzenia jaki identyfikator ma obiekt o danej nazwie. W innej strukturze mogłaby być trzymana lista identyfikatorów podperspektyw, by łatwiej można je było odnajdywać w trakcie wykonywania funkcji *nested*. Innym usprawnieniem byłaby leniwa ewaluacja wirtualnych podobiektów. Funkcja *nested* na wirtualnym identyfikatorze mogłaby zwracać jedynie zbiór specjalnych obiektów sygnalizujących, jakie wirtualne podobiekty dana perspektywa może posiadać. Dopiero podczas próby związania danej nazwy na stosie środowisk właściwa procedura zwracająca ziarna byłaby wykonywana.

Warto przetestować pomysł aby w wirtualnym identyfikatorze trzymane były nie referencje do perspektyw i ziarna ale wyniki zastosowania funkcji *nested* na tych obiektach. Obecnie w trakcie wykonywania dowolnej operacji na wirtualnym obiekcie funkcja *nested* obliczana jest za każdym razem na nowo dla każdego ziarna i dla każdej referencji perspektywy. Wyniki funkcji *nested* wkładane są na stos środowisk. Proces przygotowywania nowych sekcji stosu przed wykonaniem procedury można by przyspieszyć, gdyby wirtualny identyfikator miał zapisane przeliczone wyniki działania funkcji *nested*. Szybsze wykonanie zapytania byłoby wówczas obciążone większym zużyciem pamięci. Z tego powodu konieczne byłyby testy pozwalające odpowiedzieć na pytanie czy taka optymalizacja jest w ogóle opłacalna.

Optymalizacja samej implementacji to jedno. Równolegle warto powalczyć o to, by w trakcie przetwarzania wirtualnych obiektów, wykorzystane zostały mechanizmy optymalizujące drzewo wykonania danego zapytania. W systemie LoXiM zaimplementowano już kilka mechanizmów pozwalających na optymalizację w czasie wykonania zapytania [Sitek07]. Jest mechanizm półmocnej kontroli typów pozwalający na wykrycie zawczasu błędów w zapytaniu [Humpich08]. W trakcie parsowania zapytania wykonywana jest symulacja jego wykonania pozwalająca na przewidzenie, w których sekcjach stosu środowisk wiązane będą poszczególne nazwy występujące w zapytaniu. W systemie LoXiM zaimplementowano indeksy [Turski08]. Wszystkie te mechanizmy warto zintegrować z implementacją perspektyw. Do zaimplementowania pozostaje jeszcze wiele innych mechanizmów optymalizacji. Nazwę wirtualnego obiektu występującego w zapytaniu parser może zastąpić zapytaniem generującym ten wirtualny obiekt. Następnie całość zapytania poddać obróbce. Wydzielić i usunąć martwe podzapytania, wykorzystując zasady przemienności pewnych operatorów, wyłączyć pewne fragmenty przed dany operator. Dowolna optymalizacja stosowana dla zwykłych zapytań (zwykłych w rozumieniu – nie wykorzystujących wirtualnych obiektów) może być zastosowana także w trakcie przetwarzania obiektów wirtualnych.

Następnym rozszerzeniem obecnej implementacji perspektyw w systemie LoXiM jest zintegrowanie perspektyw i klas. Po zaimplementowani w systemie LoXiM dynamicznych ról, także z nimi warto będzie zintegrować perspektywy. Integracja taka będzie wymagać głównie przedefiniowania zestawu obiektów jakie powinny być widoczne w trakcie wykonywania operacji na wirtualnych obiektach. Być może konieczne okażą się zmiany w składni definiowania perspektyw, tak by w definicji móc zawrzeć informację, iż dana perspektywa dla przykładu dziedziczy po pewnej klasie jej zestaw metod. Relacje pomiędzy perspektywami a klasami i dynamicznymi rolami trzeba będzie także jakoś zapisać w składzie danych.

Ciekawym pomysłem wartym rozważenia są perspektywy parametryzowane. Perspektywa z parametrem działa w takim podejściu jak szablon, na podstawie którego można uzyskać całą rodzinę perspektyw. Parametr będzie wykorzystywany w trakcie obliczania procedury zwracającej ziarna wirtualnych obiektów. W obecnej implementacji procedura ta jest bezparametrowa. Parametrem może być dowolne zapytanie. Wynik takiego zapytania przekazywany jest do procedury poprzez włożenie zawierającej go sekcji na wierzch stosu środowisk. Zaraz po wykonaniu procedury obliczającej ziarna sekcja zawierająca parametr zdejmowana jest ze stosu. Parametr wykorzystywany jest tylko do obliczania tej procedury. To powinno wystarczyć. Gdyby jednak okazało się, iż potrzebne będzie użycie tego parametru w trakcie wykonywania procedur „*on_*", konieczne będzie zapisanie tego parametru w wirtualnym identyfikatorze i wkładanie go na stos środowisk podczas wykonywania tych procedur.

Przykładem na zastosowanie parametryzowanych perspektyw może być zaprezentowana w dodatku A perspektywa zwracająca średnie zarobki w poszczególnych działach. W perspektywie tej nazwa działu zapisana jest wewnątrz definicji perspektywy. W danym momencie perspektywa zwraca średnie zarobki z jednego, konkretnego działu. By uzyskać wirtualny obiekt ze średnią zarobków w innym dziale, konieczna jest zmiana wartości pomocniczego obiektu zapisanego w definicji perspektywy. Perspektywa ze stanem realizowałaby taką funkcjonalność automatycznie. Nazwa działu, dla którego w danym momencie miałyby być obliczone wirtualne dane przekazywana byłaby jako parametr przy każdym użyciu perspektywy.

W podobny sposób w systemie LoXiM zaimplementowany jest operator **virtualize as**. Operator ten umożliwia zwirtualizowanie obiektu zgodnie z definicją wskazanej perspektywy. Wirtualizowany obiekt przekazywany jest poprzez stos środowisk jako parametr dla procedury on_virtualize pobranej z odpowiedniej perspektywy. Procedura ta działa podobnie jak procedura zwracająca ziarna z tą różnicą, że zamiast wyszukiwać wszystkie obiekty spełniające wyrażony w definicji perspektywy warunek, jedynie sprawdza czy warunek taki spełnia przekazany jako parametr obiekt. Operator **virtualize as** wraz z odpowiednio sformułowaną procedurą *on_virtualize* może posłużyć do zasymulowania sparametryzowanej perspektywy. Podobnie zresztą jak użycie pomocniczego obiektu pozwoliło na taką symulację w omawianym przykładzie.

Ani operator **virtualize as**, ani pomocnicze obiekty zapisane w definicji perspektywy nie są jednak pełnym rozwiązaniem dla parametryzowanych perspektyw. Konieczna byłaby praca badawcza pozwalająca ustalić, jakie możliwości takie perspektywy przynoszą. Czy wystarczające

do zbudowania takich perspektyw są dotychczas zaimplementowane środki, czy konieczne jest dopisanie dedykowanego dla takiej funkcjonalności kodu.

Składniowo wirtualny obiekt generowany przez perspektywę z parametrem będzie prawdopodobnie bardziej przypominał wywoływanie procedury niż zwykły obiekt. Używanie takich obiektów nie będzie dla użytkownika przezroczyste. Zwykłe obiekty nie są parametryzowane i w obecnie zaimplementowanej w LoXiM składni języka SBQL nie ma konstrukcji pozwalającej na zapisanie takiego sparametryzowanego wirtualnego obiektu. Składnia *nazwa*(*parametr*) zarezerwowana jest już dla wołania procedur. Osoba implementująca parametryzowane perspektywy, będzie musiała opracować dla nich nowy fragment gramatyki języka.

Na zakończenie wspomnę jeszcze o możliwości rozszerzenia dotychczasowej implementacji perspektyw w systemie LoXiM o graficzny interfejs, wspomagający tworzenie i edycję perspektyw. Interfejs taki mógłby automatycznie generować standardowe wersje procedur. W większości przypadków, pewne operacje będą wyglądały identycznie w różnych perspektywach. Procedura *on_retrieve* na ogół zwracać będzie wynik dereferencji na ziarnie. Perspektywa *on_delete* będzie na ogół wykonywać instrukcję usuwania ziarna, Procedura *on_update* w większości wypadków podstawiać będzie swój parametr na ziarno. Takie standardowe części definicji perspektywy mogłyby być generowane automatycznie. Osoba definiująca perspektywy mogłaby wykorzystując taki dedykowany interfejs skupić się jedynie na definiowaniu ciekawszych, niestandardowych operacji.

Podsumowanie

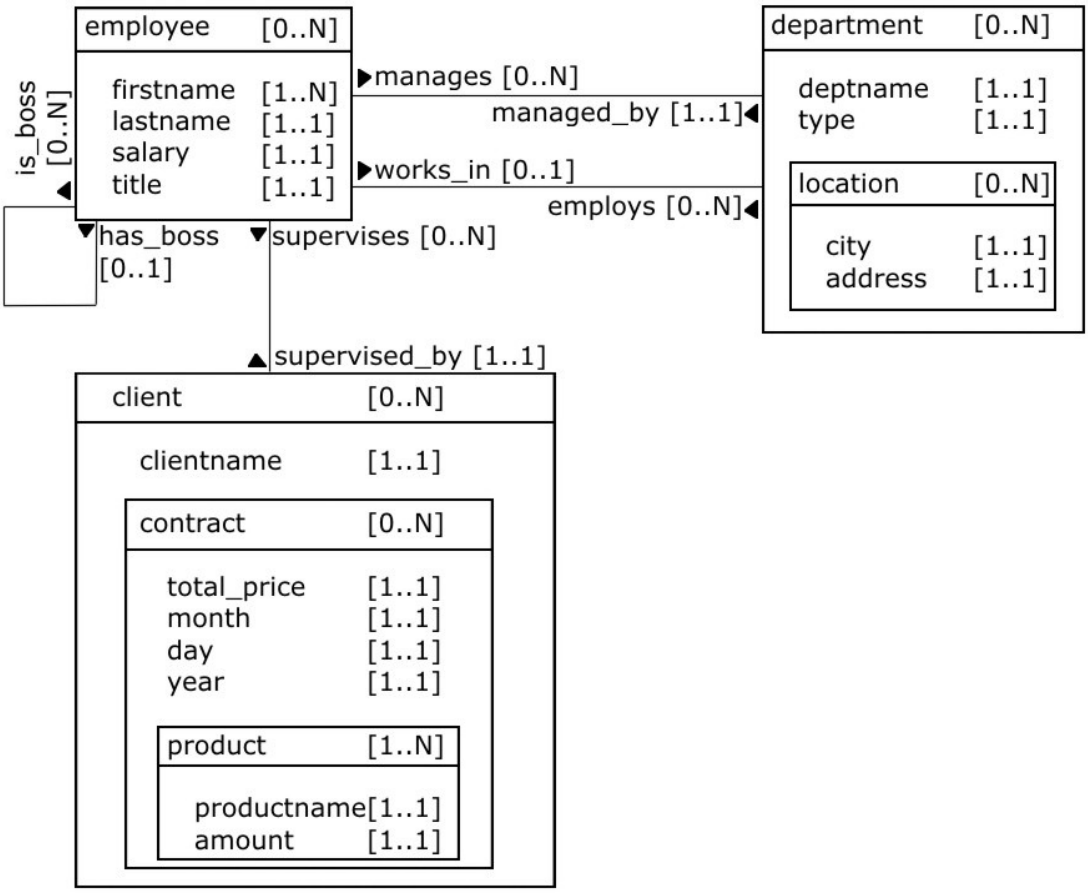
Głównym celem mojej pracy były zaimplementowanie aktualizowalnych perspektyw obiektów na bazie podejścia stosowego SBA. Implementacja powiodła się co dowodzi, iż teoria przedstawiona w pracy [Kozankiewicz04] jest implementowalna. Na bazie tej implementacji udało mi się przetestować działanie różnych perspektyw oraz aktualizowanie generowanych przez nie obiektów. Część z przykładowych pomysłów na wykorzystanie takich perspektyw przedstawiam w dodatku. Przykłady nie są abstrakcyjne. Każdy z nich realizuje pewną konkretną intencję aktualizacyjną dającą się łatwo uzasadnić biznesowo. W szczególności przy użyciu zaimplementowanych przez mnie perspektyw możliwe było wykonania takich aktualizacji wirtualnych obiektów, o jakich sądzono że są nie możliwe do wykonania. Dodatkowym celem mojej pracy było zmierzenie się z problemem wirtualnych wskaźników. Zaimplementowane przeze mnie wskaźniki można aktualizować. Mogą być wykorzystywane dokładnie w taki sam sposób jak zwykłe niewirtualne obiekty wskaźnikowe.

Wynikiem mojej pracy jest możliwość definiowania w systemie LoXiM perspektyw o pełnej mocy. Osoba tworząca perspektywę może zapisać w jej definicji dowolną intencję aktualizacyjną. Generowane przez perspektywę wirtualne obiekty są (prawie) w pełni przezroczyste dla użytkownika. Osoba definiująca perspektywę ma pełną kontrolę nad tym, co się stanie z obiektami, na podstawie których generowane są wirtualne obiekty, w momencie gdy te wirtualne obiekty są aktualizowane. Może się on zabezpieczyć przed niezamierzonymi skutkami aktualizacji. Poszczególne operacje definiowane są w postaci dowolnie złożonych procedur. Język SBQL, w którym te procedury są definiowana, ma pełną moc algorytmiczną. Nie ma żadnych ograniczeń co do możliwych zastosowań takich perspektyw. Perspektywy mogą wykorzystywać rekurencję, mogą mieć swoje lokalne środowisko, dodatkowe obiekty pozwalające zasymulować stan perspektywy. Aktualizacja wirtualnych danych może powodować dowolne uboczne efekty.

Dopracowania wymaga jedynie efektywność korzystania z wirtualnych obiektów. Konieczne jest zaimplementowanie różnego typu optymalizacji, dzięki którym użytkownik nie będzie odczuwał różnicy czasu wykonania zapytań operujących na wirtualnych i niewirtualnych obiektach.

Dodatek A
Przykłady zastosowań aktualizowalnych perspektyw

W prezentowanych przykładach wykorzystany będzie schemat danych zaprezentowany w rozdziale 2.1.1 niniejszej pracy. Przykładowy skład danych zawiera obiekty korzeniowe *employee*, *department* i *client*. Schemat przedstawia rysunek 2.1:



Rysunek 2.1: Schemat danych użyty w przykładach.

Przykład 1. Zmiana osoby nadzorującej danego klienta.

```
create view supervisionDef {
      virtual objects supervision() {return client as c}

      view clientDef {
            virtual objects client() {return c . clientname as cn}
            procedure on_retrieve() {return deref (cn)}
      }

      view supervisorDef {
            virtual objects supervisor() { return c .
supervised_by . employee as s }
            procedure on_retrieve() { return deref (s . lastname)
}
            procedure on_update(new_sup_name) {
                  (employee where lastname = new_sup_name)
                  as new_sup .
                  (delete (s . supervises) where client = c;
                  c . supervised_by := ref ( new_sup );
                  new_sup :< c as supervises)
      }
}
```

Powyższa perspektywa zwraca dla wszystkich klientów zapisanych w bazie ich nazwę oraz nazwisko pracownika opiekującego się danym klientem. Podstawienie na nazwisko spowoduje zmianę osoby wyznaczonej do opieki nad danym klientem. Na obiektach reprezentujących nazwę klienta i nazwisko jego opiekuna zdefiniowano operację dereferencji. Pozostałych operacji nie zdefiniowano, nie mogą być wykonywane na wirtualnych obiektach.

```
deref(supervision.(client, supervisor))
```

Powyższe zapytanie zwróci nazwy wszystkich klientów wraz z nazwiskami pracowników opiekujących się nimi.

```
deref((supervision where supervisor = "Dutesky" ).client)
```

To zapytanie zwróci nazwy wszystkich firm, którymi opiekuje się pracownik o nazwisku Dutesky.

```
(supervision where client = "PIP Institute").supervisor :=
"Black"
```

Po wykonaniu powyższej instrukcji opiekunem firm o nazwie *PIP Institute* stanie się pracownik o nazwisku *Black*. W składzie danych dokonana zostanie odpowiednia modyfikacja wskaźników. Dotychczasowy opiekun tego klienta straci wskaźnik *supervises* wskazujący na niego. Wskaźnik taki zyska za to pracownik o nazwisku *Black*. Dodatkowo zaktualizowany zostanie wskaźnik *supervised_by* w obiekcie client.

Podobną perspektywę przedstawiłem w rozdziale 1.1. Była to perspektywa dla pacjentów i opiekujących się nimi lekarzy. Wykonanie podstawienia na nazwisku lekarza miało zmienić danemu pacjentowi lekarza. W rzeczywistości jednak zmieniane było jedynie nazwisko obecnego lekarza. Taka anomalia aktualizacyjna, była powodem uznania podobnych perspektyw za niemożliwe do aktualizowania. Jak widać zaimplementowane przeze mnie perspektywy nie mają takiego ograniczenia. Powyższa sytuacja zabroniona jest nie tylko przez standard SQL. Także komercyjne systemy baz danych nie zezwalają na tego typu aktualizację perspektyw ze względu na zastosowany w definicji operator złączenia. Mimo to, powyższa perspektywa ma bardzo precyzyjną i jasną logikę biznesową i można przedstawić rozsądny sposób jej wykorzystania. Widać, że opisane przez mnie w pierwszym rozdziale kryteria, na podstawie których próbowano ustalać jakie perspektywy można aktualizować a jakich nie, są jedynie konsekwencją złego podejścia do tematu aktualizowania wirtualnych danych. Po zastosowaniu odpowiedniego podejścia żadne anomalie w opisanym przypadku nie występują.

Przykład 2. Aktualizacja wirtualnej danej zagregowanej.

W pierwszym rozdziale opisując różne proponowane kryteria wspomniałem o przyjętej powszechnie niemalże jak aksjomat zasadzie, według której niemożliwe jest aktualizowanie wirtualnych danych powstałych w wyniku zastosowania operatora agregującego. Problem z aktualizacją takich danych wynika z faktu, iż funkcja agregująca nie ma jednoznacznego odwrotnego odwzorowania. W przypadku tradycyjnego podejścia do definiowania perspektyw, twórca perspektywy nie ma środków, by jednoznacznie zdefiniować jaką intencje chciałby przypisać takiej aktualizacji. Aparat wykonawczy pozbawiony takiej informacji nie jest w stanie sam określić jakie operacje ma wykonać, by zrealizować zlecenie użytkownika zmiany wartości takiej wirtualnej danej. Dodatkową przeszkodą pojawiającą się w takiej sytuacji jest fakt, iż taka perspektywa zwraca w wyniku konkretną wartość zamiast referencji. Ponieważ nie ma referencji nie możliwa jest próba aktualizacji przez jakieś efekty uboczne. Perspektywy oparte o podejście SBA radzą sobie jednak z taką sytuacją bez żadnych problemów.

```
create view avgSalInDeptDef {
      virtual objects avgSalInDept() {
            return ((department where deptname =
            cur_dept).employs.employee.salary) group as tmp}
      procedure on_retrieve() {return avg (tmp)}
      procedure on_update(val) {
            if val >= avg (tmp) then
                  (count(tmp) as emps_count) .
                  (sum(tmp) as emps_sal_sum) .
                  (tmp as tmp) .
                  tmp := tmp * val * emps_count / emps_sal_sum
            else throw "WrongParameterException" fi }
      create „Retail sales” as cur_dept
}
```

Powyższa perspektywa zwraca średnią zarobków w jednym z działów firmy. Nazwa działu dla, którego średnią zarobków zwraca perspektywa zapisana jest w definicji perspektywy na pomocniczym obiekcie *cur_dept*. Wartość tego obiektu może być zmieniana. Po jej zmianie perspektywa będzie zwracać średnią z innego działu. Perspektywa generuje jeden atomowy

obiekt wirtualny reprezentujący średnią. Na ten obiekt można podstawić wartość. Podstawienie wartości mniejszej niż aktualna średnia dla danego działu spowoduje przerwanie zapytania w wyniku wyrzucenia wyjątku przez procedurę *on_update*. Gdy podstawiana na średnią wartość będzie większa od aktualnej, aparat wykonawczy zwiększy średnią zarobków w dziale poprzez podniesienie poszczególnym pracownikom pensji. W definicji procedury *on_update* zawarty jest przepis pozwalający wyliczyć, ile po zmianie powinna wynosić pensja każdego z pracowników. Definicja powyższej perspektywy zakłada proporcjonalny wzrost. Jeżeli średnia wzrasta o *x* procent, to zarobki każdego z pracowników także wzrastają o *x* procent. Osoba definiująca perspektywę może jednak zdefiniować dowolny sposób dystrybucji podwyżki pomiędzy pracowników danego działu. Ma ona pełną kontrolę nad sposobem w jaki aparat wykonawczy wykonuje poszczególne operacje na wirtualnym obiekcie.

Załóżmy, że następujące zapytanie:

```
deref((department where deptname = "Retail sales").
employs.employee.salary)
```

zwraca w wyniku zbiór liczb bag{3200, 2000, 6000, 4400, 2400}. Przy takich zarobkach pracowników działu "*Retail sales*" zapytanie:

```
deref(avgSalInDeptDef )
```

zwraca w wyniku 3600. Taka jest średnia zarobków w tym dziale. Po wykonaniu operacji:

```
avgSalInDeptDef := 4500
```

powyższe zapytanie zwracające zarobki poszczególnych pracowników działu "*Retail sales*" zwróci już zmienione wartości: bag{4000, 2500, 7500, 5500, 3000}. Średnia wzrosła do 4500, aktualizacja wirtualnego obiektu reprezentującego średnią zakończyła się poprawnie.

Przykład 3. Tworzenie złożonych wirtualnych obiektów.

```
create view virtual_employeeDef {
      virtual objects virtual_employee() {
            return employee as ve }

      view v_firstnameDef {
            virtual objects v_firstname() {
                  return ve.firstname as v }
            procedure on_retrieve() {return deref(v)}
      }
      view v_lastnameDef {
            virtual objects v_lastname() {
                  return ve.lastname as v }
            procedure on_retrieve() {return deref(v)}
      }
      view v_salaryDef {
            virtual objects v_salary() {
```

```
                return ve.salary as v }
            procedure on_retrieve() {return deref(v)}
        }
        view v_titleDef {
            virtual objects v_title() {
                return ve.title as v}
            procedure on_retrieve() {return deref(v)}
        }
        view v_works_inDef {
            virtual objects v_works_in () {
                return ve.works_in as v}
            procedure on_retrieve() {return deref(v)}
        }
        view v_supervisesDef {
            virtual objects v_supervises () {
                return ve.supervises as v }
            procedure on_retrieve() {return deref(v)}
        }
        procedure on_create(val) {
            if ((count(val.v_firstname) >= 1) and
            (count(val.v_lastname) = 1) and
            (count(val.v_salary) = 1) and
            (count(val.v_title) = 1) and
            ((count(val.v_works_in) = 0) or
            (count(val.v_works_in) = 1)) then
            (val.v_firstname as first) . (
                ((create (first as firstname, (val.v_lastname) as
                lastname, (val.v_salary) as salary, (val.v_title
                as title)) as employee)
                as new_emp) . (
                    for each ((val.v_firstname) minus first) as
                    next) do
                    new_emp :< next as firstname od;
                    for each (val.v_works_in as next) do
                    new_emp :< next as works_in od;
                    for each (val. v_supervises as next) do
                    new_emp :< next as supervises od;
                )
            ) fi
        }
}
```

Wirtualne obiekty *virtual_employee* generowane przez powyższą perspektywę mają wirtualne podobiekty *v_firstname*, *v_lastname*, *v_salary*, *v_title*, *v_works_in* i *v_supervises*. Dla wirtualnych obiektów zdefiniowano procedurę tworzenia oraz procedury dereferencji dla wszystkich wirtualnych podobiektów. Wewnątrz procedury *on_create* zapisana została logika, na podstawie której nowotworzony obiekt jest sprawdzany co do zgodności ze jego budowy ze schematem danych. Nowotworzony obiekt *virtual_employee* musi mieć dokładnie po jednym podobiekcie *v_lastname*, *v_title* i *v_salary*. Musi on mieć przynajmniej jeden podobiekt *v_firstname*. Może mieć co najwyżej jeden podobiekt *v_works_in* oraz dowolnie wiele podobiektów *v_supervises*. Polecenie utworzenia nowego wirtualnego obiektu jest tłumaczone na

utworzenie obiektu rzeczywistego. Obiekt zostanie dodany do składu jedynie wówczas, gdy jego definicja spełnia te warunki. Procedura *on_create* zdefiniowana dla powyższej perspektywy działa następująco. Najpierw tworzony jest obiekt *employee* wraz ze wszystkich obligatoryjnymi podobiektami. Następnie dodawane są te obiekty, które oprócz tego że są obowiązkowe to jeszcze mogą być wielokrotne. Na koniec wstawiane są do nowoutworzonego obiektu wszystkie obiekty zdefiniowane jako opcjonalne.

```
create ("Dariusz" as v_firstname, "Gryglas" as v_lastname, 9000
as v_salary, "Consultant" as v_title, (department where deptname
= "IT") as v_works_in) as virtual_employee
```

Powyższe zapytanie spowoduje wstawienie do składu danych obiektu:

```
<i306, employee {
      <i301, firstname, "Dariusz">,
      <i302, lastname, "Gryglas">,
      <i303, salary, 9000>,
      <i304, title, "Consultant">,
      <i305, works_in, i35>
}>
```

Przykład 4. Wykorzystanie perspektyw ze stanem i z pomocniczymi procedurami

```
create view favoritesDef {
      virtual objects favorites() {
            return distinct(favorite_client.client) as f
      }

      procedure on_retrieve() {return deref(f)}

      procedure add_favorite(name) {
            favoritesDef :<
            (client where clientname = name) as favorite_client
      }
      procedure remove_favorite(name) {
            delete favoritesDef.favorite_client
            where client.clientname = name
      }

}
```

Powyższa perspektywa przechowuje informacje o ulubionych klientach. Perspektywa ma dwie pomocnicze procedury. Procedura *add_favorite* dodaje do grupy ulubionych klienta o wskazanej nazwie. Procedura *remove_favorite* usuwa z grupy ulubionych klienta o danej nazwie. Dane o ulubionych klientach przechowywane są w postaci obiektów wskaźnikowych wewnątrz definicji perspektywy. Ulubienie klienci widoczni są jako wirtualne obiekty *favorites*. Na obiektach *favorites* można wykonywać dereferencję.

By dodać do grupy ulubionych klienta o nazwie *PIP Institute* należy wykonać następującą instrukcję:

```
favoritesDef.addFavorite("PIP Institute")
```

Bibliografia

[AKSS08] R. Adamus, K. Kaczmarski, K. Stencel, K.Subieta. SBQL Object Views - Unlimited Mapping and Updatability. ICOODB, Berlin, Niemcy 2008

[Codd74] E. F. Codd. Recent investigations In a relational database systems. Information Processing, Vol.74 (Proc. IFIP Congres Stockholem), North Holland, 1017-1021, 1974

[HKKS04] P. Habela, K. Kaczmarski, H. Kozankiewicz, M. Lentner, K. Stencel, K. Subieta. Data-Intensive Grid Computing Based on Updatable Views. IPI PAN, Warszawa 2004.

[Humpich08] M. Humpich. Implementacja kontroli typów w systemie LoXiM. Warszawa 2008

[KLS02] H. Kozankiewicz, J. Leszczylowski, K. Subieta. Updatable Object Views. IPI PAN, Warszawa 2002

[KLS03] H. Kozankiewicz, J. Leszczylowski, K. Subieta. Updatable XML Views. ADBIS, Dresden, Niemcy 2003

[Kozankiewicz04] H. Kozankiewicz. Updateable Object Views. IPI PAN, Warszawa 2004

[LoXiM] Strona systemu LoXiM: http://loxim.mimuw.edu.pl/

[PH02] R. Hryniów, T. Pieciukiewicz. A Stack-Based XML Query Language. PJWSTK, Warszawa 2002

[Płodzień00] J. Płodzień. Optimization Methods in Object Query Languages. IPI PAN, Warszawa 2000

J. Sitek. Implementacja optymalizacji zapytań w systemie LoXiM, Warszawa 2007

[SBA] Strona poświęcona SBA i SBQL: http://www.sbql.pl/

[sql92] ISO/IEC 9075:1992 Standard języka SQL

[Subieta04] K. Subieta. Teoria i konstrukcja obiektowych języków zapytań. Wydawnictwo PJWSTK, Warszawa 2004

[Turski08] P. Turski. Implementacja indeksów w systemie LoXiM. Warszawa 2008