Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jakub Sitek
Nr albumu: 209298

Implementacja optymalizacji zapytań w systemie LoXiM

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Stencla
Instytut Informatyki

Wrzesień 2007

Oświadczenie kierującego pracą

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

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

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

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

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

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiam opis zaimplementowanych przeze mnie optymalizacji zapytań opartych na przepisywaniu. Optymalizacje zostały wdrożone do języka SBQL. Jest on językiem zapytań do obiektowej semistrukturalnej bazy danych LoXiM opartej na podejściu stosowym autorstwa profesora Krzystofa Subiety.

Słowa kluczowe

baza danych, parser, optymalizacje, przepisywanie zapytań, LoXiM, sbql, metabaza

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2. Database Management
H.2.8. Database Applications

Tytuł pracy w języku angielskim

An implementation of a query optimizations in the LoXiM system

Spis treści

Wprowadzenie

LoXiM jest obiektową semistrukturalną bazą danych rozwijaną w ramach seminarium z Obiektowych Baz Danych oraz Systemów Zarządzania Bazami Danych prowadzonych przez dr Krzysztofa Stencla na Wydziale Matematyki Inforamtyki i Mechaniki Uniwersytetu Warszawskiego. Językiem zapytań w LoXiMie jest *SBQL - Stack Based Query Language*. W ramach seminarium magisterskiego zajmowałem się implementacją optymalizacji zapytań. Przedmiotem tej pracy jest opis zaimplementowanych w LoXiMie metod optymalizacji opartych na przepisywaniu zapytań.

W pierwszym rozdziale omówię od strony teoretycznej zaimplementowane przeze mnie metody - wyłączania niezależnych podzapytań przed operator niealgebraiczny, usuwania martwych podzapytań, usuwania zbędnych nazw pomocniczych oraz zastąpienia złączenia nawigacyjnego operatorem kropki. W drugim rozdziale opiszę czym jest metabaza czyli statyczny odpowiednik składowiska danych, format i sposób fizycznego przechowywania metadanych w bazie danych oraz jak wykorzystuję je w implementacji wyżej wymienionych metod optymalizacji. W rozdziale trzecim opiszę algorytm statycznej ewaluacji oraz używane w nim struktury danych. Ostatni rozdział poświęcony jest szczegółom mojej implementacji. W dodatku A przedstawiony jest kod źródłowy zapytań tworzących przykładowy schemat danych. Na tym schemacie testowałem moją implementację. Na nim także opierają się przykładowe zapytania podawane w tej pracy.

Rozdział 1

Metody optymalizacji

1.1. Wstęp

W tym rozdziale opiszę poszczególne metody oparte na przepisywaniu zapytań, oraz wytłumaczę sposób ich działania na przykładach. Ten rozdział opracowałem na podstawie [Subieta01]

1.2. Wyłączanie niezależnych podzapytań przed operator niealgebraiczny

1.2.1. Wstęp

Podzapytania SBQL są nazywane niezależnymi, jeżeli mogą być ewaluowane na zewnątrz pętli implikowanych przez ich niealgebraiczny operator. Takie podzapytania warte są analizy, ponieważ zwykle niosą możliwość optymalizacji. Używamy specjalnej techniki zwanej metodą niezależnych podzapytań do optymalizacji zapytań, które zawierają niezależne części. Technicznie metoda ta polega na analizowaniu, w których sekcjach są wiązane poszczególne nazwy występujące w zapytaniu. Okazuje się, że jeżeli żadna z nazw w danym podzapytaniu nie jest wiązana w przestrzeni otwieranej przez obecnie ewaluowany operator niealgebraiczny, wtedy to podzapytanie może zostać zewaluowane wcześniej, niż to wynika z jego tekstowego położenia w zapytaniu którego jest częścią. Ta metoda modyfikuje tekstową postać zapytania (bez zmieniania jego rezultatu) w ten sposób, że wszystkie jego podzapytania są ewaluowane tak wcześnie jak to tyko możliwe.

Według semantyki SBQL w czasie wykonania, każdy niealgebraiczny operator występujący w zapytaniu otwiera swoją własną przestrzeń na stosie środowisk i każda nazwa tego zapytania jest wiązana w pewnej sekcji tego stosu. Ponieważ zapytanie możne być wywołane w dowolnym środowisku (tzn. przy dowolnym stanie stosu środowisk stanu wykonania, ograniczonym przez schemat bazy danych), nie jest możliwe dokładne określenie, w których sekcjach jego poszczególne nazwy będą wiązane, za każdym razem, gdy jest ewaluowane. Dla rożnych ewaluacji mogą to być sekcje na rożnych poziomach stosu, ponieważ liczba i zawartość sekcji bazowych (sekcji które są na stosie środowisk w momencie wywołania zapytania) może być rożna dla rożnych wykonań zapytania. Aby poradzić sobie z tym problemem, podzielimy wszystkie nazwy występujące w zapytaniu na dwie grupy: nazwy wiązane w przestrzeniach bazowych oraz poza nimi. Dla naszej metody nie jest ważne, w której sekcji należącej do przestrzeni bazowej jest wiązana nazwa, ponieważ jedyną własnością takich nazw używaną przez tę metodę jest fakt, że są one wiązane w sekcjach bazowych. Dlatego więc bez straty ogólności traktujemy wszystkie sekcje należące do przestrzeni bazowej jako jedną sekcję.

Przy tym założeniu możemy określić, w której sekcji niebazowej względem bazowej są wiązane nazwy. Aby określić, w którym obszarze nazwy występujące w zapytaniu są wiązane w czasie wykonania, dokonujemy statycznej analizy zapytania (w sposób opisany w rozdziale opisującym statyczną ewaluację) i w jej trakcie dodatkowo:

- każdemu operatorowi niealgebraicznemu przypisujemy numer obszaru, który otwiera
- każdej nazwie przypisujemy dwie liczby:
  - rozmiar stosu - liczba sekcji na statycznym stosie środowisk ES w momencie wiązania danej nazwy
  - numer sekcji - numer sekcji na statycznym stosie środowisk ES w której jest wiązana dana nazwa

Wyciąganie niezależnych podzapytań ma swoje korzenie w optymalizacji zagnieżdżonych zapytań w modelu relacyjnym, ale jest bardziej ogólne. Następujący przykład ilustruje ogólną ideę naszej metody. Poniższe zapytanie zwraca osoby, które są starsze niż Nowak :

1.2.2. Prosty przykład

Z *1*

Manager ***where*** *salary* $\geqslant$ *((Manager* ***where*** *name = 'Nowak').salary)*

Zauważmy, że podzapytanie obliczające zarobki kierownika o nazwisku Nowak:

Z *2*

(Manager ***where*** *name = 'Nowak').salary*

jest ewaluowane dla każdego obiektu Manager z bazy danych, tzn. dla każdego identyfikatora zwróconego przez lewe podzapytanie zewnętrznego operatora *where*. Jednakże wystarczy wykonać zapytanie (2) tylko raz, gdyż jego ewaluacja zwraca za każdym razem ten sam wynik.

Przyjrzyjmy się dokładniej zapytaniu (1). Rozmiary stosu, numery sekcji dla nazw i numery sekcji otwierane przez operatory niealgebraiczne występujące w tym podzapytaniu są pokazane poniżej:

Manager where salary $\geqslant$ ((Manager where name = 'Nowak').salary)
(1,1) 2 (2,2) (2,1) 3 (3,3) 4 (3,3)

Jak widzimy, żadna z nazw występujących w podzapytaniu (2) nie jest wiązana w sekcji 2 otwieranej przez zewnętrzny operator *where*, ich numery sekcji wynoszą 1 i 3, co oznacza, że są wiązane albo w sekcji pierwszej albo w sekcji trzeciej. Nazywamy to zapytanie niezależnym od jego bezpośredniego (tzn. najbliższego) operatora niealgebraicznego. Dla zapytania (i) mówimy, że podzapytanie (ii) jest niezależne od zewnętrznego operatora where; jego właściwością jest to, że nie używa sekcji włożonych przez ten operator na stos środowisk ES, tzn. żadna z jego nazw nie jest wiązana w tych sekcjach.

Ponieważ podzapytanie (2) jest niezależne od operatora *where*, może być obliczone zanim ten operator otworzy swoje środowisko na stosie. Jest tak dlatego, że rezultat tego podzapytania po takiej transformacji jest dokładnie taki sam jak przed nią. Dzięki temu możemy wyciągnąć niezależne podzapytanie przed operator niealgebraiczny i zmienić tekstową formę zapytania. W tym celu robimy co następuje:

- Po pierwsze wprowadzamy nową, unikalną nazwę pomocniczą w celu nazwania rezultatu tego podzapytania. (Nazwa musi być unikalna, aby jej wprowadzenie nie zmieniło rezultatu całego zapytania.)
- Następnie niezależne zapytanie jest nazywane tą nazwą, wyciągane przed całe zapytanie i łączone z nim za pomocą operatora projekcji (kropka) (aby włożyć rezultat na stos środowisk ES).
- Na koniec pomocnicza nazwa jest wstawiana w początkowe miejsce niezależnego podzapytania.

Po wyciągnięciu podzapytania (2) zapytanie (1) ma postać:

Z *3* *(((Manager where name = 'Nowak').salary) group as s).(Manager where salary ⩾ s)*

gdzie *s* jest pomocniczą nazwą wprowadzoną przez metodę . Teraz podzapytanie (2) jest ewaluowane zanim jego wynik jest użyty, a pomocnicza nazwa *s* nazywająca jego wynik sprawia, że można go później odczytać. Zwróćmy uwagę, że w zapytaniu (3) mamy teraz dwa niezależne podzapytania:

- Drugie podzapytanie *Manager* jest niezależne od drugiej kropki.
- Podzapytanie *s* jest niezależne od operatora *where* otwierającego sekcję 3.

Jednakże nie należy ich faktoryzować.

1.2.3. Kiedy nie faktoryzować

Ważnym przypadkiem, gdy niezależne podzapytanie nie może być faktoryzowane, jest podzapytanie składające się z pojedynczej nazwy. Pierwszą, nie krytyczną przyczyną, jest to że pomocnicza nazwa wprowadzona dla wyniku wiązania niezależnego podzapytania składającego się z pojedynczej nazwy byłaby ewaluowana tę samą ilość razy co podzapytanie przed faktoryzacją. Ponieważ transformacja powoduje pewne dodatkowe operacje, w tym wypadku metoda ta, co prawda tylko nieznacznie, ale zmniejszyłaby wydajność. Krytycznym powodem, dla którego nie możemy wyciągać podzapytań składających się z pojedynczej nazwy jest to, że proces ten nigdy by się nie zakończył. Jest tak dlatego, że nowa pomocnicza nazwa wprowadzona przez metodę jest niezależna od wszystkich tych operatorów niealgebraicznych, od których jest niezależne podzapytanie którego rezultat ona nazywa. Każde kolejne zastosowanie metody faktoryzowałoby taką pojedynczą nazwę i wprowadzało kolejną pomocniczą.

Wyciąganie pojedynczej nazwy można stosować w przypadku gdy nazwa dotyczy podprogramu, ponieważ wtedy faktoryzacja zmniejsza ilość (możliwe że kosztownych) wykonań.

1.2.4. Bardziej skomplikowane przykłady

Weźmy zapytanie zwracające projekty i liczbę wykonujących je pracowników, których imię jest takie samo jak imię kierownika kierującego tym projektem:

Z *4*

Project join count((done_by.Emp) where name = managed_by.Manager.name)
(1.1) 2 (2,2) 3 (3,3) 3 (3,3) (3,2) 4 (4,4) 4 (4,4)

Zauważmy, że gdy zaczyna się ewaluacja podzapytania zwracającego imię kierownika kierującego projektem

Z *5 managed_by.Manager.name*

na stosie środowisk ES znajdują się dwie sekcje niebazowe: pierwsza jest otwierana przez operator złączenia nawigacyjnego *join*, druga przez operator *where*. Podzapytanie (5) jest niezależne od operatora *where*, ponieważ wszystkie nazwy występujące w nim są wiązane w sekcji 2 lub 4, a operator ten otwiera sekcję 3. Zależy ono od operatora *join*, który otwiera 2 sekcję (jest w niej wiązana nazwa managed_by). Zapytanie (5) może być zatem ewaluowane zanim operator selekcji włoży swoją sekcję na stos, ale nie przed włożeniem na stos sekcji przez operator złączenia nawigacyjnego. Optymalizujemy zapytanie (4) przez wyciągnięcie (5) przed operator *where*:

```
Project join count(((managed_by.Manager.name) group as n).((done_by.Emp) where name=n))
  (1,1)   2                (2,2)      3   (3,3) 3  (3,3)              3   (3,2) 4 (4,4)    4     (4,4)(4,3)
```

Zwróćmy uwagę, że w tym przypadku niezależne podzapytanie nie zostało wyciągnięte przed całe zapytanie, a jedynie przed lewe podzapytanie operatora niealgebraicznego, od którego jest niezależne. Jest to ogólna zasada, a wyciąganie przed całe zapytanie jak w (1) i (3) jest tylko szczególnym jej przypadkiem.

Rozpatrzmy zapytanie zwracające projekty (nazwane pomocniczą nazwą *p*) i tych pracowników pracujących przy nich, którzy są starsi od kierujących nimi kierowników:

```
(Project group as p) join (p . (done_by.(Emp where age ⩾ (p . managed_by.Manager.age))))
  (1,1)                2 (2,2) 3   (3,3) 4  (4,4)   5     (5,5)(5,2) 6   (6,6)      6 (6,6) 6 (6,6)
```

analizując sekcje, w których wiązane są nazwy widzimy, że: *p.managed_by.Manager.age* jest niezależne nie tylko od operatora *where* dodającego sekcję 5 stosu ES, ale także od niealgebraicznych operatorów otwierających sekcje 4 i 3, a zależy od operatora złączenia nawigacyjnego. Tak więc podzapytanie to może być wyciągnięte przed te wszystkie trzy operatory niealgebraiczne:

```
(Project group as p) join
  (1,1)                2
(((p . managed_by.Manager . age) group as a) . ( l . (attended_by.(Student where age ⩾ a))))
(2,2)3  (3,3)     3   (3,3)   3 (3,3)             3(3,2)4         (4,4) 5    (5,5)     6 (6,6) (6,3)
```

1.2.5. Kiedy nie da się faktoryzować

Wyciąganie przed nawias nie zawsze jest możliwe. Rozpatrzymy następujące zapytanie zwracające kierowników mających najwyższe zarobki w działach:

```
(Manager group as m) where m . salary=max((Manager where works_in.Dept =
    (1,1)                  2   (2,2)3  (3,3)            (2,1)     3       (3,3)   4  (4,4)
                                                                    m . works_in.Dept) . salary)
                                                                    (3,2) 4 (4,4) 5 (5,5) 3 (3,3)
```

Podzapytanie
max((Manager where works_in.Dept = m . works_in.Dept) . salary)
nie jest niezależne od zewnętrznego operatora *where*, gdyż występująca w nim nazwa *m* jest wiązana w drugiej sekcji stosu.

1.2.6. Wyciąganie przed dowolny operator

Wyciąganie niezależnych podzapytań może być zastosowane dla dowolnego operatora niealgebraicznego (nie tylko dla operatora selekcji *where*). W SBQL wszystkie operatory niealgebraiczne działają w bardzo podobny sposób, w szczególności wszystkie wykonują pętlę, przed którą wyciąganie jest istotą tej metody. Na przykład zapytanie:
(Project group as p) where (((p.done_by.Emp) group as e) for_each
(e.age ⩾ p.managed_by.Manager.age))
może zostać przepisane do postaci:
(Project group as p) where (((p.managed_by.Manager.age) group as a).(((p.done_by.Emp) group as e) for_each(e.age ⩾ a)))

1.2.7. Ogólna postać

Ogólna zasada naszej metody faktoryzacji niezależnych podzapytań może być sformułowana następująco. Rozpatrzmy zapytanie:

$q_1\ \Theta\ q_2$

gdzie Θ jest dowolnym operatorem niealgebraicznym, a q_1 i q_2 są podzapytaniami. Niech q_2 ma postać:

$\alpha_1\ q_3\ \alpha_2$

gdzie q_3 jest dowolnym syntaktycznie poprawnym podzapytaniem połączonym z resztą q_2 dowolnymi operatorami:

$q_1\ \Theta\ (\alpha_1\ q_3\ \alpha_2)$

Jeżeli q_3 jest niezależne od operatora Θ i wszystkich innych (jeżeli takie są) operatorów niealgebraicznych których sekcje są na stosie środowisk ES w momencie rozpoczęcia ewaluacji podzapytania q_3, wtedy zapytanie to może być przepisane do postaci:

$(q_3$ *group as x).(* $q_1\ \Theta\ (\alpha_1\ x\ \alpha_2))$

Co więcej, jeżeli q_1 nie jest zapytaniem składającym się z pojedynczej nazwy (lub jest nazwą podprogramu) wtedy nasza metoda zwiększa wydajność i dlatego powinna zostać zastosowana do optymalizacji tego zapytania.

1.3. Usuwanie martwych podzapytań

1.3.1. Wstęp

Martwe podzapytanie jest to taki fragment zapytania, który nie ma żadnego wpływu na jego końcowy rezultat. Usunięcie całego martwego podzapytania wraz z operatorami łączącymi je z reszta zapytania nie wpływa na wynik końcowy zapytania. Martwe podzapytania mogą pojawić się w wyniku stosowania perspektyw rozwijanych w miejscu wywołania, na których stosuje się dalsze modyfikacje używające jedynie części zwracanego przez nie wyniku. Mogą być również skutkiem wielokrotnych przeróbek programów w wyniku których końcowa intencja zapytania różni się od początkowej, ale o tej początkowej programista już nie pamięta. Inną przyczyną możne być automatyczne generowanie zapytań (np. przez interfejs graficzny) które przewiduje pewien ogólny sposób generowania zapytań, ale w konkretnej sytuacji niektóre elementy tak wygenerowanych zapytań nie są używane.

W tym paragrafie opiszę metodę optymalizacji polegającą na usuwaniu martwych podzapytań. Pokażę motywujące przykłady, zapytania z martwymi podzapytaniami, które można usunąć oraz przykłady pozornie martwych podzapytań, których jednak nie można usunąć ze względu na ich liczność. Opiszę sposób wyszukiwania i usuwania martwych podzapytań.

1.3.2. Przykłady

Rozpatrzmy zapytanie zwracające imiona kierowników:

Z *6 (Manager join (works_in.Dept)).name*

W zapytaniu tym kosztowny operator złączenia nawigacyjnego *join* działa na podzapytanie

Z *7 (works_in.Dept)*

ale wynik tego podzapytania nie jest potem używany. Końcowy wynik wyznacza podzapytanie *name* połączone operatorem kropki, a to podzapytanie korzysta wyłącznie z wyniku podzapytania *Manager*. Zatem podzapytanie *(works_in.Dept)* oraz operator *join* można w tym przypadku usunąć, optymalizując to zapytanie do postaci:

Z *8 Manager.name*

Rozpatrzmy zapytanie:

Z *9 Manager.(name group as managerName, (salary/avg(Manager.salary) * 100) group as avgProc, (works_in.Dept.name) as deptName)*

Jeżeli chcielibyśmy poznać jedynie imiona kierowników, moglibyśmy używając powyższego zapytania uzyskać zapytanie:

Z *10 Manager.(name group as managerName, (salary/avg(Manager.salary) * 100) group as avgProc, (works_in.Dept.name) as deptName). managerName*

Liczenie podzapytań *avgProc* oraz *deptName* jest tutaj zbędne, gdyż końcowy wynik od nich nie zależy. Jednocześnie policzenie tych podzapytań może być bardzo kosztowne, gdyż pierwsze z nich wymaga policzenia średnich zarobków, a w tym celu trzeba odczytać z dysku wszystkie obiekty *Manager.salary*. Zatem możemy usunąć te martwe podzapytania nie zmieniając ostatecznego wyniku przez przepisanie zapytania do postaci:

Z *11 Manager.(name group as managerName).managerName*

1.3.3. Zapytania pozornie martwe

Rozpatrzmy zapytanie zwracające nazwiska kierowników projektów:

Z *12 (Manager join manages.Project).name*

rezultat podzapytania

Z *13 manages.Project*

nie jest wykorzystywany, jednakże nie może zostać ono usunięte i nie możemy przepisać zapytania do postaci:

Z *14 Manager.name*

gdyż podzapytanie (13) ma liczność rożną od 1 wiec wpływa na liczność wyniku końcowego, tym samym zmienia liczność całego zapytania. Zapytania (13) i (14) nie są syntaktycznie równoważne.

1.3.4. Algorytm metody

Zauważmy że:

- Powodem powstawania martwych podzapytań są te operatory, które dają w wyniku działania struktury. Tylko dla nich funkcja nested sumuje bindery pochodzące z ich części. W języku SBQL mamy dwie konstrukcje dające w wyniku struktury: operator *join* oraz operator przecinka , . Jeżeli pewne zapytanie nie ma operatora generującego struktury, to nie może mieć martwych podzapytań. Dla operatora *join* martwe może być tylko podzapytanie z jego prawej strony. Dla operatora przecinka , dowolny jego składnik może być potencjalnie martwy.

- Tylko niektóre operatory niealgebraiczne mogą prowadzić do martwych podzapytań, mianowicie, operator kropki oraz kwantyfikatory. Tylko te operatory działające na strukturach mogą nie wykorzystać niektórych ich składników. Jeżeli pewne zapytanie po operatorze generującym struktury nie ma operatora kropki lub kwantyfikatora, to nie może mieć martwych podzapytań.

- Zapytanie może być pozornie martwe, gdyż jego wynik może wpływać na liczność wyniku końcowego. Na przykład, jeżeli w zapytaniu (12) założylibyśmy, że obiekt pointerowy *works_in* jest opcjonalny dla obiektów *Manager* (niektórzy kierownicy mogą nie być przypisani do działów), wówczas podzapytanie (*works_in.Dept*) wpłynie na liczność wyniku, powodując, że (12) i (14) nie będą semantycznie równoważne.

Uwzględniając powyższe obserwacje przystąpimy do opisu budowy grafu wynikania. Jest on potrzebny do zbadania, czy dane podzapytanie nie jest martwe. Graf wynikania nakłada się na drzewo syntaktyczne zapytania - jego węzłami są węzły drzewa syntaktycznego. Chodzi o wiązanie nazw i wskazanie podzapytania w wyniku realizacji, którego pewna nazwa mogła być związana (mówimy wtedy, że dana wynika z podzapytania). Założenia grafu wynikania są następujące:

- Strzałki oznaczające wynikanie prowadza od podzapytania będącego rezultatem wynikania do podzapytania, z którego ten rezultat wynika.

- Dla operatorów algebraicznych strzałki wynikania prowadzą od wyniku operatora do wszystkich jego argumentów.

- Podobnie dla wszystkich operatorów niealgebraicznych z wyjątkiem operatora kropki i kwantyfikatorów.

- Dla operatora kropki i kwantyfikatorów strzałkę wynikania rysujemy tylko od operatora do jego prawej strony, lewą stronę pomijamy.

- Dla każdej nazwy występującej w drzewie zapytania rysujemy strzałkę wynikania od tej nazwy do podzapytań, z których ta nazwa wynika. Wyznaczanie strzałek wynikania dla nazw odbywa się poprzez modyfikacje procedury statycznej ewaluacji. Każdy binder na stosie środowisk ES wskazuje na węzeł, z którego wynika (może wynikać tylko z jednego węzła, może być to tylko węzeł z nazwą lub operator group as). Przy wiązaniu nazwy na stosie środowisk ES z każdego statycznego bindera, w którym ta nazwa się związała odwzorowuje wynikanie tego bindera na wynikanie dla nazwy, która jest w nim wiązana. Zauważmy, że strzałka wynikania z nazwy może prowadzić tyko do innego węzła z nazwą lub do węzła operatora group as. Zauważmy także, że z jednego węzła z nazwą

może wychodzić wiele strzałek wynikania. Dla nazw wiązanych w sekcjach bazowych nie rysujemy strzałek wynikania.

- Dla każdego podzapytania wyznaczana jest liczność wyniku. Liczność ustala w dużym przybliżeniu ile elementów może dane podzapytanie odłożyć na stosie wyników QRES. Liczność jest obliczana w trakcie statycznej ewaluacji. Na bazie liczności zawartych w metabazie obliczamy liczność wyników poszczególnych podzapytań i umieszczamy w węzłach drzewa syntaktycznego. (Możliwe wartości liczności opisane są w rozdziale dotyczącym metabazy.) Dla każdego operatora języka można określić prostą arytmetykę i za jej pomocą określać generowaną liczność. Usuwać można tylko podzapytania o liczności 1. Tylko w tym przypadku mamy gwarancję, że usunięcie tego podzapytania nie zmieni liczności końcowego wyniku.

Po wyznaczeniu grafu wynikania dalsze postępowanie następuje wyłącznie na drzewie syntaktycznym zapytania i składa się z następujących kroków:

- Zaczynając od korzenia drzewa poruszamy się po drzewie zgodnie z kierunkiem strzałek wynikania, markując wszystkie węzły znajdujące się na ścieżkach wynikania
- Zaczynając od każdego markowanego węzła idziemy w górę drzewa (do ojca) markując wszystkie napotkane po drodze węzły.
- Węzły które pozostały niezamarkowane reprezentują potencjalnie martwe podzapytania. Idąc od góry drzewa syntaktycznego usuwamy wszystkie poddrzewa, które nie są zamarkowane i dla których liczność wynosi 1.
- Po usunięciu węzłów konieczne jest usunięcie węzłów operatora *join*, o ile jego prawy argument został w całości usunięty. Węzeł ten usuwa się w naturalny sposób, poprzez przełączenie pointerów w drzewie syntaktycznym. Analogicznie usuwa się operator przecinka , . W jego wypadku mógł zostać usunięty lewy lub prawy argument.

1.4. Usuwanie zbędnych nazw pomocniczych

1.4.1. Wstęp

Nazwy pomocnicze są bardzo użytecznym narzędziem przy tworzeniu zapytań. Umożliwiają nazywanie rożnych części wyniku zapytania, aby później mieć dostęp dokładnie do tych części których potrzebujemy. Dostajemy się do nich poprzez projekcję na odpowiednią nazwę pomocniczą. Dogłębna analiza tych dwóch operacji (tzn. nazwania części wyniku pomocniczą nazwą i późniejsza projekcja na tą nazwę) pokazuje, że często są one wykonywane niepotrzebnie, ponieważ moglibyśmy dostać się do tego częściowego wyniku bezpośrednio. Daje to nam kolejna możliwość optymalizacji - usuwanie zbędnych nazw pomocniczych. Problem nie jest jednak tak prosty, ponieważ okazuje się, że w pewnych przypadkach usunięcie nazwy pomocniczej zmienia wynik całego zapytania, przez co nie może zostać wykonane. Zbędną nazwą pomocniczą nazywamy pomocniczą nazwę wprowadzoną przez operator *group as*, której usunięcie nie zmienia wyniku całego zapytania. W tym rozdziale omówię przypadek takich nazw.

1.4.2. Przykłady

Rozpatrzymy zapytanie zwracające kierowników zarabiających powyżej 3000 i nazywające wynik pomocniczą nazwą m:

Z *15 (Manager where salary ⩾ 3000) group as m*

Rezultatem tego zapytania są bindery z nazwą m i wartością id odpowiedniego obiektu Manager. Jeżeli chcielibyśmy dostać się do identyfikatorów obiektów Manager używając powyższego zapytania, otrzymalibyśmy zapytanie:

Z *16*

((Manager where salary ⩾ 3000) group as m).m
(1,1) 2 (2,2) 2 (2,2)

Jak widzimy pomocnicza nazwa *m* w zapytaniu spełnią dwie funkcje:

- najpierw nazywa wynik podzapytania *(Manager where salary ⩾ 3000)*
- następnie nawiguje do tego wyniku.

Zauważmy, że wynik zapytania (16) nie zmieni się, jeżeli usuniemy pomocnicza nazwę *m* (przez usunięcie nazwy pomocniczej rozumiemy usunięcie jej definicji za pomocą operatora group as oraz wszystkich jej użyć). Zapytanie (16) może zostać przepisane do postaci:
Manager where salary ⩾ 3000

Innym przypadkiem jest sytuacja, w której pomocnicza nazwa umożliwia dalsza nawigację. Używając zapytania (16) możemy pobrać również atrybuty obiektów, które zwraca (poniższe zapytanie pobiera imię i wiek kierowników zarabiających powyżej 3000):

Z *17*

((Manager where salary ⩾ 3000) group as m) . (m .name, m . age)
(1,1) 2 (2,2) 2 (2,2) 3(3,3)(2,2)3 (3,3)

Jak widać atrybuty obiektu *Manager* są dostępne bezpośrednio, a wiec nie ma potrzeby używania nazwy pomocniczej:
(Manager where salary ⩾ 3000).(name, age)

Przykład z wielokrotnym wykorzystaniem nazwy pomocniczej:
((Emp group as e) where e.age ⩽ 20).e
zapytanie to może zostać przepisane do postaci:
Emp where age ⩽ 20

Poniższy przykład pokaże sytuacje, w której nazwa pomocnicza nie może zostać usunięta. Zapytanie zwraca dla każdego pracownika jego imię i imiona kierowników, których projekty wykonuje:

Z *18*

Emp.(does.Project.managed_by.(Manager group as m).(name, m . name))
(1,1)2(2,2)3(3,3)3 (3,3) 3 (3,3) 3(3,2)(3,3)4(4,4)

Zauważmy, że atrybut *name* jest atrybutem zarówno obiektu *Emp* jak i *Manager*. W podzapytaniu:

(name, m.name)

pierwsza nazwa *name* jest wiązana w sekcji otwartej dla wyniku podzapytania *Emp*, a druga jest wiązana w sekcji otwartej dla wyniku podzapytania *Manager group as m* (z powodu nawigacji do nazwy *m*). Jeśli usunęlibyśmy pomocniczą nazwę *m*:

Z *19*

Emp.(does.Project.managed_by.(Manager).(name, name))
(1,1)2(2,2)3(3,3)3 (3,3) 3 (3,3) 3 (3,3) (3,3)

wtedy obydwie nazwy *name* byłyby wiązane w tej samej sekcji (otwieranej dla wyniku podzapytania *Emp*). Zapytanie (19) zwraca dla każdego pracownika zduplikowane imiona jego kierowników, nie zwraca imienia pracownika. Zapytania (18) i (19) nie są semantycznie równoważne, co oznacza, że nie możemy wykonać takiego przekształcenia.

W poniższych dwóch przykładach również nie można usunąć nazw pomocniczych:

Z *20 (Manager join (age group as a)).a*

nie byłoby równoważne z

(Manager join age)

a zapytanie zwracające sumę zbiorów kierowników i pracowników:

Z *21 ((Manager group as m) join (Emp group as e)).e*

przepisane do

Manager join Emp

zwracałoby iloczyn kartezjański.

1.5. Zastąpienie złączenia nawigacyjnego operatorem kropki

1.5.1. Opis

Idea tej metody jest następująca: jeżeli bindery z pewnego obiektu będące częścią sekcji wkładanej na stos środowisk ES przez operator kropki nie są używane, to nie ma potrzeby aby były tam wkładane. Dzięki takiej optymalizacji stos zużywa mniej pamięci, a ewaluacja jest wykonywana szybciej.

Rozpatrzmy zapytanie zwracające projekt wraz z kierownikiem nim kierującym:

Z *22 Project join managed_by.Manager*

jeżeli przy pomocy tego zapytania chcielibyśmy uzyskać jedynie imię kierownika kierującego projektem otrzymalibyśmy:

(Project join managed_by.Manager).name

Ponieważ nazwa *name* ma znaczenie jedynie w kontekście nazwy *Manager*, i nie ma żadnego znaczenia w kontekście nazwy *Project*, końcowa nawigacja odnosi się tyko do prawego argumentu operatora *join*. Oznacza to, że podczas ewaluacji nawigacji nie potrzebujemy dostępu do wyniku lewego argumentu operatora *join*. Tak wiec przy wiązaniu nazwy *name* wystarczy,

jeżeli na stosie środowisk ES będą jedynie bindery pochodzące z wyniku prawego argumentu operatora *join*. W SBQL uzyskujemy to zastępując operator złączenia nawigacyjnego operatorem kropki i przepisując zapytanie do postaci:

(Project . managed_by.Manager).name

teraz podczas ewaluacji nazwy *name* najwyższa sekcja stosu środowisk ES zawiera jedynie bindery pochodzące z obiektu *Manager*.

Weźmy zapytanie zwracające nazwy działów w których pracują pracownicy:

Z *23 (Emp join does.Project join managed_by.Manager join works_in.Dept).dname*

jest ono równoważne z :

(Emp join does.Project) join managed_by.Manager join works_in.Dept).dname

podzapytanie *dname* korzysta jedynie z wyniku podzapytania *works_in.Dept*, co oznacza, że możemy przepisać zapytanie do postaci:

(((Emp join does.Project) join managed_by.Manager) . works_in.Dept).dname

podzapytanie *works_in* korzysta jedynie z wyniku podzapytania *managed_by.Manager*, więc możemy wykonać transformację do postaci:

(((Emp join does.Project) . managed_by.Manager) . works_in.Dept).dname

postępując dalej w ten sam sposób otrzymujemy ostateczną postać:

(((Emp . does.Project) . managed_by.Manager) . works_in.Dept).dname

która jest równoważna z:

Emp join does.Project. managed_by.Manager.works_in.Dept.dname

Rozdział 2

Metabaza

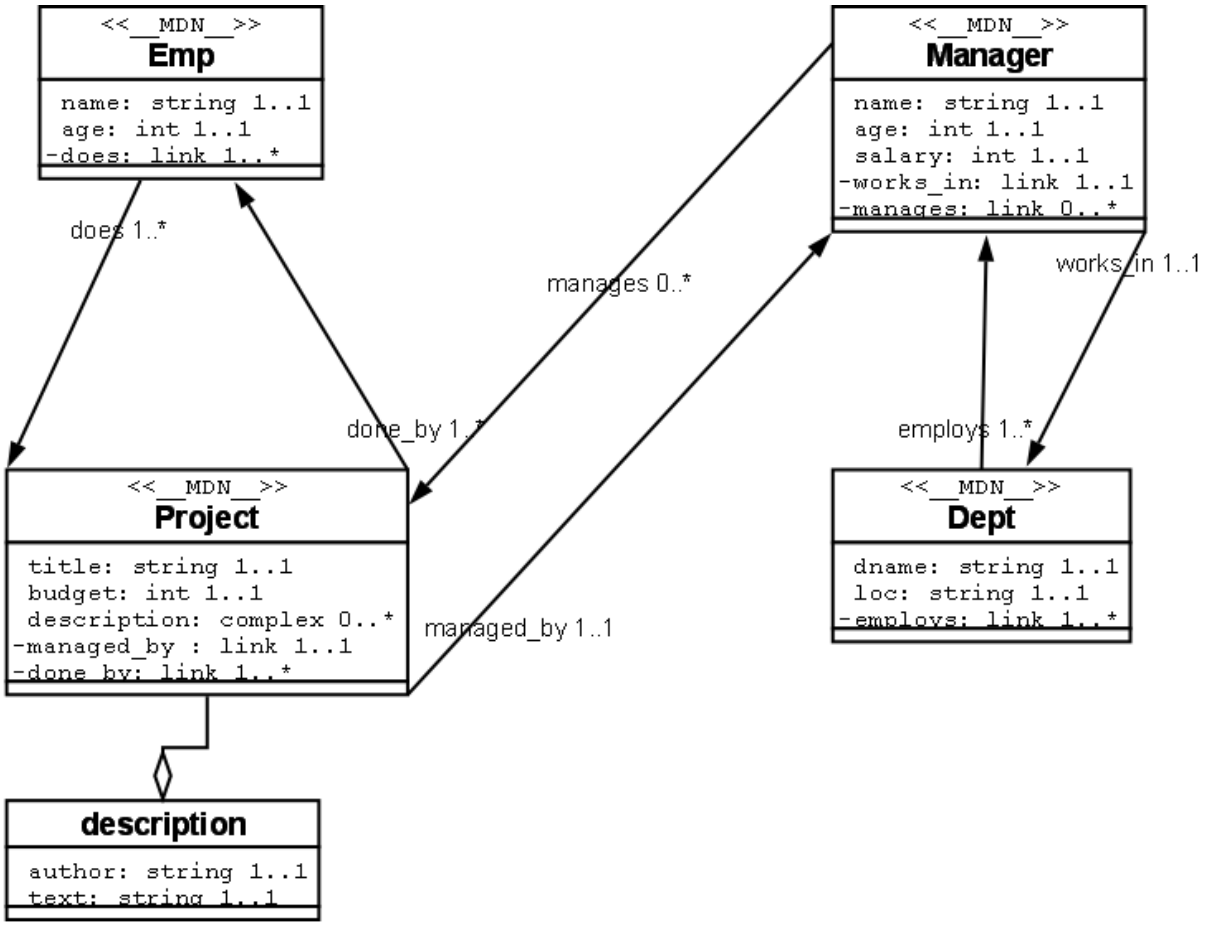
2.1. Wstęp

Ten rozdział opracowany jest na podstawie [Stencel01] oraz [Subieta01]. Metabaza jest statycznym odpowiednikiem składowiska danych. Jej głównym składnikiem jest graf schematu bazy danych. Poza grafem schematu metabaza może zawierać także inne dane takie jak: liczba instancji danej klasy, definicje indeksów, statystyki dostępu. Graf schematu powstaje w wyniku kompilacji schematu danych i modeluje statycznie skład danych. Węzły grafu zawierają definicje bytów znajdujących się w bazie danych (obiektów, metod i procedur, perspektyw). Krawędzie grafu modelują powiązania między tymi definicjami. Dla naszych potrzeb wystarczające są powiązania:

- *owner* (linia ciągła) pomiędzy podobiektem obiektu złożonego, a obiektem złożonym, którego jest atrybutem
- *target* referencja do wskazywanego obiektu.

2.2. Przechowywanie i wykorzystywanie metadanych

Metabaza, czyli schemat danych jest przechowywany w taki sam sposób jak wszystkie inne informacje w bazie danych tzn. jest zbiorem obiektów. Pozwala to na używanie tego samego języka zapytań (SBQL) do przetwarzania zarówno danych jak i metadanych. Jest to zgodne z dobrymi praktykami projektowania środowisk programowania dla baz danych i systemów zarządzania bazami danych (patrz postulaty Codda). Metabazę wyróżnia to, że jej obiekty bazowe muszą mieć konkretną, ustaloną nazwę, ustalone podobiekty ustalonego typu, etc. - czyli format, opisany w kolejnym podrozdziale. Optymalizator, chcąc korzystać z metabazy, ma do dyspozycji odpowiednie struktury w pamięci, które odzwierciedlają tą metabazę. Skąd ma te struktury? Serwer na początku sesji wywołuje odpowiednią metodę optymalizatora (dając mu dostęp do składu danych), która wyłuskuje z bazy schemat danych zwykłymi zapytaniami i wczytuje go do pamięci - zapamiętuje w odpowiednich strukturach i optymalizator podczas statycznej kompilacji zapytania korzysta właśnie z tych pomocniczych struktur. W związku z tym, że ze względów optymalizacyjnych schemat ładowany jest raz na początku sesji i nie jest automatycznie odświeżany przy modyfikacjach, zmiana schematu wymusza restart sesji.



Rysunek 2.1: Powyższy rysunek przedstawia przykładowy schemat danych. Wszystkie zapytania z tej pracy dotyczą tego właśnie schematu. Kod źródłowy którego wynikiem kompilacji jest powyższy schemat zawarty jest w dodatku A.

2.3. Format obiektów metabazy

W samej bazie danych (do której chcemy stworzyć metabazę) są rożnego rodzaju obiekty. Obiekty atomowe, wskaźnikowe, złożone. (W przyszłości również mogą to być: procedury, metody, klasy, perspektywy, ale tym się na razie nie zajmujemy). W zależności od tego jaki rodzaj obiektu opisuje dana jednostka metabazy, rożna będzie jej budowa. Poniżej znajdują się przykłady obiektów ze schematu danych, w zależności od tego jaki byt opisują:

- Opisujące obiekt atomowy. (atomic)

```
<id1, __MDN__ ,
 {
  <id2, name, "nazwaDzialu">  /* nazwa opisywanego obiektu              */
  <id3, kind, "atomic">       /* rodzaj obiektu - tu atomowy            */
  <id4, type, "string">       /* tu może być jeszcze "int" lub "double" */
  <id5, card, "1..1">         /*  liczebność, patrz opis niżej          */
  <id6, owner, id18>          /* ref. do nadobiektu, czyli do obiektu
                            złożonego, dla którego ten jest podobiektem. */
  /* jeśli dany obiekt jest bazowy,                                      */
  /* to tego podobiektu Owner w ogóle nie ma.                            */
 }
>
```

- Opisujące obiekt złożony. (complex)

```
<id1, __MDN__ ,
 {
  <id2, name, "Worker">       /* nazwa opisywanego obiektu              */
  <id3, kind, "complex">      /* rodzaj obiektu - tu złożony            */
  <id4, card, "0..*">
  <id6, owner, id18>          /* ref. do nadobiektu, czyli do obiektu
                            złożonego, dla którego ten jest podobiektem  */
  <id6, subobject, id30>      /* obiekty wskaźnikowe                    */
  <id7, subobject, id36>      /* referencje do podobiektów              */
  <id8, subobject, id22>
 }
>
```

- Opisujące obiekt wskaźnikowe. (link)

```
<id1, __MDN__ ,
 {
  <id2, name, "WorksIn">      /* nazwa opisywanego obiektu              */
  <id3, kind, "link">         /* rodzaj obiektu - tu wskaźnikowy        */
  <id4, card, "1..*">
  <id5, target, id20>         /* referencja do wskazywanego obiektu     */
  <id6, owner, id18>          /* ref. do nadobiektu, czyli do obiektu
                           złożonego, dla którego ten jest podobiektem */
 }
>
```

2.4. Card czyli liczebność obiektu

Ten atrybut opisuje ile jednostek takiego obiektu może się znajdować w bazie. W metabazie jest on napisem, a rożne wartości oznaczają:

- "0..1" (obiekt opcjonalny, może mieć 0 lub 1 instancje)
- "1..1" (obiekt pojedynczy, nieopcjonalny - wiadomo, że jest 1 inst.)
- "0..*" (obiekt wielokrotny, opcjonalny - 0, 1 lub wiele instancji)
- "1..*" (obiekt wielokrotny, nieopcjonalny - przynajmniej 1 instancja)

Domyślne wartości:
Jeśli nie zdefiniowane przez tworzącego metabazę użytkownika, przyjmowane są następujące wartości:

- card : "1..1"
- owner: domyślnie go w ogóle nie ma (domyślnie obiekty są bazowe).

Rozdział 3

Statyczna ewaluacja

3.1. Wstęp

Statyczna analiza jest podstawą metod optymalizacji opartych na przepisywaniu zapytań. W trakcie statycznej analizy generowane jest drzewo syntaktyczne zapytania, które jest potem modyfikowane przez metody opisane w rozdziale 1.

3.2. Struktury danych

Poniżej znajduje się opis podstawowych struktur danych używanych podczas statycznej ewaluacji zapytania. Są one statycznymi odpowiednikami struktur używanych przez Executor.

3.2.1. Metabaza

Jest statycznym odpowiednikiem składowiska danych. Głównym jej składnikiem jest graf schematu bazy danych. Została szczegółowo opisana w rozdziale 2.

3.2.2. Statyczne stosy

Zadaniem statycznych stosów jest dokładne zasymulowanie działania ewaluacji zapytań podczas ich kompilacji. Statyczny stos środowisk ES jest odpowiednikiem stosu środowisk czasu wykonania, a statyczny stos wyników QRES jest odpowiednikiem stosu wyników używanego przez Executor. Statyczny stos środowisk ES składa się z sekcji. Nowe sekcje kładzione są na stos przez operatory niealgebraiczne. Sekcja na stosie ES jest kolekcją statycznych binderów. Statyczny binder jest odpowiednikiem binderów czasu wykonania. **Statyczne bindery** są postaci:

- n(typ): odpowiada sytuacji gdy na stosie ES ma pojawić się binder n(v), gdzie n ϵ N, v ϵ V, *typ* jest typem wartości v
- n(ref):odpowiada sytuacji gdy na stosie ES ma pojawić się binder n(i), gdzie i ϵ I jest identyfikatorem obiektu składu danych. W tym wypadku *ref* jest referencją do metabazy, identyfikatorem tego jej węzła, który w czasie wykonania odpowiada identyfikatorowi *i*.
- n(sygn) gdzie sygn ϵ *Sygnatura* (zdefiniowana poniżej): odpowiada sytuacji gdy na stosie ES ma pojawić się binder n(rezultat) gdzie *rezultat* jest wynikiem dowolnego zapytania.

Elementami stosu wyników QRES są sygnatury. Sygnatura jest statycznym odpowiednikiem rezultatu dowolnego zapytania czasu wykonania, czyli tego co może się znaleźć na stosie QRES Executora. **Definicja** zbioru **sygnatur**:

- każdy typ elementarny należy do zbioru Sygnatura
- każda referencja do węzła grafu schematu należy do zbioru Sygnatura
- każdy statyczny binder należy do zbioru Sygnatura
- kolekcja sygnatur jest sygnaturą, (kolekcje są wynikiem operatora *join* lub *przecinek*(,))

3.2.3. Drzewo syntaktyczne zapytania

Drzewo syntaktyczne zapytania jest strukturą danych powstałą w wyniku rozbioru gramatycznego (parsingu) zapytania. Zawiera węzły odpowiadające wszystkim semantycznie istotnym elementom zapytania, powiązanym w taki sposób, aby oddać budowę zapytania, z pominięciem syntaktycznego lukru. Drzewo zapytania po opuszczeniu parsera jest daną wejściową dla Executora. Optymalizacje przez przepisywanie polegają na zmianie drzewa syntaktycznego. Aby optymalizacje były możliwe, węzły drzewa poza informacjami ważnymi z punktu widzenia Executora, rozszerzane są o pewne dane używane jedynie przez parser. Są to:

- Numer otwieranej na stosie sekcji dla operatorów niealgebraicznych.
- Wysokość stosu w momencie wiązania oraz numer sekcji, w której jest wiązana nazwa dla węzłów z nazwą. Powyższe dane używane są przez metodę niezależnych podzapytań. Numer sekcji, w której jest wiązana nazwa może zostać wykorzystany również przez Executor - nie musi on przeszukiwać całego stosu środowisk, może od razu przejść do sekcji w której nazwa jest wiązana.
- Dla węzłów z nazwą wektor boundIn węzłów NameNode lub GroupAsNode, z których pochodzą statyczne bindery, w których dana nazwa jest wiązana, oraz symetrycznie, wektor usedBy węzłów NameNode, przez które są używane statyczne bindery pochodzące z danego węzła. Te dwa wektory są używane przez metody usuwania martwych podzapytań oraz zbędnych nazw pomocniczych.

Dane te obliczane są w trakcie statycznej analizy zapytania.

3.3. Statyczna analiza

Podczas statycznej analizy zapytania nie jest znany rzeczywisty stan składu danych. Podstawą analizy jest graf schematu, traktowany w podobny sposób jak skład danych. Wynikiem statycznej ewaluacji zapytania jest jego sygnatura. Celem statycznej analizy jest uzupełnienie drzewa syntaktycznego zapytania o dodatkowe dane opisane w rozdziale 3.2.3 wykorzystywane przez część prasera odpowiedzialną za optymalizacje zapytania oraz przez Executor do przyspieszenia jego wykonania.

3.3.1. Wiązanie nazw

Mechanizm wiązania pozwala nam określić znaczenie każdej z nazw. Wiązanie nazwy *n* polega na szukaniu sekcji statycznego stosu środowisk zawierającej statyczny binder *n(x)*. Przeszukiwanie stosu zaczynamy od jego wierzchołka - czyli ostatnio dodanej sekcji, i jeżeli w danej sekcji nie znajdziemy bindera *n(x)* to przechodzimy do kolejnej sekcji. Wynikiem wiązania nazwy *n* jest sygnatura *x* bindera *n(x)*. Jeżeli dana sekcja stosu zawiera kilka binderów *n(x1) n(x2) n(x3)*...to wynikiem związania nazwy *n* w takiej sekcji będzie kolekcja *x1, x2, x3*... W podejściu stosowym na początku stos ES zawiera jedną sekcję zawierającą statyczne bindery dla wszystkich obiektów bazowych. W trakcie ewaluacji zapytania stos środowisk rośnie i kurczy się zgodnie z zagnieżdżeniem zapytań, ale na koniec jego stan jest taki sam jak przed wykonaniem zapytania.

3.3.2. Funkcja static_nested

Jest odpowiednikiem czasu kompilacji dla funkcji nested używanej przez Executor. Buduje nowe sekcje, które są wkładane na statyczny stos środowisk ES, podobnie do funkcji nested budującej sekcje wkładane na stos środowisk czasu wykonania. Główną różnicą jest, że nie używa składu danych ale metabazy. Dla każdego (pod)obiektu (a dokładniej dla jego definicji w schemacie danych) tworzy jeden statyczny binder bez względu na liczność tego (pod)obiektu.

3.3.3. Algorytm statycznej analizy zapytania

Procedura static_eval jest odpowiednikiem czasu kompilacji procedury eval wykonywanej przez Executor. Jest w stosunku do niej znacznie uproszczona. Ogólny opis statycznej ewaluacji (szczegóły znajdują się w rozdziale 4):

- Nie występują w niej pętle implikowane przez operatory niealgebraiczne.
- Analizę zapytania przeprowadza się na drzewie syntaktycznym zapytania.
- Na początku statyczny stos środowisk ES zawiera jedną sekcję ze statycznymi binderami do definicji obiektów bazowych (root).
- Sygnaturą literału występującego w zapytaniu jest jego oczekiwany typ. Sygnaturę tę wstawia się na statyczny stos środowisk QRES.
- Sygnaturą nazwy n występującej w z zapytaniu jest rezultat statycznego wiązania na stosie środowisko ES. Poszukiwany jest tam poczynając od czubka stosu statyczny binder n(i). W rezultacie na stos QRES wstawiana jest sygnatura i. Dodatkowo węzeł drzewa syntaktycznego zawierający aktualnie przetwarzaną nazwę n zostaje uzupełniony o aktualny rozmiar stosu środowisk ES, numer sekcji, w której ta nazwa została związana oraz o węzły, z których pochodzą statyczne bindery, w których ta nazwa została związana (wektor dependsOn).
- Jeżeli zapytanie q zwraca sygnaturę s, to zapytanie q group as n zwraca na stos QRES sygnaturę n(s).
- Dla operatorów algebraicznych sygnaturę wyznaczamy na podstawie operatora oraz sygnatur jego argumentów.
- Podobnie dla operatorów niealgebraicznych. Tu dodatkowo zwiększa się wysokość stosu środowisk ES.

3.3.4. Przykład statycznej analizy

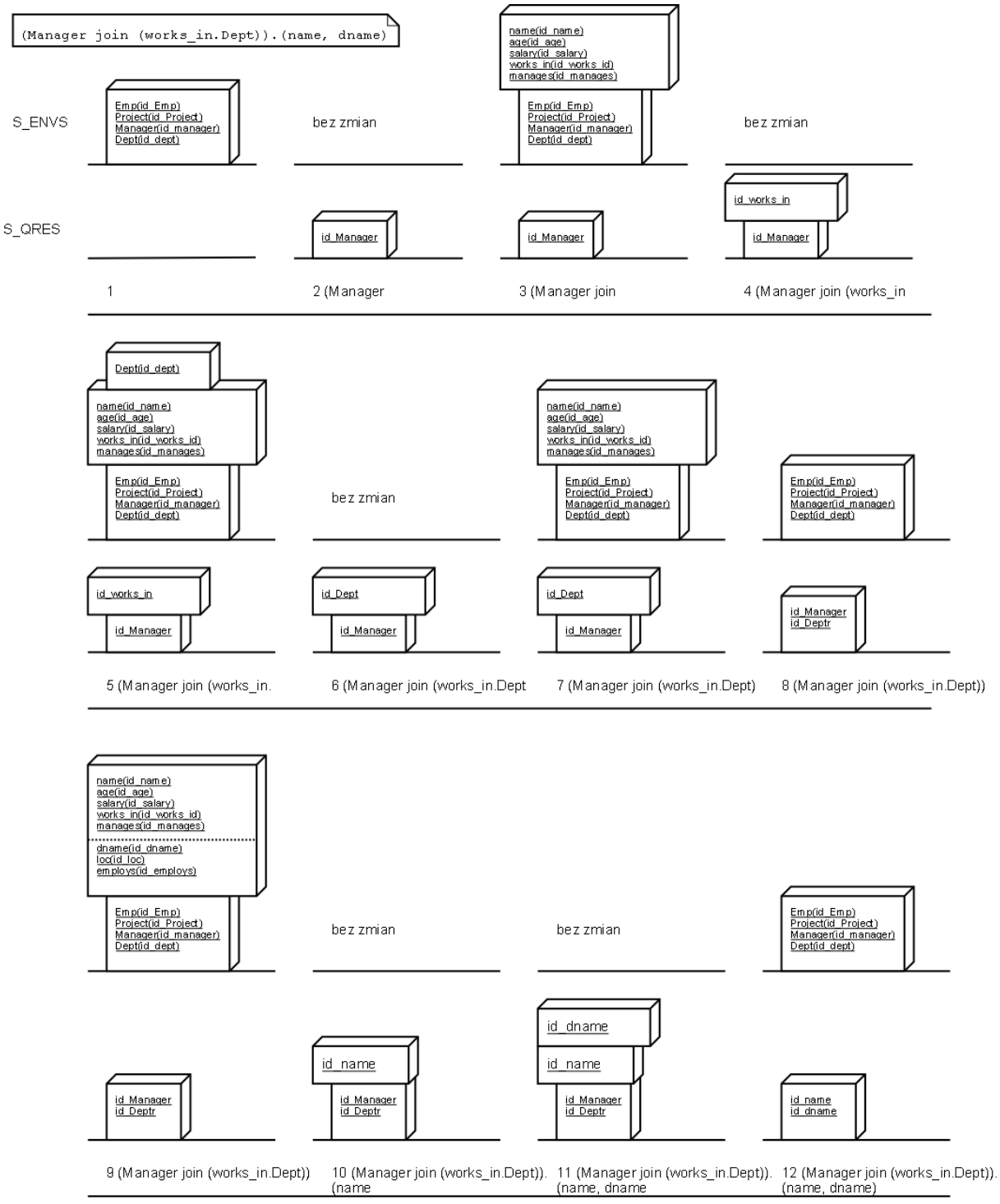
Rozpatrzmy zapytanie:

Z *24 (Manager join (works_in.Dept)).(name, dname)*

Na rysunku 4.2.3 przedstawiam 12 kroków ewaluacji tego zapytania. W każdym kroku podaję dotychczas zewaluowane zapytanie, stan statycznego stosu rezultatów S_QRES i stan statycznego stosu środowisk S_ENVS.

1. Stan przed rozpoczęciem ewaluacji.
2. Wiązana jest nazwa *Manager*, wstawiana sekcja na S_QRES, węzeł drzewa syntaktycznego zawierający nazwę *Manager* jest uzupełniany informacją o wysokości stosu i numerze sekcji, w której wiązana jest nazwa oraz wypełniam wektory boundIn i usedBy.
3. Operator join otwiera na S_ENVS sekcję o nr 2, do jego węzła w drzewie syntaktycznym jest wpisywany przedział <2, 2>.
4. Wiązana jest nazwa *works_in*, jej węzeł drzewa syntaktycznego jest uzupełniany o parę <2,2>, uzupełniane wektory boundIn i usedBy.
5. Operator kropki otwiera na S_ENVS sekcję o numerze 3, do jego węzła syntaktycznego jest wpisywany przedział numerów <3,3>.
6. Wiązana jest nazwa *Dział*, jej węzeł drzewa syntaktycznego jest uzupełniany o parę <4,4>, uzupełniane wektory.
7. Zmniejszenie stosu S_ENVS wskutek zakończenia podzapytania.
8. Zmniejszenie stosu S_ENVS wskutek zakończenia podzapytania.
9. Operator kropki otwiera na S_ENVS sekcję o numerze 2, wkładane są do niej statyczne bindery pochodzące z obiektów *Manager* oraz *Dept*, węzeł drzewa syntaktycznego odpowiadający temu operatorowi kropki jest uzupełniany o przedział numerów <2,2>.
10. Nazwa *Name* jest wiązana w drugiej sekcji stosu czego skutkiem jest zapełnienie S_QRES, oraz uzupełnienie jej węzła drzewa syntaktycznego o przedział <2,2> oraz jego wektorów boundIn i usedBy.
11. Nazwa dname jest wiązana w drugiej sekcji stosu czego skutkiem jest zapełnienie S_QRES, oraz uzupełnienie jej węzła drzewa syntaktycznego o przedział <2,2> oraz jego wektorów boundIn i usedBy .
12. Koniec ewaluacji zapytania, S_ENVS powraca do stanu początkowego, S_QRES zawiera sygnaturę wyniku.

Na potrzeby samego parsera i jego optymalizacji pierwsza liczba z pary liczb <c,d> wpisywanych do węzłów niealgebraiczne wydaje się zbędna. Jednak nasze dotychczasowe rozważania są względne - nie uwzględniają faktycznej wysokości stosu traktując wszystkie dolne sekcje jako jedną. Executor mógłby efektywnie wykorzystać liczbę d w celu uniknięcia przeszukiwania stosu podczas wykonania, ale względność naszego algorytmu byłaby wtedy przeszkodą. Dzięki liczbie c wiemy, jak daleko od wierzchołka stosu znajduje się sekcja, w której ma nastąpić wiązanie.


(Manager join (works_in.Dept)).(name, dname)
S_ENVS
S_QRES
Emp(id_Emp)
Project(id_Project)
Manager(id_manager)
Dept(id_dept)
bez zmian
name(id_name)
age(id_age)
salary(id_salary)
works_in(id_works_id)
manages(id_manages)
id_Manager
id_works_in
1
2 (Manager
3 (Manager join
4 (Manager join (works_in
Dept(id_dept)
id_Dept
id_Manager
id_Deptr
5 (Manager join (works_in.
6 (Manager join (works_in.Dept
7 (Manager join (works_in.Dept)
8 (Manager join (works_in.Dept))
dname(id_dname)
loc(id_loc)
employs(id_employs)
id_dname
id_name
9 (Manager join (works_in.Dept))
10 (Manager join (works_in.Dept)). (name
11 (Manager join (works_in.Dept)). (name, dname
12 (Manager join (works_in.Dept)). (name, dname)

Rozdział 4

Implementacja i problemy

4.1. Struktury danych

Klasa Signature - jest odpowiednikiem sygnatury czasu wykonania, jest podstawową abstrakcyjną klasą o wielu podklasach. Sygnatury znajdują się zarówno na statycznym stosie rezultatów jak i środowisk (statyczne bindery). Jej podklasy to:

- SigRef - sygnatura z referencja do obiektu bazodanowego,
- SigAtom - sygnatura odpowiadająca obiektowi atomowemu (int, double, string, bool),
- SigColl - sygnatura będąca kolekcją sygnatur, powstaje w wyniku działania operatorów przecinka lub złączenia nawigacyjnego na inne sygnatury, lub w wyniku działania funkcji staticNested na obiekt złożony, zaimplementowana jako lista z dowiązaniami
- StatBinder - statyczny binder, odpowiednik bindera czasu wykonania, składa się z nazwy oraz wartości którą jest inna sygnatura

Klasa DataObjectDef - implementuje definicję pojedynczego obiektu bazowego w pamięci parsera

Klasa DataScheme - jest implementacją grafu schematu, w jej skład wchodzi kolekcja obiektów DataObjectDef wszystkich obiektów bazowych schematu bazy danych.

Statyczne stosy - dziedziczą po jednej wspólnej nadklasie QStack - klasa ta implementuje interfejs typowy dla stosu, jej podklasy StatEnvStack i StatQResStack dodają funkcjonalność typową dla statycznego stosu środowisk i rezultatów. Zostały zaimplementowane od początku przez nas w postaci łączonych list - nie użyliśmy niestety dostępnych gotowych rozwiązań z biblioteki STL. Sekcjami na stosie środowisk są obiekty klasy StatEnvSection, każdy taki obiekt zawiera listę statycznych binderów oraz wskaźnik na kolejną sekcję. Elementami stosu rezultatów są obiekty klasy StatQResElt. Podobnie do sekcji stosu środowisk zawierają wskaźnik do kolejnego elementu na stosie, a ich właściwą zawartość stanowi wskaźnik do sygnatury. Elementy stosów podobnie jak stosy dziedziczą po wspólnej nadklasie - ListQStackElem.

Uważam, że implementacja stosów i ich elementów nie wyszła tak dobrze jak by mogła. Zostały one od podstaw zaimplementowane przez nas w postaci łączonych list, nie zastosowaliśmy niestety gotowych dostępnych bibliotek. W dodatku zdarzało się nam w jednej klasie implementować część funkcjonalności typowego stosu - czyli np. dodawanie elementu

do stosu oraz logikę typową dla stosu środowisk jak funkcja bindName. Uważam, że nawet dokonując własnej implementacji stosów i ich elementów lepiej byłoby wydzielić całkowicie funkcjonalność stosów do osobnych klas. Wydaje mi się, że jeszcze lepsze byłoby użycie gotowych kolekcji z STL'a i udekorowanie ich metodami specyficznymi dla stosów środowisk i rezultatów. W chwili obecnej dziwię się i żałuję, że tego nie zrobiliśmy. Żałowałem i ponosiłem skutki tego wielokrotnie, gdy w późniejszych etapach rozwoju musiałem modyfikować i rozwijać te fragmenty kodu. Dlaczego tak zrobiliśmy? Przypuszczalnie najlepiej brzmiałyby wytłumaczenia:

- Chcieliśmy zrobić to bardzo wydajnie. Rzeczywiście udało się to nam, jednak myślę, że STL byłby wystarczająco wydajny.
- Chcieliśmy mieć pełną kontrolę nad tworzonym kodem, móc go dowolnie rozwijać i dodawać dowolne metody i funkcjonalności. Tu jednak skutek wyszedł trochę odwrotny do zamierzonego, kod stał się mało czytelny, bardzo mocno uzależniliśmy się od naszej implementacji, w wielu miejscach, mimo że nie powinno się tak pisać, są wykorzystywane jej szczegóły.

Wydaje mi się jednak, że faktycznymi przyczynami podjętych decyzji były:

- Brak znajomości STL'a. Zamiast poświęcić trochę czasu na jego zapoznanie zabraliśmy się za wynajdywanie koła na nowo.
- Brak doświadczenia w pracy przy dużych projektach, oraz poświęcenie zbyt mało czasu na analizę i projektowanie.
- Nieznajomość podstawowych wzorców projektowych. Do stosów doskonale nadawałby się dekorator. W naszych klasach zaimplementowalibyśmy tylko metody specyficzne dla stosów, sekcji, sygnatur, ale przy użyciu api gotowych kolekcji z biblioteki.

Uważam, że początkowe złe decyzje projektowe przyczyniły się do powstania kodu trudnego do zrozumienia i dalszego utrzymania.

4.2. Statyczna analiza

W tym rozdziale opiszę implementacje podstawowych procedur statycznej analizy: staticEval, statNested i bind_name.

4.2.1. staticEval

Domyślna implementacja B.1.1(w klasie TreeNode) zwraca -1 czyli konczy się błędem . Nie zmieniają jej np. węzły wywołania procedury lub perspektywy, w większości klas jest jednak przedefiniowana. Tutaj opiszę schemat działania w poszczególnych klasach ją przedefiniowujących, bardziej szczegółowe fragmenty kodu znajdują się w B.1.1:

- W klasie ValueNode (wezły tej klasy nie posiadają dzieci) działa według schematu:
  - wkładamy na stos sygnaturę atomową z odpowiednim typem;
  - następnie w procedurze evalCard na podstawie grafu schematu obliczamy liczność tego węzła;
  - zwracamy 0 - czyli sukces.

- W klasie NameNode działamy według schematu:
  - wiążemy nazwę z ewaluowanego węzła na stosie środowisk;
  - w wyniku wiązania dostajemy wektor statycznych binderów;
  - jeżeli wektor ten jest pusty to oznacza to, że nazwa nie została związana - zwracamy -1 i statyczna ewaluacja kończy się z błędem;
  - wpp. ustawiamy numer sekcji, w której została związana i obecną wysokość stosu środowisk envs->size();
  - obliczamy liczność wyniku, i zapamiętujemy w tym węźle wszystkie węzły w których ta nazwa została związana (w wektorze boundIn);
  - wkładamy na stos rezultatów sygnaturę będącą sumą wszystkich sygnatur, które są nazywane przez bindery, w których została związana ta nazwa.
- W klasie NameAsNode metoda staticEval działa według schematu:
  - wykonujemy statyczną ewaluację dla swojego argumentu (syna);
  - zdejmujemy ze stosu StatQResStack jej rezultat;
  - wkładamy na stos nowy statyczny binder nazywający sygnaturę będącą wynikiem statycznej ewaluacji swojego argumentu;
  - w procedurze evalCard() wyliczamy liczność (w tym wypadku po prostu jest ona równa liczności syna).
- W klasie UnOpNode schemat jest następujący:
  - dokonujemy statycznej analizy poddrzewa będącego argumentem tego węzła;
  - zdejmujemy ze stosu qres wynik ewaluacji poddrzewa;
  - sprawdzamy czy sygnatura ma typ zgodny z oczekiwanym przez operator unarny reprezentowany przez dany węzeł, jeżeli nie to zwracamy -1 czyli błąd;
  - wpp. wkładamy na stos qres sygnaturę będącą wynikiem działania tego operatora na sygnaturę będącą wynikiem ewaluacji jego argumentu;
  - na koniec obliczamy liczność wyniku.
- W klasie AlgOpNode metoda działa według następującego schematu:
  - dokonujemy statycznej ewaluacji lewego i prawego argumentu (kolejność nie ma znaczenia);
  - na zmiennych pomocniczych zapamiętujemy sygnatury będące wynikami ich ewaluacji i zdejmujemy je ze stosu qres;
  - obliczamy wynik zastosowania tego operatora do wcześniej wyliczonych sygnatur i wkładamy go na stos qres;
  - obliczamy liczność dla tego węzła;
  - zwracamy 0 jeżeli procedura zakończyła się bez błędu lub -1 gdy z błędem (np gdy nie została jeszcze zaimplementowana obsługa tego operatora lub gdy statyczna ewaluacja jednego z argumentów zakończyła się błędem).
- W klasie NonAlgOpNode działa według następującego schematu:

- analizujemy statycznie lewy argument;
- sygnaturę będącą jej wynikiem zapamiętujemy;
- ustawiamy numer pierwszej otwieranej sekcji dla tego węzła;
- wkładamy na stos envs wynik działania staticNested na obliczonej sygnaturze;
- w kontekście zmodyfikowanego środowiska (o bindery pochodzące z sygnatury będącej wynikiem ewaluacji lewego argumentu) wykonujemy statyczną ewaluację prawego argumentu;
- zapamiętujemy sygnaturę będącą jej wynikiem;
- ustawiamy numer ostatniej otwieranej sekcji dla tego węzła;
- zdejmujemy ze stosu envs ostatnią sekcję (wynik staticNested na lewym arg);
- zdejmujemy ze stosu qres wynik ewaluacji prawego oraz lewego podzapytania;
- wkładamy na stos qres wynik zastosowania tego operatora do zapamiętanych sygnatur;
- obliczamy liczność dla tego węzła.

4.2.2. statNested

Domyślna implementacja w klasie Signature zwraca NULL . Tutaj opiszę schemat działania w poszczególnych klasach ją przedefiniowujących, bardziej szczegółowe fragmenty kodu znajdują się w B.1.2:

- W klasie SigRef czyli w sygnaturze będącej referencją do obiektu w bazie danych na podstawie id wskazywanego obiektu pobierana jest jego definicja z grafu schematu bazy danych i w zależności od typu obiektu zwracane jest:
  - dla obiektów atomowych pusta lista statycznych binderów;
  - dla węzłów odpowiadających obiektom wskaźnikowym zwracana jest lista z jednym statycznym binderem o nazwie takiej jak wskazywany obiekt i zawierającym sygnaturę SigRef wskazującą na ten obiekt;
  - dla obiektów złożonych zwracana jest lista statycznych binderów - dla każdego podobiektu tworzony jest statyczny binder z nazwą tego podobiektu zawierający sygnaturę wskazującą na ten obiekt w bazie danych.
- W klasie StatBinder czyli dla statycznego bindera zwracana jest kolekcja zawierająca po prostu ten binder.
- W klasie SigColl czyli dla kolekcji sygnatur statNested zwraca kolekcję binderów będącą sumą rezultatów wywołań funkcji statNested na każdej sygnaturze będącej składnikiem tej kolekcji.

4.2.3. Wyłączanie niezależnych podzapytań przed operator niealgebraiczny

Główna pętla znajduje się w metodzie parseIt klasy Optimzer i wygląda następująco:

```
int optres = -2;
if (this->statEvaluate(nt) != 0) optres = -1;
while (optres == -2) {
```

```
  optres = nt->optimizeTree();
  while (nt->getParent() != NULL) nt = nt->getParent();
  /*one more static eval, to make sure nodes have the right info.. */
  if (this->statEvaluate(nt) != 0) optres = -1;
}
```

nt jest wskaźnikiem na korzeń optymalizowanego drzewa. Na wstępie wykonujemy statyczną ewaluację. W niej wyznaczane są dane, na podstawie których wyznaczane są niezależne podzapytania - numery otwieranych sekcji dla operatorów niealgebraicznych oraz numer sekcji wiązania i wysokość stosu dla węzłów z nazwą. W pętli wykonujemy metodę optimizeTree klasy TreeNode. Metoda ta stara się znaleźć i wyciągnąć niezależne podzapytanie przed operator niealgebraiczny. Jeżeli jej się to uda zwraca -1, wpp. zwraca 0. Zauważmy, że w wyniku refaktoryzacji dotychczasowy wskaźnik na korzeń drzewa *nt* może przestać wskazywać na korzeń. Tak jest na przykład w zapytaniu:

Z *25 Emp where (age = ((Manager where name = "Kowalski").age)*

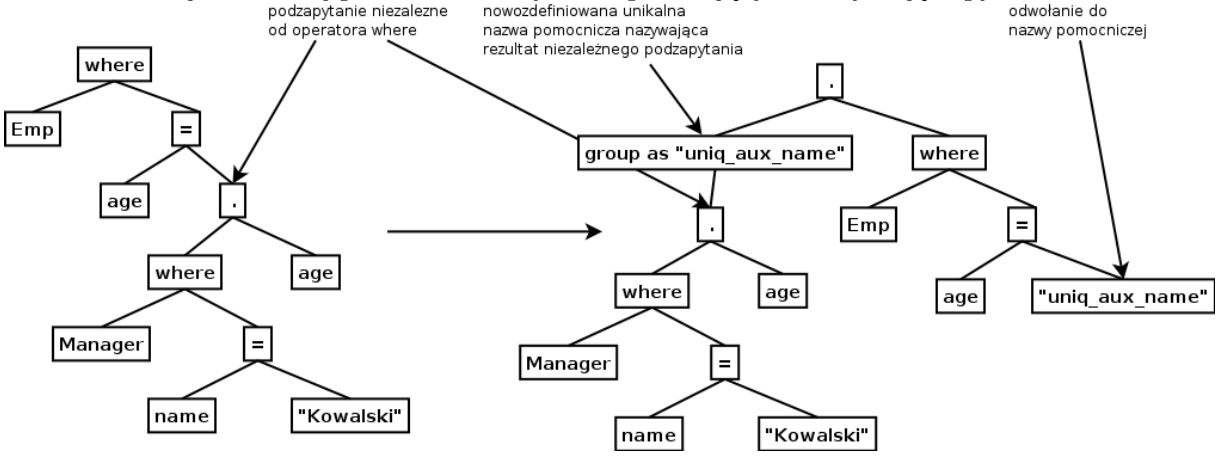
Dlatego po wywołaniu optimizeTree aktualizujemy wskaźnik na korzeń: *while (nt->getParent() != NULL) nt = nt->getParent();* . Następnie wykonujemy statyczną ewaluację na zrefaktoryzowanym drzewie w celu uaktualnienia atrybutów węzłów, na podstawie których wyznaczam niezależne podzapytania i rozpoczynam pętlę od nowa.
Najważniejsze metody:

- statEvaluate(TreeNode *nt) - opisane w sekcji 4.2 tego rozdziału;
- optimizeTree() klasy TreeNode - metoda ta zwraca 0 jeżeli nie znalazła niezależnego podzapytania, -1 jeżeli wystąpił błąd i -2 jeżeli znalazła i przeniosła niezależne poddrzewo. Metoda dla większości węzłów (zdefiniowana na poziomie TreeNode) zwraca 0. Dla klas NameAsNode i UnOpNode wywołuje się rekurencyjnie i zwraca wynik wywołania dla swojego argumentu, dla AlgOpNode wywołuje się dla lewego syna i jeżeli rezultat jest mniejszy od zera (błąd lub przeniesione) to go zwraca, wpp. zwraca wynik wywołania dla prawego syna (kolejność synów w wywołaniach jest bez znaczenia). Nieco bardziej skomplikowana jest dla węzłów niealgebraicznych NonAlgOpNode (na ich poziomie odbywa się optymalizacja - szukamy niezależnych od nich podzapytań):

  ```
  int pRes = 0;
  Optimizer *optimiser = new Optimizer();
  TreeNode *hInd = optimiser->getIndependant(this);
  if (hInd == NULL) {
    if ((pRes = larg->optimizeTree()) < 0) return pRes;
    else return rarg->optimizeTree();
  }
  TreeNode *prt = this->parent;
  if (prt != NULL) {
    prt-> swapSon (this, this->factorSubQuery(hInd, randomString()));
  } else this->factorSubQuery(hInd, randomString());
  return -2;
  ```

  Najpierw tworzymy klasę odpowiedzialną za znajdowanie niezależnych podzapytań. Za pomocą jej metody TreeNode *getIndependant(NonAlgOpNode *node) próbujemy znaleźć zapytanie niezależne od węzła operatora niealgebraicznego, w którym się znajdujemy. Jeżeli takiego nie znaleźliśmy, to postępujemy tak jak w przypadku węzła operatora

algebraicznego - staramy się zrefaktoryzować któregoś z węzłów będących argumentami. Jeżeli znaleźliśmy niezależne podzapytanie (getIndependant zwraca wskaźniki do węzła będącego jego korzeniem), to przy pomocy metod swapSon(TreeNode , TreeNode*) i factorSubQuery(TreeNode* node, char* auxName) dokonujemy refaktoryzacji. Wyróżniamy w niej dwa przypadki - gdy aktualny węzeł jest korzeniem całego drzewa i nie ma ojca (w tym przypadku po refaktoryzacji przestaje być korzeniem), oraz gdy nie jest korzeniem i ma ojca. Poniżej przedstawiam rysunek pokazujący faktoryzację zapytania 25:



- metody swapSon i factorSubQuery - tworzą nowe węzły i przy pomocy operacji na wskaźnikach wykonują wyżej opisaną refaktoryzację:

  –
  ```
  TreeNode* TreeNode::factorSubQuery(TreeNode *subT, string newName) {
    subT->getParent()->swapSon((QueryNode *) subT,
                            new NameNode(newName));
    TreeNode *nickNode = new NameAsNode ((QueryNode *) subT,
                                         newName, true);
    TreeNode *topNode = new NonAlgOpNode ((QueryNode *) nickNode,
                                (QueryNode *) this, NonAlgOpNode::dot);
    return topNode;
  }
  ```
  metoda ta faktoryzuje z danego węzła operatora niealgebraicznego niezależne od niego podzapytanie *subT* (znalezione przez metodę getIndependant). W tym celu tworzy węzeł NameAsNode z pomocniczą nazwą *newName* i niezależnym podzapytaniem *subT* jako argumentem. W ojcu niezależnego podzapytania podmienia drzewo tego niezależnego podzapytania *subT* na nowy węzeł NameNode z wywołaniem pomocniczej *newName*. Następnie tworzy i zwraca nowy węzeł operatora kropki, którego lewym argumentem jest węzeł *nickNode* definiujący nową nazwę pomocniczą, a prawym węzeł operatora niealgebraicznego *this*, na którym została wywołana metoda i z którego faktoryzujemy niezależne podzapytanie. W wyniku tych operacji niezależne podzapytanie nie jest liczone w każdym obrocie pętli implikowanym przez operator niealgebraiczny, jedynie raz przed jego wykonaniem i jego rezultat odkładany jest na stos rezultatów QRES. Operator kropki powoduje, że jest on wkładany na stos środowisk przed ewaluacją operatora niealgebraicznego.
  – int TwoArgsNode::swapSon(TreeNode* oldSon, TreeNode* newSon) - zamienia argument *oldSon* na węzeł *newSon*, nowemu węzłowi ustawia jako ojca węzeł operatora który wywołał tą metodę.

- TreeNode *Optimizer::getIndependant(NonAlgOpNode *opNiealg) metoda ta zwraca wskaźnik do węzła będącego korzeniem podzapytania niezależnego od przekazanego jako argument operatora niealgebraicznego lub *NULL*, jeżeli takiego podzapytania nie ma. Jeżeli w danym poddrzewie jest kilka podzapytań niezależnych od operatora *opNiealg*, to zwracamy wskaźnik do poddrzewa o najmniejszej głębokości. Metoda ta działa w ten sposób, że dla każdego operatora niealgebraicznego *oNAlg* z poddrzewa *opNiealg* przechodzi całe jego poddrzewo i sprawdza, czy nazwy w nim występujące są wiązane poza sekcjami otwieranymi przez *opNiealg* - jeżeli tak to oznacza to, że drzewo operatora *oNAlg* jest niezależne od operatora *opNiealg*. W mojej implementacji dla każdej nazwy *nnode* wykonywane jest:

```
if (nnode->getBindSect() < opNiealg.getFirstOpenSect() ||
    nnode->getBindSect() > opNiealg.getLastOpenSect())
  return true;
else
  return false;
```

W tej chwili *opNiealg.getFirstOpenSect()* i *opNiealg.getLastOpenSect()* są sobie równe bo w chwili implementacji nie było dziedziczenia klas i wszystkie bindery powstające w wyniku wywołania staticNested trafiają do tej samej sekcji - inaczej jest gdy jest dziedziczenie - bindery z kolejnych podklas trafiają do kolejnych sekcji na stosie środowisk.

4.3. Usuwanie martwych podzapytań

Jest realizowane przez metodę rmvDeath(TreeNode *qTree) klasy DeathRmver. Metoda ta jako argument przyjmuje drzewo syntaktyczne zapytania, w którym usuwa martwe podzapytania. Metoda ta działa według następującego schematu:(szczegóły w B.3)
W pętli:

- Najpierw wykonujemy statyczną ewaluację. W jej trakcie obliczane są wektory boundIn węzłów NameNode.
- Potem wywoływana jest metoda markNeeded, która używa ich do oznaczenia martwych oraz używanych węzłów.
- Następnie metoda qTree->getDeath() wraca martwe poddrzewo lub *NULL* gdy takiego nie ma (wtedy wychodzimy z pętli).
- Jeśli metoda getDeath() znalazła martwe poddrzewo to usuwamy je i przechodzimy do następnego obrotu pętli, w którym szukamy kolejnego martwego poddrzewa.

Usunięcie martwego poddrzewa uzyskujemy poprzez zamianę wskaźników:

- Jeżeli ojciec martwego drzewa jest korzeniem całego drzewa, to korzeniem całego drzewa staje się jego brat.
- W przeciwnym razie u dziadka martwego drzewa (w metodzie *swapSon*) wskaźnik wskazujący na ojca martwego poddrzewa ustawiamy na jego brata, a u brata wskaźnik na ojca ustawiamy na dziadka.

Metoda GetDeath dla węzłów UnOpNode i jego podklas zwraca wynik rekurencyjnego wywołania się na swoim argumencie. Podobnie jest dla operatorów dwu argumentowych (zwraca wynik rekurencyjnego wywołania dla swych argumentów) za wyjątkiem algebraicznego operatora przecinka ',' i niealgebraicznego operatora *join*. W klasie NonAlgOpNode w tej metodzie dla operatora *join* sprawdzamy czy jego prawy argument jest niepotrzebny i czy ma liczność 1..1, jeżeli tak to go zwracamy, wpp. wywołujemy metodę getDeath z nadklasy. Podobnie jest dla operatora przecinka, z tą różnicą, że szukamy w jego prawym i lewym poddrzewie. To czy dany węzeł jest potrzebny czy nie (atrybut *needed* zwracany przez metodę getNeeded - w konstruktorze ustawiany na false) jest obliczane przez metodę markNeeded. Metoda ta domyślnie ustawia atrybut *needed* na true, wywołuje się rekurencyjnie dla wszystkich dzieci danego węzła i kończy swoje działanie. Inaczej jest w przypadku węzłów NameNode i operatora kropki w węźle NonAlgOpNode:

- Dla węzłów NameNode ustawia attr needed na true oraz dla wszystkich węzłów z wektora boundIn - czyli węzłów (NameNode lub NameAsNode), których rezultatów używa jest wywoływana metoda markNeededUp. Metoda ta markuje wszystkie węzły od danego węzła (łącznie z nim) na ścieżce do korzenia drzewa jako potrzebne. Zwróćmy uwagę, że nie zajmujemy się argumentem węzła NameAsNode - gdyż on sam może zawierać martwe podzapytanie.
- Dla węzłów NonAlgOpNode z operatorem kropki ustawia attr needed na true oraz wywołuje się rekurencyjnie jedynie dla prawego argumentu (pomijając lewy) - w wyniku tej operacji lewy argument operatora kropki domyślnie pozostaje martwy. Jako potrzebny może zostać oznaczony w metodzie markNeededUp wywoływanej w metodzie markNeeded na węzłach z wektora boundIn jakiegoś węzła NameNode.

4.4. Usuwanie zbędnych nazw pomocniczych

Jest realizowane przez metodę rmvAux(TreeNode *qTree) klasy AuxRmver. Metoda ta jako argument przyjmuje drzewo syntaktyczne zapytania, w którym usuwa zbędne nazwy pomocnicze. Metoda ta działa według następującego schematu (szczegóły w B.4):

W pętli (pętla kończy się, gdy nie uda się usunąć żadnej nazwy pomocniczej):

- Na początku wykonujemy statyczną ewaluację. W jej trakcie wypełniane są wektory boundIn wszystkich węzłów z nazwami. Wstawiane są do nich węzły, w których te nazwy są wiązane (mogą to być inne węzły z nazwami NameNode bądź węzły groupAsNode).
- Następnie próbujemy usunąć jakąś zbędną nazwę pomocniczą.
- Aby stwierdzić, czy dana nazwa pomocnicza może zostać usunięta:
  - w pierwszym kroku wywołujemy metodę bool canTryToRemoveAux(NameAsNode * auxNode, vector<NameNode*>* nameVec) metoda ta zwraca false jeżeli wektor nameVec nazw, które używają danej nazwy, jest pusty (czyli nigdy nie nawigujemy do bindera utworzonego przez daną nazwę pomocniczą) wpp. dla wszystkich 'użyć' nazwy definiowanej w auxNode, czyli węzłów NameNode, które są wiązane w statycznym binderze włożonym przez daną nazwę pomocniczą, sprawdza czy:
    - ojciec danej nazwy jest operatorem kropki - tzn. albo nawiguje, albo umożliwia dalszą nawigację, jeżeli nazwa pomocnicza jest użyta w inny sposób nie możemy jej usunąć i zwracamy false

* dana nazwa jest wiązana w sekcji o rozmiarze 1 wtedy zwraca true wpp. false - wydaje się, że wystarczające byłoby gdyby była wiązana we wszystkich binderach z danej sekcji i co najwyżej jeden z nich miał liczność *, to że nie może być wiązana jedynie w jednym z kilku binderów pokazuje przykład:
  - *(Manager join (age group as a)).a*
    nie jest równoważne z
  - *(Manager join age)*
    a o tym, że dwa bindery z sekcji nie mogą mieć liczności * świadczy przykład:
    zapytanie zwracające sumę zbiorów kierowników i pracowników:
  - *((Manager group as m) join (Emp group as e)).e*
    przepisane do
  - *Manager join Emp*
    zwracałoby iloczyn kartezjański.

– jeżeli metoda canTryToRemoveAux zwróci true to w kolejnym kroku wywołujemy metodę bool removedAux(NameAsNode * auxNode, vector<NameNode*> *toRemoveVec, TreeNode *qTree). Metoda ta zwraca true, jeżeli dana nazwa pomocnicza auxNode oraz wszystkie jej użycia z toRemoveVec mogą zostać usunięte oraz usuwa tą definicję tej nazwy i wszystkie jej użycia i przypisuje zmodyfikowane drzewo na zmienną qTree, jeżeli nazwa nie może zostać usunięta zwraca false i nie modyfikuje drzewa qTree.
Aby stwierdzić czy daną nazwę pomocnicza auxNode i wszystkie jej użycia z toRemoveVec można usunąć metoda bool removedAux(NameAsNode * auxNode, vector<NameNode*> *toRemoveVec, TreeNode *qTree) kopiuje drzewo qTree, usuwa z niego węzeł auxNode oraz nazwy z toRemoveVec, na zmodyfikowanym drzewie wykonuje statyczną ewaluację. Następnie sprawdza czy wszystkie węzły z nazwami ze zmodyfikowanego drzewa są wiązane w tych samych węzłach co odpowiadające im węzły z oryginału. Jeżeli można, zwraca true i zmodyfikowane drzewo, wpp. zwraca false i niezmienione drzewo.

4.5. Zastąpienie złączenia nawigacyjnego operatorem kropki

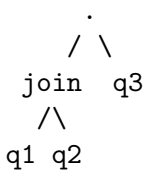
Jest realizowane przez metodę replaceJoin(TreeNode *qTree) klasy JoinRpcer. Metoda ta jako argument przyjmuje drzewo syntaktyczne zapytania w którym, jeżeli jest to możliwe, to dokonuje zamiany złączeń nawigacyjnych kropką. Metoda ta działa według następującego schematu (szczegóły w B.5):

Metoda ta na początku wykonuje statyczną ewaluację. W jej trakcie wypełniane są wektory boundIn wszystkich węzłów z nazwami. Wstawiane są do nich węzły, w których te nazwy są wiązane (mogą to być inne węzły z nazwami NameNode bądź węzły groupAsNode). Następnie w pętli tak długo jak udaje się zastąpić jakiś operator join kropką:

- wywoływana jest metoda void getSupposed(vector<NonAlgOpNode*> *toReplaceVec, TreeNode *qTree)
  metoda ta wstawia do wektora toReplaceVec wszystkie węzły z operatorem kropki, których lewym synem jest operator join;

- następnie dla każdego takiego węzła wywoływana jest metoda
  bool JoinRpcer::canReplace(NonAlgOpNode* dotNode).

Metoda ta sprawdza czy operator join będący lewym synem operatora kropki może zostać zastąpiony kropką. Jest tak wtedy, gdy żadna nazwa występująca w prawym poddrzewie (na poniższym schemacie q3) nie jest wiązana w lewym poddrzewie (q1) operatora join. Jeżeli można zastąpić, to jest wywoływana metoda replaceJoinWithDot(NonAlgOpNode* joinNode) oraz zwracane true. Jedyne co robi metoda replaceJoinWithDot to zmiana typu operatora: joinNode->setOp(NonAlgOpNode::dot);



Rozdział 5

Podsumowanie

Uważam, że projekt LoXiM jako całość, nawet pomimo znanych i nadal nie poprawionych błędów, należy uznać za udany. Podstawą do takiej opinii jest to, że projekt mimo początkowej fazy zawiera już bardzo bogatą funkcjonalność (dla przykładu zaimplementowane są transakcje które w MySQL pojawiają się dopiero w wersji 3.23.15). W dodatku nasza baza działa, czego w przypadku wielu dużych projektów zarówno open-source jak i komercyjnych nie udaje się osiągnąć.

Przechodząc do mojej implementacji, tu podobnie jak w przypadku całego projektu za główne osiągnięcie uważam to, że poprawnie działa. Moim celem było zaimplementowanie jak największej liczby optymalizacji. Zostały one dokładnie opisane w rozdziałach 1 i 4. Nie udało mi się znaleźć zapytania, które zostałoby zoptymalizowane do postaci, która nie byłaby równoważna z zapytaniem wejściowym. Optymalizacje działają również dla rozbudowanych zapytań, w których zastosowanie znajduje kilka sposobów optymalizacji. Dla takich zapytań optymalizator z powodzeniem stosuje wszystkie rodzaje zaimplementowanych optymalizacji. Myślę, że nie było czegoś co kompletnie by się nie udało. Jednak jest wiele rzeczy, które mogłyby zostać zrobione lepiej. Zalicza się do tego opisana już w 4.1 implementacja struktur danych. Mogłyby zostać przepisane z użyciem STL'a i wzorca dekorator.
Parser jest miejscem gdzie swój kod umieszczali nie tylko jego twórcy czyli ja i Maksymilian Humpich, ale także wielu innych twórców LoXiM'a. Przede wszystkim było to dodawanie nowej składni oraz klas, metod i atrybutów na potrzeby implementowanych przez nich funkcjonalności. Uważam, że powinien zostać zaprojektowany jakiś system modułów lub przynajmniej struktura katalogowa powinna zostać zorganizowana w taki sposób, aby odseparować kod realizujący różną funkcjonalność. Uważam, że również funkcjonalność optymalizatora powinna zostać bardziej oddzielona od samego parsera. Za to byłem osobiście odpowiedzialny i nie do końca to zrealizowałem. Klasy dokonujące poszczególnych optymalizacji zostały dobrze oddzielone od siebie nawzajem i od parsera. Można łatwo (manipulując co prawda kodem) wyłączyć dowolną optymalizację lub zmienić ich kolejność. Przez wyłączenie rozumiem tutaj to, że zapytanie nie zostanie pod tym kątem zoptymalizowane. Uważam jednak, że dane na których operują te klasy i ich inicjalizacja i wyliczanie nie zostały dobrze odseparowane od procesu statycznej ewaluacji. W chwili obecnej wygląda to tak, że w metodzie staticEval są wyliczane dane dla wszystkich metod optymalizacji bez względu nawet na to, czy ta metoda jest włączona czy nie. Myślę, że powinno to zostać w przyszłości poprawione.
Przed parserem i optymalizatorem pozostają do zaimplementowania kolejne ciekawe optymalizacje. Są to optymalizacje procedur i funkcji zarówno wewnątrz jak i ich wywołań, optymalizacje perspektyw. Optymalizacja wyrażeń ścieżkowych przez wyłączanie wspólnych podścieżek lub metodą ASR (dokładny opis tych i innych metod znajduje się w [Plodzien01]).

Dodatek A

Kod źródłowy zapytań tworzących przykładowy schemat danych

Kod źródłowy zapytań tworzących schemat danych w bazie danych Loxim. Wynikiem kompilacji tego schematu jest graf z rysunku 2.1 (Przykładowy schemat danych)

```
begin;
/
create ("Emp" as name, "0..*" as card, "complex" as kind) as __MDN__;
/
create ("Manager" as name, "0..*" as card, "complex" as kind) as __MDN__;
/
create ("Project" as name, "0..*" as card, "complex" as kind) as __MDN__;
/
create ("Dept" as name, "0..*" as card, "complex" as kind) as __MDN__;
/

create ("name" as name, "1..1" as card, "atomic" as kind, "string" as type)
                                                                  as subobject;
/
(subobject where name="name") :< (create (__MDN__ where name="Emp")
                                                        as owner);
/
(__MDN__ where name="Emp") :< (subobject where name="name");
/

create ("age" as name, "1..1" as card, "atomic" as kind, "int" as type)
                                                          as subobject;
/
(subobject where name="age") :< (create (__MDN__ where name="Emp")
                                                       as owner);
/
(__MDN__ where name="Emp") :< (subobject where name="age");
/


create ("name" as name, "1..1" as card, "atomic" as kind, "string" as type)
```

```
                                                                as subobject;
/
(subobject where name="name") :< (create (__MDN__ where name="Manager")
                                                                  as owner);
/
(__MDN__ where name="Manager") :< (subobject where name="name");
/

create ("age" as name, "1..1" as card, "atomic" as kind, "int" as type)
                                                                 as subobject;
/
(subobject where name="age") :< (create (__MDN__ where name="Manager")
                                                                  as owner);
/
(__MDN__ where name="Manager") :< (subobject where name="age");
/

create ("salary" as name, "1..1" as card, "atomic" as kind, "int" as type)
                                                                  as subobject;
/
(subobject where name="salary") :< (create (__MDN__ where name="Manager")
                                                                   as owner);
/
(__MDN__ where name="Manager") :< (subobject where name="salary");
/

create ("dname" as name, "1..1" as card, "atomic" as kind, "string" as type)
                                                                    as subobject;
/
(subobject where name="dname") :< (create (__MDN__ where name="Dept")
                                                                  as owner);
/
(__MDN__ where name="Dept") :< (subobject where name="dname");
/

create ("loc" as name, "1..1" as card, "atomic" as kind, "string" as type)
                                                                  as subobject;
/
(subobject where name="loc") :< (create (__MDN__ where name="Dept")
                                                                as owner);
/
(__MDN__ where name="Dept") :< (subobject where name="loc");
/



create ("title" as name, "1..1" as card, "atomic" as kind, "string" as type)
                                                                  as subobject;
/
```

```
(subobject where name="title") :< (create (__MDN__ where name="Project")
                                                              as owner);
/
(__MDN__ where name="Project") :< (subobject where name="title");
/

create ("budget" as name, "1..1" as card, "atomic" as kind,
                                            "string" as type) as subobject;
/
(subobject where name="budget") :< (create (__MDN__ where name="Project")
                                                               as owner);
/
(__MDN__ where name="Project") :< (subobject where name="budget");
/


create ("description" as name, "0..*" as card, "complex" as kind)
                                                      as subobject;
/
create ("author" as name, "1..1" as card, "atomic" as kind,
                                            "string" as type) as subobject;
/
(subobject where name="author") :< (create (subobject
                  where name="description") as owner);
/
(subobject where name="description") :< (subobject where name="author");
/

create ("text" as name, "1..1" as card, "atomic" as kind, "string" as type)
                                                              as subobject;
/
(subobject where name="text") :< (create (subobject where
                             name="description") as owner);
/
(subobject where name="description") :< (subobject where name="text");
/

(subobject where name="description") :< (create (subobject where
                                        name="Project") as owner);
/
(__MDN__ where name="Project") :< (subobject where name="description");
/


create ("does" as name, "1..*" as card, "link" as kind) as subobject;
/
(subobject where name="does") :< (create (__MDN__ where name="Project")
                                                          as target);
/
```

```
(subobject where name="does") :< (create (__MDN__ where name="Emp")
                                                        as owner);
/
(__MDN__ where name="Emp") :< (subobject where name="does");
/
create ("done_by" as name, "1..*" as card, "link" as kind) as subobject;
/
(subobject where name="done_by") :< (create (__MDN__ where name="Emp")
                                                          as target);
/
(subobject where name="done_by") :< (create (__MDN__ where name="Project")
                                                           as owner);
/
(__MDN__ where name="Project") :< (subobject where name="done_by");
/

create ("manages" as name, "0..*" as card, "link" as kind) as subobject;
/
(subobject where name="manages") :< (create (__MDN__ where name="Project")
                                                          as target);
/
(subobject where name="manages") :< (create (__MDN__ where name="Manager")
                                                           as owner);
/
(__MDN__ where name="Manager") :< (subobject where name="manages");
/
create ("managed_by" as name, "0..*" as card, "link" as kind) as subobject;
/
(subobject where name="managed_by") :< (create (__MDN__ where
                                    name="Manager") as target);
/
(subobject where name="managed_by") :< (create (__MDN__ where
                                    name="Project") as owner);
/
(__MDN__ where name="Project") :< (subobject where name="managed_by");
/



create ("works_in" as name, "1..1" as card, "link" as kind) as subobject;
/
(subobject where name="works_in") :< (create (__MDN__ where name="Dept")
                                                        as target);
/
(subobject where name="works_in") :< (create (__MDN__ where name="Manager")
                                                           as owner);
/
(__MDN__ where name="Manager") :< (subobject where name="works_in");
/
```

```
create ("employs" as name, "1..*" as card, "link" as kind) as subobject;
/
(subobject where name="employs") :< (create (__MDN__ where name="Manager")
                                                              as target);
/
(subobject where name="employs") :< (create (__MDN__ where name="Dept")
                                                            as owner);
/
(__MDN__ where name="Dept") :< (subobject where name="employs");
/


end;
/
```

Dodatek B

Najciekawsze fragmenty kodu źródłowego

B.1. Statyczna analiza

Podstawowe metody algorytmu statycznej analizy.

B.1.1. staticEval

```
int TreeNode::staticEval (StatQResStack *&qres, StatEnvStack *&envs) {
         // dla wszystkich poza przedefiniowanymi poniżej zwraca błąd
         return -1;
}

int ValueNode::staticEval (StatQResStack *&qres, StatEnvStack *&envs) {
  // wkładamy na stos sygnaturę atomowa z odpowiednim typem
  // następnie w procedurze evalCard na podstawie grafu schematu obliczamy
  // liczność tego węzła
  switch (this->type()) {
    case TreeNode::TNINT: {qres->pushSig (new SigAtomType ("int")); break;}
    case TNSTRING: { qres->pushSig (new SigAtomType ("string")); break;}
    case TNDOUBLE: { qres->pushSig (new SigAtomType ("double")); break;}
    default: {return -1; break;}
  }
  this->evalCard();
  return 0;
}

int NameNode::staticEval (StatQResStack *&qres, StatEnvStack *&envs) {

/*  wiążę tę nazwę na stosie środowisk - w wyniku dostaje
 *  wektor statycznych binderów, jeżeli wektor jest pusty to oznacza to,
 *  że nazwa nie została związana - statyczna ewaluacja kończy się z błędem,
 *  wpp. ustawiamy numer sekcji, w której została związana i
 *  obecna wysokość stosu środowisk envs->size(),
```

```
 *  obliczam liczność wyniku, i zapamiętuję w tym węźle
 *  wszystkie węzły, w których ta nazwa została związana (w wektorze boundIn),
 *  wkładam na stos rezultatów qres sygnaturę będącą sumą wszystkich sygnatur,
 *  które są nazywane przez bindery, w których została związana ta nazwa
 */
  vector<BinderWrap*> *vec = new vector<BinderWrap*>();
  BinderWrap *bw = NULL;
  envs->bindName2(this->name, vec);
  if (vec->size() > 0){
    bw = vec->at(0); // wszystkie bindery pochodzą z tej samej sekcji
  }
  if (bw == NULL) {
    return -1;
  }
  this->setBindSect(bw->getSectNumb());
  this->setStackSize(envs->getSize());
  this->setCard(bw->getBinder()->getCard()); // to jest zamiast wyw evalCard

  this->setStatEnvSecSize(((BinderWrap*)envs->getElt(this->getBindSect())->
                                                    getContent())->size());

  Signature *sig = bw->getBinder()->getValue()->clone(); //death

  for (int i = 1; i < vec->size(); i++){ // dodaje do sygnatury pozostałe
    BinderWrap *pombw = vec->at(i);
      if (sig->type() == Signature::SSTRUCT) {
        ((SigColl *) sig)->addToMyList (pombw->getBinder()->getValue()->clone());
  } else{
        SigColl *s = new SigColl(Signature::SSTRUCT);
        s->setElts(sig); s->addToMyList(pombw->getBinder()->getValue()->clone());
        sig = s;
      }
  }
  qres->pushSig (sig);
  calculateBoundUsed(vec);
  return 0;
}

int NameAsNode::staticEval (StatQResStack *&qres, StatEnvStack *&envs) {
  /*
   *  wykonuje statyczną ewaluację dla swojego argumentu,
   *  zdejmuje ze stosu StatQResStack jej rezultat,
   *  wkłada na stos nowy statyczny binder nazywający sygnaturę będącą wynikiem
   *  statycznej ewaluacji swojego syna
   *  w procedurze evalCard() wylicz liczność
   *  (w tym wypadku po prostu jest ona równa liczności syna)
   */
/*zrób staticEval dla this->arg. potem sprawdź top na qres,
  zrób pop, zrób z tego bindera, wsadź z powrotem.*/
```

```
    if (this->arg->staticEval(qres, envs) == -1 ) return -1;
    StatBinder *sb = new StatBinder (this->name, qres->topSig()->clone());
    qres->pop();
    qres->pushSig (sb);
    this->evalCard();
    return 0;
}

int AlgOpNode::staticEval (StatQResStack *&qres, StatEnvStack *&envs) {
  /*  działa według następującego schematu:
   *  dokonuje statycznej ewaluacji lewego i prawego argumentu (kolejność nie
   *  ma znaczenia),
   *  zapamiętujemy sygnatury będące wynikami ich ewaluacji na zmiennych
   *  pomocniczych i zdejmujemy je ze stosu qres,
   *  obliczamy wynik zastosowania tego operatora do wcześniej wyliczonych
   *  sygnatur i wkładamy go na stos qres,
   *  obliczamy liczność dla tego węzła
   *  zwracamy 0 jeżeli procedura zakończyła się bez błędu lub -1 gdy z błędem
   *  (np gdy nie została jeszcze zaimplementowana obsługa tego operatora lub
   *  gdy statyczna ewaluacja jednego z argumentów zakończyła się błędem)
   *
  */
  if (this->larg->staticEval(qres, envs) == -1) return -1;
    Signature *lSig = qres->topSig();
    if (this->rarg->staticEval(qres, envs) == -1) return -1;
    Signature *rSig = qres->topSig();
    qres->pop();
    qres->pop();
    this->evalCard();
    if (op == AlgOpNode::eq || op == AlgOpNode::neq || op == AlgOpNode::lt ||
        op == AlgOpNode::gt || op == AlgOpNode::le || op == AlgOpNode::ge ||
        op == AlgOpNode::boolAnd || op == AlgOpNode::boolOr) {
      qres->pushSig (new SigAtomType ("bool"));
      return 0;
    }
    if (op == AlgOpNode::plus || op == AlgOpNode::minus ||
        op == AlgOpNode::times || op == AlgOpNode::divide) {
      qres->pushSig (new SigAtomType ("double"));
      return 0;
    }
    if (op == AlgOpNode::bagUnion || op == AlgOpNode::bagIntersect ||
        op == AlgOpNode::bagMinus) {
      Deb::ug("can't handle Union,intersect and Minus now.. ");
      return -1;
    }
    if (op == AlgOpNode::comma) {
      if (lSig->type() == Signature::SSTRUCT) {
        ((SigColl *) lSig)->addToMyList (rSig);
```

```
        qres->pushSig (lSig);
      } else {
        SigColl *s = new SigColl(Signature::SSTRUCT);
        s->setElts(lSig); s->addToMyList(rSig);
        qres->pushSig(s);
      }
      return 0;
    }
    this->evalCard();
    return 0;
}

int NonAlgOpNode::staticEval (StatQResStack *&qres, StatEnvStack *&envs) {
/*  działa według następującego schematu:
 *  analizujemy statycznie lewy argument,
 *  sygnaturę będącą jej wynikiem zapamiętujemy,
 *  ustawiamy numer pierwszej otwieranej sekcji dla tego węzła
 *  wkładamy na stos envs wynik działania staticNested na obliczonej sygnaturze,
 *  w kontekście zmodyfikowanego środowiska (o bindery pochodzące z sygnatury
 *  będącej wynikiem ewaluacji lewego argumentu) wykonujemy statyczną ewaluację
 *  prawego argumentu, zapamiętujemy sygnaturę będącą jej wynikiem,
 *  ustawiamy numer ostatniej otwieranej sekcji dla tego węzła
 *  zdejmujemy ze stosu envs ostatnią sekcję (wynik staticNested na lewym arg)
 *  zdejmujemy ze stosu qres wynik ewaluacji prawego oraz lewego podzapytania
 *  wkładamy na stos qres wynik zastosowania tego operatora do zapamiętanych
 *  sygnatur,
 *  obliczamy liczność dla tego węzła
*/
  if (this->larg->staticEval(qres, envs) == -1) return -1;
  Signature *lSig = qres->topSig();
  this->setFirstOpenSect(1 + envs->getSize());
  Signature *toPush = lSig->clone();
  envs->pushBinders(toPush->statNested());
  this->setLastOpenSect(envs->getSize());
  if (this->rarg->staticEval(qres, envs) == -1) return -1;
  Signature *rSig = qres->topSig();
  envs->pop();
  qres->pop();    /*pop rSig*/
  qres->pop();    /*pop lSig*/
  switch (op) {
    case NonAlgOpNode::dot :{/*we just take the part to the right of the dot*/
      qres->pushSig(rSig->clone());
      break;}
    case NonAlgOpNode::where: {
      if (rSig->type() != Signature::SBOOL) {
        qres->pushSig(lSig->clone());
      break;}
    case NonAlgOpNode::join: {
      SigColl *s = new SigColl(Signature::SSTRUCT);
```

```
      s->setElts(lSig); s->addToMyList(rSig);
      qres->pushSig(s->clone());
      break;}
    default: {return -1;}
  }
  this->evalCard();
  return 0;
}
```

B.1.2. statNested

```
virtual BinderWrap * Signature::statNested() {
/* domyślnie zwraca NULL */
return NULL;
}

BinderWrap* SigRef::statNested() {
  /*  dla sygnatury będącej referencja do obiektu w bazie danych na podstawie
   *  id wskazywanego obiektu pobierana jest jego definicja z grafu schematu
   *  bazy danych i w zależności od typu obiektu zwracane jest:
   *  dla obiektów atomowych pusta lista statycznych binderów
   *  dla węzłów odpowiadających obiektom wskaźnikowym zwracana jest lista
   *    z jednym statycznym binderem o nazwie takiej jak wskazywany obiekt
   *    zawierającym sygnaturę SigRef wskazującą na ten obiekt
   *  dla obiektów złożonych zwracana jest lista statycznych binderów -
   *    dla każdego podobiektu tworzony jest statyczny binder z nazwą tego
   *    podobiektu zawierający sygnaturę wskazującą na ten obiekt w bazie
  */
  return DataScheme::dScheme()->statNested(this->refNo);
}

BinderWrap* StatBinder::statNested() {
  /* dla statycznego bindery zwracana jest kolekcja zawierająca po prostu ten
     binder */
  return new BinderList((StatBinder *) this);
}

BinderWrap* SigColl::statNested() {
  /* dla kolekcji sygnatur statNested zwraca kolekcję binderów będącą sumą
   * rezultatów wywołań funkcji statNested na każdej sygnaturze będącej
   * składnikiem tej kolekcji
  */
  Signature *pt = this->getMyList();
  BinderWrap *bindersCol = NULL;
  BinderWrap *resultBinders = NULL;
  while (pt != NULL) {
    bindersCol = NULL;
```

```
    bindersCol = pt->statNested();
    BinderWrap *one;
    if (resultBinders == NULL){
      resultBinders = bindersCol;
    } else {
      one = bindersCol;
      while(one!=NULL){
        BinderWrap *nextPom = one->getNext();
        resultBinders =  resultBinders->addOne(one);
        one = nextPom;
        // UWAGA tak byłoby źle: (bo addOne zmienia obiekt który dodaje)
        //one =  one->getNext();
      }
    }
    pt = pt->getNext();
  }
  return resultBinders;
}
```

B.2. Wyłączanie niezależnych podzapytań przed operator nie-algebraiczny

Główna pętla znajduje się w metodzie parseIt klasy Optimzer i wygląda następująco:

```
int optres = -2;
if (this->statEvaluate(nt) != 0) optres = -1;
while (optres == -2) {
  optres = nt->optimizeTree();
  while (nt->getParent() != NULL) nt = nt->getParent();
  /*one more static eval, to make sure nodes have the right info.. */
  f (this->statEvaluate(nt) != 0) optres = -1;
}
```

B.3. Usuwanie martwych podzapytań

Schemat klasy odpowiedzialnej za tą optymalizację:

```
class DeathRmver {
    public:
        DeathRmver(QueryParser *qParser);
        virtual ~DeathRmver();
        int rmvDeath(TreeNode *&qTree);
}
```

Schemat działania metody rmvDeath(TreeNode *qTree):

```
while(true){
  qParser->statEvaluate(qTree);
```

```
  qTree->markNeeded2();
  TreeNode * death = qTree->getDeath();
  if (death == NULL) break;
  TwoArgsNode * parent = (TwoArgsNode*) death->getParent();
  TreeNode * live =
            (parent->getLArg() == death)?parent->getRArg():parent->getLArg();
  if (parent == qTree){
    qTree = live;
    qTree->setParent(NULL);
  } else {
    parent->getParent()->swapSon(parent, live);
  }
}
```

Metoda getDeath w klasie TreeNode i jej podklasach:

```
virtual TreeNode * TreeNode::getDeath(){
  return NULL;
}

virtual TreeNode * UnOpNode::getDeath(){
  return this->getArg()->getDeath();
}

virtual TreeNode * TwoArgsNode::getDeath(){
  TreeNode *pom = this->getLArg()->getDeath();
  if (pom != NULL)
    return pom;
  return this->getRArg()->getDeath();
}

virtual TreeNode * NonAlgOpNode::getDeath(){
  if (this->getOp() == NonAlgOpNode::join){
    if((!this->getRArg()->getNeeded())&&(this->getRArg()->getCard()=="1..1")){
      return this->getRArg();
    }
  }
  return TwoArgsNode::getDeath();
}

virtual TreeNode * AlgOpNode::getDeath(){
  if (this->getOp() == AlgOpNode::comma){
    if((!this->getRArg()->getNeeded()&&(this->getRArg()->getCard()=="1..1"))){
      return this->getRArg();
    }
    if((!this->getLArg()->getNeeded()&&(this->getLArg()->getCard()=="1..1"))){
      return this->getLArg();
    }
  }
```

```
  return TwoArgsNode::getDeath();
}
```

B.4. Usuwanie zbędnych nazw pomocniczych

Schemat klasy odpowiedzialnej za tą optymalizację:

```
class AuxRmver {
public:
        AuxRmver(QueryParser *qParser);
        virtual ~AuxRmver();
        int rmvAux(TreeNode *&qTree);
protected:
     bool canTryToRemoveAux(NameAsNode * auxNode, vector<NameNode*>* nameVec);
     bool removedAux(NameAsNode * auxNode, vector<NameNode*> *toRemoveVec,
                                                                TreeNode *&qTree);
};
```

Schemat działania głównej metody:

```
rmvAux(TreeNode *&qTree){
  bool removed = true;
  while(removed){
    removed = false;
    qParser->statEvaluate(qTree);
    vector<NameAsNode*> * auxVec = new vector<NameAsNode*>();
    qTree->getAllNameAsNodes(auxVec); // wstawia do wektora auxVec
                                      // wszystkie węzły nameAsNode

    dla każdego węzła z NameAsNode *nameAsNode z drzewa qTree wykonaj:{
    vector<NameNode*> * usedBy =  (vector<NameNode*>*)nameAsNode->getUsedBy();

    bool canTryToRemove = true;

    if (usedBy->size() == 0) {
      canTryToRemove = false;
    }
    for(vector<NameNode*>::iterator usedIter =usedBy->begin();
                                    usedIter != usedBy->end(); usedIter++){
      /*
       * tu sprawdzam czy (*usedIter) jest wiązany w sekcji o rozmiarze 1 i
       * czy jego ojcem jest kropka, jeżeli tak to kopiuje sobie znowu
       * oryginał i usuwam z niego dekl i wszystkie wywołania i ewaluuje
       * potem sprawdzam czy wszystkie z used by są wiązane w tym samym węźle
       */
      if (!canTryToRemoveAux(*nameAsNode,  usedBy)){
        canTryToRemove = false;
```

```
      } else {
        // można usunąć
      }
    }

    if (canTryToRemove){
      if (removedAux(*nameAsNode, usedBy, qTree)){
        // został usunięty
        removed = true;
        break;
      }
    }
  }
}
```

B.5. Zastąpienie złączenia nawigacyjnego kropką

Schemat klasy odpowiedzialnej za tą optymalizację:

```
class JoinRpcer {
  public:
         virtual int replaceJoin(TreeNode *&qTree);
  protected:
         virtual void getSupposed(vector<NonAlgOpNode*> *toReplaceVec,
                                                        TreeNode *&qTree);
         virtual bool canReplace(NonAlgOpNode* dotNode);
         virtual void replaceJoinWithDot(NonAlgOpNode* joinNode);
};

replaceJoin( TreeNode *&qTree){

  qParser->statEvaluate(qTree);
  bool replaced = true;
  while(replaced){
    replaced = false;
    vector<NonAlgOpNode*> *toReplaceVec =  new vector<NonAlgOpNode*>();
    getSupposed(toReplaceVec, qTree);
    // tu robie zastępowanie joina kropka, jeżeli się uda,
    // ustawiam replaced = true;
    for(vector<NonAlgOpNode*>::iterator iter = toReplaceVec->begin();
                                      iter != toReplaceVec->end(); iter++){
      if (canReplace(*iter)){
        replaceJoinWithDot((NonAlgOpNode*) (*iter)->getLArg());
        replaced = true;
      }
    }
  }
```

```
}


void JoinRpcer::getSupposed(vector<NonAlgOpNode*> *toReplaceVec, TreeNode *&qTree){
  vector<TreeNode*> *listVec =  new vector<TreeNode*>();
  qTree->getInfixList(listVec);
  for(vector<TreeNode*>::iterator iter = listVec->begin();
                                                  iter != listVec->end(); iter++){
    if ((*iter)->type()==TreeNode::TNNONALGOP){
      NonAlgOpNode * dot = (NonAlgOpNode*)(*iter);
        if (dot->getOp() == NonAlgOpNode::dot){
          if (dot->getLArg()->type() == TreeNode::TNNONALGOP){
            if (((NonAlgOpNode*)dot->getLArg())->getOp() == NonAlgOpNode::join){
              toReplaceVec->push_back(dot);
            }
          }
        }
    }
  }
}


     /*
                .
               / \
             join  q3
              /\
            q1 q2
     */
bool JoinRpcer::canReplace(NonAlgOpNode* dotNode){
  // wiem ze lewym drzewem jest join
  TreeNode* q1 = ((NonAlgOpNode*)dotNode->getLArg())->getLArg();
  TreeNode* q2 = ((NonAlgOpNode*)dotNode->getLArg())->getRArg();
  TreeNode* q3 = (NonAlgOpNode*)dotNode->getRArg();

  vector<NameNode*> *q3NamesVec = new vector<NameNode*>();
  vector<TreeNode*> *q3NodesVec =  new vector<TreeNode*>();
  q3->getInfixList(q3NodesVec);

  for(vector<TreeNode*>::iterator iter = q3NodesVec->begin();
                                          iter != q3NodesVec->end(); iter++){
    if ((*iter)->type()==TreeNode::TNNAME){
      q3NamesVec->push_back((NameNode*)(*iter));
    }
  }

  for(vector<NameNode*>::iterator iter = q3NamesVec->begin();
                                          iter != q3NamesVec->end(); iter++){
    vector<TreeNode*> *boundIn = (*iter)->getBoundIn();
```

```
    for(vector<TreeNode*>::iterator q1NodeIter = boundIn->begin();
                                    q1NodeIter != boundIn->end(); q1NodeIter++){
      long oid = (*q1NodeIter)->getOid();
      if (q1->containsOid(oid)){
        return false;
      }
    }
  }
  return true;
}
```

Dodatek C

Opis zawartości załączonej płyty

Do pracy dołączona jest płyta CD, która zawiera:

- Kod źródłowy tej pracy magisterskiej w formacie LaTeX
- Pracę magisterską w formacie pdf
- Skompilowany system zarządzania bazą danych LoXiM oraz jego dokumentację
- Kod źródłowy LoXiMa - w tym zaimplementowane przeze mnie optymalizacje i parser
- Dokumentację LoXiMa
- Przykładowe skrypty tworzące schemat danych oraz zapytania testujące na jego podstawie zaimplementowane metody optymalizacji.

Sposób uruchomienia przykładowych optymalizacji:
Kopiujemy katalog LoXiM i testy do dowolnego katalogu.
Wchodzimy do katalogu LoXiM/Server i uruchamiamy serwer poleceniem ./Listener.
W następnej konsoli wchodzimy do katalogu LoXiM/SBQLCli i odpalamy klienta poleceniem ./SBQLCli. Logujemy się jako użytkownik root z dowolnym hasłem. Teraz już możemy wydawać polecenia sbqlowe. Rozpoczynamy transakcje poleceniem begin, kończymy poleceniem end. Skrypty wykonujemy komendą @/path_to/file.sbql np. @../../testy/schemat_1.sbql wczyta nam używany w tej pracy schemat danych. Po stworzeniu schamtu restartujemy sesję i możemy wykonać kilka zapytań:

```
begin
/
  Manager where salary > ((Manager where name = "Nowak").salary)
  /
  (Manager join (works_in.Dept)).name
  /
  ((Emp group as e) where e.age < 20).e
  /
  (Emp join does.Project join managed_by.Manager join works_in.Dept).dname
  /
end
/
```

Zoptymalizowane zapytania wypisywane są na standardowe wyjście serwera.

Bibliografia

[Stencel01] Krzysztof Stencel, *Półmocna kontrola typów w językach programowania baz danych*, Wydawnictwo PJWSTK, Warszawa 2006.

[Stencel02] Lech Banachowski, Krzysztof Stencel, *Bazy danych. Projektowanie aplikacji na serwerze*, EXIT, 2001.

[Subieta01] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, Wydawnictwo PJWSTK, Warszawa 2004.

[Özsu01] M. Tamer Özsu, Patrick Valduriez, *Principles of Distributed Database Systems*, Second Edition Prentice-Hall 1999.

[Plodzien01] Jacek Płodzień *Optimization Methods in Object Query Languages*, Warszawa, 2000.