AUTOMATYKA • 2009 • Tom 13 • Zeszyt 3

Tomasz M. Kowalski*, Paweł Cebula*, Kamil Kuliberda*,
Jacek Wiślicki*, **, Radosław Adamus*, **

Metody optymalizacji przez indeksowanie dla obiektowego języka zapytań***

1. Wprowadzenie

Indeks to pomocnicza (nadmiarowa) struktura danych przechowywana na serwerze. Administrator baz danych zarządza pulą indeksów, generując nowe albo usuwając istniejące w zależności od aktualnych potrzeb, mając na celu zwiększenie tym samym całkowitej wydajności aplikacji. Tak jak w przypadku tradycyjnych książek, indeksy na końcu książki pozwalają na szybkie odnalezienie pożądanej strony, podobnie indeks bazodanowy umożliwia szybki dostęp do obiektów (lub rekordów) spełniających określone kryteria. W związku z faktem, że indeksy mają relatywnie mały rozmiar (w porównaniu do całej bazy danych), zajęcie dodatkowej przestrzeni dyskowej do ich przechowywania jest całkowicie uzasadnione poprzez zysk na wydajności działania bazy danych. Bardzo efektywna fizyczna organizacja indeksów umożliwia osiągnięcie nawet o kilka rzędów wielkości wyższej wydajności.

Ogólna idea indeksowania w zorientowanych obiektowo bazach danych nie różni się od indeksowania w bazach relacyjnych [2]. Wiele metod może być zaadaptowanych z systemów relacyjnych, a nawet ich zastosowanie może być znacząco rozszerzone. Bywają sytuacje, w których metody indeksacji wzięte z relacyjnych baz danych są przestarzałe dla baz danych zorientowanych obiektowo. W szczególności operacje złączenia nie wymagają szczególnej optymalizacji, ponieważ w obiektowych bazach danych potrzeba złączeń jest znacznie mniejsza ze względu na obecność identyfikatorów obiektów oraz jawnych wskaźnikowych powiązań.

* Katedra Informatyki Stosowanej, Politechnika Łódzka

** Stypendysta projektu „Innowacyjna dydaktyka bez ograniczeń – zintegrowany rozwój Politechniki Łódzkiej – zarządzanie uczelnią, nowoczesna oferta edukacyjna i wzmacnianie zdolności do zatrudniania, także osób niepełnosprawnych" współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

*** Praca naukowa finansowana ze środków na naukę w latach 2008/2009 jako projekt badawczy nr N516 383334

ODRA (*Object Database for Rapid Applications development*) jest prototypem systemu zarządzania zorientowaną obiektowo bazą danych opartą na podejściu stosowym SBA (*Stack Based Architecture*) [13, 14]. Głównym celem projektu ODRA jest tworzenie i rozwijanie nowych paradygmatów tworzenia aplikacji bazodanowych oraz wprowadzenie nowego, uniwersalnego, deklaratywnego języka programowania wraz z rozproszonym środowiskiem wykonawczym, zorientowanym na bazy danych i obiektowość. ODRA zawiera własny język zapytań o nazwie SBQL (*Stack Basek Query Language*), który jest zintegrowany z możliwościami programistycznymi i abstrakcjami, w tym abstrakcjami bazodanowymi: aktualizowane widoki, procedury składowane, transakcje.

Ważną cechą systemu ODRA jest silnik optymalizacyjny odpowiedzialny za zwiększanie wydajności wykonywania zapytań bazodanowych. Zasadniczym komponentem silnika jest moduł, który optymalizuje zapytania używając indeksacji. Główne cechy implementacji indeksowania to: przezroczysty wybór odpowiednich indeksów dla zadanego zapytania (jeśli to możliwe), automatyczna aktualizacja indeksów w odpowiedzi na zmiany w powiązanych danych oraz zarządzanie administracyjne indeksami.

Niniejszy artykuł prezentuje wspomniane trzy aspekty implementacji indeksacji w ODRA. Sekcja 2 przedstawia ogólną architekturę optymalizacji zapytań w ODRA. Sekcja 3 omawia cechy systemu indeksowania w ODRA. Sekcja 4 opisuje narzędzia zarządzania indeksami w ODRA. Sekcja 5 ilustruje optymalizację zapytań bazodanowych opartą o indeksację. Sekcja 6 prezentuje zysk wydajności wynikający z proponowanych rozwiązań na podstawie przykładowych zapytań. Sekcja 7 zawiera podsumowanie artykułu i wnioski.

2. Idea indeksowania w ODRA

W uproszczeniu indeks może być postrzegany jako tabela składająca się z dwóch kolumn, gdzie pierwsza z nich zawiera unikalną wartość kluczową, natomiast druga przechowuje wartość niekluczową. W większości przypadków wartość niekluczową stanowi referencja do obiektu.

Wartość kluczowa jest używana jako dana wejściowa dla procedur wyszukujących opartych o indeksy. W rezultacie, procedura zwraca odpowiadającą wartość z drugiej kolumny tego samego wiersza tabeli. Klucze są zazwyczaj wartościami wybranych atrybutów obiektów (gęste indeksy) lub reprezentują zakres tych wartości (indeksy zakresowe).

Wartości kluczowe mogą być obliczane przy użyciu wyrażeń, które zawierają wbudowane funkcje języka zapytań lub funkcje definiowane przez użytkownika (indeksy funkcyjne [1]). To podejście umożliwia administratorowi stworzenie indeksu odpowiadającego dokładnie predykatom w obrębie najczęściej pojawiających się zapytań, tak by ich ewaluacja stała się szybsza i używała minimalnej ilości operacji we/wy.

W optymalizacji zapytań indeksy są używane w kontekście operatora *where*, w przypadku gdy lewy argument jest zindeksowany według wartości kluczowej prawego argu-

mentu stanowiącego predykat selekcji. Dla dużych baz danych zastąpienie klauzuli *where* wywołaniem funkcji indeksu, może powodować zysk wydajności nawet o kilka rządów wielkości. Niemniej do osiągnięcia tego wzrostu serwer baz danych powinien zapewniać przezroczystość indeksów. Oznacza ona, że programista aplikacji z bazą danych nie musi być świadomy istnienia indeksów. Najczęściej optymalizator zapytań jest odpowiedzialny za automatyczne wykorzystanie indeksów. Drugi ważny aspekt przezroczystości jest związany z utrzymywaniem spójności między indeksami a indeksowanymi danymi. Jest to problem tzw. automatycznej aktualizacji indeksu. Modyfikacje w bazie powinny być automatycznie wykrywane i odzwierciedlane w odpowiednich indeksach.

2.1. Cechy indeksów w ODRA

Obecnie implementacja wspiera indeksy oparte na liniowym haszowaniu [5], które może być łatwo rozszerzane do rozproszonej wersji SDDS [6] w celu optymalnego wykorzystania zasobów obliczeniowych systemów rozproszonych. Niemniej jednak, istnieje szeroki zakres różnych struktur indeksów, które mogą być użyte do indeksowania w obiektowo zorientowanych bazach danych podobnie do rozwiązań obecnych w relacyjnych bazach danych [1, 2, 8, 10]: B-Drzewa, indeksy bitmapowe, itp.

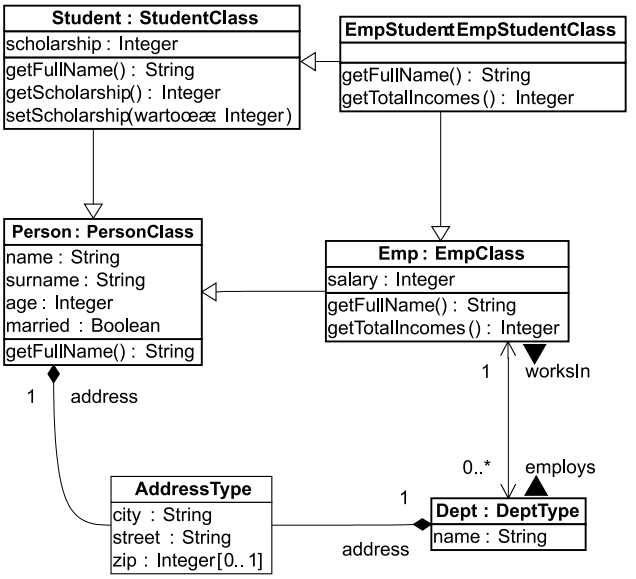
Implementacja indeksowania w ODRA wspiera indeksy z wieloma kluczami. W dodatku do typów kluczy wspomnianych wcześniej (gęstych i zakresowych), wprowadzono typ *enum* m.in. do zwiększenia elastyczności indeksowania na wielu kluczach. Ponadto, dzięki własnościom języka SBQL, tj. ortogonalności i kompozycyjności, zaimplementowane rozwiązania dostarczają generycznego wsparcia dla dowolnych definicji indeksów np. złożonych z wyrażeń zawierających polimorficzne metody i operatory agregujące.

3. Zarządzanie indeksami

Wszystkie indeksy znajdujące się w bazie danych są zarejestrowane i zarządzane przez menadżera indeksów systemu ODRA. Lista wszystkich indeksów i pomocnicze informacje wymagane dla optymizatora są przechowywane wewnątrz specjalnego modułu administracyjnego. Każdy indeks jest skojarzony z modułem, w którym został stworzony, a jego nazwa musi być unikalna. Dlatego też menadżer indeksów sprawdza czy dany indeks istnieje na liście referencji do obiektów metabazy opisujących indeksy używając kombinacji składającej się z nazwy modułu oraz nazwy indeksu: „nazwa_modułu.nazwa_indeksu”.

3.1. Przykładowy schemat

Schemat na rysunku 1 jest wprowadzony jako podstawa dla przedstawionych dalej przykładów użycia indeksów.



Rys. 1. Przykład schematu zorientowanego obiektowo

Przykładowy schemat ilustruje rekordy personelu pewnej firmy. Wprowadzono kilka klas *PersonClass*, *StudentClass*, *EmpClass*, *EmpStudentClass* oraz dwa typy strukturalne *DeptType* i *AddressType*. Trwałe instancje klas wymienionych powyżej mogą być dostępne za pośrednictwem nazw klas ich instancji *Person*, *Student*, *Emp* i w końcu *EmpStudent*. Obiekty reprezentujące departamenty firmy, nazwane *Dept* mają strukturę *DeptType* z atrybutem głównym *name*. Instancje klasy *EmpClass* reprezentują obecnych pracowników firmy oraz rozszerzają obiekty *Person* o atrybut *salary*. Obiekty *Emp* oraz *Dept* są powiązane obiektami wskazującymi nazwanymi odpowiednio *worksIn* oraz *employs*. Kolejna klasa, która rozszerza klasę *PersonClass* to *StudentClass*. Wprowadza ona atrybut *scholarship*. Ostatnia klasa pokazana na schemacie, nazwana *EmpStudentClass,* dziedziczy po *EmpClass* oraz *StudentClass*. Została wprowadzona, aby reprezentować studentów, którzy są jednocześnie pracownikami firmy. Użycie nazwy *Person* w zapytaniu SBQL daje w rezultacie wszystkie instancje klasy *PersonClass* i jej podklas. Podobnie poprzez nazwę *Emp* programista odnosi się do instancji obu klas *EmpClass* i *EmpStudentClass*.

Klasy mogą zawierać metody korzystające z polimorfizmu (nadpisywanych w podklasach). Np. metoda *getTotalIncomes*() klasy *EmpClass* zwraca wartość atrybutu *salary*, ale dla instancji klasy *EmpStudentClass* zwraca sumę atrybutów *salary* i *scholarship*.

3.2. Typy indeksów

Składnia umożliwiająca stworzenie indeksów pozwala administratorowi wyspecyfikować ogólne właściwości klucza indeksu, tj. dotyczące wartości kluczowych lub celu opty-

malizacji. Jest to osiągane poprzez wprowadzenie opcjonalnego typu wskaźnika: *dense*, *range* oraz *enum*.

Wskaźnik *dense* powoduje, że optymalizacja zapytań, które używają danego klucza jako warunku, będzie stosowana tylko dla predykatów selekcji opartych na operatorach „=" oraz *in*. Dlatego dystrybucja zindeksowanych obiektów w indeksie (tj. w tabeli haszującej) może być bardziej losowa. Kolejność wartości kluczowych nie ma znaczenia dla procesu indeksacji. Typ *dense* jest zawsze używany do wartości wskaźnikowych (bez względu czy zostało to określone przez administratora). Ponadto jest to domyślny typ wskaźnika dla takich typów wartości kluczowych: *integer*, *string*, *double*.

add index *idxEmpSalary*(**dense**) **on** Emp(*salary*)

Wskaźnik *range* powoduje, że optymalizowane będą predykaty selekcji oparte nie tylko na operatorach „=" oraz *in* ale również na operatorach „>", „≥", „<" i „≤". W obrębie indeksu funkcja haszująca grupuje obiekty zgodnie z zakresami wartości kluczowych. W obecnej implementacji, zakresy są dynamicznie dzielone, ponieważ każdy zakres jest skojarzony z indywidualnym fragmentem liniowej mapy haszującej zwanym kubełkiem (*bucket*).

add index *idxDeptSalary*(**range**) **on** *Dept*(**sum**(*employs.Emp.salary*))

Indeks *idxDeptSalary* zwraca referencje do departamentu zgodnie z wartością (lub zakresem wartości) sumy wynagrodzeń pracowników departamentu. Jego zaletą jest unikanie wyliczania wartości złożonych predykatów selekcji wiele razy, ponieważ jest ona wcześniej wyznaczona w momencie tworzenia indeksu. Z drugiej strony koszt utrzymania indeksu *idxDeptSalary* jest bardzo duży i może powodować poważne zmniejszenie wydajności operacji aktualizacji bazy danych.

Wskaźnik *enum* jest stosowany w sytuacji, gdy chcemy wykorzystać własności klucza ze skończonym, policzalnym zbiorem odrębnych wartości, tj. klucze z niską licznością (kardynalnością) wartości. Wydajność indeksu znacznie pogorsza się jeżeli wartości kluczowe mają niską liczność, np. kolor oczu osób, stan cywilny (wartość boolowska) lub rok urodzenia. Używając klucza indeksu typu *enum* powoduje, że wewnętrznie są składowane wszystkie możliwe wartości kluczowe (lub zakres dla całkowitych wartości) a informacje te są używane do optymalizacji struktury indeksów.

Typ klucza *enum* powoduje optymalizacje predykatów selekcji dokładnie tak jak w przypadku wskaźnika *range*, np. dla operatorów „=", „>", „≥", „<" i „≤".

Inna ważna właściwość klucza typu *enum* pojawia się w momencie kiedy indeks jest założony na wielu kluczach, wówczas optymizator może niektóre z nich pominąć jeśli jest to potrzebne w trakcie optymalizacji zapytań. Jeżeli *enum* jest ustawiony na wszystkich kluczach indeksu i ilość zindeksowanych obiektów jest duża, wówczas ewaluacja wywoła-

nia indeksu powinna zadziałać w sposób bardzo wydajny (każda kombinacja wartości kluczowych wskazuje na osobną tablicę referencji obiektów zwaną kubełkiem)

add index *idxPerAge&Mar&City*(**enum**|**enum**|**enum**) **on** *Person* (*age*, *married*, *address.city*)

Inne przykłady poleceń tworzących indeksy przedstawiono poniżej:

add index *idxPerZip*(enum) **on** *Person*(*address.zip*)

Indeks *enum* który zwraca obiekty *Person*, którego argumentem jest podobiekt podobiektu *address* czyli kod pocztowy osoby. Należy zwrócić uwagę na fakt, iż atrybut kod pocztowy (*zip*) na schemacie na rysunku 1 jest opcjonalny i dlatego ten indeks przechowuje tylko obiekty *Person*, które go zawierają.

add index *idxPerBirthYear*(**range**) **on** *Person*(2009 – *age*)

Indeks w tym przypadku zwraca obiekty *Person* według wartości wyrażenia „2009 – *age*". Zakłada się, że ten indeks jest w stanie przetwarzać zapytania zakresowe.

add index *idxEmpTotalIncomes* **on** *Emp*(*getTotalIncomes*())

Indeks gęsty używający metody *getTotalIncomes*() klasy *Emp* jako klucza dla selekcji obiektów *Emp*. Ta metoda jest nadpisana dla instancji klasy *EmpStudent*.

Jedyną akcją wymaganą od administratora, aby skorzystać z zalet indeksowania jest stworzenie odpowiednich indeksów. Pozostała część optymalizacji jest przezroczysta dla programistów. Następna sekcja opisuje zasady stosowane przez optymizator wykorzystujący indeksy.

4. Optymalizacja zapytań

W systemie ODRA użycie indeksów jest całkowicie przezroczyste dla kodu aplikacji. Programista nie musi być świadomy o istnieniu indeksów, ponieważ kod zupełnie od nich nie zależy. Optymalizator indeksów automatycznie stosuje wszystkie możliwe indeksy w trakcie procesu kompilacji zapytań.

Oprócz tej możliwości użytkownik może również używać indeksów jawnie. Ta cecha jest wdrożona do celów testowych, aby sprawdzić równość semantyczną opracowanych metod optymalizacji przez indeksowanie i badań nad nowymi możliwościami wykorzystania indeksowania.

Poniższa sekcja opisuje ogólne zasady używane w rozwiązywaniu problemów semantycznej równoważności zapytań oryginalnych, wejściowych i przetworzonych przez

optymalizator. Większość poniższych zasad dotyczy optymalizacji zapytań zakresowych. Optymalizator analizuje prawy argument niealgebraicznego operatora *where* biorąc pod uwagę wszystkie predykaty selekcji połączone operatorami koniunkcji (*and*) i alternatywy (*or).*

Podstawowa procedura optymalizacji przez indeksowanie działa na selektywnych zapytaniach gdzie lewą stronę operatora *where* stanowi kolekcja zaindeksowana przez jeden lub więcej indeksów. Algorytm analizuje wszystkie predykaty selekcji połączone operatorem *and* i próbuje znaleźć indeks, którego klucz pasuje do predykatów. Jeżeli więcej niż jeden indeks zostaje znaleziony, optymalizator wybiera jeden z najlepszą selektywnością (na podstawie istniejącego modelu kosztów)

4.1. Składnia użycia indeksów

Z punktu widzenia składni języka SBQL inwokacja indeksu jest po prostu wywołaniem procedury:

```
<indexname>( <key_param_1> [; <key_param_2> ...] )
```

Liczba parametrów jest równa liczbie kluczy indeksu. Każdy parametr klucza definiuje pożądaną wartość kluczową. Wywołanie funkcji indeksu zwraca referencje do obiektów odpowiadających określonym kryteriom.

Wyrażenie parametru klucza może definiować pojedynczą wartość jako kryterium. W takim przypadku ewaluacja powinna zwrócić wartość typu *integer*, *double*, *string*, *reference* lub *Boolean* albo wskaźnik do takowej wartości. Poniżej zaprezentowaliśmy wywołanie dla przykładowego indeksu idxDeptName:

idxDeptName(“HR” **groupas** *$equal*)

Pojedyncza wartość kluczowa może być przekazana przez wartość *bindera* nazwanego “$equal”. *Bindery* są używane do zwiększenia czytelności oraz aby umożliwić łatwiejsze wprowadzanie typów parametrów dla wywołania indeksów.

Aby określić zakres jako kryterium wartości kluczowej, wyrażenie powinno zwracać strukturę składającą się z czterech parametrów:

(<lower_limit>, <upper_limit>, <lower_closed>, <upper_closed>)

gdzie:

- *<lower_limit>* i *<upper_limit>* są wartościami klucza określającymi zakres
- *<lower_closed>* jest wartością typu *Boolean* określającą czy *<lower_limit>* należy do zakresu kryterium
- *<upper_closed>* jest wartością typu *Boolean* określającą czy *<upper_limit>* należy do zakresu kryterium

Przykłady wywołań indeksów:

```
idxPerBirthYear((1978, 1982, true, true) groupas $range)
```

```
idxPerBirthYear((1900, (sum(Person.(2009 – age)) / count(Person)), true, false)
groupas $range);
```

Ostatni przykład zwraca referencje do osoby, której rok urodzenia jest wcześniejszy niż średnia wieku wszystkich osób zawartych w bazie danych. Podobnie jak w przypadku pojedynczych parametrów wartości klucza, parametry określające zakres są przekazywane z użyciem wartości *bindera* o nazwie „$range".

Parametr klucza może specyfikować również kolekcję wartości kluczowych jako kryterium.

```
idxEmpAge&WorkCity((25 union 30 union 35) groupas $in;
"Boston"groupas $equal)
```

Binder o nazwie "$in" jest użyty do przekazania kolekcji wartości kluczowych.

Jeśli kryterium zwraca zbiór pusty, wówczas wywołanie indeksu również zwraca pusty zbiór.

4.2. Równoważność semantyczna w optymalizacji obejmującej klucze opcjonalne

W pierwszej kolejności rozważmy jak liczność klucza [0..1] wpływa na optymalizacje. Używanie kryteriów o opcjonalnej liczności może spowodować wystąpienie błędu czasu wykonania, ponieważ predykat selekcji oparty na operatorach „=", „>", „≥", „<" i „≤" wymusza używanie pojedynczej wartości zarówno w przypadku lewego jak i prawego argumentu. Nieoczekiwana ilość wartości argumentów powoduje błąd czasu wykonania. Użycie wywołania indeksu w optymalizacji z takowymi predykatami wyeliminowałoby niebezpieczeństwo wystąpienia błędu a zatem zoptymalizowane zapytania nie byłyby semantycznie równe oryginalnym. W takich przypadkach optymalizacja jest dozwolona tylko jeśli operator *in* jest użyty jako predykat ponieważ nie ogranicza on liczności prawego argumentu.

Poniżej prezentowany jest przykład niebezpiecznej ewaluacji predykatu, który może spowodować błąd czasu wykonania. Lewa strona predykatu selekcji ma liczność [0..1] zgodnie z kardynalnością atrybutu kodu pocztowego (*zip*).

```
Person where address.zip = 94107
```

Aby uniknąć możliwości wystąpienia błędu czasu wykonania powinien być użyty „bezpieczny" operator *in:*

Person **where** 94107 **in** *address.zip*

W omawianym przypadku optymalizator indeksów wspiera optymalizacje, kiedy predykaty są zdefiniowane przy użyciu operatorów „=", „>", „≥", „<" i „≤" pod warunkiem użycia odpowiedniego predykatu egzystencjalnego *exists*. Przykład bezpiecznej ewaluacji predykatu z użytym operatorem „=" przedstawiony jest poniżej:

Person **where exists**(*address.zip*) **where** *address.zip* = 94107

Jedynie w przypadku dwóch ostatnich przykładów zapytań optymalizator wykorzystujący indeksy może stosować następującą transformację zapytania:

idxPerZip(94107 **groupas** *$equal*)

Minimalna liczność klucza równa zero oznacza, że indeks może nie zawierać referencji do wszystkich obiektów zdefiniowanej przy tworzeniu indeksu indeksowanej kolekcji. W przypadku indeksu na wielu kluczach, jeśli taki klucz byłby pominięty w predykacie selekcji, zaistniałaby możliwość, że ewaluacja operatora *where* zwróci referencje do obiektu, który nie jest przechowywany wewnątrz indeksu. Zatem optymalizator nie zastosowałby optymalizacji używając takiego indeksu. Reasumując, klucze z minimalną licznością równą zero są obligatoryjne, pomimo zadeklarowania przy pomocy wskaźnika typu *enum*.

4.3. Aspekty optymalizacji predykatów zakresowych

Jeżeli predykat selekcji zoptymalizowanego zapytania określa tylko jeden limit zakresu (dolny lub górny), wówczas drugi limit jest generowany automatycznie, tj. możliwie najmniejsza lub największa wartość dla danego klucza. Przykładowo poniższe zapytanie dotyczy oddziału zlokalizowanego w Warszawie, którego pracownicy sumarycznie zarabiają mniej niż najlepiej opłacany pracownik w całej firmie.

Zapytanie oryginalne:

Dept **where sum**(*employs.Emp.salary*) **<** **max**(*Emp.salary*)
and *address.city* = "Warszawa"

Zapytanie zoptymalizowane

idxDeptSalary((–2147483648, **max**(*Emp.salary*), **true**, **false**) **groupas** *$range*)
where *address.city* = "Warszawa"

Jeżeli mamy do czynienia z więcej niż jednym predykatem dotyczącym jednego limitu (np. dolnego) zakresu danego klucza, wówczas wyrażenia *min, max, union* oraz porównania są używane do wyznaczenia odpowiedniego parametru klucza zakresu.

Zapytanie oryginalne:

```
((sum(Person.(2009 – age)) / count(Person)) as avgyear).
(Person where 2009 – age > avgyear and 1970 <= 2009 – age
and 2009 – age < 1980)
```

Zapytanie zoptymalizowane:

```
(sum(Person.(2009 – age)) / count(Person)) as avgyear).
idxPerBirthYear((max(avgyear union 1970), 1980, 1970 > avgyear, false)
groupas $range)
```

4.4. Pomijanie kluczy w wywołaniu indeksu

W indeksach opartych na wielu kluczach, klucze *enum* mogą być zwykle pomijane w wywołaniach indeksów. Optymalizator indeksów, aby pominąć klucz, kiedy żaden predykat selekcji nie został określony, ustawia oba – dolne oraz górne ograniczenia – do najniższej oraz najwyższej możliwej wartości kluczowej.

Zapytanie oryginalne:

```
Person where true = married and address.city in „Wrocław”
```

Zapytanie zoptymalizowane:

```
idxPerAge&Mar&City ((–2147483648 , 2147483647 , true , true)
groupas $range; true groupas $equal ; „Wrocław” groupas $equal)
```

Aby pominąć klucz typu *boolean* w wywołaniu indeksu używa się zbioru kryteriów parametru kluczowego złożonego z dwóch wartości *fałsz* i *prawda*.

Zapytanie oryginalne:

```
Person where age > 30 and 33 >= age and address.city in „Wrocław”
```

Zapytanie zoptymalizowane:

```
idxPerAge&Mar&City((30 , 33 , false , true) groupas $range;
(false union true) groupas $in ; „Wrocław” groupas $equal)
```

4.5. Alternatywa predykatów oraz uwzględnienie dziedziczenia

Optymalizator indeksów jest przygotowany również do przetwarzania zapytań, w których predykaty selekcji są połączone operatorem *or*. Jako że dysjunkcja osłabia selekcję, komplikuje to również proces optymalizacji. Dlatego jeżeli zastosowanie indeksów jest możliwe bez uwzględnienia predykatów połączonych operatorem o*r,* wówczas optymalizator może pominąć głębsze analizy. W przeciwnym wypadku, aby sprawdzić wszystkie możliwości indeksowania, optymalizator usuwa operatory *or*, rozdzielając niealgebraiczne wyrażenie *where* na dwa częściowe wyrażenia selekcji. Obiekty zwracane przez obydwa wyrażenia mogą być zduplikowane więc istnieje potrzeba pozostawienia tylko różnych referencji obiektów; jest to osiągane poprzez użycie wyrażenia *uniqueref*. Indeksowanie umożliwia redukcje ilości danych przetwarzanych przez zapytania wtedy, gdy może być zastosowane do obu częściowych wyrażeń. Ta procedura jest rekursywna w przypadku, gdy mamy do czynienia z więcej niż jednym operatorem o*r.* Przeanalizujmy poniższy przykład optymalizacji.

Emp **where** *age* = 28 **and** *married* = **true and** (*address.city* = "Szczecin"
or "Szczecin" **in** worksIn.*Firm.address.city*)

Zapytanie może być podzielone przez optymalizator indeksów zgodnie z poniższą formą:

uniqueref((*Emp* **where** *age* = 28 **and** *married* = **true and** *address.city* =
„Szczecin") **union** (*Emp* **where** *age* = 28 **and** *married* = **true**
and „Szczecin" **in** *worksIn.Firm.address.city*))

Zależnie od obecnego modelu kosztów oraz istniejących indeksów, optymalizator może zastosować następującą transformację:

uniqueref(((*Emp*) *idxPerAge&Mar&City*(28 **groupas** *$equal*; **true**
groupas *$equal*; „Szczecin" **groupas** *$equal*))**union**
(*idxEmpAge&WorkCity*(28 **groupas** *$equal*; „Szczecin" **groupas**
$equal) **where** *married* = **true**)

Predykat selekcji oparty na wyrażeniach *age*, *married oraz address.city* dotyczy nadklasy klasy *EmpClass*, tj. *PersonClass* i z tego powodu administrator może wyposażyć całą kolekcję *Person* w indeks *idxPerAge&Mar&City*. Indeks ten może zwracać instancje, które nie należą do *EmpClass*, stąd optymalizator musi wykorzystać narzędzie do usuwania instancji nienależących do *EmpClass* z rezultatu inwokacji indeksu. Może to być zrealizowane poprzez operator koercji języka SBQL. Składnia operatora koercji została wzięta z typowej konwencji syntaktycznej znanej z takich języków jak C, C++ jako rzutowanie. W rezultacie wywołanie indeksu *idxPerAge&Mar&City* zwraca wynik, który jest automatycznie rzutowany na kolekcję *Emp*, ponieważ oryginalne zapytanie dotyczy tylko pracowników.

W zaprezentowanym podejściu do ponownego użycia indeksu w hierarchii dziedziczenia kolekcji, indeksy dotyczące klasy, która wprowadza dany klucz, są bardziej wszechstronne i uniwersalne, jako że mogą być użyte w optymalizacji zapytań adresowanych również do kolekcji określonych przez podklasy.

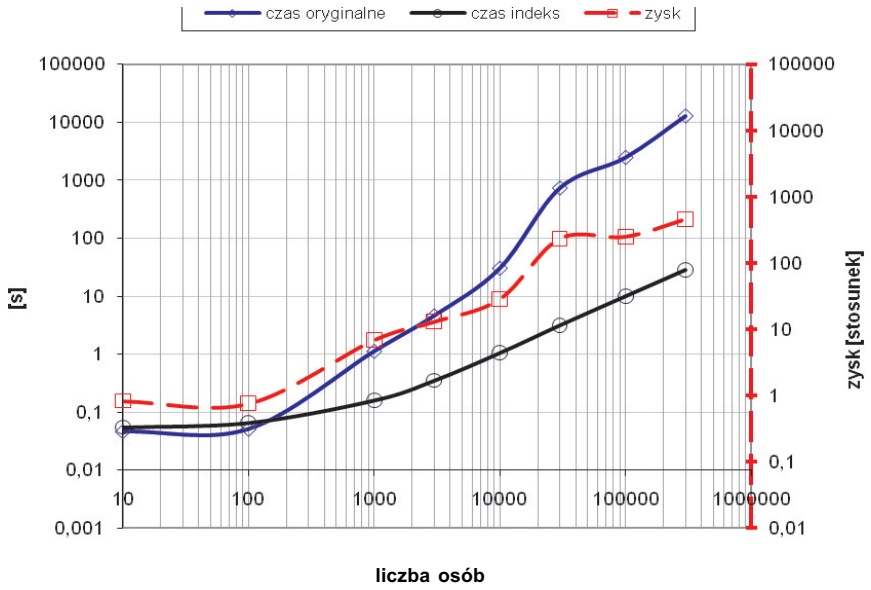
5. Zysk optymalizacji

Rozważmy następujący przykład testowy. Jeżeli wywołanie indeksu jest umieszczone po prawej stronie niealgebraicznego operatora, na przykład operatora „kropki", wówczas jest prawdopodobne, że będzie ono ewaluowane więcej niż jeden raz podczas wykonywania zapytania. Jest to przedstawione w następującym przykładzie z indeksem *idxEmpTotalIncomes*. Zapytanie w tabeli 1, dla 61-letnich pracowników w stanie małżeńskim, zamieszkujących w Łodzi lub Wrocławiu, zwraca imię połączone z nazwiskiem oraz liczbę pracowników z równą ilością całkowitych przychodów. Na rysunku 2 przedstawiono czasy wykonania obu postaci zapytania z tabeli 1 oraz zysk wynikający z wykorzystania indeksowania.

Testy prototypu indeksowania w bazie danych ODRA przeprowadzono na komputerze klasy PC z procesorem Intel Mobile Core 2 Duo T2300, 1,66 GHz, z pamięcią RAM 2,00 GB na systemie operacyjnym MS Windows Server 2003 R2 Service Pack 2, 32 bit.

Tabela 1

Zapytanie oryginalne	((*Emp* **where** *address.city* = "Łódź" **and** *worksIn.Dept.address.city* **in** ("Łódź" **union** "Wrocław") **and** *married* = **true and** *age* = 61) **as** *e*) .(*e.name* + " " + *e.surname*, **count**(*Emp* **where** *getTotalIncomes*() = *e.getTotalIncomes*()))
Zapytanie zoptymalizowane	((*Emp* **where** *address.city* = "Łódź" **and** *worksIn.Dept.address.city* **in** ("Łódź" **union** "Wrocław") **and** *married* = **true and** *age* = 61) **as** *e*) .(*e.name* + " " + *e.surname*, **count**(*idxEmpTotalIncomes*(*e.getTotalIncomes*()))

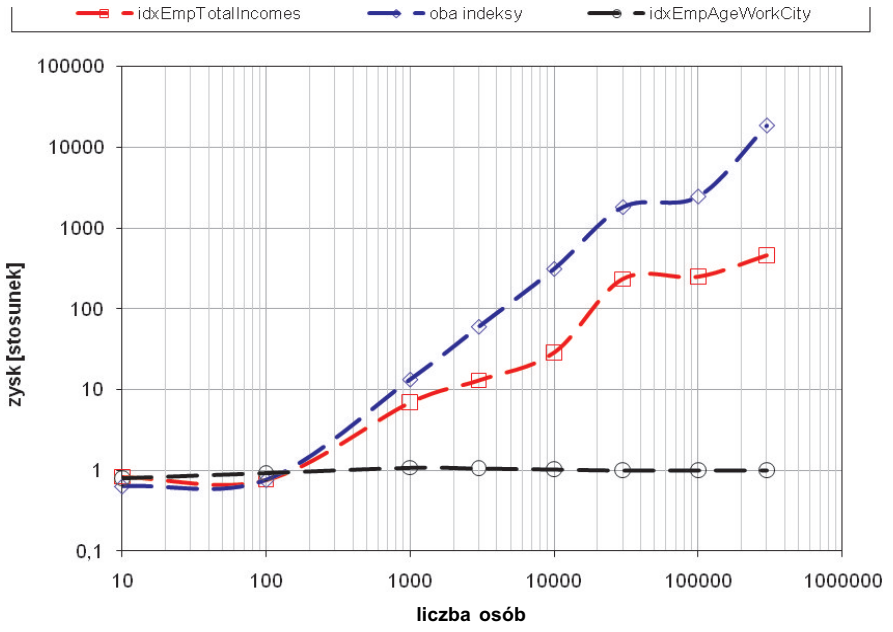


Rys. 2. Porównanie czasów wykonania i zysku optymalizacji zapytania

Dodatkowo wprowadźmy inny indeks – idxEmpAge&WorkCity – tak, aby zoptymalizować ewaluacje pierwszej części zapytania, co może znacząco wpłynąć na wzrost wydajności. W tabeli 2 wyszczególniono zapytania wykorzystujące różne kombinacje indeksów. Na rysunku 3 znajduje się wykres z odpowiadającymi im zyskami optymalizacji uzależnionymi od liczby osób w bazie danych.

Tabela 2

Zapytanie z indeksem *idxEmpTotalIncomes*	((*Emp* **where** *address.city* = "Łódź" **and** *worksIn.Dept.address.city* in ("Łódź" **union** "Wrocław") **and** *married* = **true and** *age* = 61) **as** *e*). (*e.name* + " " + *e.surname*, **count**(*idxEmpTotalIncomes*(*e.getTotalIncomes*()))
Zapytanie z indeksem *idxEmpAgeWorkCity*	((*idxEmpAge&WorkCity*(61 **groupas** *$equal*; ("Łódź" **union** "Wrocław") **groupas** *$in*) **where** *address.city* = "Łódź" **and** *married* = **true**) **as** *e*). (*e.name* + " " + *e.surname*, (*e.name* + " " + *e.surname*, **count**(*Emp* **where** *getTotalIncomes*() = *e.getTotalIncomes*()))))
Zapytanie z dwoma indeksami	((*idxEmpAge&WorkCity*(61 **groupas** *$equal*; ("Łódź" **union** "Wrocław") **groupas** *$in*) **where** *address.city* = "Łódź" **and** *married* = **true**) **as** *e*). (*e.name* + " " + *e.surname*, **count**(*idxEmpTotalIncomes*(*e.getTotalIncomes*()))



Rys. 3. Zysk optymalizacji indeksów dla zapytania

Dla bazy danych zawierającej 300000 obiektów odpowiadających osobom, dwa indeksy dają zysk w przybliżeniu 40 razy większy. Pomimo takiej różnicy najważniejszy jest indeks wołany wielokrotnie, tj. *idxEmpTotalIncomes*. Bez tego indeksu nie ma zauważalnej poprawy wydajności.

6. Wnioski i plany dalszych prac

W niniejszym artykule zostały skrótowo opisane zasady dotyczące tworzenia i wykorzystywania indeksów w prototypie ODRA. W zaprezentowanym podejściu optymalizacja jest osiągnięta poprzez opisaną transformację zapytań. Zaproponowana implementacja indeksacji w ODRA umożliwia tworzenie i przezroczystą, automatyczną aktualizację indeksów ułatwiających przetwarzanie predykatów selekcji opartych na dowolnych, deterministycznych wyrażeniach składających się z wyrażeń ścieżkowych, funkcji agregujących, inwokacji metod klas (włącznie z dziedziczeniem oraz polimorfizmem). Wszystkie funkcjonalności wymagane do kompletnego działania systemu indeksowania zostały już zaimplementowane. Niemniej jednak, indeksacja w ODRA nadal jest rozwijana i wymaga dalszych badań. Plany przyszłych prace obejmują zastosowanie innych struktur indeksów (np. B-Drzewa) oraz implementacji nowych metod optymalizacji, korzystających z zalet indeksów (np. optymalizacja zapytań rankingowych). Kolejne prace dotyczą *techniki ulotnego indeksowania*, która może być stosowana do przetwarzania danych heterogenicznych oraz wirtualnie udostępnionych poprzez perspektywy SBQL. Technika ta wykazuje skuteczność w przetwarzaniu złożonych zapytań, w których indeks jest wywoływany więcej niż jeden raz. Dodatkowo rozważamy rozszerzenia możliwości indeksowania na rozproszone środowisko przy użyciu skalowalnego i rozproszonego indeksu SDDS.

Literatura

[1] Burleson D., T*urbocharge SQL with advanced Oracle9i indexing*. March 26, 2002, http://www.dba-oracle.com/art_9i_indexing.htm.
[2] Elmasri R., Navathe S.B., *Fundamentals of Database Systems 4th ed.* Pearson Education, Inc., 2004, ISBN: 83-7361-716-7.
[3] GemStone Systems, Inc. www.gemstone.com.
[4] Java API User Guide ObjectStore, Release 7.1 for all platforms, August 2008.
[5] Litwin W., *Linear Hashing: a new tool for file and tables addressing.* Reprinted from VLDB-80 in READINGS IN DATABASES. 2-nd ed., Morgan Kaufmann Publishers, Inc., 1994, Stonebraker, M. (Ed.).
[6] Litwin W., Nejmat M.A., Schneider D.A., *LH*: Scalable, Distributed Database System.* ACM Trans. Database Syst., 21(4), 1996, 480–525.
[7] Meier D., Stein J., *Indexing in an object-oriented DBMS.* Proceedings of the OODBS, IEEE Computer Society Press, 1986, 171–182.
[8] O'Neil P.E., Quasi D., *Improved Query Performance with Variant Indexes.* Proceedings of SIGMOD, 1997, 38–49.
[9] Objectivity for Java Programmer's Guide, Release 9.3, October 13, 2006.

[10] Oracle9i Data Warehousing Guide Release 2 (9.2). Part Number A96520-01.
[11] Płodzień J., *Optimization Methods In Object Query Languages*. IPIPAN, Warszawa, 2000 (Ph.D. Thesis).
[12] Płodzień J., Kraken A., *Object Query Optimization in the Stack-Based Approach.* Proc. of 3rd ADBIS Conf., Maribor, Slovenia, Springer LNCS 1691, 1999, 303–316.
[13] SBA & SBQL Web pages: http://www.sbql.pl/.
[14] Subieta K. *Theory and Construction of Object-Oriented Query Languages* (in Polish). PJIIT – Publishing House, 2004, 522.
[15] VERSANT Database Fundamentals Manual, (Release 7.0.1.0) July 2005.