Implementacja języka zapytań oparta na drzewach rozbioru

(Implementation of a query language
based on partition trees)

Damian Górski

Praca inżynierska

Promotor: dr Wiktor Zychla

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

30 czerwca 2017 r.

Damian Górski

.....

.....
(adres zameldowania)

.....

.....
(adres korespondencyjny)

PESEL: .....

e-mail: .....

Wydział Matematyki i Informatyki
stacjonarne studia I stopnia
kierunek: informatyka
nr albumu: 273212

Oświadczenie
o autorskim wykonaniu pracy dyplomowej

Niniejszym oświadczam, że złożoną do oceny pracę zatytułowaną *Implementacja języka zapytań oparta na drzewach rozbioru* wykonałem/am samodzielnie pod kierunkiem promotora, dr Wiktora Zychli. Oświadczam, że powyższe dane są zgodne ze stanem faktycznym i znane mi są przepisy ustawy z dn. 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity: Dz. U. z 2006 r. nr 90, poz. 637, z późniejszymi zmianami) oraz że treść pracy dyplomowej przedstawionej do obrony, zawarta na przekazanym nośniku elektronicznym, jest identyczna z jej wersją drukowaną.

Wrocław, 30 czerwca 2017 r.

(czytelny podpis)

Streszczenie

Tutaj coś będzie.

Something goes here.

Spis treści

Rozdział 1.

Preliminaria

W celu zrozumienia mechanizmu budowy zapytania SQL-owego, trzeba najpierw zrozumieć sposób działania języka LINQ, który jest punktem wejścia, oraz struktury drzewa rozbioru składniowiowego, będącym przedmiotem translacji LINQ-to-SQL. Zakładam, że czytelnikowi znane są podstawowe pojęcia związane z programowaniem obiektowym, takie jak metoda, kolekcja, dziedziczenie, typ generyczny. W niniejszym rozdziale poruszone zostaną następujące tematy:

- Sposób przetrzymywania kolekcji wyliczalnych w .NET-cie.
- Opis i motywacja powstania języka zapytań LINQ.
- Struktura drzewa wyrażeń `IQueryable`, i dlaczego takie drzewa są trudne do odwiedzania w celu zrealizowania zadania LINQ-to-SQL.
- Biblioteka re-linq uproszczająca powyższe drzewa, obiekty `QueryModel`.

1.1. Słowo o `IEnumerable<T>` i `IQueryable<T>`

We frameworku .NET wszystkie kolekcje, które możemy wyliczyć (a takie nas interesują, bo pracujemy z relacyjną bazą danych), implementują interfejs `IEnumerable<T>`, gdzie `T` jest typem obiektu, który jest przetrzymywany w kolekcji. Ten interfejs definiuje metodę `GetEnumerator()`, który zwraca obiekt typu `IEnumerator<T>`, który ma właściwość `Current` oraz metodę `MoveNext()`, pozwalając na przejście po uporządkowanym ciągu obiektów typu `T` oraz określenie obecnej pozycji. Korzystając z tych dwóch informacji, jesteśmy w stanie rozszerzyć `IEnumerable<T>` o metody takie jak wyznaczenie długości, filtrowanie kolekcji, łączenie dwóch kolekcji ze sobą, mapowanie funkcji na wszystkie obiekty znajdujące się w kolekcji. Dokładna lista metod rozszerzających `IEnumerable<T>` jest dostępna w oficjalnej dokumentacji MSDN.

Rozszerzeniem `IEnumerable<T>` jest interfejs `IQueryable<T>`, który de facto implementuje `IEnumerable<T>`. Zasadniczą różnicą między tymi dwoma interfejsami jest to, że w momencie wywołania ciągu metod rozszerzających `IEnumerable<T>`, każda z tych metod jest wywoływana jedna po drugiej, co może obciążyć moc obliczeniową procesora. Natomiast kolekcja `IQueryable<T>` jest świadoma, że nie musi wykonywać tych metod od razu, tylko przetrzymuje je w postaci drzewa wyrażeń (o wyrażeniach w następnej sekcji), które dopiero przy wywołaniu metody wyliczającej elementy z kolekcji zostaje wykonane w całości w efektywny sposób. Takie rozwiązanie jest idealne dla kolekcji, które łączą się z zewnętrzną bazą danych, aby istniała możliwość wybrania danych za pomocą jednego dużego zapytania SQL-owego.

1.2. Language INtegrated Query

Programiści na codzień pracują z danymi w różnych formach - zapisanych w plikach XML i JSON, przetrzymywanych w bazie danych, czy też po prostu z kolekcjami obiektów. Nie jest sztuką zauważyć, że trudnością dla programisty będzie odnalezienie się w projekcie, który korzysta z wielu źródeł danych, ponieważ wybranie danych z każdego z nich wymaga znajomości metod używania tych źródeł. To dało do myślenia architektom z Microsoftu, którzy „postanowili uogólnić problem [wyboru danych] i dodać możliwość wykonywania zapytań w sposób kompatybilny ze wszystkimi źródłami danych, nie tylko relacyjnymi i XML-owymi. Rozwiązanie to nazwali **L**anguage **IN**tegrated **Q**uery" [1], i zostało bardzo ciepło przyjęte przez programistów .NET. Zapytanie LINQ jest automatycznie tłumaczone do docelowego języka zapytań, którego programista C# lub VB nie musi znać - a więc jest w stanie wybierać dane z niemal każdego źródła z użyciem tej samej składni.

Poniżej zostało przedstawione przykładowe zapytanie LINQ, które wybiera imiona i nazwiska osób z kolekcji pracowników, którzy zarabiają więcej niż 3000 złotych, posortowane alfabetycznie po nazwiskach:

```
var linqQuery =
    from e in db.Employees
    where e.Salary > 3000.0
    orderby e.LastName
    select new
    {
        FirstName = e.FirstName,
        LastName = e.LastName
    };
```

Takie zapytanie można również zapisać za pomocą metod z użyciem wyrażeń lambda (powyższe zapytanie jest tłumaczone przez kompilator do poniższego):

```
var linqQuery2 = db.Employees
    .Where(e => e.Salary > 3000.0)
    .OrderBy(e => e.LastName)
    .Select(e => new
    {
        FirstName = e.FirstName,
        LastName = e.LastName
    });
```

Pisząc zapytanie LINQ, tak naprawdę wykonywane są metody na kolekcjach `IEnumerable<T>`, z którymi była okazja zapoznać się w trakcie czytania sekcji traktującej o kolekcjach, które implementują ten interfejs. Każde z tych zapytań zwraca kolekcję `IEnumerable<T>` (w przypadku danych wybieranych z zewnętrznego źródła - `IQueryable<T>`), gdzie `T` jest typem anonimowym zawierającym dwie właściwości `FirstName` i `LastName`. Tą kolekcję można w łatwy sposób przerzutować na dowolną kolekcję używając odpowiedniej metody (na przykład `.ToList()` albo `.ToArray()`). Przykłady bardziej skomplikowanych zapytań można znaleźć w folderze `Thesis.Relinq.Tests` w plikach z rozszerzeniem `.cs` zawierających klasy testujące system, który stanowi załącznik do tej pracy.

1.3. Drzewa wyrażeń `IQueryable`

Wynikiem zapytania LINQ jest obiekt, który implementuje interfejs `IQueryable`. Poniższy fragment kodu pochodzi z biblioteki .NET i pokazuje sposób, w jaki `IQueryable` rozszerza `IEnumerable`:

```
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}
```

Pierwsza właściwość zawiera oczywiście typ obiektów, których kolekcja jest wynikiem zapytania. Trzecia właściwość to instancja klasy, który implementuje interfejs `IQueryProvider`. Dostarczenie takiej implementacji jest zadaniem programisty, i o tym traktuje następna część rodziału. Natomiast przedmiotem tej sekcji jest właściwość druga, o tajemniczym typie `Expression`.

Prawdziwym „zapytaniem” ukrytym pod interfejsem `IQueryable` jest obiekt `Expression`, który reprezentuje wejściowe zapytanie LINQ jako drzewo operatorów i metod, które zostały w tym zapytaniu użyte [2]. Po głębszej analizie kodu źródłowego biblioteki .NET okazuje się, że `IQueryable` jest tak naprawdę mechanizmem wykorzystującym metody typowe dla kolekcji do budowania drzewa rozbioru

składniowego w postaci obiektu `Expression`, który (wraz z `ElementType`) jest wykorzystywany przez `Provider` do wykonania zapytania.

Może się wydawać, że mamy wszystko - przecież wystarczy zaimplementować `IQueryProvider` w taki sposób, by tłumaczył drzewa `Expression` na zapytanie do języka, który nas interesuje. Okazuje się, że te drzewa mogą być problematycznym modelem do odwiedzania. Idąc śladem Fabiana Schmieda [3], weźmy na warsztat pewne zapytanie i zobaczmy, w jaki sposób obiekt `Expression` jest budowany:

```
var linqQuery3 =
    from c in QueryFactory.CreateLinqQuery<Customer>()
    from o in c.Orders
    where o.OrderNumber == 1
    select new { c, o };
```

Na początku, takie zapytanie jest tłumaczone na równoważny ciąg wywołań metod (równie dobrze programista mógł napisać w kodzie to, co jest poniżej):

```
QueryFactory.CreateLinqQuery<Customer>()
    .SelectMany(c => c.Orders, (c, o) => new {c, o})
    .Where(trans => trans.o.OrderNumber == 1)
    .Select(trans => new {trans.c, trans.o})
```

Kompilator tłumaczy powyższe wywołania metod na wywołania statycznych metod `IQueryable`, oraz opakowuje wyrażenia lambda w obiekty `Expression.Lambda`, które są ich abstrakcyjną reprezentacją:

```
Queryable.Select(
  Queryable.Where(
    Queryable.SelectMany(
      QueryFactory.CreateLinqQuery<Customer>(),
      Expression.Lambda(Expression.MakeMemberAccess(...)),
      Expression.Lambda(Expression.New(...))),
    Expression.Lambda(Expression.MakeBinary(...))),
  Expression.Lambda(Expression.New (...)))
```

Z tej reprezentacji korzystają obiekty `IQueryable`, które budują poszukiwany obiekt `Expression`, który wreszcie jest abstrakcyjną reprezentacją zapytania, które jest przekazywane do dostawcy LINQ w celu budowy zapytania:

```
MethodCallExpression("Select",
  MethodCallExpression("Where",
    MethodCallExpression("SelectMany",
      CostantExpression(IQueryable<Customer>),
      UnaryExpression(...), UnaryExpression(...)),
    UnaryExpression(...)),
  UnaryExpression(...))
```

W tym miejscu warto zauważyć, że `Expression` jest oczywiście tylko klasą abstrakcyjną dla klas określających konkretne wyrażenia, które po niej dziedziczą, takie jak `MethodCallExpression`, `UnaryExpression`, czy `BinaryExpression`.

Problemem z drzewami `Expression` jest fakt, że kolejność wykonywanych metod nie jest z góry określona - **jakakolwiek** metoda może nastąpić po **jakiejkolwiek** metodzie, przez co drzewa bardzo szybko stają się skomplikowane. Ponadto, jedna metoda może służyć w kilku kontekstach, np. `SelectMany` może służyć zarówno za część odpowiadającą za budowę podzapytania, jak również wybór dodatkowego źródła danych (następna tabela dla części `FROM` zapytania SQL-owego). Ponadto, dostawca LINQ musi przejść po wszystkich wyrażeniach lambda nawet na samą górę drzewa, aby znaleźć odpowiedni kontekst, o który chodziło użytkownikowi w zapytaniu. Stąd wniosek nasuwa się jeden - budowa dostawcy LINQ, który ma większe możliwości niż podstawowe operacje na pojedynczej tabeli, jest trudnym zadaniem, jeśli chciałoby się to zrobić na drzewach `Expression`.

Kończąc powyższe rozważania, Schmied zauważył że logika przetwarzania drzew `Expression` jest w każdym dostawcy LINQ niepotrzebnie duplikowana. W tym miejscu zadał pytanie: „*Czy inteligentniejszym rozwiązaniem nie byłaby **jednokrotna** implementacja logiki przetwarzania drzew w sposób generyczny, z której mogą korzystać wszyscy dostawcy LINQ*”? To pytanie było motywacją do powstania biblioteki re-linq. Autor pracy dyplomowej skorzystał z tej biblioteki, i o sposobie jej działania oraz użycia poświęcona została cała następna sekcja.

1.4. re-linq i obiekty `QueryModel`

W sekcji traktującej o drzewach wyrażeń `IQueryable` dowiedziono, że ze względu na skomplikowaną strukturę tych drzew, budowa zapytania docelowego na podstawie tych drzew jest trudna. W związku z tym, alternatywnym rozwiązaniem jest wspomniana już biblioteka re-linq, która tłumaczy drzewa wyrażeń `IQueryable` na drzewa rozbioru składniowego o wiele przystępniejsze do przeglądania, a dokładniej na obiekty `QueryModel`, które o wiele bardziej przypominają oryginalne zapytanie LINQ. Te obiekty mają cztery właściwości:

- `SelectClause` - klauzula `SelectClause` określająca element, który jest wybierany w zapytaniu `select` z końca zapytania).
- `MainFromClause` - klauzula `MainFromClause` określająca główne źródło, z którego wybierane są informacje w zapytaniu (najbardziej zewnętrzny `from`).
- `BodyClauses` - zbiór wyrażeń implementujących `IBodyClause`, które definiują jakie dane są wybierane w zapytaniu i w jakiej kolejności (słowa kluczowe `where`, `orderby`, `join`, wewnętrzne `from`-y, które są przetrzymywane w klauzulach `AdditionalFrom`).

- `ResultOperators` - zbiór wyrażeń dziedziczących po `ResultOperatorBase`, które wykonują logikę na zbiorze wynikowym (na przykład metody agregujące `Count()`, `Average()`, `Distinct()` i im podobne, operacje na zbiorach `Union()`, `Distinct()` i im podobne).

Biblioteka re-linq, poza przekształceniem obiektów `Expression` na `QueryModel`, pozwala również na znaczne uproszczenie implementacji `IQueryProvider`, udostępniając klasę abstrakcyjną `QueryableBase`, po której dziedziczy klasa budująca zapytanie docelowe. Klasa ta musi posiadać metodę `CreateQueryProvider`, która zwraca obiekt typu `IQueryProvider` wykorzystywany przez `IQueryable`. Takim obiektem może być oferany przez re-linq `DefaultQueryProvider`, który jest budowany z trzech argumentów: typu docelowego implementującego `IQueryable`, obiektu `QueryParser` dokonującego translacji drzewa `Expression` do obiektu `QueryModel` (istnieje możliwość napisania własnego tłumacza, ale autor pracy korzysta z domyślnego, który został dostarczony razem z biblioteką), oraz własnej implementacji interfejsu `IQueryExecutor` (patrz: `Thesis.Relinq/PsqlQueryable.cs`). Taka implementacja powinna posiadać trzy metody:

- `IEnumerable<T> ExecuteCollection<T>(QueryModel queryModel)`,
- `T ExecuteScalar<T>(QueryModel queryModel)`,
- `T ExecuteSingle<T>(QueryModel queryModel, bool defaultWhenEmpty)`.

Wybór wywoływanej przez `IQueryExecutor` metody zależy od oczekiwanego wyniku zapytania (cała kolekcja, skalar, pojedynczy element z kolekcji). W rezultacie, pisząc zapytanie LINQ, dostajemy obiekt w pełni implementujący `IQueryable`, na którym wywołanie metody wyciągającej wynik z bazy danych zwróci wynik jednej z powyższych trzech metod. Teraz jedyne, co nas dzieli od oczekiwanego rezultatu, jest ich implementacja, która przechodząc przez drzewo `QueryModel` buduje zapytanie, wykonuje je korzystając z zewnętrznej biblioteki łączącą się z bazą danych PostgreSQL, konwertuje wynik zapytania do oczekiwanego typu i go zwraca.

Sposobem budowy zapytania na podstawie obiektu `QueryModel` jest implementacja wzorca projektowego Odwiedzający (Visitor), którego zadaniem jest przejście przez wnętrze tego obiektu. Biblioteka re-linq oczywiście udostępnia bazowe klasy abstrakcyjne, które wystarczy przeciążyć w celu wykonania tego zadania, i o tym poświęcony został następny rozdział niniejszej pracy. Przy okazji warto jeszcze wspomnieć, że biblioteka re-linq jest na tyle potężnym narzędziem, że na jej użycie zdecydowali się nawet autorzy NHibernate oraz Entity Framework 7, które są najpopularniejszymi bibliotekami ORM w .NET.

Rozdział 2.

Proces budowy zapytania

Znając sposób działania dostawców LINQ oraz budowę drzewa `QueryModel`, wystarczy opracować metodę przechodzenia przez te drzewa w celu budowy zapytania do bazy PostgreSQL. Punktem wejściowym dla projektu, który jest załącznikiem do niniejszej pracy, jest artykuł [4], opisujący przykładową implementację dostawcy LINQ dla NHibernate.

2.1. Implementacja `QueryModelVisitorBase`

Korzystając z dotychczasowej wiedzy, następnym krokiem do wykonania jest implementacja metod odwiedzających nowe drzewo rozbioru składniowego. Również w tym przypadku biblioteka re-linq asystuje programistę w tym zadaniu, udostępniając klasę `QueryModelVisitorBase`, która implementuje zbiór metod odwiedzających obiekt `QueryModel`. Stawianym przed programistą zadaniem jest napisanie klasy dziedziczącej po `QueryModelVisitorBase`, która wykona dodatkową logikę na argumentach implementowanych metod, oraz wywoła bazową logikę z użyciem słowa kluczowego `base` w celu akceptowania odwiedzanych elementów.

Argumentami każdej z metod, które będą nadpisywane, są różne klauzule - skondensowane do postaci wygodnych obiektów - które występują w zapytaniu LINQ. Ich właściwościami są znane już obiekty `Expression`, jednak są one na tyle proste, że można łatwo się zająć ich odwiedzeniem, i o tym będzie traktować następna sekcja tego rozdziału. Na chwilę obecną załóżmy, że posiadamy generyczną metodę, która odwiedza każdy możliwy podtyp `Expression`, i na jego podstawie buduje fragment zapytania SQL-owego. Taki fragment jest przekazywany do instancji klasy `QueryPartsAggregator`, służącej do łączenia takich fragmentów w pełne zapytanie SQL. Dokładna implementacja klasy, która jest tematem niniejszego podrozdziału, znajduje się w pliku `PsqlGeneratingQueryModelVisitor.cs`. Autor pracy zachęca czytelnika do zapoznawania się z nim w trakcie czytania następnych podsekcji.

2.1.1. Metoda `VisitQueryModel`

Punkt wejściowy dla całego procesu odwiedzania całego zapytania. Dla zadanego `QueryModel`, wywołuje metody `VisitSelectClause`, `VisitMainFromClause` oraz zbiór metod odwiedzających po kolei każdy z elementów właściwości `BodyClauses` i `ResultOperators`.

2.1.2. Metoda `VisitSelectClause`

Odwiedza klauzulę `SelectClause`, która definiuje właściwości obiektu, który zostanie zbudowany w wyniku zapytania (buduje część `SELECT` zapytania SQL-owego).

2.1.3. Metoda `VisitMainFromClause`

Odwiedza klauzulę `MainFromClause`, która definiuje źródło, na podstawie którego obiekt zostanie zbudowany w wyniku zapytania (dodaje pierwszą tabelę do części `FROM` w zapytaniu SQL-owym).

2.1.4. Metoda `VisitWhereClause`

W przypadku, gdy kolekcja `BodyClauses` zawiera klauzulę `WhereClause` (inaczej - zapytanie LINQ zawiera metodę `Where`), dodaje warunek, który wybrane dane muszą spełniać (dodaje element do części `WHERE` zapytania SQL-owego).

2.1.5. Metoda `VisitOrderByClause`

W przypadku, gdy kolekcja `BodyClauses` zawiera klauzulę `OrderByClause` (zapytanie LINQ zawiera metodę `OrderBy` lub `OrderByDescending`), dodaje porządek, według którego dane zostaną posortowane (dodaje element do części `ORDER BY` zapytania SQL-owego).

2.1.6. Metoda `VisitJoinClause`

W przypadku, gdy kolekcja `BodyClauses` zawiera klauzulę `JoinClause` (zapytanie LINQ zawiera metodę `Join`), dodaje złączenie wewnętrzne (ang. *inner join*) do poprzedniego dodanego źródła danych w zapytaniu (dokleja `INNER JOIN [table]` do odpowiedniej części `FROM` zapytania SQL-owego, a dokładniej do tabeli, która jest łączona).

2.1.7. Metoda `VisitAdditionalFromClause`

W przypadku, gdy kolekcja `BodyClauses` zawiera klauzulę `FromClause` (zapytanie LINQ zawiera więcej niż jedną część `from`, która została zakumulowana razem z poprzednimi do większego obiektu metodą `SelectMany`), dodaje następne źródło, na podstawie którego obiekt zostanie zbudowany w wyniku zapytania (dodaje następną tabelę do części `FROM` zapytania SQL-owego po przecinku, co w rezultacie tworzy iloczyn kartezjański dwóch tabel, ang. *cross join*).

2.1.8. Metoda `VisitGroupJoinClause`

W przypadku, gdy kolekcja `BodyClauses` zawiera klauzulę `GroupJoinClause` (zapytanie LINQ zawiera metodę `GroupJoin`), dodaje lewostronne złączenie zewnętrzne (ang. *left join*) do poprzedniego dodanego źródła danych w zapytaniu (dokleja `LEFT JOIN [table]` do odpowiedniej części `FROM` zapytania SQL-owego, a dokładniej do tabeli, która jest łączona).

2.1.9. Metoda `VisitResultOperator`

W odróżnieniu od wszystkich powyższych klauzul, które implementują `IBodyClause`, re-linq niestety nie udostępnia wygodnego modelu odwiedzania dla obiektów `ResultOperatorBase`, w związku z tym ta metoda jest wywoływana dla każdego obiektu zawartego we właściwości `QueryModel.ResultOperators`.

Dostawca LINQ, który jest tematem niniejszej pracy, dzieli operatory wynikowe na pięć kategorii, zależnych od właściwego typu obiektu, który dziedziczy po `ResultOperatorBase` (każdy z nich nazywa się `SomeResultOperator`, dla prostoty każda nazwa została w poniższym spisie skrócona):

a) `Count`, `Average`, `Min`, `Max`, `Sum`, `Distinct` (operatory agregujące, które jako argument przyjmują zbiór wybranych danych i na ich podstawie zwraca pojedynczą wartość (w przypadku `Distinct` - unikatowe krotki)) - otacza wybraną część `SELECT` zapytania SQL-owego w odpowiadającą danemu operatorowi funkcję.

b) `Union, Intersect, Concat, Except` (operatory, które jako argumenty przyjmują zbiory danych i zwracają nowy zbiór) - sygnatury odpowiadających w języku C# metod w jednym ze swoich argumentów mają zbiór, na którym ma zostać wykonana dana operacja. Ten zbiór jest oczywiście kolejnym drzewem `Expression`, które zostaje przetłumaczone na `QueryModel`, w związku z tym budowane zostaje podzapytanie, a zapytanie końcowe jest wynikiem złączenia zapytania głównego i podrzędnego.

c) `Take`, `Skip` (operatory stronicowania) - dodaje do zapytania odpowiednią część odpowiedzialną za stronicowanie `(LIMIT X/OFFSET X)`.

d) `Any` (operator określający istnieje obiektu, który spełnia pewien warunek) - dolecowo użyty do wybierania obiektu na podstawie stwierdzenia, czy istnieje obiekt w wyniku innego zapytaniu, który spełnia podany na zewnątrz warunek. Poniższe zapytanie LINQ:

```
QueryFactory.CreateLinqQuery<Customer>()
    .Where(c => QueryFactory.CreateLinqQuery<Order>()
    .Any(o => o.CustomerID == c.CustomerID));
```

zostaje tłumaczone na odpowiadające mu zapytanie w SQL-u:

```
SELECT * FROM customers WHERE EXISTS
  (SELECT * FROM orders WHERE
    (customers.CustomerID = orders.CustomerID));
```

e) `All` (operator określający spełnienie pewnego warunku przez wszystkie obiekty w kolekcji) - dolecowo użyty do wybierania obiektu na podstawie stwierdzenia, czy wszystkie obiekty w wyniku innego zapytania spełniają podany na zewnątrz warunek. To stwierdzenie jest równoważne stwierdzeniu, że **nie istnieje** obiekt, który **nie spełnia** danego warunku (na zajęciach *Logika dla informatyków* w IIUWr można dowiedzieć się, że $\forall x\phi \Leftrightarrow \neg\exists x\neg\phi$). Korzystając z tego faktu, poniższe zapytanie LINQ:

```
QueryFactory.CreateLinqQuery<Customer>()
    .Where(c => QueryFactory.CreateLinqQuery<Order>()
    .All(o => o.CustomerID != c.CustomerID));
```

zostaje tłumaczone na następujące zapytanie w SQL-u:

```
SELECT * FROM customers WHERE NOT EXISTS
  (SELECT * FROM orders WHERE NOT
    (customers.CustomerID != orders.CustomerID));
```

2.2. Implementacja `RelinqExpressionVisitor`

Implementacja klasy `QueryModelVisitorBase`, opisana w rozdziale 2.1, zajmuje się odwiedzaniem obiektu `QueryModel` oraz przetwarzaniem wygenerowanych części zapytania SQL-owego do postaci pary napisu przedstawiającego zapytanie oraz słownika z parametrami. W tym rozdziale opisana została implementacja udostępnianej przez re-linq klasy abstrakcyjnej `RelinqExpressionVisitor`, która dziedziczy po .NET-owym `ExpressionVisitor`. Służy ona do generowania kluczowych części zapytania oraz parametrów, które dane zapytanie będzie wykorzystywać.

Argumentami każdej z metod, które będą nadpisywane, są obiekty dziedziczące po `Expression`. Na ich podstawie budowany jest napis w klasie `StringBuilder`, który po zakończeniu odwiedzania wyrażenia zostaje przekazany do omawianej już implementacji `QueryModelVisitorBase`.

2.2.1. Metoda `VisitQuerySourceReference`

Ta metoda odwiedza źródło danych, z którego wybrane zostaną dane. Rozpatrywane są dwa przypadki:

a) Źródło jest klauzulą `GroupJoinClause` - aby zrozumieć postać tej klauzuli, rozważmy najpierw następujące zapytanie:

```
from c in QueryFactory.CreateLinqQuery<Customer>()
join o in QueryFactory.CreateLinqQuery<Order>()
on c.CustomerID equals o.CustomerID into orders
select new
{
    Customer = c.CustomerID,
    Orders = orders
};
```

W niniejszym zapytaniu obiekt `orders` jest kolekcją `IEnumerable<Order>` zamówień wykonanych przez konkretnych użytkowników, a zapytanie wynikowe tworzy obiekty anonimowe postaci numeru ID klienta i kolekcji zamówień, które dany klient zamówił. Przetłumaczenie tego zapytania do SQL jest trudne, ze względu na konieczność grupowania kolekcji i zwrócenia jej w postaci obiektu. Rozwiązanie, które wykorzystuje LINQ to SQL - oraz biblioteka autora pracy - jest dosyć sprytne: wykonywane jest złączenie zewnętrzne lewostronne tablicy grupującej z grupowaną, całość zostaje posortowana względem porównywanych kluczy, oraz dodawana jest nowa kolumna, która jest wynikiem podzapytania zliczającego obiekty w każdej grupie. Powyższe zapytanie LINQ-owe tłumaczone jest na:

```
SELECT
  customers.CustomerID AS CustomerID, [...],
  (SELECT COUNT(*) FROM orders AS temp
    WHERE temp.CustomerID = customers.CustomerID)
  AS Orders.__GROUP_COUNT
FROM customers LEFT OUTER JOIN orders ON
  customers.CustomerID = orders.CustomerID
ORDER BY customers.CustomerID, orders.CustomerID;
```

`GroupJoinClause` posiada właściwość `JoinClause`, z której wybierane są właściwości `[Outer/Inner]KeySelector`, na podstawie których doklejany zostaje powyższy kawałek zapytania umożliwiający grupowanie danych z poziomu LINQ.

b) Źródło nie jest klauzulą `GroupJoinClause` - w tym przypadku odwiedzana jest po prostu tablica w bazie danych. W zależności od tego, czy obecne wywołanie metody zostało wykonane przez metodę `VisitMember` lub nie, do zapytania doklejana jest nazwa tablicy lub ciąg postaci `[tablica].[kolumna1], [tablica].[kolumna2], ...`, który definiuje całą tablicę `tablica`.

2.2.2. Metoda `VisitSubQuery`

Wyciąga z `SubQueryExpression` dodatkowy `QueryModel`, buduje na jego podstawie zapytanie i dodaje je do nadrzędnej klasy obsługującej budowę głównego zapytania. Jest to jedyna metoda, która nie generuje napisu w `StringBuilder`, a wykonuje logikę bezpośrednio na obiekcie odwiedzającym `QueryModel`. Takie zapytania są później łączone w całość za pomocą odpowiadających im operatorów wynikowych.

2.2.3. Metoda `VisitBinary`

Wyrażenia `BinaryExpression`, jak można się domyślić, mają jako właściwości wyrażenia `Left` i `Right` oraz operator łączący je. Odwiedza lewe wyrażenie, dokleja do wyniku napis odpowiadający operatorowi łączącemu, odwiedza prawe wyrażenie.

2.2.4. Metoda `VisitConditional`

Rozważmy następujące zapytanie LINQ:

```
from e in QueryFactory.CreateLinqQuery<Entity>()
select new
{
    Result = (e.Property < 5
        ? "less than five"
        : e.Property == 5
            ? "five"
            : "more than five")
};
```

W ramach przypomnienia: operator `?` jest operatorem warunkowym, który ewaluuje wyrażenie boolowskie i zwraca wartość przed dwukropkiem dla prawdy, po dwukropku dla fałszu. W kontekście budowy zapytania, jest ono przetrzymywane w postaci `ConditionalExpression`, które zawiera właściwości `Test`, `IfTrue`, `IfFalse`. W szczególności, w `IfTrue` i `IfFalse` może być następne wyrażenie warunkowe. Metoda `VisitConditional` przechodzi po drzewie takich wyrażeń i tłumaczy je do SQL z użyciem funkcji `CASE`. Dla powyższego zapytania, odpowiadające mu zapytanie SQL wygląda następująco:

```
SELECT
  CASE WHEN entities.Property < 5 THEN 'less than five'
       WHEN entities.Property = 5 THEN 'five'
       ELSE 'more than five'
  END AS "Result"
FROM entities;
```

2.2.5. Metoda `VisitConstant`

Aby zapobiec atakowi typu SQL injection, należy parametryzować zapytanie. Odwiedzając wartość stałą (jest nią np. napis, liczba, itp.), metoda tworzy nowy parametr w zapytaniu, nadaje mu nazwę i dokleja tą nazwę do zapytania.

2.2.6. Metoda `VisitMember`

Odwiedzana przy wyborze właściwości z modelu tabeli w bazie danych. Dokleja do zapytania (do części `SELECT`) napis `[table].[column]`, pozwalając na wybór pojedynczych kolumn w wyniku zapytania.

2.2.7. Metoda `VisitMethodCall`

Opakowuje metodę C#-ową w odpowiadającą funkcję w zapytaniu SQL-owym. Przekazuje tej funkcji argumenty w sposób określony przez jej sygnaturę, po czym dopisuje dany fragment zapytania do bufora.

2.2.8. Metoda `VisitNew`

W przypadku, gdy zapytanie LINQ zwraca nowy obiekt anonimowy, ta metoda pozwala na przejście po wszystkich właściwościach nowego obiektu i ich odwiedzenie.

2.2.9. Metoda `VisitUnary`

Wykorzystywana w negacji wyrażenia boolowskiego lub do przekazania tego wyrażenia jako `MemberExpression`.

2.3. Czynności wykonywane po budowie zapytania

Potrafiąc zbudować dowolne zapytanie SQL-owe na podstawie zapytania LINQ, pozostaje już tylko kwestia wykonania go i zwrócenia wyniku w postaci obiektowej. Zapytanie może wykonać dowolny obiekt `DbConnection` kompatybilny z PostgreSQL (na przykład pochodzący z biblioteki Npgsql), rozszerzony przez bibliotekę Dapper o metody `Query` i `Query<T>`, które umożliwiają wykonanie zapytania, którego wynik jest automatycznie rzutowany do postaci obiektowej.

Korzystając z Dappera, wykonywanie i mapowanie zapytań jest banalnie proste:

```
var result = connection.Query<T>(statement, parameters);
```

W tym miejscu zmienna `result` jest typu `IEnumerable<T>`, gdzie `T` jest typem obiektu, który zwraca oryginalne zapytanie LINQ (jest ono typu `IQueryable<T>)`. Dapper wymaga, aby nazwy kolumn w relacjach mapowanych na obiekty były takie same, jak nazwy właściwości w klasie modelowej - stąd, podczas budowy zapytania, każda kolumna została przemianowana na nazwę odpowiadającej właściwości w klasie z użyciem słowa kluczowego `AS`.

Dapper niestety nie radzi sobie z typami anonimowymi ze względu na to, że nie istnieje publiczny konstruktor obiektów tego rodzaju. Obejściem tego problemu jest dostarczona przez autora pracy metoda rozszerzająca `QueryAnonymous<T>`. Dla typów zawierających tylko i wyłącznie właściwości proste (inaczej mówiąc: dla krotek, które zawierają tylko kolumny z bazy danych), wystarczy każdą kolumnę przerzutować do postaci tablicy i za pomocą statycznej metody `Activator.CreateInstance` utworzyć nowy obiekt anonimowy. Metoda ta pozwala również na grupowanie obiektów dla zapytań używających metody `GroupJoin` (patrz: 2.2.1a), która korzysta z dodatkowego pola w celu określenia liczby obiektów grupowanych.

Rozdział 3.

Testy jakości i wydajności

Tutaj coś jeszcze dzisiaj napiszę.

3.1. Możliwe funkcjonalności

Autor niniejszej pracy podczas realizowania projektu używał techniki *test-driven development*, tj. przed implementacją pewnej funkcjonalności napisał test, który ją pokrywa. Wszystkie testy są dostępne w katalogu `Thesis.Relinq.Tests`.

3.2. Wydajność a inne rozwiązania

Z poprzedniej sekcji wynika, że przeciętny student informatyki w ciągu czterech miesięcy jest w stanie napisać w ramach pracy dyplomowej dostawcę LINQ z nietrywialnymi funkcjonalnościami. Czas sprawdzić wydajność takiego dostawcy, porównując go z komercyjnym LinqConnect firmy DevArt, dołączonym do próbnej wersji biblioteki dotConnect for PostgreSQL 7.9 Professional, oraz open-sourcowym LINQ to DB, które jest rozwijane od kilku lat. W celu sprawdzenia wydajności tych trzech dostawców LINQ dla PostgreSQL, użyta została biblioteka do testowania wydajności w .NET o nazwie BenchmarkDotNet.

Rozdział 4.

Podsumowanie

Tutaj coś będzie jak złapię wenę.

Bibliografia

[1] Don Box, Anders Hejlsberg, LINQ: .NET Language-Integrated Query, 2007.

[2] Matt Warren, LINQ: Building an IQueryable Provider, 2007.

[3] Fabian Schmied, re-linq - A General Purpose LINQ Foundation, 2009.

[4] Markus Giegl, re-linq | ishing the Pain: Using re-linq to Implement a Powerful LINQ Provider on the Example of NHibernate, 2010.

[5] Microsoft, 101 LINQ Samples, 2012.

[6] The PostgreSQL Global Development Group, Documentation, 1996-2017.

Dodatek A

Instrukcja obsługi

Tutaj coś będzie jak złapię wenę.