

# 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

<b>1. Preliminaria</b>	<b>7</b>
1.1. Słowo o IEnumerable<T> i IQueryable<T> . . . . .	7
1.2. Language INtegrated Query . . . . .	8
1.3. Drzewa wyrażeń IQueryable . . . . .	9
1.4. re-linq i QueryModel . . . . .	10
<b>2. Proces budowy zapytania</b>	<b>13</b>
2.1. Sekcja . . . . .	13
<b>3. Testy jakości i wydajności</b>	<b>15</b>
3.1. Sekcja . . . . .	15
<b>4. Podsumowanie</b>	<b>17</b>
<b>Bibliografia</b>	<b>19</b>
<b>Dodatki</b>	<b>21</b>
A Instrukcja obsługi . . . . .	21



# 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 składniowego, 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 **Language INtegrated Query**” [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 rozdziału. 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.

W tym momencie może się wydawać, że mamy wszystko - wystarczy zaimplementować `IQueryProvider` w taki sposób, by tłumaczył drzewa `Expression` na zapytanie do języka, który nas interesuje. Po dłuższej analizie tych drzew okazuje się, że są one bardzo niewygodnym modelem do odwiedzania. TODO.

## 1.4. re-linq i QueryModel

W sekcji traktującej o drzewach wyrażeń `Expression` można zobaczyć, że ze względu na skomplikowaną strukturę tych drzew, budowa docelowego zapytania może być (i jest) trudna. W związku z tym, alternatywnym rozwiązaniem jest biblioteka `re-linq`, która tłumaczy drzewa wyrażeń `IQueryable` na tytułowe abstrakcyjne drzewa składniowe, a dokładniej - na obiekty `QueryModel`, które o wiele bardziej przypominają oryginalne zapytanie LINQ. Jest ono rozbite na cztery właściwości:

- `SelectClause` - zawiera wyrażenie określające element, który jest wybierany w zapytaniu LINQ.
- `MainFromClause` - określa główne źródło, z którego wybierane są informacje (w przypadku zapytań SQL - pierwsza tabela z części `FROM`).
- `BodyClauses` - TODO.
- `ResultOperators` - TODO.

Biblioteka `re-linq`, poza przekształceniem obiektów `Expression` na `QueryModel`, pozwala również na znaczne uproszczenie implementacji `IQueryProvider`, udostępniając interfejs `QueryableBase`, po którym dziedziczy klasa budująca zapytanie docelowe. Klasa ta musi posiadać metodę `CreateQueryProvider`, która zwraca obiekt typu `IQueryProvider` wykorzystywany przez `IQueryable`. Takim obiektem jest oferowany 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 łączącej 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

Tutaj coś będzie jak złapię wenę.

### 2.1. Sekcja

Tutaj też.



## Rozdział 3.

# Testy jakości i wydajności

Tutaj coś będzie jak złapię wenę.

### 3.1. Sekcja

Tutaj też.





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



## Dodatek A

# Instrukcja obsługi

Tutaj coś będzie jak złapię wenę.