Języki i środowiska przetwarzania danych rozproszonych

Optymalizacja poprzez modyfikację zapytań

Wykładowca: **Tomasz Kowalski**

Wstęp - Jak bardzo LINQ jest deklaratywne?

- Przykład *maxQuery*, zapytanie złożone:

```
from Product p in products where
  (from Product p2 in products select p2.unitPrice).Max()
      == p.unitPrice
select p.productName
```

- Przy założeniu ewaluacji LINQ to Objects:
  - Dwie zagnieżdżone pętle iterujące:
    - zewnętrzna po *p*,
    - wewnętrzna po *p2*.
  - Zapytanie zagnieżdżone będzie ewaluowane dla każdego produktu *p*.
  - Złożoność obliczeniowa kwadratowa (zamiast oczekiwanej liniowej).

- Czy **optymalizacja** działania takiej konstrukcji jest prostym zadaniem?

Tuning zapytania – trudności/wyzwania

- Podstawowe założenia optymalizacji:
  - poprawna dla możliwych danych wejściowych,
  - dla tego samego stanu źródła danych ten sam rezultat zapytania,
  - nie gorszy czas działania w kluczowych miejscach.
- Potencjalne wyzwania optymalizacji laziness-seeking constructs:
  - zmiana stanu źródła danych w czasie wykonania,
  - zachowanie miejsca wykonania zapytania (nie w miejscu definicji),
  - nieskończone źródła danych (np. unikanie materializacji),
  - itd..
- Czy tuning ma brać uwagę efekty uboczne?
- Zachowanie innych właściwości konstrukcji właściwych dla języka.

Cechy funkcyjnych Collection Pipeline

- Zestaw cech zależy od konkretnego języka

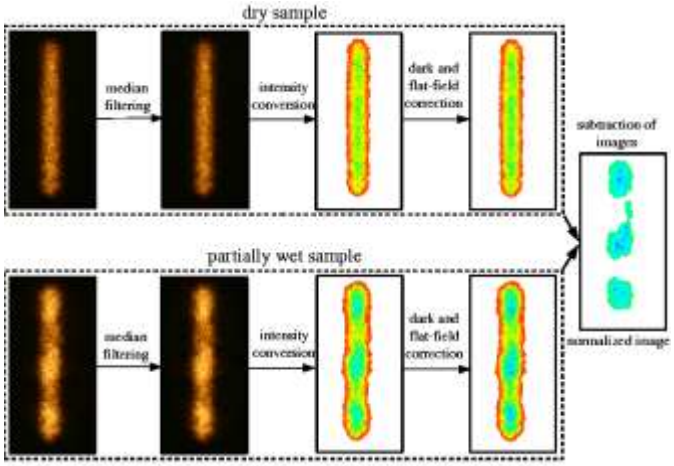
- Java 8 Streams API:
  - **No storage**
  - **Functional in nature**
  - **Laziness-seeking**
  - **Possibly unbounded**
  - **Consumable**

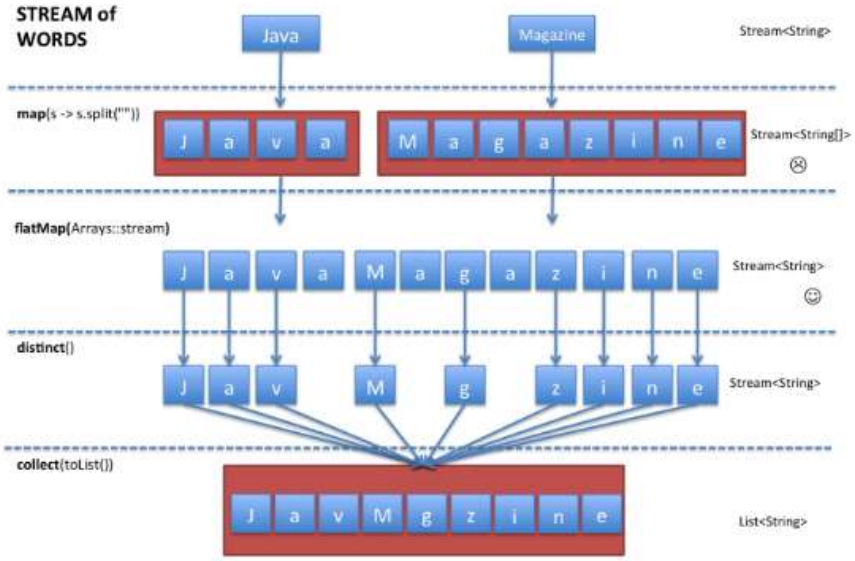
No storage & Possibly Unbounded

- to dane **iterowalne** (no storage) reprezentujące różne typy danych:
  - dane skończone
    - kolekcje utrwalone
      - bazy danych (sql, nosql)
      - pliki XML
    - kolekcje ulotne
      - tablice/kolekcje obiektów w pamięci
  - dane nieskończone **(possibly unbounded)**
    - dane generowane
      - liczby losowe
      - liczby pierwsze
    - strumienie I/O
      - zdarzenia (akcje GUI)
      - pisanie na klawiaturze
      - ruchy i akcje mysz
      - big-data (np. transakcje, komentarze, tweety)

Functional in nature & Laziness-seeking

- operacje na kolekcjach nie modyfikują danych:
  - np. filtracja nie usuwa elementów ze źródła danych
  - minimalizacja efektów ubocznych
  - ułatwia optymalizacje zapytań





- dwa typy „**laziness-seeking**” operatorów :
  - pełne **lazy evaluation** – np. *filter* i *map*
  - tylko **deferred execution** (pełna materializacja na żądanie) – np. w LINQ *GroupBy* oraz *OrderBy*

Consumable

- zapytania collection pipeline mogą być **jednorazowe** np.
  - ewaluacja natychmiastowa (**eager evaluation)** – np. operatory *map* i *filter* w python.

- Czy zapytania LINQ są **consumable**?
  - NIE

- Czy leniwa ewaluacja może być **consumable**?
  - OCZYWIŚCIE (np. Java 8 Streams)
  - jedna instancja iteratora na strumień

Tuning niezależnego pod-zapytania (1)

- Przykład ze wstępu bez syntax-sugar:

```
var maxQuery = products.Where(p =>
       products.Max(p2 => p2.unitPrice) == p.unitPrice).
    Select(p => p.productName);
```

- Naiwna optymalizacja (rozbicie na dwie instrukcje)

```
var nestedMaxPrice = products.Max(p2 => p2.unitPrice);
var maxQuery = products.
               Where(p => nestedMaxPrice == p.unitPrice).
               Select(p => p.productName);
```

- Wady rozwiązania:
  - ***nestedMaxPrice*** materializowane nie obok miejsca wykonania ***maxQuery***, więc rezultat ***maxQuery*** polega na potencjalnie nieaktualnej wartości ***nestedMaxPrice***,
  - ...?

Tuning niezależnego pod-zapytania (2)

- Przykład ze wstępu bez syntax-sugar:

```
var maxQuery = products.Where(p =>
       products.Max(p2 => p2.unitPrice) == p.unitPrice).
    Select(p => p.productName);
```

- Natychmiastowa materializacja w ramach funkcji:

```
var nestedMaxPrice = products.Max(p2 => p2.unitPrice);
return products.
                Where(p => nestedMaxPrice == p.unitPrice).
                Select(p => p.productName).ToList();
```

- Wady rozwiązania:
  - każde wykonanie zapytania tworzy nowe iteratory (drobiazg),
  - mniejsza deklaratywność (kwestia względna),
  - **dla pustej kolekcji products zapytanie zmodyfikowane rzuci wyjątkiem!**

Tuning niezależnego pod-zapytania (3)

- Odroczenie wykonania pod-zapytania:

```
var maxPriceThunk =
    new Lazy<Double>(() => products.Max(p2 => p2.unitPrice));
return products.
                Where(p => maxPriceThunk.Value == p.unitPrice).
                Select(p => p.productName).ToList();
```

- ***Thunk*** – objekt w pamięci reprezentujący niewykonane (zawieszone w czasie) obliczenie (wykorzystywane w strategii call-by-need).

- Wady rozwiązania (w przypadku ogólnym):
  - klienta zapytania może interesować nie cały rezultat ale np. kilka pierwszych elementów (spowolnienie działania programu),
  - nie jest to tuning zawsze bezpieczny, bo materializuje wynik zapytania a potencjalnie źródło danych może być nieskończone,
  - zmiana algorytmu (wpływ na wystąpujące efekty uboczne).

Pod-zapytanie skorelowane.

- Przykład *uniqueQuery*, zapytanie złożone:

```
var uniqueQuery = products.Where(p =>
      products.Where(p2 =>
         p.category.Equals(p2.category)).Count() == 1).
   Select(p => p.productName);
```

- Przy założeniu ewaluacji LINQ to Objects:
  - Dwie zagnieżdżone pętle iterujące:
    - zewnętrzna po *p*,
    - wewnętrzna po *p2*.
  - Zapytanie zagnieżdżone będzie ewaluowane dla każdego produktu *p*.
  - Złożoność obliczeniowa kwadratowa.
  - Możliwa implementacja quasi-liniowego rozwiązania.

- Jak zoptymalizować przy użyciu LINQ?

Tuning skorelowanego pod-zapytania (1)

- Przykład *uniqueQuery*, zapytanie złożone:

```
var uniqueQuery = products.Where(p =>
      products.Where(p2 =>
         p.category.Equals(p2.category)).Count() == 1).
   Select(p => p.productName);
```

- Alternatywne zapytanie przy użyciu ***GroupBy***:

```
var uniqueQueryAlt = products.GroupBy(p => p.category).
   Where(prodGroup => prodGroup.Count() == 1).
   SelectMany(prodGroup =>
      prodGroup.Select(p => p.productName))
```

- Wady rozwiązania:
  - grupowanie wymaga materializacji (dodatkowa pamięć),
  - kolejność rezultatów może być inna niż w oryginale,
  - dla kategorii **null** oryginalne zapytanie rzuca wyjątkiem (inna semantyka).

Tuning skorelowanego pod-zapytania (2)

- Zastosowanie indeksu (***Lookup*** w C#) + podział na dwie instrukcje + odroczenie + ToList + funkcja:

```
var productsLookupThunk = new Lazy<Double>(
    () => products.ToLookup(p2 => p2.category));
return products.Where(p =>
       productsLookupThunk.Value[p.category].Count() == 1).
    Select(p => p.productName).ToList();
```

- Wady rozwiązania:
  - indeks wymaga materializacji (dodatkowa pamięć),
  - dla kategorii **null** oryginalne zapytanie rzuca wyjątkiem (inna semantyka).

- Zachowuje kolejność oryginalnego rezultatu.
- Prostsza reguła trasformacji (**Lookup** zastępuje **Where**)!

Podsumowanie

- Ograniczona deklaratywność z uwagi na wydajność zapytań.

- Niemożliwa w pełnie przezroczysta optymalizacja:
  - zamierzone i niezamierzone (przez dewelopera) efekty uboczne,
  - zbyt małe możliwości wyrazu LINQ.

- Rafy semantyczne optymalizacji:
  - leniwa, odroczona lub natychmiastowa ewaluacja,
  - wyjątki,
  - materializacja a nieskończone źródła danych.