Katedra
Informatyki i Automatyki
Politechniki Rzeszowskiej

Bazy Danych

5. Optymalizacja zapytań

Opracował: Maciej Penar

Spis treści

1. Zanim zaczniemy

PRZYPOMNIJMY SOBIE ALGEBRĘ RELACJI

Materiały:

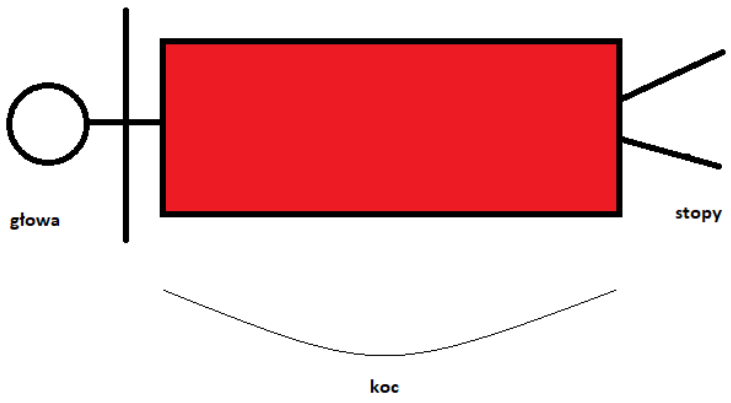
- Google: https://www.google.pl/search?q=algebra+relacji&oq=algebra+relacji
- Podstawowy kurs systemów baz danych, rozdział 2 oraz 5.2, J. Ullman, J. Widom

OPTYMALIZACJA BAZ DANYCH

Optymalizacja tzw. strojenie baz danych (ang. Database Tuning / Query Tuning) to proces ujednolicenia wydajności bazy danych – najczęściej zapytań oraz komend DML (INSERT/DELETE/UPDATE).

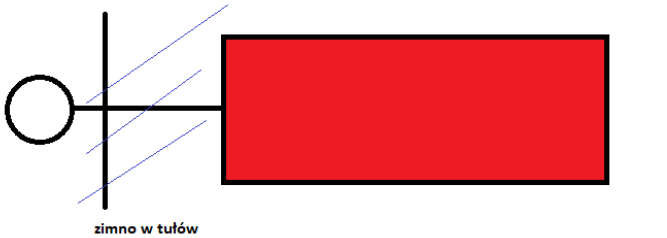
KOCYK

Strojenie bazy danych można porównać do problemu zimnej nocy i krótkiego kocyka. Mamy człowieka z krótkim kocem w nocy, tak jak na moim pięknym rysunku:



Mając ograniczony zasób – kocyk, możemy:

- Przesunąć kocyk na stopy – i będzie nam zimno w tułów, ale ciepło w stopy
- Przesunąć kocyk na brzuch – i będzie nam zimno w stopy, ale ciepło w brzuch



W zależności od tego jaki efekt chcemy osiągnąć – należy przesunąć kocyk w odpowiednie miejsce. Nie musi być to pozycja skrajna – być może jesteśmy w stanie tolerować zimne palce u stóp, bo wolimy mieć ciepły brzuszek.

Złośliwy powiedziałby: „i tak źle, i tak niedobrze”

KOCYK, A BAZY DANYCH

Analogicznie do przypadku kocyka – Bazę Danych w radykalnym scenariuszu możemy dostroić:

- Albo do operacji DML:
  - czyli mieć tanie operacje INSERT – skrajnie O(1)
  - ale każdy SELECT/UPDATE/DELTE – kosztem O(n)
- Albo do operacji odczytu:
  - Czyli SELECT/UPDATE/DELETE – kosztem O(logN) [punktowy odczyt]
  - INSERT – kosztem O(logN) – dla każdego indeksu

Podsumowując: podczas strojenia Bazy Danych nie można mieć ciasta i zjeść ciasta.

ARSENAŁ

Do szukania szeroko rozumianej wydajności w Bazach Danych mogą służyć nam:

STRUKTURA TABELI:

Do dyspozycji mamy różne struktury które służą do fizycznej alokacji danych. Pliki stertowe (powiedzmy, że to listy łączone) oraz Indeks Klastrujący (inna nazwa B+ drzewo).

Struktura	Opis
Pliki stertowe	Niedomyślna struktura w SQL Serverze. Dla SELECT’ów punkt wejścia to wykonanie przeglądu całej przestrzeni danych. Tanie INSERT’y. Tabela **nieposortowana**.
Indeks klastrowany	Rekordy w tej strukturze są fizycznie posortowanie wg. klucza klastrującego. INSERT’y wymagają znalezienia pozycji na której wsadzamy nowy rekord. SELECTy na warunku klucza klastrujacego (lub jego części) są tanie – tak samo zapytania zakresowe.

W SQL Serverze jeśli tabel ma indeks pomocniczy typu CLUSTERED to traktowana jest jako B+ drzewo. Czasem to widać która tabela jest B+ drzewem, czasem nie.

Poniższa tabela jest B+ drzewem:

```
CREATE TABLE Test_1(
	ID INT PRIMARY KEY
);
```

Bo SQL Server tak naprawdę wykonuje:

```
CREATE TABLE Test_1(
	ID INT PRIMARY KEY CLUSTERED
);
```

Jeśli chcemy składować dane nieposortowane to w SQL Serverze PK musimy jawanie oznaczyć jako:

```
CREATE TABLE Test_2(
	ID INT PRIMARY KEY NONCLUSTERED
);
```

Pamiętajcie:

Używając plików sterowych – tabela jest nieposortowana, przez co odczytu nie da się efektywnie wykonać. Nie bez indeksów pomocniczych.

Z drugiej strony – indeks klastrujący wymusza na BD utrzymywanie ustalonego porządku (co wpływa na INSERT)… i spoiler alert: BD **sprawia wrażenie** jakby dane były uporządkowane **logicznie,** ale fizycznie posortowane są tylko **bloki** danych. To oznacza, że procesor musi sortować „w locie" dane na stronie (co robi szybko, ale co wpływa marginalnie na szybkość SELECT).

INDEKSY – POMOCNICZE STRUKTURY DOSTĘPOWE:

Indeksy to nic innego jak przepisanie tabeli i „uzyskanie świadomości" przez BD, że w przepisanej tabeli istnieje porządek. Porządek który można eksploatować w celu przyśpieszania niektórych zapytań… kosztem złożoności operacji INSERT. Idealną sytuacją jest gdy tabela bazowa w żaden sposób nie jest angażowana do wykonania zapytania.

Do utworzenia indeksu służy wyrażenie CREATE INDEX:

```
CREATE INDEX [nazwa] ON [tabela]([kolumna_1],…,[kolumna_2]);
```

Dobre praktyki związane z indeksami to:

- Indeksy pokrywające (tzw. Covering indexes - link) – czyli takie które zawierają dodatkowe kolumny – indeksy rosną, wymagają więcej miejsca, ale często nie trzeba dotykać tabeli bazowej (i najczęściej wykonywać swap na buforach bazy danych)
- Nie tworzyć dużo indeksów (książką Bill Karwin – SQL Antipatterns, rodział „Index Shotgun") – optymalizator może zgłupieć, bazy danych puchną, wolniejsze inserty. Myśleć o tym że **K indeksów to K-razy wolniejszy insert.**

W książce Billa Karwin – SQL Antipatterns, rodział „Index Shotgun"- znajduje się metodologia dostrajania indeksów tzw. MENTOR (Measure, Explain, Nominate, Test, Optimize, Rebuild), innymi słowy:

- Najpierw mierzymy wydajność dostrajanych zapytań
- Potem sprawdzamy plany wykonania zapytania
- Potem wybieramy atrybuty na których postawimy indeksy
- Stawiamy i testujemy
- Optymalizujemy pod kątem tego jak będziemy z tabeli korzystać
- Na koniec przebudowywujemy indeksy (link)

PARTYCJONOWANIE TABEL

W ramach ciekawostki:

Bardzo skuteczna (i niedoceniana) forma zwiększania wydajności Bazy Danych – zarówno INSERTów jak i SELECT'ów – kosztem małego narzutu na każde wchodzące zapytanie do Bazy Danych. **Jest to tak efektywna forma że jest dostępna zazwyczaj w wersji Enterprise**.

Istnieją dwie formy partycjonowania danych:

- W poziomie – czyli tabelę dzielimy wierszami
- W pionie – czyli dzielimy tabelę kolumnami

KOMPRESJA DANYCH

W ramach ciekawostki:

Kompresja danych = więcej danych na tej samej przestrzeni.

Więcej danych na tej samej przestrzeni = Więcej danych per bufor

Więcej danych per bufor = mniej swap'owania

Mniej swapowania = mniejsze średnie czasy odpowiedzi

Kompresja w bazach danych to darmowe przyśpieszenie…. Dostępne w wersji Enterprise.

JESZCZE UWAGI ODNOŚNIE INDEKSOWANIA

Utworzenie indeksu na tabeli jest porównywalne do utrzymywanie przez BD wyniku analogicznego SELECT-a. Na przykład:

```
CREATE INDEX SomeIndex ON Kontakt(TelDomowy, EMail);
```

Można przyrównać do posiadania przez BD dodatkowej tabeli, zgodnej co znaczenia z:

```
SELECT TelDomowy, Email FROM Kontakt ORDER BY TelDomowy, EMail;
```

Tak naprawdę to byłoby to takie zapytanie:

```
SELECT TelDomowy, Email + PRIMARY KEY FROM Kontakt ORDER BY TelDomowy, EMail;
```

I clue programu. **Mające takie dane:**

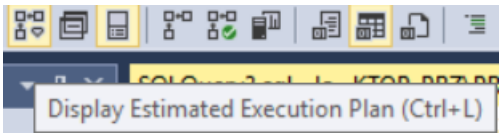
- **nie możemy efektywnie znaleźć E-maila dopóki nie mamy numeru telefonu – kolejność ma znaczenie**
- Nie możemy odpytać o żadną inną daną (poza TelDomowy i Email)

Ten indeks byłby wykorzystywany dla zapytań postaci operujących na TelDomowy lub <TelDomowy, Email>

SQL SERVER – PRZYGOTOWANIE DO INDEKSACJI

PLAN ZAPYTANIA

By wyświetlić plan zapytania (estymowany) należy nacisnąć CTRL+L



Plan powinien wyświetlić się w zakładce Execution Plan. Powinien przypominać drzewa Algebry Relacji – z tą różnicą, że operatory relacyjne powinny zostać zamienione na algorytmy realizujące je. Wyjątkowo w SQLServerze selekcja (WHERE) może być realizowane w każdym bloku.

SŁOWA KLUCZOWE

Scan oznacza przeczytanie całej struktury – i jest to raczej sytuacja niepożądana. **Seek** oznacza „wstrzelenie" się zapytania w indeks (bardzo dobrze). **Spool** oznacza tworzenie indeksu na poczekaniu (bardzo, bardzo źle). **Key Lookup** oznacza odwołanie się z jednego indeksu do drugiego – niby ok, ale meh… można lepiej.

Jeśli chodzi o złączenia to najmniej efektywnym jest algorytm **Nested Loop.** Hash join i Merge Join są prawie tak samo efektywne z tym zastrzeżeniem, że Merge Join **wymaga sortowania… i po fazie sortowania Merge Join jest nieblokujący. Hash join zawsze blokuje zwrot wyników dopóki nie wyliczy się całość złączenia.**

SQL SERVER – JAK INDEKSOWAĆ – CASE STUDY

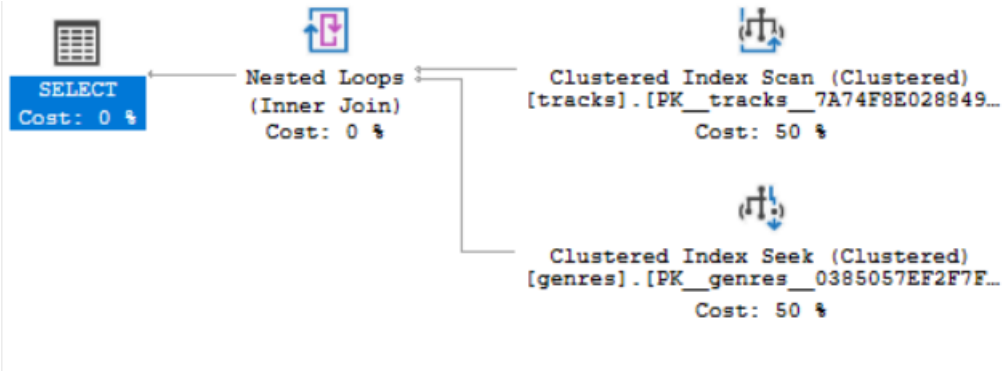
Najlepiej mieć dwie kopie bazy danych: bez indeksów i tą której będziemy dłubać. Załóżmy, że mamy tabele:

```
CREATE TABLE [dbo].[tracks](
	[TrackId] [int] PRIMARY KEY,
	[Name] [nvarchar](200) NOT NULL,
	[AlbumId] [int] NOT NULL,
	[MediaTypeId] [int] NOT NULL,
	[GenreId] [int] NOT NULL,
	[Composer] [nvarchar](200) NULL,
	[Milliseconds] [int] NOT NULL,
	[Bytes] [int] NOT NULL,
	[UnitPrice] [nvarchar](100) NOT NULL
);
CREATE TABLE dbo.genres(
	[GenreId] [int] PRIMARY KEY,
	[Name] [nvarchar](50)
 );
```

Oraz zapytanie pobierające te utwory i kompozytorów których utwory trwają od 1 do 2 minut i są klasyfikowane jako pop:

```
SELECT
	t.Name,
	t. Composer
FROM
	dbo.tracks t
	INNER JOIN dbo.genres g ON t.GenreId = g.GenreId
WHERE
	g.Name = 'Pop'
	AND t.Milliseconds BETWEEN 60000 AND 120000;
```

W bezindeksowym „vaniliowym” scenariuszu plan wygląda tak:

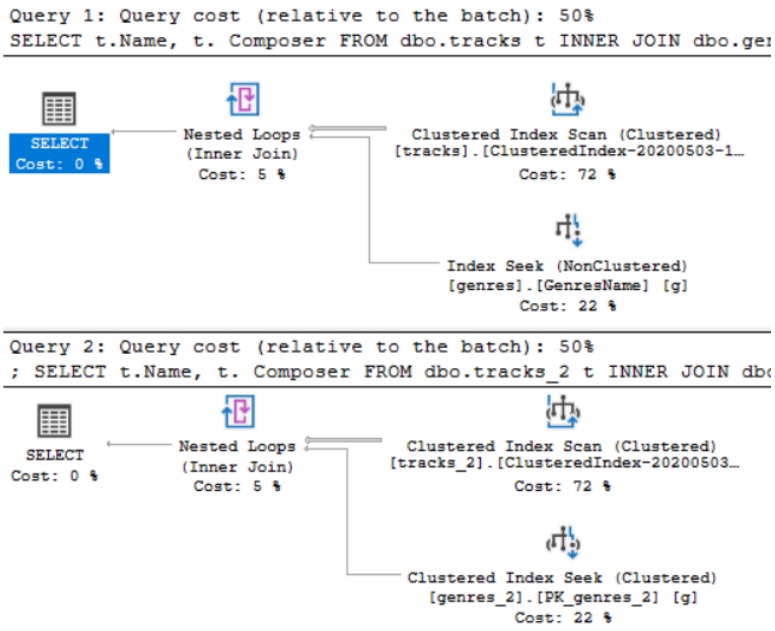


Mamy tutaj 2 pełny odczyt tabeli tracks. Dla każdego wiersza (Nested Loop) wykonywany jest pobranie wiersza z genres po genreId (`t.GenreId = g.GenreId`) – dlatego genres jest odczytywane w trybie **Index Seek.**

PODEJŚCIE NUMER 1

Stosujemy indeks:

```
CREATE INDEX GenresName ON dbo.genres(Name)
```

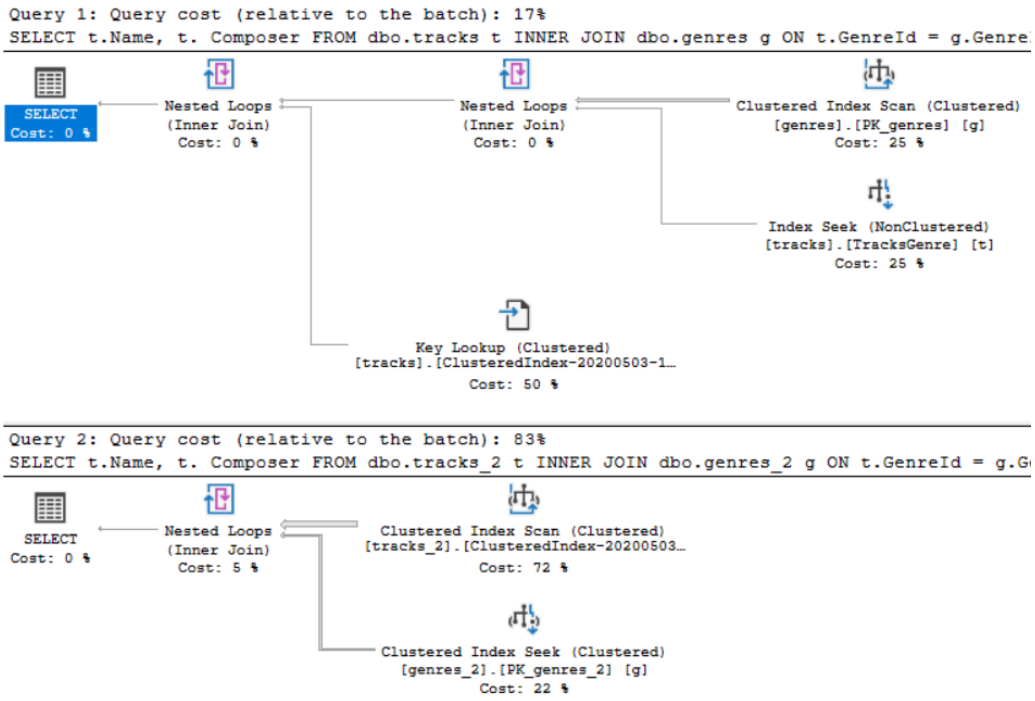


Pierwsze podejście okazuje się być fiaskiem – nowy plan w opozycji do starego ma taki sam udział w szacowanym czasie ☹. Jedyne co udało się ugrać, to zmianę klastrowanego indeksu na zwykły (a to żaden zysk).

PODEJŚCIE NUMER 2

Stosujemy indeks:

```
CREATE INDEX TracksGenre ON dbo.tracks(GenreId, Milliseconds);
```



Po dodaniu tego indeksu zapytanie zajmuje tylko 17% czasu (4 krotne przyśpieszenie). Problem w tym, że nasz indeks nie jest pokrywający – to oznacza, że w zapytaniu biorą udział: 2 tabele i indeks – stąd potrzeba dwóch złączeń.

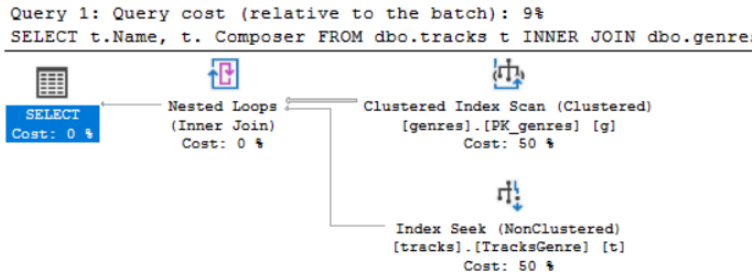
PODEJŚCIE NUMER 3

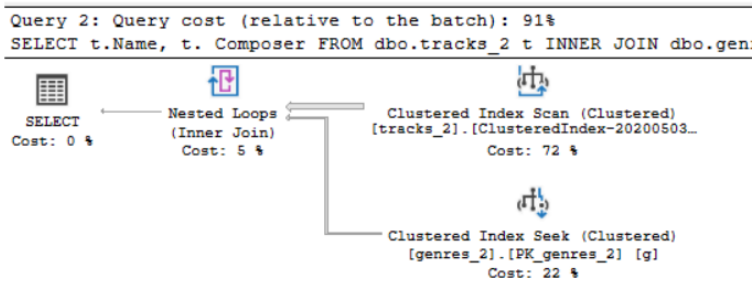
Stosujemy indeks:

```
CREATE INDEX TracksGenre ON dbo.tracks(GenreId, Milliseconds) INCLUDE(Name, Composer);
```

(Albo)

```
CREATE INDEX TracksGenre ON dbo.tracks(GenreId, Milliseconds, Name, Composer);
```



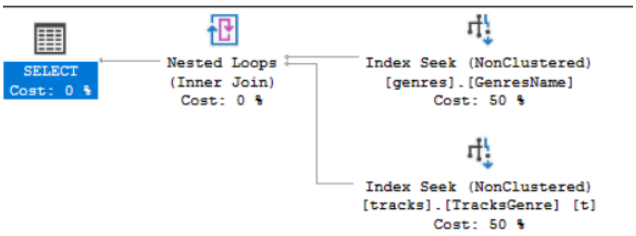


Po rozszerzeniu poprzedniego indeksu by pokrywał zapytanie (tj. by zawierał kolumny Name i Composer) zapytanie zajmuje tylko 9% czasu (10 krotne przyśpieszenie). Jeszcze pozostaje kwestia skanu na tabeli Genres.

Można to zrobić dodając indeks z pkt 1)

```
CREATE INDEX GenresName ON dbo.genres(Name)
```

Uzyskując tym samym plan:



I to nie powinno nigdy stęknąć.

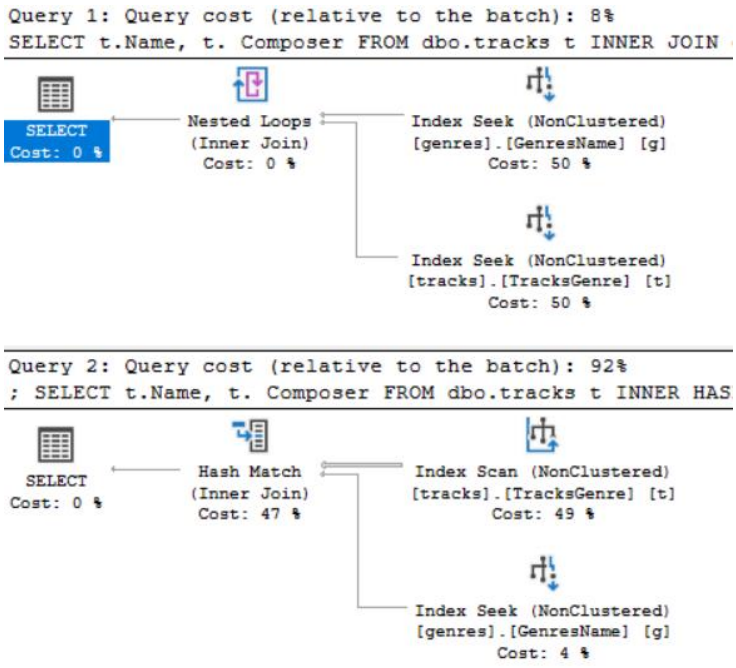
HINTY

Czasem jest tak, że może Wam się wydawać, że wiecie lepiej od kompilatora/optymalizatora jaki powinien być fizyczny plan wykonania zapytania. **Macie rację: wydaje Wam się.**

Ale gdyby jednak ktoś się upierał i chciał wymusić na BD np. konkretny algorytm wykonania złączenia to służą do tego podpowiedzi SQL (hints). Na przykład do wyrażenia INNER JOIN możemy dopisać hinty: LOOP/MERGE/HASH:

```
SELECT
	t.Name,
	t. Composer
FROM
	dbo.tracks t
	INNER HASH JOIN dbo.genres g ON t.GenreId = g.GenreId
WHERE
	g.Name = 'Pop'
	AND t.Milliseconds BETWEEN 60000 AND 120000;
```

W 99,999% przypadków osiągniecie taki efekt:



Ogólnie nie polecam – raz w życiu mi się zdarzyło, że trzeba było z hintów korzystać: case był taki, że optymalizator źle dobierał algorytm podczas procesu ETL. Dlatego ETL koniec końców zaprogramowaliśmy tak, że gdy leciał timeout to ponowienie próby dopisywało hint MERGE. Jak to timeoutowało, to HASH… i na koniec LOOP.

2. Zadanie

Dostosować poniższe zapytania do Waszej bazy danych, poindeksować, wykazać uzyskany zysk wydajnościowy. Zysk wykazywać w odniesieniu do wszystkich działających indeksów.

Podajcie statystyki IO, czasy wykonania zapytań niezoptymalizowanych/zoptymalizowanych na jakie rozwiązania się zdecydowaliście (szczególnie w query 3): `SET STATISTICS IO ON`

Sprawozdanie chcę otrzymać do 8 czerwca – UWAGA: jeśli chodzi o optymalizację BD jestem **dość** czepliwą osobą.

UPDATE: zmniejszcie ilość pamięci serwera do 512 MB RAM (PPM na serwer -> Properties -> Memory). Po czym trzeba zrestartować usługę bazodanową, żeby zmiana weszła w życie.

QUERY 1

```
SELECT DISTINCT
	m.Name
FROM
	ratings r
	INNER JOIN movies m ON r.movieId = m.movieId
	INNER JOIN user_info u ON u.userId = r.userId
WHERE
	u.Name LIKE '%Kowalski'
ORDER BY
	m.Name
```

QUERY 2

```
SELECT
	m.Name,
	AVG(r.rating)
FROM
	ratings r
	INNER JOIN movies m ON r.movieId = m.movieId
	INNER JOIN user_info u ON u.userId = r.userId
WHERE
	u.Name = 'Abel'
GROUP BY
	m.Name
```

QUERY 3

```
DECLARE @COUNTER INT;
SET @COUNTER = 0;
WHILE(@COUNTER < 5000)
BEGIN
	INSERT INTO ratings(userId, movieId, rating)
	VALUES(
		(SELECT ABS(CHECKSUM(NEWID()) % (SELECT MAX(userId) FROM users)),
		(SELECT ABS(CHECKSUM(NEWID()) % (SELECT MAX(movieId) FROM movies)) + 1,
		(SELECT ABS(CHECKSUM(NEWID()) % 5) + 1
	)
	SET @COUNTER = @COUNTER  + 1;
END
```