Universiteit Gent - TELIN - DDCM
2019-2020

Workshop: Query-optimalisatie (2)

1 Introductie

Tijdens de workshop ‘Query-optimalisatie 1’ hebben jullie al heel wat handige technieken geleerd om een `SELECT` query te optimaliseren. Op het moment dat de eerste workshop plaatsvond, hadden jullie echter enkel nog maar de basisconcepten van `SELECT` queries gebruikt. Nu zijn jullie (hopelijk) allemaal experts in het opvragen van data uit een databank en zijn subqueries, aggregatiefuncties en set-operaties voor niemand nog een geheim. In deze workshop breiden we daarom de lijst van technieken uit met tips om geavanceerdere queries te optimaliseren.
Let op, we bouwen meteen verder op alle concepten gezien tijdens de workshop ‘Query-optimalisatie 1’. Het is dus zeer sterk aangeraden om eerst deze workshop te maken vooraleer verder te gaan. Het uiteindelijke doel van beide workshops ‘Query-optimalisatie’ is om jullie in staat te stellen om een ‘slechte’ query te herkennen en technieken toe te passen om de query te verbeteren. Om dit in te oefenen, bestaat het grootste deel van deze workshop uit optimalisatie-oefeningen.

2 Eerste stappen

Om de effecten van bepaalde query-optimalisaties beter te kunnen aantonen, gaan we in deze workshop opnieuw werken met een iets grotere ‘Rolls Robin’ dataset. Opdat jullie deze nieuwe dataset niet opnieuw manueel zouden moeten gaan importeren in de verschillende tabellen, hebben we reeds een .sql-script (`rollsrobin_medium.sql`) voorzien dat een backup van de `rollsrobin`-databank bevat die gevuld is met grotere hoeveelheden data. Dit .sql-script vinden jullie op Ufora in de module van deze workshop.

Een fysieke implementatie van de `rollsrobin` databank samen met de geïmporteerde data bekomen jullie door deze backup in te laden (restore) op jullie lokale machine. Dit kan je doen via de commandolijn-applicatie `psql` met volgend commando.

```
psql
    --username postgres
    --file rollsrobin_medium.sql
```

Voor de volledigheid is het relationele databankschema van de 'Rolls Robin' databank terug te vinden in de Appendix.

> Zorg vooraleer verder te gaan dat je een volledig geïmplementeerde en gevulde `rollsrobin_medium` databank hebt aangemaakt op je lokale PostgreSQL-cluster.

3 Query-optimalisatietechnieken

In de eerste workshop zagen we optimalisatie-technieken zoals het vermijden van overbodige `JOIN`s, het toevoegen van indices,.... Hieronder gaan we aan de hand van twee voorbeelden nog een aantal dingen bekijken die je best kan vermijden.

3.1 Voorbeeld 1: wagens gehuurd door 1 persoon

In de derde reeks SQL-evaluatieoefeningen kregen jullie de vraag om alle unieke nummerplaten van wagens terug te geven die door slechts 1 persoon gehuurd zijn geweest. Een eerste poging om tot een correcte oplossing te komen is de volgende.

```
SELECT DISTINCT nummerplaat
    FROM registratieformulier r1
    WHERE nummerplaat NOT IN
        (SELECT nummerplaat
            FROM registratieformulier r2
            WHERE r1.email != r2.email);
```

> Voer bovenstaande query uit door er `EXPLAIN ANALYZE` voor te plaatsen. Indien je binnen de minuut geen resultaat krijgt, stop dan de uitvoering.

De bovenstaande query zal inderdaad de correcte oplossing geven, maar het grote probleem is dat de uitvoering lang duurt en de kost dus ook zeer hoog is. Voor elk registratieformulier wordt er namelijk gecheckt of de wagen (voorgesteld door een nummerplaat) binnen deze registratie reeds voorkomt in de verzameling van **alle** nummerplaten gehuurd door een ander persoon (voorgesteld door een e-mailadres). Dit zorgt dus intern voor de uitvoering van een geneste (en bovendien gecorreleerde) lus: de buitenste lus voor de selectie van uniek gehuurde nummerplaten en de binnenste lus voor de opbouw van de verzameling van nummerplaten gehuurd door een ander persoon. We zullen hieronder proberen om de query te optimaliseren.
Een mogelijke optimalisatie van bovenstaande query is de volgende.

```
SELECT DISTINCT nummerplaat
    FROM registratieformulier r1
    WHERE nummerplaat NOT IN
        (SELECT nummerplaat
            FROM registratieformulier r2
            WHERE r1.email != r2.email
                AND r1.nummerplaat = r2.nummerplaat);
```

Voer bovenstaande query opnieuw uit door er `EXPLAIN ANALYZE` voor te plaatsen.

Inderdaad, we moeten in de subquery niet kijken of de nummerplaat valt in de verzameling van alle nummerplaten gehuurd door een andere persoon. Het volstaat om te kijken of de huidige nummerplaat die in de buitenste query onderzocht wordt gehuurd is door een andere persoon. Je zal zien dat de uitvoering van deze query al redelijk snel gaat en de kost ook verlaagt. Het grote verschil met hierboven is dat de verzameling in de subquery **maximaal 1** nummerplaat kan bevatten, namelijk de nummerplaat van het registratieformulier uit de buitenste query. De verzameling zal deze nummerplaat bevatten indien er nog een andere persoon bestaat die deze wagen heeft gehuurd. Het blijft wel zo dat er een geneste lus uitgevoerd moet worden, maar het toevoegen van een extra conditie aan de `WHERE` clausule in de subquery zal ervoor zorgen dat er meer nummerplaten weggefilterd worden.
Tot slot nog een laatste poging om de query te optimaliseren.

```
SELECT DISTINCT r1.nummerplaat
    FROM registratieformulier r1
        LEFT JOIN registratieformulier r2
            ON r1.nummerplaat = r2.nummerplaat AND r1.email != r2.email
```

```
        WHERE r2.email IS NULL;
```

Voer bovenstaande query opnieuw uit door er `EXPLAIN ANALYZE` voor te plaatsen.

We zien dus dat we bij deze opgave helemaal geen subquery nodig hebben. We kunnen namelijk de juiste resultaten bekomen enkel door gebruik te maken van `JOIN`s met de juiste condities. Meer nog, de uitvoering gaat ook gewoon veel sneller en kost veel minder. Als we het queryplan bekijken zien we dat de query planner intern gebruik maakt van een 'Hash Anti Join'. Dit heeft als voordeel dat er geen geneste lussen meer uitgevoerd moeten worden om eventuele matches te zoeken, maar dat we nu aan 1 scan van registratieformulier, waarin intern een hashfunctie wordt uitgevoerd, genoeg hebben.
Wat we uit dit voorbeeld kunnen besluiten is dat, ook al is dit zeker geen vuistregel die altijd geldt, het vaak beter is om subqueries te vermijden, aangezien ze vaak voor geneste lussen zorgen, en te vervangen door een `JOIN`. Nu moeten we deze vaststelling in onderstaand voorbeeld wel even nuanceren.

Voer onderstaande query uit door er `EXPLAIN ANALYZE` voor te plaatsen.

```
SELECT DISTINCT nummerplaat
    FROM registratieformulier r1
    WHERE NOT EXISTS
        (SELECT 1
            FROM registratieformulier r2
            WHERE r1.email != r2.email
                AND r1.nummerplaat = r2.nummerplaat);
```

Indien je bovenstaande opdracht uitgevoerd hebt, zal je merken dat de uitvoering evenveel kost als onze oplossing zonder subquery. Je zou ook snel moeten opmerken dat het queryplan exact hetzelfde is. De interne query planner van PostgreSQL weet dus blijkbaar dat, in het geval van de bovenstaande gecorreleerde (`NOT`) `EXISTS` constructie, de query zo omgevormd kan worden dat subqueries overbodig zijn. Toch raden we jullie aan om bij de opbouw van een query subqueries zoveel mogelijk te vermijden.

3.2 Voorbeeld 2: langste registraties

In een tweede voorbeeld gaan we op zoek naar alle langste registratieformulieren op basis van het totaal aantal dagen dat registratie duurt. Een eerste mogelijke oplossing is de volgende.

```
SELECT * FROM registratieformulier
    WHERE (periode_end-periode_begin) >= ALL
        (SELECT (periode_end-periode_begin) FROM registratieformulier);
```

> Voer bovenstaande query uit door er `EXPLAIN ANALYZE` voor te plaatsen. Indien je binnen de minuut geen resultaat krijgt, stop dan de uitvoering.

Opnieuw zal deze query de correct oplossing geven, maar de kost voor de uitvoering zal zeer hoog liggen. Ook hier wordt er namelijk een dubbele lus uitgevoerd. Voor elk registratieformulier uit de buitenste query wordt de duur (`periode_end-periode_begin`) vergeleken met de duur van **alle** registratieformulieren.
Een mogelijke optimalisatie is de volgende.

```
SELECT * FROM registratieformulier
    WHERE (periode_end-periode_begin) =
        (SELECT MAX(periode_end-periode_begin) FROM registratieformulier);
```

> Voer bovenstaande query opnieuw uit door er `EXPLAIN ANALYZE` voor te plaatsen.

De kost voor de uitvoering van deze tweede query verlaagt zienderogen. De reden is dat we hier een aggregatie toepassen in de subquery (die typisch met lage kost gebeurt) en we dus elke registratie uit de buitenste query maar met 1 waarde moeten vergelijken.
Uit dit voorbeeld kunnen we besluiten dat, ook al is dit opnieuw zeker geen vuistregel die altijd geldt, het vaak beter is om te vergelijken met 1 waarde (door middel van een scalaire subquery), dan om operatoren zoals `ALL`, `ANY`,... te gebruiken.

4 Oefeningen

Hieronder vind je een aantal oefeningen terug. Jouw taak is om een query te schrijven met een zo laag mogelijk kost die natuurlijk ook het correcte antwoord terug-

geeft. Soms hebben we de vraag weggelaten en moet je dus zelf de voorbeeldquery interpreteren om de vraag te vinden. Kan jij ons verslaan?

4.1 What's in a name?

Geef de volledige naam van die personen waarvan de voornaam begint met de letter waar de achternaam mee eindigt (zonder rekening te houden met hoofdletters). In het resultaat verwachten we de kolom 'naam', die een concatenatie voorstelt van de voor- en achternaam van een persoon (gescheiden door een spatie).

Richtkost: 42.23

4.2 Populaire constructeurs

Geef per maand (in het formaat 'MM/YYYY') de constructeur die het meeste verhuurd is geweest. Een verhuur telt mee voor de maand waarin de registratie begint (op basis van periode_begin). Verwachte kolommen: 'maand', 'merk' en 'model'.

Richtkost: 2378.15

4.3 Opel Vivaro of Ford Escort?

```
SELECT DISTINCT p.email, p.voornaam, p.achternaam
FROM persoon p
    INNER JOIN registratieformulier r ON p.email = r.email
WHERE EXISTS
    (SELECT 1
    FROM registratieformulier r1
        INNER JOIN wagen w1 ON r1.nummerplaat = w1.nummerplaat
    WHERE r.email = r1.email
        AND w1.merk = 'Opel'
        AND w1.model = 'Vivaro')
OR EXISTS
    (SELECT 1
    FROM registratieformulier r2
        INNER JOIN wagen w2 ON r2.nummerplaat = w2.nummerplaat
    WHERE r.email = r2.email
        AND w2.merk = 'Ford'
        AND w2.model = 'Escort')
```

Richtkost: 404.01

4.4 Nummerplaatverzamelingen

```
SELECT w1.nummerplaat
FROM registratieformulier r1
    INNER JOIN wagen w1 ON r1.nummerplaat = w1.nummerplaat
WHERE to_char(periode_begin, 'YYYY')::integer = 2016
INTERSECT
(SELECT w2.nummerplaat
FROM registratieformulier r2
    INNER JOIN wagen w2 ON r2.nummerplaat = w2.nummerplaat
WHERE to_char(periode_begin, 'YYYY')::integer = 2017
EXCEPT
SELECT w3.nummerplaat
FROM registratieformulier r3
    INNER JOIN wagen w3 ON r3.nummerplaat = w3.nummerplaat
WHERE to_char(periode_begin, 'YYYY')::integer = 2018
)
```

Richtkost: 1747.90

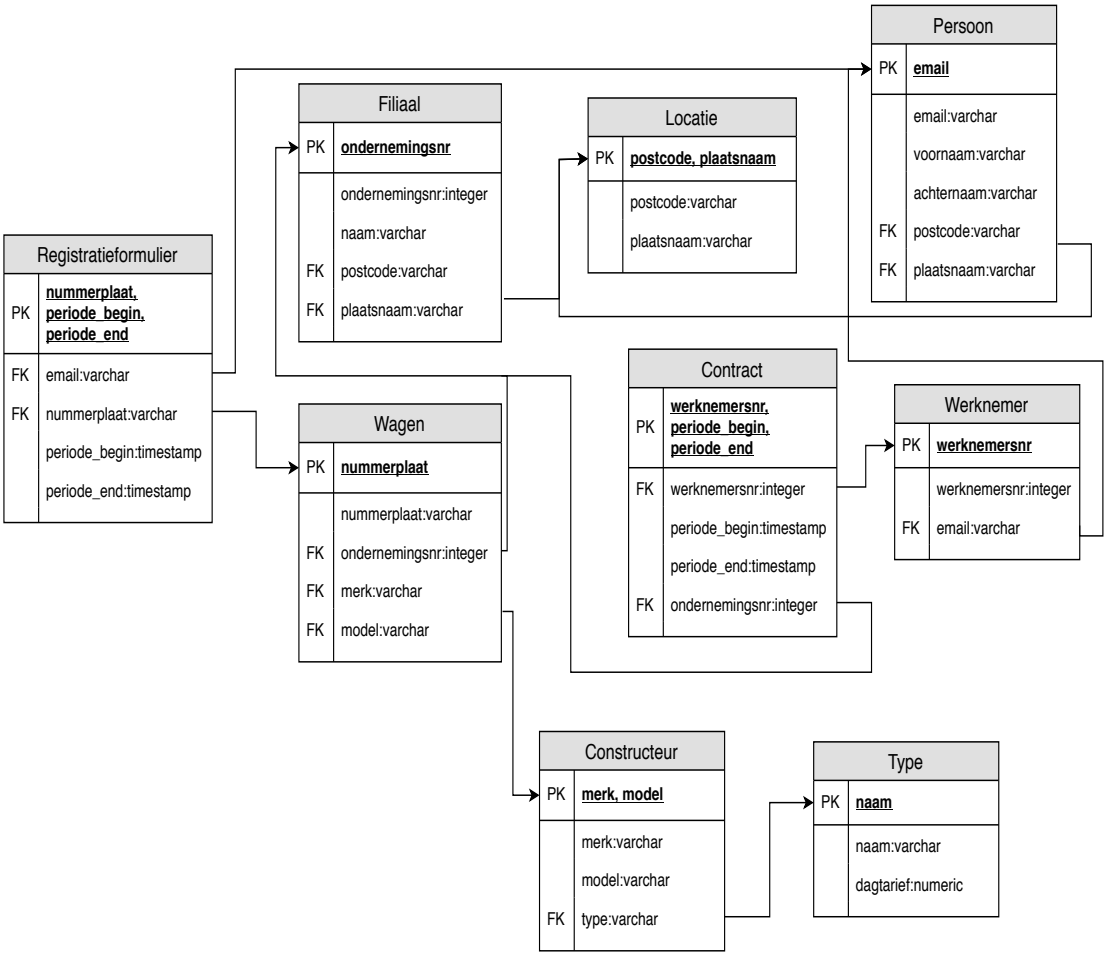
4.5 Nog een laatste oefening...

```
SELECT DISTINCT p.email, p.voornaam, p.achternaam
FROM werknemer w
    INNER JOIN persoon p ON w.email = p.email
    INNER JOIN contract c ON w.werknemersnr = c.werknemersnr
WHERE NOT EXISTS(
    SELECT 1 FROM registratieformulier r WHERE p.email = r.email
) AND (c.periode_end-c.periode_begin) >= ALL(
    SELECT c.periode_end-c.periode_begin
    FROM contract c
        INNER JOIN werknemer w ON c.werknemersnr = w.werknemersnr
    WHERE NOT EXISTS(
        SELECT 1 FROM registratieformulier r WHERE w.email = r.email
    )
)
```

Richtkost: 1433.84

Appendix: Logisch databankontwerp Rolls Robin

In onderstaande figuur vind je een schematische weergave van een mogelijk logisch ontwerp voor de Rolls Robin databank. Hierbij wordt iedere basisrelatie weergegeven door een rechthoek, die bovendien een oplijsting van alle attributen met bijhorend datatype bevat. Daarnaast worden de attributen die behoren tot de primaire sleutel bovenaan weergegeven, en worden vreemde sleutels voorgesteld door een pijl tussen de betreffende attribuutverzamelingen. Alle extra beperkingen die niet kunnen weergegeven worden in dit schema worden onderaan opgelijst.



Extra beperkingen

- Type:
  1. CHECK: dagtarief ≥ 0
- Registratieformulier:

    1. CHECK: periode_begin ≤ periode_end
    2. Insert: controleer bij toevoegen van registratieformulier dat periode niet overlapt met periode van ander registratieformulier voor dezelfde nummerplaat.

- Contract:
    1. CHECK: periode_begin ≤ periode_end
    2. Insert: controleer bij toevoegen van nieuw contract dat periode van contract niet overlapt met periode van ander contract voor hetzelfde werknemersnummer.