Universiteit Gent - TELIN - DDCM
2019-2020

Workshop: Query-optimalisatie

1 Introductie

In de voorbije lessen en workshops hebben we al heel wat geleerd over het ontwerp van een databank, het relationele databankmodel, het importeren van data naar een PostgreSQL databank en het aanpassen en verwijderen van data. Daarnaast hebben we ook reeds twee lessen data-analyse met behulp van SQL (Structured Query Language) achter de rug. Hierin hebben we de basisconcepten voor het bevragen van een databank (zoals `SELECT` en `WHERE`) bekeken en hebben we ook reeds gezien hoe we data uit verschillende tabellen met elkaar kunnen combineren door middel van een `JOIN`.
Waarschijnlijk heb je al gemerkt dat je voor een bepaalde vraagstelling meerdere correcte queries kan opstellen. Ook al geven deze queries allemaal de correcte oplossing terug, vaak hanteren ze onderliggend een andere berekeningswijze om tot die correcte oplossing te komen. Op kleine datasets (zoals die waar jullie tot nog toe op hebben gewerkt) heeft dit vaak heel weinig impact en repercussies. Wanneer je echter bevragingen wil uitvoeren op grotere volumes data, kan de exacte query die je lanceert een grote invloed hebben op de tijd die het databankbeheersysteem nodig heeft om de query uit te voeren. Om jullie een idee te geven, het is mogelijk dat de uitvoering van een ‘slechte’ query verschillende uren in beslag kan nemen, wat nefast is voor ondernemingen die grote hoeveelheden data verwerken. Je probeert dus best altijd een query te schrijven die zo snel mogelijk uitvoert en die geen onnodige berekeningen doet. In deze workshop gaan we daarom kijken naar mogelijke technieken om de uitvoeringstijd van een query te verlagen en onnodige berekeningen te vermijden. Deze technieken kunnen worden samengevat onder de term ‘query-optimalisatie’.

2 Eerste stappen

In de vorige workshops hebben we de `rollsrobin` databank fysiek geïmplementeerd en de aangemaakte tabellen vervolgens gevuld met kleine hoeveelheden data. Om de effecten van bepaalde query-optimalisaties beter te kunnen aantonen, gaan we in deze workshop echter werken met een grotere dataset. Opdat jullie deze nieuwe dataset niet opnieuw manueel zouden moeten gaan importeren in de verschillende tabellen, hebben we reeds een .sql-script voorzien dat een backup van de `rollsrobin`-databank bevat die gevuld is met grote hoeveelheden data.
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_large_dataset.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_extended` databank hebt aangemaakt op je lokale PostgreSQL-cluster.

3 Kostanalyse

Vooraleer we verschillende technieken introduceren om queries te optimaliseren, bekijken we even onderstaande queries.

```
SELECT * FROM registratieformulier
    INNER JOIN wagen USING(nummerplaat)
    INNER JOIN filiaal USING(ondernemingsnr);

SELECT * FROM filiaal
    INNER JOIN wagen USING(ondernemingsnr)
    INNER JOIN registratieformulier USING(nummerplaat);
```

Met jullie kennis SQL zou het duidelijk moeten zijn dat beide queries exact hetzelfde resultaat opleveren, namelijk een grote tabel die alle data uit de tabellen

registratieformulier, wagen en filiaal gekoppeld bevat. Het enige verschil tussen beide queries is de volgorde waarin de INNER JOIN operaties worden uitgevoerd. In het geval van de eerste query voeren we eerst een INNER JOIN uit tussen de tabel registratieformulier (met 612930 rijen) en de tabel wagen (met 4000 rijen). Als we voor elke registratie een match vinden in wagen (en dit is het geval) bekomen we een 'tussentabel' van 612930 rijen, waarop we vervolgens opnieuw een INNER JOIN uitvoeren met de tabel filiaal (met 30 rijen). In het geval van de tweede query voeren we eerst een INNER JOIN uit tussen de tabel filiaal (met 30 rijen) en de tabel wagen (met 4000 rijen). Hierdoor bekomen we een 'tussentabel' van 4000 rijen, waarop we vervolgens opnieuw een INNER JOIN uitvoeren met de tabel registratieformulier (met 612930 rijen). We zien dus dat in het geval van de tweede query het tussenresultaat een veel kleiner aantal rijen bevat. Met andere woorden, bij de tweede query moeten er minder rijen overlopen worden in de tussentabel om een match te vinden voor de join met de laatste tabel. Aangezien de complexiteit en dus de uitvoeringstijd van een INNER JOIN operatie toeneemt met het aantal rijen van de twee tabellen die gekoppeld worden, verwachten we dus dat de totale uitvoeringstijd van de tweede query kleiner zal zijn dan die van de eerste query.

Voer beide queries enkele keren uit. Analyseer het resultaat. Vergelijk de tijd die nodig is voor beide queries om het resultaat terug te geven. Merk je een groot verschil?

3.1 EXPLAIN

Als je beide queries uitvoert, zou je opgemerkt moeten hebben dat er in werkelijkheid amper een verschil in uitvoeringstijd tussen beide queries is. Zoals hierboven aangehaald, gaat dit in tegen onze verwachtingen. Om dit te onderzoeken kijken we eerst eens naar het EXPLAIN[1] keyword. Wanneer je dit keyword voor een query plaatst, zal je zien dat de uitvoering van een query niet resulteert in een resultatentabel. De output die je krijgt toont namelijk het uitvoeringsplan dat onderliggend gebruikt wordt door PostgreSQL voor de uitvoering van de query.

Voer beide queries opnieuw uit door EXPLAIN ervoor te plaatsen. Bestudeer kort het uitvoeringsplan. Merk je in dit geval een verschil?

[1] https://www.postgresql.org/docs/current/using-explain.html

In beide gevallen wordt er na uitvoering van de query onderstaand resultaat getoond.

```
QUERY PLAN
Hash Join  (cost=122.67..16336.85 rows=612930 width=164)
    Hash Cond: (wagen.ondernemingsnr = filiaal.ondernemingsnr)
        ->  Hash Join  (cost=121.00..14498.94 rows=612930 width=68)
                Hash Cond: ((registratieformulier.nummerplaat)::text =
                                (wagen.nummerplaat)::text)
                    ->  Seq Scan on registratieformulier
                            (cost=0.00..12767.30 rows=612930 width=50)
                    ->  Hash (cost=71.00..71.00 rows=4000 width=28)
                        ->  Seq Scan on wagen
                                (cost=0.00..71.00 rows=4000 width=28)
    ->  Hash  (cost=1.30..1.30 rows=30 width=100)
        ->  Seq Scan on filiaal  (cost=0.00..1.30 rows=30 width=100)
```

Dit is wat men noemt het queryplan van een query. Dit plan interpreteer je van binnen (rechts) naar buiten (links).
In het geval van bovenstaande query is het zo dat er eerst een Hash-functie wordt toegepast op iedere rij van de tabel `wagen` waarna de tabel `registratieformulier` sequentieel wordt gescand. Tijdens het scannen van `registratieformulier` wordt er voor elke rij een match gezocht in de tabel `wagen` met behulp van de gebruikte Hash-functie. Dit resulteert dus in een `Hash Join` tussen `registratieformulier` en `wagen` op basis van de gegeven `Hash Cond`itie. Voor dit practicum volstaat het om te weten dat een `Hash Join` een algoritme is om een `JOIN` uit te voeren. Daarna wordt er opnieuw een `Hash Join` uitgevoerd tussen de resulterende tabel en `filiaal`.
Je merkt dus dat, ondanks het feit dat we hierboven twee verschillende queries opstelden, beide queries op exact dezelfde manier worden uitgevoerd. PostgreSQL beslist achter de schermen om eerst `wagen` met `registratieformulier` te koppelen, en daarna pas met `filiaal`. De reden dat net dit uitvoeringsplan gekozen wordt is dat de **kost** van deze operatie (16336.85) vrij laag is. Deze kost is een arbitraire maatstaf die PostgreSQL intern berekent en aangeeft hoe 'complex' het is om een specifieke stap of specifiek queryplan uit te voeren.

3.2 Query planner

Naast alle uit te voeren operaties worden er in het uitvoeringsplan ook per operatie een lijst van metrieken gegeven. De waarden die deze metrieken aannemen zijn schattingen die worden berekend op basis van statistieken die PostgreSQL bijhoudt over alle tabellen. Deze statistieken worden bij iedere actie in de databank geüpdatet.

De metrieken die we voor de laatste `Hash Join` in het queryplan zien zijn de volgende.

- **cost=122.67..16336.85**: De geschatte kost voor de uitvoering van de operatie (uitgedrukt in een arbitraire eenheid).
- **rows=612930**: Het geschat aantal rijen dat deze operatie zal teruggeven.
- **width=104**: Het geschat aantal bytes waaruit elke rij, die de operatie zal teruggeven, bestaat.

Zoals je ziet wordt de kost voorgesteld door twee waarden. De eerste waarde is de 'start-up' kost en geeft de kost weer die nodig is om de eerste rij terug te geven wanneer de operatie wordt uitgevoerd. De tweede waarde is de totale kost die nodig is om om alle rijen terug te geven na uitvoering van de operatie. De precieze berekening van de kost is vrij complex en valt buiten de scope van deze cursus. Het is wel belangrijk om te onthouden dat zowel de **geheugenefficiëntie** als de **totale uitvoeringstijd** van de operaties een belangrijke rol spelen in deze berekening.
Het bepalen van de ideale uitvoering van een query wordt dus helemaal niet alleen beïnvloed door een databankgebruiker/beheerder. Ondertussen is duidelijk dat PostgreSQL ook zelf intern het uitvoeringsplan van een query optimaliseert, op basis van de berekening van kosten. Deze optimalisatie wordt uitgevoerd door de zogenaamde query planner/optimizer. De reden dat PostgreSQL een dergelijke query planner heeft ontwikkeld, toont aan dat voor heel wat applicaties de geheugenefficiëntie en/of de totale uitvoeringstijd van een query cruciaal zijn. Op basis van de opgegeven query en een grote verzameling gekende optimalisatie-technieken genereert de query planner verschillende uivoeringsplannen en kiest het plan met laagst geschatte kost. Wanneer je een bepaalde query opgeeft en uitvoert ben je dus helemaal niet zeker dat jouw query ook op exact deze manier wordt uitgevoerd, zoals je in bovenstaand voorbeeld zag. Het is belangrijk dat je je hiervan bewust bent. Veel optimalisaties die je op het eerste zicht zou kunnen doorvoeren aan je query hebben mogelijks helemaal geen effect op de kost omdat het databankbeheersysteem deze optimalisaties automatisch zelf zal doorvoeren.

3.3 EXPLAIN ANALYZE

Zoals aangegeven in 3.1 houdt de query planner dus enkel rekening met schattingen op basis van statistieken die je kan opvragen door middel van het `EXPLAIN` keyword. Aangezien het hier echter om schattingen gaat, vormen deze meestal echter maar een benaderend beeld van de geheugenefficiëntie en uitvoeringstijd van een query. Je kan echter ook een aantal exacte metrieken voor iedere stap in het uitvoeringsplan bekomen. Hiervoor dien je in plaats van het `EXPLAIN` commando het `EXPLAIN ANALYZE` commando te gebruiken. In tegenstelling tot `EXPLAIN` wordt de onderzochte query hierbij effectief uitgevoerd.

Bekijk het queryplan met bijhorende kost voor de hierboven vermelde queries met behulp van `EXPLAIN ANALYZE`.

Je ziet in het resultaat dat PostgreSQL hetzelfde queryplan teruggeeft als hiervoor. Naast iedere stap in het queryplan samen met de geschatte kost voor die stap, worden nu echter ook exacte waarden teruggegeven voor verschillende metrieken. Dit zijn de volgende.

- **actual time**: De tijd nodig voor de uitvoering van de operatie (uitgedrukt in ms). Er kan opnieuw een onderscheid worden gemaakt tussen 'start-up' tijd en totale tijd.
- **rows**: Het exact aantal rijen dat deze operatie zal teruggeven.
- **loops**: Het aantal keer dat een bepaalde operatie uitgevoerd werd.
- **Planning time**: De tijd die de query planner nodig had om het meest optimale uitvoeringsplan te selecteren.
- **Execution time**: De totale tijd die nodig was om de query uit te voeren.

Indien mogelijk is het altijd beter om `EXPLAIN ANALYZE` te gebruiken, aangezien je hiermee veel meer (correcte) informatie verkrijgt. Daarnaast update de uitvoering van dit commando ook steeds de statistieken die de planner gebruikt. Met andere woorden, uitvoering van `EXPLAIN ANALYZE` zal typisch ook de accuraatheid van de geschatte kosten verbeteren, indien deze nog niet accuraat genoeg waren. Enkel wanneer de query heel lang duurt of bijzonder zwaar is, kan je terugvallen op de schattingen die `EXPLAIN` voor jou maakt.

3.4 Query-optimalisatie revisited

We weten nu dat er een interne PostgreSQL query planner bestaat die reeds een groot deel van jouw werk doet. We willen er echter zeker van zijn dat deze query planner effectief optimaal werk verricht. Oorspronkelijk dachten we immers dat de tweede query uit het bovenstaande voorbeeld beter zou zijn dan de eerste query, maar volgens de planner is dit niet het geval. Daarom gaan we even zelf queryplanner spelen.
In PostgreSQL bestaat er een instelling waarmee je de volgorde van `JOIN`s kan forceren. Dit is `join_collapse_limit` met 8 als standaardwaarde. Hoe lager de waarde, hoe minder tijd de planner zal steken om de optimale volgorde van `JOIN`s te bepalen. Dit zullen we nu gebruiken om de queries te vergelijken.

Voer het commando `SET join_collapse_limit = 1` uit. Voer daarna opnieuw de bovenstaande queries uit voorafgegaan door `EXPLAIN ANALYZE`. Is er deze keer een verschil in uitvoeringstijd? En wat met de kost? Vind je ook de reden hiervoor? Vergeet niet om de instelling terug op zijn oorspronkelijke waarde te zetten vooraleer verder te gaan.

Als je bovenstaande opgave uitgevoerd hebt, zal je zien dat de kost van de tweede query veel hoger is dan de kost van de eerste query, ook al is de uitvoeringstijd lager. Dit lijkt in het begin misschien wat contra-intuïtief, maar er is effectief een logische verklaring voor: de geschatte kost voor het uitvoeren van een bepaald plan wordt, zoals reeds vermeld, niet alleen bepaald door de uitvoeringstijd, maar ook door de geheugenefficiëntie! Zonder te veel in detail te gaan mag je aannemen dat bij het uitvoeren van de tweede query het aantal nodige schijfoperaties veel hoger zal liggen dan bij het uitvoeren van de eerste query. Vandaar dat de query planner uiteindelijk toch de eerste query als meest kost-efficiënte zal beschouwen.
Een belangrijke conclusie die we hieruit kunnen trekken is dat het als eindgebruiker weinig zin heeft om aanpassingen te doen aan de volgorde waarin je `JOIN`s formuleert in een query. Dit geldt bovendien ook nog voor andere eenvoudige optimalisaties. Anderzijds betekent dit niet dat nadenken over het optimaliseren van queries in het algemeen niet noodzakelijk is. Integendeel, de query planner kan immers bepaalde (eenvoudig te detecteren) constructies (bijvoorbeeld de volgorde van joins) optimaliseren, maar zal vaak niet in staat zijn om alles te optimaliseren. De reden daarvoor is dat het efficiënter maken van een query soms heel subtiele ingrepen vergt. Het is dus heel belangrijk dat je nog steeds zelf nadenkt over je query en verschillende technieken toepast om deze minder complex te maken en niet-noodzakelijke berekeningen te vermijden! Deze technieken gaan we in het vervolg van deze workshop onderzoeken.

4 Query-optimalisatietechnieken

Naast de interne optimalisatie die uitgevoerd wordt door de query planner bestaan er ook extra technieken die je zelf kan gebruiken om een query te optimaliseren. Je zal merken dat er geen eenduidig wondermiddel bestaat dat voor iedere te optimaliseren query zal werken. Of en wanneer je een bepaalde techniek kan inzetten om een query effectief sneller te maken hangt heel hard af van de specifieke context en vereist enige ervaring. Onthoud dit dus goed als je later in deze workshop probeert te begrijpen waarom een bepaalde techniek voor één query werkt en voor een andere niet.

4.1 Indexen

Beschouw de volgende query:

```
SELECT * FROM registratieformulier
    WHERE email = 'xander.meller@hotmail.com';
```

> Verkrijg het queryplan en de kosten per stap voor deze query met behulp van de `EXPLAIN ANALYZE` functionaliteit. Onthoud de uitvoeringstijd en de kost van de volledige query.

PostgreSQL stelt voor om een parallelle sequentiële scan uit te voeren en voor iedere rij te controleren of het e-mailadres al dan niet gelijk is aan het opgegeven e-mailadres. De rijen waarvoor dit niet het geval is worden weggefilterd. Zoals je kan zien zal de sequentiële scan dus 3 loops in parallel doorlopen: 2 loops door worker processen en 1 door het leader proces. In elke loop worden 204292 rijen weggefilterd, zodat er in totaal 54 rijen overblijven en 612876 rijen weggefilterd worden. Ondanks dat het uiteindelijke resultaat dus slechts 54 rijen bevat, moet voor alle 612930 rijen in de tabel `registratieformulier` het e-mailadres worden gescand. Het is duidelijk dat dit efficiënter moet kunnen.
Een typische oplossing voor een dergelijk probleem is het introduceren van een **index**. Bij de aanmaak van een index wordt een nieuwe datastructuur (een opzoektabel) aangemaakt die het mogelijk maakt om voor de kolommen waarop een index is gecreëerd heel snel rijen met bepaalde waarden terug te vinden. Je kan het vergelijken met een opzoektabel (effectief ook index genaamd) zoals die typisch voorkomt achteraan in kookboeken: per ingrediënt worden de paginanummers opgelijst van de pagina's waarop recepten te vinden zijn waarin dat ingrediënt wordt gebruikt. Indien deze index niet beschikbaar is en je te weten zou willen komen in welke recepten er allemaal 'tomaten' gebruikt worden, moet je het hele boek doorbladeren om alle recepten te controleren (d.i. een sequentiële scan!). Dankzij de index kan je echter rechtstreeks naar de juiste recepten springen. Dit is exact wat een index in een databank ook doet. Per waarde (of groepering van waarden) van een bepaalde kolom (of combinatie van kolommen) wordt er een link aangemaakt naar de rijen die deze waarde als data hebben opgeslagen.
Een index kan in PostgreSQL met behulp van de volgende instructie worden aangemaakt.

```
CREATE INDEX indexnaam ON relatie (attr_1,...,attr_n);
```

Zoals reeds aangegeven, wordt de datastructuur die gebruikt wordt voor de index zodanig gestructureerd dat efficiënt kan worden gezocht op bepaalde informatie.

Om deze datastructuur op te bouwen wordt gebruik gemaakt van de eigenschappen van de attributen waar we snel op willen kunnen zoeken. In het geval dat je bijvoorbeeld snel wil kunnen zoeken op een `varchar`-attribuut, kan de datastructuur op zo'n manier zijn opgebouwd zodat de verwijzingen naar rijen alfabetisch gegroepeerd worden opgeslagen. Wanneer je vervolgens rijen met een specifieke waarde voor dat attribuut wil verkrijgen is geen sequentiële scan over alle rijen meer nodig. Er bestaan immers meer geavanceerde zoekalgoritmen op een gesorteerde datastructuur. Indices hoeven echter niet alleen op basis van sortering te zijn opgebouwd, maar bijvoorbeeld ook met behulp van een hashwaarde[2]...

Maak een index aan op het attribuut `email` van de tabel `registratieformulier`. Bekijk vervolgens opnieuw het queryplan van de hierboven vermelde query met behulp van `EXPLAIN ANALYZE`. Vergelijk de uitvoeringstijd en de kost van de query met de resultaten van hierboven.

Je merkt dat zowel de kost horende bij het uitvoeringsplan als de uitvoeringstijd van de query veel lager zijn dan voorheen. Zo daalt de uitvoeringstijd van een grootteorde van tientallen milliseconden naar minder dan een miliseconde. Je ziet ook dat het queryplan verschilt: in plaats van een sequentiële scan uit te voeren, wordt er nu een index scan uitgevoerd op `registratieformulier`. Er wordt dus gebruik gemaakt van de index op het attribuut `email` om de 54 rijen die we in ons resultaat verwachten zo snel mogelijk terug te vinden.
Een index verhoogt dus duidelijk de performantie van een opzoekquery enorm, waardoor het misschien een goed idee lijkt om voor alle mogelijke combinaties van attributen een index te gaan definiëren. Op die manier zou je kunnen anticiperen op alle mogelijke queries die ooit uitgevoerd kunnen worden op de databank. Dit is echter geen goed idee, aangezien het introduceren van een index ook nadelig kan zijn. Er zijn verschillende mogelijke nadelen verbonden aan een index.
Het meest voor de hand liggende nadeel is dat voor iedere index die je creëert, een nieuwe datastructuur wordt aangemaakt waarin een verwijzing naar iedere rij opgeslagen zit. Aangezien zo'n datastructuur zelf ook moet worden opgeslagen in het geheugen, kan je begrijpen dat dit voor grote tabellen een grote kost vormt (d.i. minder geheugenefficiënt!). Een tweede nadeel is dat het aantal mogelijke indices exponentieel stijgt met het aantal kolommen, aangezien je ook indices over meerdere attributen kan definiëren. Daarnaast kan je ook indices aanmaken op bepaalde kolomtransformaties, op specifieke waarden van een kolom,...Je ziet dus dat de opties om indices aan te maken eindeloos zijn. Alle mogelijke indices aanmaken is

[2] Meer info over de verschillende types indexen in PostgreSQL vind je op https://www.postgresql.org/docs/9.1/indexes-types.html

dus een heel geheugeninefficiënte manier van werken. Tot slot is het ook belangrijk om te vermelden dat een index redelijk wat onderhoudswerk vergt. Iedere keer dat je een nieuwe rij toevoegt (`INSERT`) die een attribuut met bijhorende index bevat, of een attribuutwaarde van een bestaande rij waarop een index is gedefinieerd wil aanpassen (`UPDATE`), dient de index ook aangepast te worden. Stel bijvoorbeeld dat de index die we net aangemaakt hebben voor de tabel `registratieformulier` de verwijzingen naar de rijen in de tabel alfabetisch opslaat op basis van het attribuut `email` en dat je het e-mailadres van een bepaalde rij wil aanpassen. Dan moet de verwijzing naar die rij natuurlijk worden verplaatst in de datastructuur, zodat na het uitvoeren van de `UPDATE` de datastructuur nog altijd in alfabetische volgorde is gesorteerd. Indices kunnen dus niet alleen bepaalde operaties versnellen, ze kunnen tegelijkertijd ook andere vertragen! Het is belangrijk om dit niet uit het oog te verliezen.
Wanneer is het dan een goed idee om een index op een bepaald attribuut (of meerdere attributen) te gebruiken? De belangrijkste criteria hiervoor zijn de volgende.

- Het attribuut wordt vaak gebruikt in een `WHERE`- of een `JOIN`-conditie. Door een index te definiëren op dit attribuut kan je een belangrijk deel van de queries die typisch op de databank worden uitgevoerd versnellen. Een voorbeeld zijn primaire sleutels: daar wordt inderdaad typisch heel frequent op gefilterd vanwege de uniciteit en ze komen ook heel frequent voor in `JOIN` operaties. Omwille van deze reden maakt PostgreSQL bij het aanmaken van een `UNIQUE`-beperking (die standaard ook op de primaire sleutel is gedefinieerd) automatisch een index op de combinatie van de kolommen die meespelen in deze beperking.

- Veel rijen hebben verschillende waarden voor een bepaald attribuut. Intuïtief is het logisch dat hoe meer verschillende waarden er zijn voor eenzelfde attribuut, hoe minder rijen er typisch in de resulterende tabel zullen zitten wanneer je met behulp van een `WHERE`-clausule filtert op één specifieke attribuutwaarde. Hoe minder rijen je verwacht in je uiteindelijke query-resultaat (en dus, hoe meer verschillende attribuutwaarden er voorkomen in de tabel voor het attribuut), hoe nuttiger het wordt om op een attribuut een index te bouwen. Neem opnieuw het voorbeeld van een index opgebouwd op basis van alfabetische volgorde. In het geval dat je de index definieert op een attribuut dat unieke waarden bevat (dus heel veel verschillende attribuutwaarden kan aannemen), zal de datastructuur telkens een attribuutwaarde bevatten samen met één specifieke verwijzing naar een rij (de rij die die attribuutwaarde aanneemt). Wanneer op één specifieke attribuutwaarde wordt gezocht met behulp van `WHERE`, zal dus enkel de geheugenverwijzing in de index moeten opgezocht worden die hoort bij die attribuutwaarde, en vervolgens het ene datablok waarin deze ene rij zich bevindt ingeladen moeten worden in het RAM-geheugen. Dit is

duidelijk veel efficiënter dan het scenario zonder index, waarbij alle geheugenblokken die nodig zijn om een tabel in te laden, ingeladen moeten worden. Stel nu echter dat je een index legt op een attribuut dat slechts twee verschillende attribuutwaarden kan aannemen (bv. ‘Ja’ en ‘Neen’), waarbij je er vanuit gaat dat bijvoorbeeld 50% van de rijen de waarde ‘Ja’ aanneemt en 50% ‘Neen’. In dit geval zal de index bestaan uit de attribuutwaarde ‘Ja’ met een bijhorende lijst van verwijzingen naar alle rijen die als attribuutwaarde ‘Ja’ hebben, en een attribuutwaarde ‘Neen’ die geassocieerd is met een lijst van verwijzingen naar alle rijen die als attribuutwaarde ‘Neen’ hebben. Indien je dan zoekt op een specifieke attribuutwaarde, zal je ondanks de aanmaak van een index erop, nog steeds alle geheugenblokken moeten inladen die nodig zijn om 50% van de rijen terug te kunnen krijgen. De efficiëntiewinst is in dit geval dus veel beperkter.

- Op kleine tabellen heeft het gebruik van een index heel weinig nut. Deze passen typisch in één enkel geheugenblok, waardoor je met behulp van een index het aantal geheugenblokken dat je dient in te laden in het RAM-geheugen helemaal niet meer kan optimaliseren (1 is het absolute minimum).

Tot slot willen we nog even vermelden dat je een index ook opnieuw kan verwijderen door middel van het volgende commando.

```
DROP INDEX indexnaam;
```

Verwijder de eerder aangemaakte index opnieuw.

4.2 Geen overbodige kolommen en joins

Een vrij logische techniek om een query sneller te maken is om enkel die kolommen op te vragen die je ook effectief nodig hebt. Zeker wanneer je veel tabellen met elkaar combineert met `JOIN`-operaties kan je eindigen met enorm veel kolommen. Herinner je ook de ‘width’ metriek in het queryplan die hier meer informatie over geeft.

Voer de twee onderstaande queries uit en analyseer hun queryplannen met behulp van `EXPLAIN ANALYZE`. Wat merk je op in verband met de vermelde ‘width’, de kost en de uitvoeringstijd voor beide queries?

```
SELECT * FROM registratieformulier
    INNER JOIN wagen USING(nummerplaat)
    INNER JOIN filiaal USING(ondernemingsnr);

SELECT ondernemingsnr, email FROM registratieformulier
    INNER JOIN wagen USING(nummerplaat)
    INNER JOIN filiaal USING(ondernemingsnr);
```

Je zal zien dat de kost van de query niet verandert, maar de resulterende tabel minder bytes per rij heeft en de uitvoeringstijd ook iets lager zal zijn. Wanneer het aantal bytes per rij in een tussentabel beperkt wordt zal dit echter wel een invloed hebben op de kost (zie voorbeeld Sectie 3).
Naast het beperken van het aantal opgevraagde kolommen kan je ook het aantal `JOIN`-operaties gaan reduceren. In de tweede query hierboven kunnen we bijvoorbeeld perfect zowel `ondernemingsnr` en `email` opvragen uit de tabellen `registratieformulier` en `wagen`. De tabel `filiaal` is hier dus eigenlijk overbodig.

Voer de twee onderstaande queries uit en analyseer hun queryplannen met behulp van `EXPLAIN ANALYZE`. Vergelijk de 'width', de kost en de uitvoeringstijd van beide queries.

```
SELECT ondernemingsnr, email FROM registratieformulier
    INNER JOIN wagen USING(nummerplaat)
    INNER JOIN filiaal USING(ondernemingsnr);

SELECT ondernemingsnr, email FROM registratieformulier
    INNER JOIN wagen USING(nummerplaat);
```

4.3 Functies in `WHERE` en `JOIN`

Zoals we eerder al zagen in 4.1 kan het gebruik van een index een query heel wat sneller maken. Dit is het geval wanneer het attribuut waarop de index is gedefinieerd voorkomt in een `WHERE` of `JOIN` conditie. Soms hebben we echter geen query nodig die filtert of een join uitvoert op basis van een attribuut, maar op een berekende of getransformeerde waarde van het attribuut. Een voorbeeld hiervan is onderstaande query. Deze query filtert de registratieformulieren die werden gedaan door een persoon met een e-mailadres van Telenet.

```
SELECT * FROM registratieformulier
    WHERE substring(email, position('@' in email)) = '@telenet.be';
```

Aangezien we de index hebben verwijderd op het attribuut `email`, voert PostgreSQL opnieuw een parallelle sequentiële scan uit. We zouden een index kunnen toevoegen op `email` om het gebruik van een sequentiële scan te vermijden.

> Maak opnieuw een index aan op het attribuut `email` van de tabel `registratieformulier` en analyseer het queryplan.

Je merkt echter dat ook na aanmaak van de index er nog steeds een parallelle sequentiële scan over alle rijen wordt uitgevoerd. De reden hiervoor is dat we een index hebben aangemaakt op het volledige attribuut `email`, maar niet op getransformeerde waarden ervan! In het voorbeeld filtert de query op basis van een substring van de e-mails, maar de index die we hierboven aanmaakten is helemaal niet opgebouwd op basis van deze substrings.
Het gebruik van functies kan dus vaak de uitvoeringstijd van een query langer maken zonder dat je er met een index op het basisattribuut iets aan kan veranderen. Typische voorbeelden van veel gebruikte functies zijn substrings, reguliere expressies, vergelijkingen van de vorm `LIKE '%substring'`, numerieke berekeningen,... Als een bepaalde berekening of transformatie van een attribuut echter veel gebruikt moet worden in bijvoorbeeld een `WHERE` of `JOIN` conditie, is het wel mogelijk om een index op de berekening of transformatie van dit attribuut aan te maken.

> Maak een index aan op de transformatie `substring(email, position('@' in email))` van het attribuut `email`.

Je merkt dat in dit geval de sequentiële scan over alle e-mails niet meer nodig is, maar de query optimizer opnieuw kiest voor een index scan. De query voert daardoor veel sneller uit. Het dient hierbij wel opgemerkt te worden dat indien je inderdaad heel frequent queries uitvoert op berekeningen of transformaties van attributen je er tijdens het databankontwerp misschien beter aan had gedaan om het afgeleide attribuut als aparte kolom op te slaan.

> Verwijder tot slot de aangemaakte indices opnieuw.

4.4 Gebruik van `DISTINCT` en `ORDER BY`

Zoals jullie reeds gezien hebben tijdens de eerste SQL-les kan je zorgen dat iedere rij in de resultatentabel uniek is door toevoeging van het `DISTINCT` keyword in de `SELECT` expressie.

> Onderzoek met behulp van `EXPLAIN ANALYZE` het queryplan dat PostgreSQL genereert voor de query waarmee je alle unieke e-mailadressen opvraagt die in de tabel `registratieformulier` voorkomen.

Je zou in het uitvoeringsplan moeten opmerken dat eerst een sequentiële scan wordt uitgevoerd over de tabel `registratieformulier`. Tijdens de `HashAggregate` stap die hierna volgt worden alle gelijke attribuutwaarden van het attribuut `email` gegroepeerd en wordt er per groep 1 waarde getoond in de resultatentabel. Zelfs al kan je de sequentiële scan vermijden door toevoeging van een index op het attribuut `email`, dan nog is het vergelijken van de verschillende waarden door middel van de `HashAggregate` stap een heel dure operatie. Indien het niet noodzakelijk is, is het dus beter om het gebruik van `DISTINCT` te vermijden. Dit kan je bijvoorbeeld doen door een combinatie van kolommen (zoals de primaire sleutel) te selecteren die zeker unieke rijen zullen teruggeven, zonder dat het gebruik van `DISTINCT` nog nodig is.

> Sorteer de bekomen resultaten van de vorige query in alfabetische volgorde op `email`. Vergelijk de kost en de uitvoeringstijden van beide queries.

Hetzelfde geldt dus ook voor `ORDER BY`. Het sorteren van rijen op basis van attribuutwaarden is de duurst mogelijke operatie op een databank, omwille van de rekenintensieve sorteeralgoritmen. Indien het niet noodzakelijk is, is het dus ook beter om het gebruik van `ORDER BY` te vermijden. In tegenstelling tot het vermijden van het `DISTINCT` keyword, is er voor het vermijden van `ORDER BY` geen alternatief.

4.5 Databankontwerp

Een laatste methode om queries te optimaliseren is om er tijdens het databankontwerp op in te spelen. Een voorbeeld hiervan gaven wij al aan in 4.3 om berekende of getransformeerde attributen als extra kolom toe te voegen. Ook kan je ervoor kiezen om twee tabellen ongescheiden te gaan opslaan (dus samen te voegen tot één tabel) om `JOIN`s te vermijden (dit heet denormalisatie). Let op, beide ingrepen

zullen zorgen voor de opslag van redundante informatie wat tot inconsistenties kan leiden. Doe dit dus enkel en alleen in uiterste nood, bijvoorbeeld wanneer een zware query enorm veel uitgevoerd moet worden.

5 Oefeningen

Om deze eerste workshop query-optimalisatie af te sluiten krijgen jullie hieronder opnieuw een aantal SQL-vragen. De bedoeling is om deze nu niet alleen correct op te lossen, maar om ze ook zo snel mogelijk te laten uitvoeren. Om jullie een idee te geven van hoe optimaal jouw oplossing is, hebben we bij iedere query een richttijd en een richtkost meegegeven. Kunnen jullie beter doen?

1. Geef de voornamen van de personen van wie de eerste twee letters van de voornaam een ‘a’, ‘e’ of ‘i’ bevatten, zonder rekening te houden met hoofdletters.

   Richttijd: 0.8 ms, **Richtkost:** 175.69

2. Geef de rijen terug uit de tabel `registratieformulier` waarvoor de som van de twee laatste cijfers van de nummerplaat gelijk is aan 3.

   Richttijd: 22ms, **Richtkost:** 706.68

Tot slot hebben wij van het management van Rolls Robin een query ontvangen als antwoord op onderstaande vraagstelling. Kunnen jullie als databankexpert het bedrijf helpen om deze query te optimaliseren. Let op, interpreteer de gegeven query eerst en controleer of deze een correct resultaat teruggeeft, want dit is niet gegarandeerd.

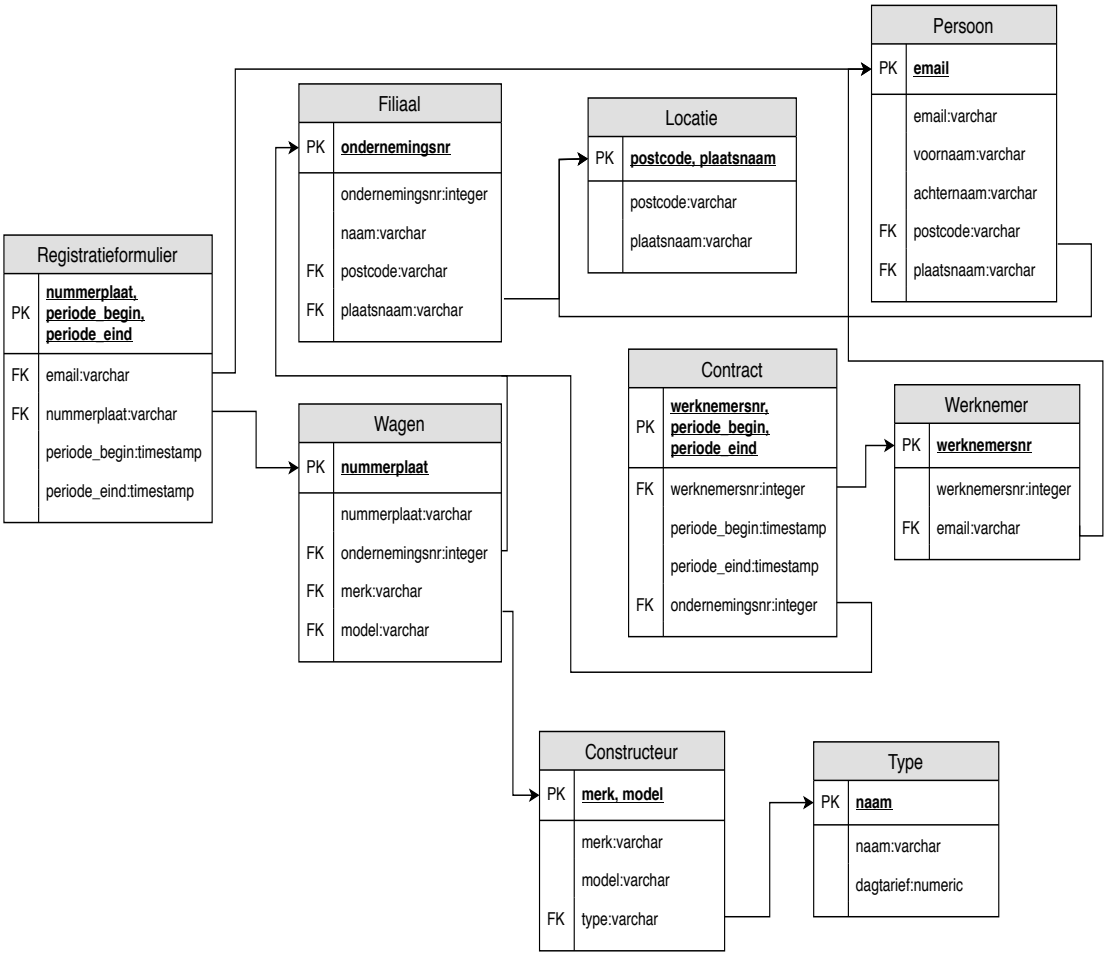
Geef een lijst van wagens die verhuurd zijn door een werknemer gedurende de periode dat deze werknemer werkt bij het bedrijf waar deze wagen gehuurd kan worden met voorwaarde dat het contract van de werknemer begon voor 2017. De verwachte kolommen zijn 'nummerplaat', 'werknemersnr', 'registratie_begin', 'registratie_einde', 'contract_begin', 'contract_einde' en 'check_date'. De kolom 'check_date' geeft aan of de verhuur begon/eindigde op dezelfde dag als waarop het contract van de werknemer begon/eindigde.

```
SELECT DISTINCT
    r.nummerplaat,
    c.werknemersnr,
    p.email,
    r.periode_begin    AS registratie_begin,
    r.periode_end      AS registratie_einde,
    c.periode_begin    AS contract_begin,
    c.periode_end      AS contract_einde,
    CASE
        WHEN c.periode_begin = r.periode_begin
            OR c.periode_end = r.periode_end THEN true
        ELSE false END AS check_date
FROM registratieformulier r
    INNER JOIN wagen w ON r.nummerplaat = w.nummerplaat
    INNER JOIN persoon p ON r.email = p.email
    INNER JOIN werknemer wn ON p.email = wn.email
    INNER JOIN contract c ON wn.werknemersnr = c.werknemersnr
WHERE
    c.ondernemingsnr = w.ondernemingsnr AND
    c.periode_begin <= r.periode_begin AND
    c.periode_end >= r.periode_end AND
    c.periode_begin < '2017-01-01'::timestamp AND
    r.periode_begin < '2017-01-01'::timestamp;
```

Richttijd: 80 ms, **Richtkost:** 11199.60

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_eind
    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_eind
    2. Insert: controleer bij toevoegen van nieuw contract dat periode van contract niet overlapt met periode van ander contract voor hetzelfde werknemersnummer.