Advanced Database Technologies

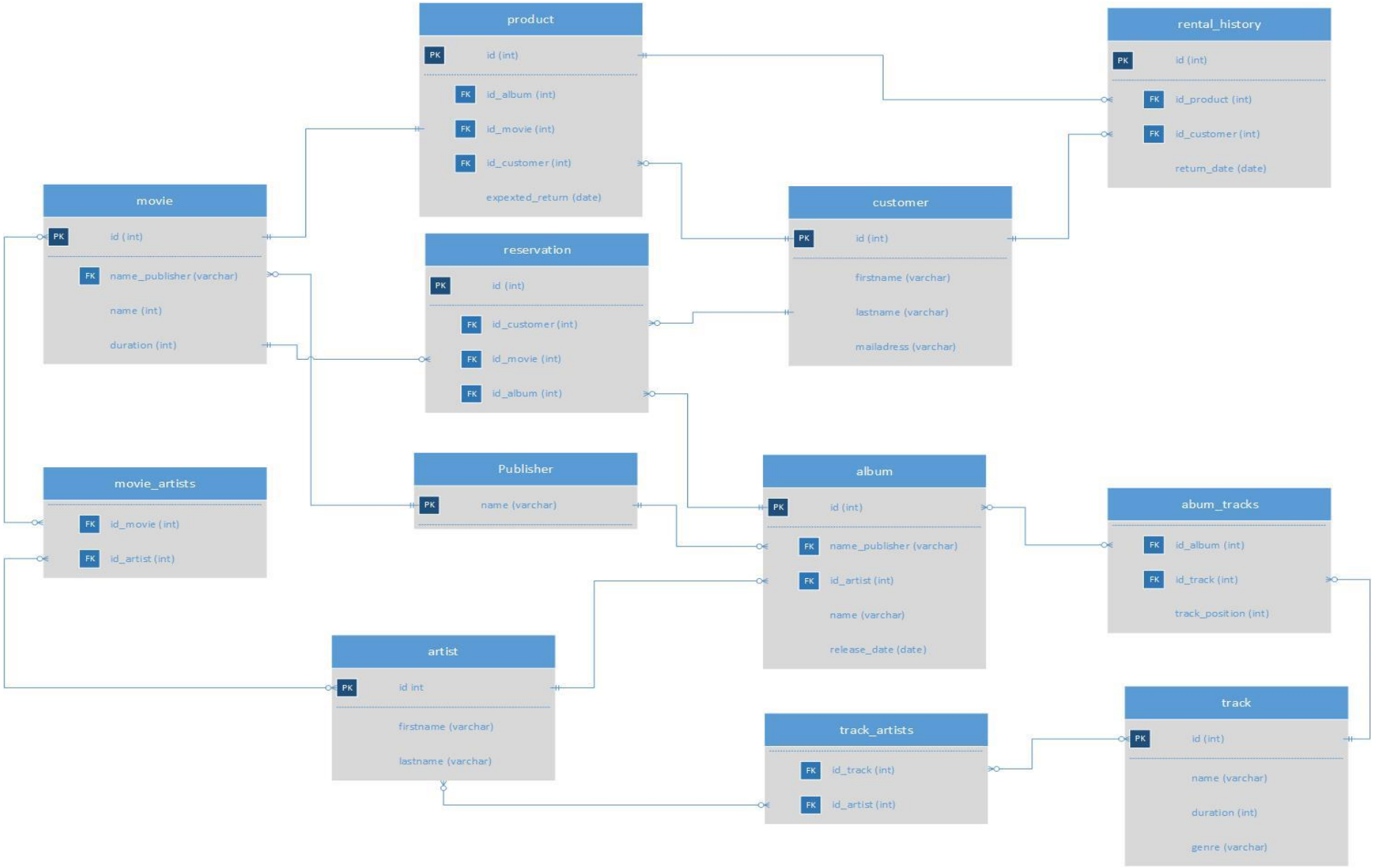
Relationele Databases

Tim Jongsma en Tom Schuitert

1 INLEIDING

Tijdens de eerste twee weken van dit vak zijn we bezig geweest met relationele databases. We hebben hiervoor postgressql gebruikt en hier een java applicatie voor geschreven. Het is van belang om vooraf te weten dat een van de doelen van deze opdracht was om zoveel mogelijk functionaliteit in de database zelf te implementeren. In dit document kunt u lezen hoe deze database is opgezet en welke keuzes er tijdens het ontwerp gemaakt zijn. Aan de java applicatie zelf zal minder aandacht worden besteed aangezien de nadruk van deze opdracht op database ontwerp lag.

2 DIAGRAM


product
PK id (int)
FK id_album (int)
FK id_movie (int)
FK id_customer (int)
expexted_return (date)
rental_history
PK id (int)
FK id_product (int)
FK id_customer (int)
return_date (date)
movie
PK id (int)
FK name_publisher (varchar)
name (int)
duration (int)
customer
PK id (int)
firstname (varchar)
lastname (varchar)
mailadress (varchar)
reservation
PK id (int)
FK id_customer (int)
FK id_movie (int)
FK id_album (int)
movie_artists
FK id_movie (int)
FK id_artist (int)
Publisher
PK name (varchar)
album
PK id (int)
FK name_publisher (varchar)
FK id_artist (int)
name (varchar)
release_date (date)
abum_tracks
FK id_album (int)
FK id_track (int)
track_position (int)
artist
PK id int
firstname (varchar)
lastname (varchar)
track_artists
FK id_track (int)
FK id_artist (int)
track
PK id (int)
name (varchar)
duration (int)
genre (varchar)

3 ONTWERPKEUZES

Tijdens het ontwerpen van de database hebben wij een aantal keuzes moeten maken deze zullen wij hieronder uitlichten. In de context van deze opdracht is er voor gekozen om zoveel mogelijk functionaliteit in de database zelf te implementeren.

3.1 VERHUURPROCES

Een van de kernfunctionaliteiten van het systeem is het uitlenen van albums en films tijdens het implementeren hiervan zijn enkele keuzes gemaakt. Hieronder wordt uitgelegd hoe dit proces nu werkt.

Albums en films worden in de database gerepresenteerd door de gelijk genaamde tabellen. Het kan echter zo zijn dat er van één film of album meerdere exemplaren bestaan vandaar dat de tabel ‘products’ is toegevoegd. Een rij hierin representeert het fysieke medium waar een film of album op staat. Vandaar ook dat er een relatie is tussen deze tabellen.

Wanneer een klant een album of film wil huren wordt er gekeken hoeveel producten er aan hun eisen voldoen en geen relatie met een andere klant hebben in de vorm van een ‘customer_id’. Is dit aantal groter dan nul dan is er nog een product beschikbaar en wordt er een relatie gelegd met de nieuwe klant door het customer_id aan te passen bij het product.

Zijn er geen producten meer beschikbaar dan kan een klant ook een reservering doen Deze worden weergegeven in de database door een rij aan te maken in de ‘reservations’ tabel. Om een geschiedenis bij te houden wordt wanneer een product wordt teruggebracht naast dat de relatie met de klant wordt verwijdert ook een rij aangemaakt in de tabel ‘rental_history’ met daarin het product-id, het klant-id en de datum waarop het product is teruggebracht.

3.2 FUNCTIES

Bepaalde functionaliteit is geïmplementeerd door functies in aan de database toe te voegen.
Hierdoor zijn de volgende functies ontstaan:

- add_album
- add_movie
- add_product
- add_track
- is_movie_available
- rent_album
- rent_movie

Voor de functies die gebruikt worden om bijvoorbeeld films en albums toe te voegen geldt dat er gekozen is voor een functie omdat er in meerdere tabellen rijen moeten worden toegevoegd.
Een alternatief zou zijn om de applicatie die de database gebruikt meerdere query's uit te laten voeren echter door dit als functies in de database te implementeren behoudt de database zelf de controle over zijn integriteit. Hierdoor kan een slecht geschreven applicatie minder invloed hebben op de betrouwbaarheid van de gegevens in de database.

Voor de functies die te maken hebben met het verhuur process geldt ook dat er vaak meerdere queries zouden moeten worden uitgevoerd om tot het gewenste resultaat te komen. Vandaar dat er ook hier voor is gekozen dit op te lossen door functies te schrijven die deze queries combineren. Waardoor de applicatie slechts één aanroep hoeft te doen richting de database.

3.3 TRIGGERS

Er zijn een tweetal triggers geimplementeerd namelijk:

- reservation()
- rental_history()

De eerste wordt gebruikt om op het moment dat er een nieuwe reververing aan de database wordt toegevoegd een aantal controles uit te voeren. Zo wordt er gecontroleerd of het om een reservering voor een album of een film gaat. Vervolgens wordt er gekeken of er nog exemplaren beschikbaar zijn. Door dit als trigger te implementeren kan de database garanderen dat deze controles onafhankelijk van de applicatie altijd worden uitgevoerd.

Voor de verhuur historie geldt dat deze trigger er voor zorgt dat wanneer een product wordt teruggebracht er automatisch een regel wordt toegevoegd aan de tabel die de verhuur historie bijhoudt. Zo hoeft de applicatie hier geen rekening mee te houden en hoeft deze functionaliteit maar op één plek geimplementeerd te worden.

3.4 INDEXES

Om te onderzoeken wat het effect van indexes is op zoek query's hebben we een drietal queries uitgevoerd zonder toegevoegde indexes. Vervolgens hebben we indexes toegevoegd en deze queries nogmaals uitgevoerd de resultaten vind u hieronder.

3.4.1 Find album by track

Query: *SELECT album.name FROM track INNER JOIN album_tracks ON track.id=album_tracks.id_track INNER JOIN album on album_tracks.id_album = album.id WHERE track.name = 'SDFS'*

3.4.1.1 Query plan zonder indexes:

```
"Nested Loop  (cost=1.07..39.52 rows=1 width=32)"
"  Join Filter: (album_tracks.id_track = track.id)"
"  ->  Seq Scan on track  (cost=0.00..1.13 rows=1 width=4)"
"        Filter: ((name)::text = 'SDFS'::text)"
"  ->  Hash Join  (cost=1.07..38.03 rows=29 width=36)"
"        Hash Cond: (album_tracks.id_album = album.id)"
"        ->  Seq Scan on album_tracks  (cost=0.00..29.40 rows=1940 width=8)"
"        ->  Hash  (cost=1.03..1.03 rows=3 width=36)"
"              ->  Seq Scan on album  (cost=0.00..1.03 rows=3 width=36)"
```

3.4.1.2 Query plan met indexes (album_tracks.id_track, album_tracks.id_album, album.name):

```
"Nested Loop  (cost=0.00..3.22 rows=1 width=32)"
"  Join Filter: (album_tracks.id_album = album.id)"
"  ->  Nested Loop  (cost=0.00..2.15 rows=1 width=4)"
"        Join Filter: (track.id = album_tracks.id_track)"
"        ->  Seq Scan on track  (cost=0.00..1.13 rows=1 width=4)"
"              Filter: ((name)::text = 'SDFS'::text)"
"        ->  Seq Scan on album_tracks  (cost=0.00..1.01 rows=1 width=8)"
"  ->  Seq Scan on album  (cost=0.00..1.03 rows=3 width=36)"
```

Kosten verschil: 1.07..39.52 tegen 0.00..3.22

3.4.2 Find movie by artist

Query: *SELECT movie.name FROM artist INNER JOIN movie_artists ON artist.id=movie_artists.id_artist INNER JOIN movie on movie_artists.id_movie = movie.id where artist.firstname = 'Tom' AND artist.lastname = 'Schuitert'*

3.4.2.1 Query plan zonder indexes:

```
"Nested Loop  (cost=1.02..41.83 rows=1 width=32)"
"  Join Filter: (movie_artists.id_artist = artist.id)"
"  ->  Seq Scan on artist  (cost=0.00..1.14 rows=1 width=4)"
"        Filter: (((firstname)::text = 'Tom'::text) AND ((lastname)::text = 'Schuitert'::text))"
"  ->  Hash Join  (cost=1.02..40.56 rows=11 width=36)"
"        Hash Cond: (movie_artists.id_movie = movie.id)"
"        ->  Seq Scan on movie_artists  (cost=0.00..31.40 rows=2140 width=8)"
"        ->  Hash  (cost=1.01..1.01 rows=1 width=36)"
"              ->  Seq Scan on movie  (cost=0.00..1.01 rows=1 width=36)"
```

3.4.2.2 Query plan met indexes (artist.firstname, artist.lastname, movie_artist.id_movie, movie_artist.id_artist/):

```
"Nested Loop  (cost=0.00..3.23 rows=1 width=32)"
"  Join Filter: (movie_artists.id_movie = movie.id)"
"  ->  Nested Loop  (cost=0.00..2.20 rows=1 width=4)"
"        Join Filter: (artist.id = movie_artists.id_artist)"
"        ->  Seq Scan on artist  (cost=0.00..1.14 rows=1 width=4)"
"              Filter: (((firstname)::text = 'Tom'::text) AND ((lastname)::text = 'Schuitert'::text))"
"        ->  Seq Scan on movie_artists  (cost=0.00..1.03 rows=3 width=8)"
"  ->  Seq Scan on movie  (cost=0.00..1.01 rows=1 width=36)"
```

Kosten verschil: 1.02..41.83 tegen 0.00..3.23

3.4.3 Find rental history by album:

Query: *SELECT customer.firstname, customer.lastname, rental_history.return_date FROM rental_history INNER JOIN product ON rental_history.id_product = product.id INNER JOIN album ON id_album = album.id INNER JOIN customer ON rental_history.id_customer = customer.id WHERE album.name = 'HardMetal'*

3.4.3.1 Query plan zonder indexes:

```
"Nested Loop  (cost=2.16..4.34 rows=1 width=68)"
"  Join Filter: (rental_history.id_customer = customer.id)"
"  ->  Hash Join  (cost=2.16..3.27 rows=1 width=8)"
"        Hash Cond: (rental_history.id_product = product.id)"
"        ->  Seq Scan on rental_history  (cost=0.00..1.07 rows=7 width=12)"
"        ->  Hash  (cost=2.15..2.15 rows=1 width=4)"
"              ->  Nested Loop  (cost=0.00..2.15 rows=1 width=4)"
"                    Join Filter: (product.id_album = album.id)"
"                    ->  Seq Scan on album  (cost=0.00..1.04 rows=1 width=4)"
"                          Filter: ((name)::text = 'HardMetal'::text)"
"                    ->  Seq Scan on product  (cost=0.00..1.05 rows=5 width=8)"
"  ->  Seq Scan on customer  (cost=0.00..1.03 rows=3 width=68)"
```

3.4.3.2 Query plan met indexes (album.name, rental_history.id_customer, rental_history.id_product):

```
"Nested Loop  (cost=2.16..4.34 rows=1 width=68)"
"  Join Filter: (rental_history.id_customer = customer.id)"
"  ->  Hash Join  (cost=2.16..3.27 rows=1 width=8)"
"        Hash Cond: (rental_history.id_product = product.id)"
"        ->  Seq Scan on rental_history  (cost=0.00..1.07 rows=7 width=12)"
"        ->  Hash  (cost=2.15..2.15 rows=1 width=4)"
"              ->  Nested Loop  (cost=0.00..2.15 rows=1 width=4)"
"                    Join Filter: (product.id_album = album.id)"
"                    ->  Seq Scan on album  (cost=0.00..1.04 rows=1 width=4)"
"                          Filter: ((name)::text = 'HardMetal'::text)"
"                    ->  Seq Scan on product  (cost=0.00..1.05 rows=5 width=8)"
"  ->  Seq Scan on customer  (cost=0.00..1.03 rows=3 width=68)"
```

Kosten verschil: 2.16..4.34 tegen 2.16..4.34

3.5 CONCLUSIE:

Bij de eerste twee queries zien we zoals verwacht duidelijke verbeteringen. Bij de laatste query ontbreken deze. Hoe dit komt is op dit moment nog onduidelijk.