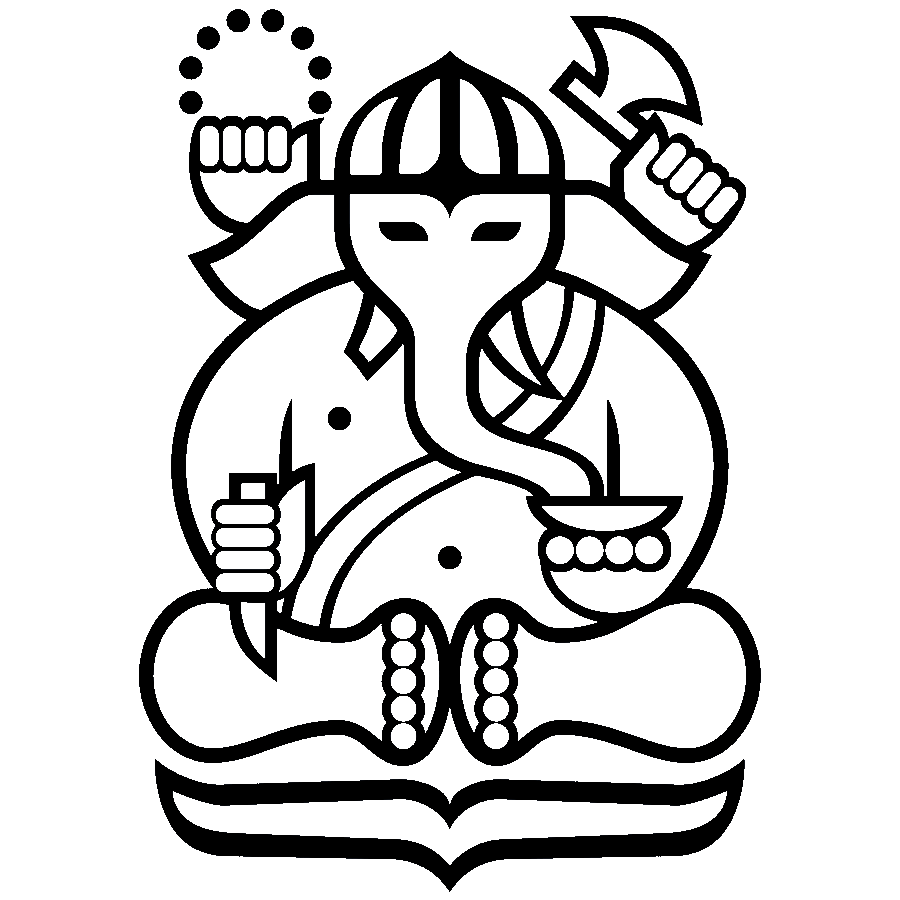
**Laporan Tugas Besar I**

**IF3140 Manajemen Basis Data**

****

**Disusun oleh:**

**Kelompok K01-G09**

10022191 Muhammad Meiditya Putra

13520034 Bryan Bernigen

13520133 Jevant Jedidia Augustine

13520157 Thirafi Najwan Kurniatama

**Program Studi Teknik Informatika**

**Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung**

**Jl. Ganesha 10, Bandung 40132**

**2022**

# Bab I Dasar Teori

Jalannya sistem basis data tidak lepas dari pemasukan dan pemrosesan *query*. Terdapat 3 langkah umum yang dilakukan oleh sistem basis data dalam melakukan pemrosesan query: *parsing* dan *translating,* optimisasi, dan evaluasi. *Parsing* dan *translating* adalah tahap dimana sistem mentranslasi query menjadi aljabar relasional, optimisasi adalah tahap dimana sistem mencari rencana evaluasi dari aljabar relasional yang paling murah, dan evaluasi adalah tahap dimana sistem menjalankan rencana evaluasi paling murah yang didapat dari tahap optimisasi untuk mendapatkan hasil dari query masukan.

Pada proses optimisasi, aljabar relasional yang didapat dari tahap sebelumnya akan dilihat dan diperiksa bagaimana caranya untuk mendapatkan hasil yang diinginkan dengan harga yang paling murah. Operasi *join, sorting,* dan *selection* memiliki harga yang berbeda-beda berdasarkan teknik yang digunakan. Setiap teknik memiliki kelebihan dan kekurangannya tersendiri, maka dari itu untuk mendapatkan rencana evaluasi yang paling murah, perlu dipilih teknik yang sesuai dengan kasus yang sedang dihadapi.

Selain memilih teknik yang sesuai, optimisasi dari query juga dapat dilakukan dengan mengubah urutan dari aljabar relasional. Dengan melakukan beberapa perubahan urutan operasi pada aljabar relasional, maka dimungkinkan untuk mendapatkan query plan dengan harga yang murah. Akan tetapi perlu diperhatikan bahwa perubahan urutan operasi pada aljabar relasional harus mengikuti aturan yang ada agar hasil query yang dihasilkan sesuai dengan query masukan.

Untuk mempercepat laju aktivitas dari basis data, maka dapat dilakukan sebuah teknik yang bernama *database* *tuning*. Tuning itu sendiri dapat dibagi menjadi 2 macam, *schema tuning* dan *index tuning*. *Schema* *tuning* merupakan jenis *tuning* yang dilakukan dengan cara mengubah skema dari basis data yang ada. Terdapat 3 cara melakukan *schema tuning*: *splitting table*, denormalisasi, dan materialized view. *Index tuning* merupakan jenis *tuning* yang dilakukan dengan cara mengindeks tabel yang ada di basis data. *Index tuning* memiliki beberapa jenis: clustering indices, non-clustering indices, B+ tree, hash index, dan bitmap index. Perlu diperhatikan bahwa kedua jenis tuning tersebut hanya akan efektif dilakukan pada kasus-kasus tertentu saja, maka dari itu perlu diketahui terlebih dahulu kasus-kasus yang terdapat pada sebuah basis data sebelum dapat mulai melakukan tuning.

Untuk menjaga integritas dari sebuah data yang ada di basis data, digunakan sebuah teknik yang dinamakan *trigger*. *Trigger* itu sendiri merupakan perintah dan pernyataan SQL yang disimpan di basis data dan akan otomatis dieksekusi saat terjadi masukan perintah *insert, update,* dan *delete*. Pada saat perintah *insert, update,* atau *delete* dilaksanakan, dengan adanya *trigger*, maka semua relasi yang berhubungan dengan relasi yang secara langsung diubah oleh ketiga perintah tersebut akan diubah untuk menjaga integritas data.

# Bab II Isi Laporan

## Skema Basis Data

Berikut merupakan query untuk memasukkan data (sekaligus membuat tabel):

| psql dvdrental < dvdrental.sql |
| --- |

(pastikan sudah membuat database dvdrental dengan query “CREATE DATABASE dvdrental” terlebih dahulu di postgresql)

Berikut merupakan query untuk membuat skema seperti pada spesifikasi:

| CREATE TABLE category (  category\_id SMALLINT,  name VARCHAR(25) NOT NULL,  last\_update DATE NOT NULL,  PRIMARY KEY(category\_id)  ); |
| --- |
| CREATE TABLE language(  language\_id SMALLINT,  name VARCHAR(25) NOT NULL,  last\_update DATE NOT NULL,  PRIMARY KEY(language\_id)  ); |
| CREATE TABLE actor(  actor\_id INTEGER,  first\_name VARCHAR(25) NOT NULL,  last\_name VARCHAR(25),  last\_update DATE,  PRIMARY KEY (actor\_id)  ); |
| CREATE TABLE country(  country\_id SMALL INT,  country VARCHAR(255),  last\_update DATE,  PRIMARY KEY(country\_id)  ); |
| CREATE TABLE city(  city\_id INTEGER,  city VARCHAR(25) NOT NULL,  country\_id SMALLINT NOT NULL,  last\_update DATE NOT NULL,  PRIMARY KEY (city\_id),  FOREIGN KEY country\_id  REFERENCES country(country\_id)  ON DELETE CASCADE  ON UPDATE CASCADE  ); |
| CREATE TABLE address(  address\_id INTEGER,  address VARCHAR(255) NOT NULL,  address2 VARCHAR(255),  district VARCHAR (25) NOT NULL,  city\_id INTEGER NOT NULL,  postal\_code SMALLINT NOT NULL,  phone VARCHAR(15) NOT NULL,  last\_update DATE NOT NULL,  PRIMARY KEY(address\_id),  FOREIGN KEY(city\_id)  REFERENCES city(city\_id)  ON DELETE CASCADE  ON UPDATE CASCADE  ); |
| CREATE TABLE store(  store\_id SMALL INT,  manager\_staff\_id INTEGER NOT NULL,  address\_id INTEGER NOT NULL,  last\_update DATE NOT NULL,  PRIMARY KEY(store\_id),  FOREIGN KEY(address\_id)  REFERENCES address(address\_id)  ON UPDATE SET NULL,  ON DELETE SET NULL  ); |
| CREATE TABLE staff(  staff\_id MEDIUM INT,  first\_name VARCHAR(25) NOT NULL,  last\_name VARCHAR(25),  address\_id INTEGER NOT NULL,  email VARCHAR(52) NOT NULL,  store\_id SMALL INT NOT NULL,  active BOOLEAN NOT NULL,  username VARCHAR(25) NOT NULL,  password VARCHAR(256) NOT NULL,  last\_update DATE NOT NULL,  picture BYTEA,  PRIMARY KEY(staff\_id),  FOREIGN KEY(address\_id)  REFERENCES address(address\_id)  ON UPDATE SET NULL  ON DELETE SET NULL,  FOREIGN KEY(store\_id)  REFERENCES store(store\_id)  ON UPDATE SET NULL  ON DELETE SET NULL  ); |
| ALTER TABLE store FOREIGN KEY(manager\_staff\_id)  REFERENCES staff(staff\_id)  ON UPDATE SET NULL  ON DELETE SET NULL; |
| CREATE TABLE film(  film\_id INTEGER,  title VARCHAR(255) NOT NULL,  description TEXT,  release\_year DATE NOT NULL,  language\_id SMALLINT NOT NULL,  rental\_duration SMALLINT NOT NULL,  rental\_rate SMALLINT NOT NULL,  length SMALLINT,  replacement\_cost SMALLINT NOT NULL,  rating NUMERIC(3,2),  last\_update DATE NOT NULL,  special\_features TEXT,  fulltext TEXT,  PRIMARY KEY (film\_id),  FOREIGN KEY(language\_id)  REFERENCES language(language\_id)  ); |
| CREATE TABLE film\_category(  film\_id INT,  category\_id SMALLINT,  last\_update DATE NOT NULL,  PRIMARY KEY(film\_id,category\_id),  FOREIGN KEY(film\_id)  REFERENCES film(film\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY(category\_id)  REFERENCES category(category\_id)  ON DELETE SET NULL  ON UPDATE SET NULL  ); |
| CREATE TABLE film\_actor(  actor\_id INTEGER,  film\_id INTEGER,  last\_update DATE,  PRIMARY KEY(actor\_id, film\_id),  FOREIGN KEY(actor\_id)  REFERENCES actor(actor\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY(film\_id)  REFERENCES film(film\_id)  ON DELETE SET NULL  ON UPDATE SET NULL  ); |
| CREATE TABLE inventory(  inventory\_id INTEGER,  film\_id INTEGER,  store\_id INTEGER,  last\_update DATE NOT NULL,  PRIMARY KEY(inventory\_id),  FOREIGN KEY (film\_id)  REFERENCES film(film\_id)  ON DELETE CASCADE  ON UPDATE CASCADE,  FOREIGN KEY (store\_id)  REFERENCES store(store\_id)  ON DELETE CASCADE  ON UPDATE CASCADE  ); |
| CREATE TABLE customer(  Customer\_id INTEGER,  store\_id INTEGER,  First\_name VARCHAR(25) NOT NULL,  Last\_name VARCHAR(25),  Email VARCHAR(50) NOT NULL,  Address\_id INTEGERm  Activebool BOOLEAN NOT NULL,  Create\_date DATE NOT NULL,  Last\_update DATE NOT NULL,  Active boolean NOT NULL,  PRIMARY KEY (customer\_id),  FOREIGN KEY (store\_id)  REFERENCES store(store\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY (address\_id)  REFERENCES address(address\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  ); |
| CREATE TABLE rental(  rental\_id INTEGER,  rental\_date DATE NOT NULL,  inventory\_id INTEGER,  customer\_id INTEGER,  return\_date DATE,  staff\_id INTEGER,  last\_update DATE NOT NULL,  PRIMARY KEY (rental\_id),  FOREIGN KEY (inventory\_id)  REFERENCES inventory(inventory\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY (staff\_id)  REFERENCES staff(staff\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY (customer\_id)  REFERENCES customer(customer\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  ); |
| CREATE TABLE payment(  Payment\_id INTEGER,  Customer\_id INTEGER,  Staff\_id INTEGER,  Rental\_id INTEGER,  Amount INTEGER,  Payment\_date DATE NOT NULL,  PRIMARY KEY (payment\_id),  FOREIGN KEY (customer\_id)  REFERENCES customer(customer\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY (staff\_id)  REFERENCES staff(staff\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  FOREIGN KEY (rental\_id)  REFERENCES rental(rental\_id)  ON DELETE SET NULL  ON UPDATE SET NULL,  ); |

## 

## Query Processing

Legenda Simbol

| No | Simbol | Arti | Statement SQL yang bersesuaian |
| --- | --- | --- | --- |
| 1 |  | projection | SELECT |
| 2 | 𝛔 | selection | WHERE |
| 3 | 𝒢 | aggregation | SUM, COUNT, AVERAGE |
| 4 | ⋈ | join | NATURAL JOIN |

| Query Tipe 1 |
| --- |
| Query Plan 1 |

Detail Query Tipe 1

| no | Query | Computation Cost ( | Actual Time Cost (ms) | Number of Tuple | Byte per Tuple |
| --- | --- | --- | --- | --- | --- |
| 1 | σ(film\_category) sequential scan | 0-16 = 16 | 0.013-0.244 = 0.231 | 1000 | 4 |
| 2 | Hash film\_id | 16.0-16.0 = 0 | 0.527-0.528 = 0.001 | 1000 | 4 |
| 3 | σ(actor) sequential scan | 0-4 = 4 | 0.014-0.059 = 0.045 | 200 | 10 |
| 4 | Hash actor\_id | 4-4 = 0 | 0.139-0.140 = 0.001 | 200 | 10 |
| 5 | σ(film\_actor)sequential scan | 0-84.62 = 84.62 | 0.036-0.741 = 0.705 | 5462 | 4 |
| 6 | ⋈ hash join on actor\_id | 6.50-105.76 = 99.26 | 0.180-3.699 = 3.519 | 5462 | 12 |
| 7 | ⋈ hash join on film\_id | 35-209.36 = 174.36 | 0.719-6.579 = 5.86 | 5462 | 12 |
| 8 | quick Sort on actor\_id | 548.42-562.08 = 13.66 | 8.365-8.881 = 0.516 | 5462 | 12 |
| 9 | 𝒢group by actor.actor\_id | 548.42-591.39 = 42.97 | 8.390-11.506 = 3.116 | 200 | 18 |
| Total | | 591.39 satuan | 11.817 ms | 200 | 18 |

| Query Tipe 2 |
| --- |
| Query Plan 2 |

Detail Query Tipe 2

| no | Query | Computation Cost | Actual Time Cost (ms) | Number of Tuple | Byte per Tuple |
| --- | --- | --- | --- | --- | --- |
| 1 | σ(film\_category) sequential scan | 0-16 = 16 | 0.013-0.244 = 0.231 | 1000 | 4 |
| 2 | Hash film\_id | 16-16 = 0 | 0.547-0.548 = 0.001 | 1000 | 4 |
| 3 | σ(film\_actor) sequential scan | 0-84.62 = 84.62 | 0.018-0.750 = 0.732 | 5462 | 4 |
| 4 | ⋈ hash join on film\_id | 28.50-188.22 = 159.72 | 0.583-4.095 = 3.512 | 5462 | 4 |
| 5 | quick Sort on actor\_id | 527.28-540.94 = 13.66 | 5.653-6.179 = 0.526 | 5462 | 4 |
| 6 | 𝒢group by actor.actor\_id | 527.28-540.94 = 13.66 | 5.653-6.179 = 0.526 | 5462 | 4 |
| 7 | σ(actor) using index | 0.14-16.16 = 16.02 | 0.013-0.104 = 0.091 | 200 | 10 |
| 8 | ⋈ merge join on actor\_id | 527.43-591.41 = 63.98 | 5.699-8.981 = 3.282 | 200 | 18 |
| Total | | 591.41 | 9.308 | 200 | 18 |

| Query Tipe 3 |
| --- |
| Query Plan |

Detail Query Tipe 1

| no | Query | Computation Cost ( | Actual Time Cost (ms) | Number of Tuple | Byte per Tuple |
| --- | --- | --- | --- | --- | --- |
| 1 | σ(film\_category) sequential scan | 0-16 = 16 | 0.013-0.245 = 0.232 | 1000 | 4 |
| 2 | Hash film\_id | 16.0-16.0 = 0 | 0.527-0.528 = 0.001 | 1000 | 4 |
| 3 | σ(actor) sequential scan | 0-4 = 4 | 0.013-0.057 = 0.044 | 200 | 10 |
| 4 | Hash actor\_id | 4-4 = 0 | 0.131-0.132 = 0.001 | 200 | 10 |
| 5 | σ(film\_actor)sequential scan | 0-84.62 = 84.62 | 0.018-0.749 = 0.731 | 5462 | 4 |
| 6 | ⋈ hash join on actor\_id | 6.50-105.76 = 99.26 | 0.165-3.636 = 3.471 | 5462 | 12 |
| 7 | ⋈ hash join on film\_id | 35-209.36 = 174.36 | 0.705-6.679 = 5.952 | 5462 | 12 |
| 8 | quick Sort on actor\_id | 548.42-562.08 = 13.66 | 8.367-8.870 = 0.503 | 5462 | 12 |
| 9 | 𝒢group by actor.actor\_id | 548.42-591.39 = 42.97 | 8.394-11.562 = 3.168 | 200 | 18 |
| Total | | 591.39 satuan | 11.841 ms | 200 | 18 |

**Analisis Query:**

1. Query Tipe 1 memiliki query execution plan yang sama dengan Query Tipe 3 sehingga keduanya memiliki hasil yang mirip (hanya berbeda di actual time saja).
2. Query Tipe 2 dan Query Tipe 1 (termasuk 3) memiliki pipelining dan materialisasi yang cukup mirip yakni ada beberapa data yang di pipeline dan ada juga beberapa data yang di materialisasi.
3. Secara computational cost, Query Tipe 2 memiliki cost yang sedikit lebih berat dibanding dengan Query Tipe 1 (dan 3), yakni 591.41 satuan untuk Query 2 dan 591.39 satuan untuk Queary Tipe 1 (dan 3). Namun secara actual time, Query 2 memiliki waktu yang jauh lebih cepat, yakni sekitar 20% lebih cepat dibanding dengan Query tipe 1 (dan 3).
4. Query 2 lebih cepat secara realtime dari Query 1 (dan 3) karena Query 2 melakukan penggabungan data besar di akhir sedangkan Query 1 (dan 3) melakukannya di awal. Penggabungan relasi aktor dengan relasi lainnya membuat ukuran data membengkak dari 4 byte per tuple menjadi 18 byte per tuple. Hal tersebut membuat jumlah tuple per block memori menjadi lebih kecil sehingga jumlah transfer menjadi lebih banyak. Dengan melakukan join tersebut di akhir, Query 2 dapat melakukan operasi pada bagian awal dengan lebih optimal karena jumlah transfer block lebih sedikit sehingga mempercepat waktu komputasi.

**Alternatif Query Yang Lebih Cepat**

Menurut kami tidak ada query yang lebih baik dibanding Query 2 karena Query tersebut sudah mengikuti ketiga heuristic untuk query optimisation yakni:

1. pushing projection

Tidak terlalu perlu dilakukan karena hampir seluruh data dari sebuah relasi dibutuhkan sehingga percuma melakukan projection early karena ukuran data tidak akan menyusut banyak. Jika dipaksakan untuk dilakukan, maka komputasi akan menjadi berat karena akan ada step tambahan yang tidak terlalu berguna untuk komputasi setelahnya.

1. pushing selection

Selection selalu dilakukan early yakni selalu di daun. Dengan demikian, maka tuple yang digunakan di operasi setelahnya hanyalah tuple-tuple yang relevan. Oleh karena itu, komputasi diatasnya menjadi lebih effisien.

1. perform most restrictive selection and join operation

Terdapat 3 relasi yang akan di join yakni film\_category, film\_actor, dan actor.

1. Film\_category ⋈ actor → tidak bisa karena tidak ada kolom yang sama
2. Film\_category ⋈ film\_actor → menghasilkan 5462 tuple dengan masing-masing tuple sebesar 4 byte sehingga ukuran hasil adalah 21848 byte.
3. Actor ⋈ film\_actor → menghasilkan 5462 tuple dengan masing-masing tuple sebesar 12 byte sehingga ukuran hasil adalah 65544 byte.

Query 2 melakukan film\_category ⋈ film\_actor yang ukuran hasilnya lebih kecil terlebih dahulu baru dilakukan join dengan aktor yang hasilnya lebih besar. Oleh karena itu, query tersebut sudah mengikuti heuristik tersebut.

## Database Design Tuning

Query awal

| SELECT full\_name, jumlah\_peminjaman + jumlah\_pembayaran poin,  CASE  WHEN jumlah\_peminjaman + jumlah\_pembayaran < 40 then 10  WHEN jumlah\_peminjaman + jumlah\_pembayaran >= 40 AND jumlah\_peminjaman + jumlah\_pembayaran < 60 then 20  WHEN jumlah\_peminjaman + jumlah\_pembayaran >= 60 then 30  END as diskon\_dalam\_persen  FROM  (SELECT c.customer\_id, concat (first\_name, ' ', last\_name) as full\_name, count(\*) as jumlah\_peminjaman  FROM rental r join customer c ON r.customer\_id = c.customer\_id  GROUP BY c.customer\_id) as A  JOIN  (SELECT c.customer\_id, count(\*) as jumlah\_pembayaran  FROM payment p join customer c ON p.customer\_id = c.customer\_id  GROUP BY c.customer\_id) as B ON B.customer\_id = A.customer\_id; |
| --- |

### Duplicate Column

Duplicate Table payment\_id dan customer\_id

| CREATE TABLE p\_id\_c\_id (  payment\_id INTEGER,  customer\_id INTEGER,  PRIMARY KEY (payment\_id, customer\_id),  FOREIGN KEY (payment\_id)  REFERENCES payment(payment\_id) on delete cascade on update cascade,  FOREIGN KEY (customer\_id)  REFERENCES customer(customer\_id) on delete cascade on update cascade  );  INSERT INTO p\_id\_c\_id (SELECT payment\_id, customer\_id FROM payment);  CREATE INDEX p\_id\_c\_id\_customer\_id ON p\_id\_c\_id(customer\_id); |
| --- |

Duplicate Table rental\_id dan customer\_id

| CREATE TABLE r\_id\_c\_id (  rental\_id INTEGER,  customer\_id INTEGER,  PRIMARY KEY (rental\_id, customer\_id),  FOREIGN KEY (rental\_id)  REFERENCES rental(rental\_id) on delete cascade on update cascade,  FOREIGN KEY (customer\_id)  REFERENCES customer(customer\_id) on delete cascade on update cascade  );  INSERT INTO r\_id\_c\_id (SELECT rental\_id, customer\_id FROM rental);  CREATE INDEX r\_id\_c\_id\_customer\_id ON r\_id\_c\_id(customer\_id); |
| --- |

Tuning menggunakan duplicate table dilakukan karena pada query mencari bonus, hanya diperlukan data berupa customer\_id, rental\_id, dan payment\_id. Sisa data lainnya tidak dibutuhkan di relasi tersebut sehingga sisa tersebut akan menjadi overhead pada saat query dilakukan. Data relasi rental\_id dan customer\_id memiliki ukuran 6 byte sedangkan relasi rental memiliki ukuran 36 byte sehingga overhead dari relasi tersebut adalah 30 byte. Begitu pula dengan data payment\_id dan customer\_id yang hanya berukuran 6 byte sedangkan relasi payment memiliki ukuran 30 byte sehingga membuat overhead sebesar 24 byte.

Dengan membuat tabel dengan kolom yang diperlukan saja, maka kita dapat membuat query menjadi lebih optimal karena dengan tabel baru, maka jumlah block transfer dapat berkurang sebanyak 83% pada relasi rental dan 80% pada relasi customer. Hal tersebut dapat terjadi karena ukuran tuple yang mengecil (36 menjadi 6 dan 30 menjadi 6) sehingga memperbanyak jumlah tuple per blok yang mengakibatkan jumlah blok untuk seluruh tuple berkurang. Dengan berkurangnya jumlah blok untuk seluruh tuple, maka berkurang juga jumlah transfer blok untuk sequential scan sehingga query akan menjadi lebih optimal.

| Pengambilan data menggunakan seluruh relasi payment = 315.02 satuan cost |
| --- |
| Pengambilan data menggunakan hanya kolom yang dibutuhkan = 272.02 satuan cost |

Skema yang dihasilkan

|  |
| --- |

### Derived Column

Penambahan kolom jml\_pinjam pada tabel customer

| ALTER TABLE customer  ADD jml\_pinjam INTEGER DEFAULT 0;  UPDATE customer c  SET jml\_pinjam = sq.total  FROM (  SELECT COUNT(\*) total, r.customer\_id  FROM r\_id\_c\_id r  GROUP BY r.customer\_id  ) sq WHERE c.customer\_id = sq.customer\_id; |
| --- |

Penambahan kolom jml\_bayar pada tabel customer

| ALTER TABLE customer  ADD jml\_bayar INTEGER DEFAULT 0;  UPDATE customer c  SET jml\_bayar = sq.total  FROM (  SELECT COUNT(\*) total, p.customer\_id  FROM p\_id\_c\_id p  GROUP BY p.customer\_id  ) sq WHERE c.customer\_id = sq.customer\_id; |
| --- |

Tuning dengan menambahkan derived column dilakukan karena pada query, banyak dilakukan agregasi untuk mencari jumlah peminjaman atau pembayaran setiap orang. Agar tidak perlu dilakukan perhitungan secara terus menerus setiap kali query dieksekusi, maka kami memilih untuk menambahkan derived column jml\_pinjam untuk mencatat banyaknya peminjaman yang dilakukan setiap user dan kolom jml\_bayar untuk mencatat jumlah pembayaran yang dilakukan oleh setiap user. Dengan melakukan hal tersebut, query akan menjadi lebih optimal karena query tidak perlu lagi melakukan join antara customer, rental, dan payment lalu melakukan agregasi untuk mencari jumlah peminjaman dan pembayaran setiap user. Query hanya perlu mengambil data peminjaman dan pembayaran yang ada di setiap kolom. Dengan demikian, query dipastikan akan lebih ringan secara komputasi dan lebih cepat secara waktu.

| Query sebelum penambahan derived column = 936.47 satuan cost |
| --- |
| SELECT full\_name, jumlah\_peminjaman, jumlah\_pembayaran,  CASE  WHEN jumlah\_peminjaman + jumlah\_pembayaran <40 then 10  WHEN jumlah\_peminjaman + jumlah\_pembayaran >= 40 AND jumlah\_peminjaman + jumlah\_pembayaran <60 then 20  WHEN jumlah\_peminjaman + jumlah\_pembayaran >= 60 then 30  END as diskon\_dalam\_persen  FROM  (SELECT c.customer\_id, concat (first\_name, ' ', last\_name) as full\_name, count(\*) as jumlah\_peminjaman  FROM rental r join customer c ON r.customer\_id = c.customer\_id  GROUP BY c.customer\_id) as A  JOIN  (SELECT c.customer\_id, count(\*) as jumlah\_pembayaran  FROM payment p join customer c ON p.customer\_id = c.customer\_id  GROUP BY c.customer\_id) as B ON B.customer\_id = A.customer\_id; |
| Query setelah penambahan derived column = 38.97 satuan cost |
| SELECT CONCAT(first\_name,' ',last\_name)AS full\_name,  jml\_pinjam + jml\_bayar poin,  CASE  WHEN jml\_pinjam + jml\_bayar <40 then 10  WHEN jml\_pinjam + jml\_bayar >= 40 AND jml\_pinjam + jml\_bayar <60 then 20  WHEN jml\_pinjam + jml\_bayar >= 60 then 30  END as diskon\_dalam\_persen  FROM customer |

Skema akhir tabel customer

|  |
| --- |

### Materialized View

Query untuk membuat materialized view poin\_diskon\_customer

| create materialized view poin\_diskon\_customer as  select c.customer\_id, concat(c.first\_name, ' ', c.last\_name) as full\_name, c.jml\_pinjam + c.jml\_bayar as poin,  case  WHEN c.jml\_pinjam + c.jml\_bayar < 40 then 10  WHEN c.jml\_pinjam + c.jml\_bayar >= 40 AND c.jml\_pinjam + c.jml\_bayar < 60 then 20  WHEN c.jml\_pinjam + c.jml\_bayar >= 60 then 30  end as diskon\_dalam\_persen  from customer c |
| --- |

Pembuatan materialized view ini dilakukan dengan asumsi query awal akan diakses sangat sering (*frequent access*). Meski tuning pada 2.3.2 membuat query sudah tidak memerlukan join, operasi-operasi tambahan seperti penentuan persen diskon dan penggabungan nama depan dan belakang masih relatif lebih mahal dibandingkan dengan melakukan operasi tersebut sekali, lalu disimpan dalam bentuk materialized view.

| Cost Query dengan Derived Column: 38.97 |
| --- |
| SELECT CONCAT(first\_name,' ',last\_name)AS full\_name,  jml\_pinjam + jml\_bayar poin,  CASE  WHEN jml\_pinjam + jml\_bayar <40 then 10  WHEN jml\_pinjam + jml\_bayar >= 40 AND jml\_pinjam + jml\_bayar <60 then 20  WHEN jml\_pinjam + jml\_bayar >= 60 then 30  END as diskon\_dalam\_persen  FROM customer |
| Cost Query dengan Materialized View: 10.99 |
| select full\_name, poin, diskon\_dalam\_persen from poin\_diskon\_customer; |

Skema Materialized View

|  |
| --- |

## 

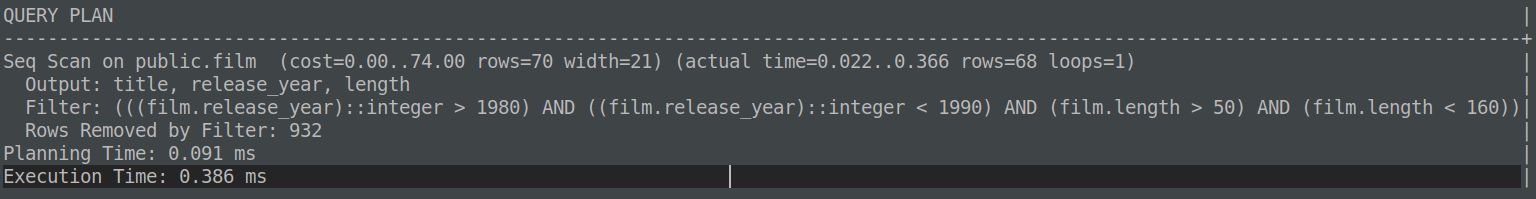
## Database Index Tuning

Query yang akan dianalisis adalah query yang akan menampilkan film yang di rilis pada tahun 1980-1990 (eksklusif) dan memiliki durasi 50 - 160 (eksklusif). SQL Statement query tersebut adalah sebagai berikut:

| SELECT title,release\_year,length  FROM film  WHERE release\_year > 1980 and release\_year < 1990 and length > 50 and length <160; |
| --- |

### Tanpa Indeks

Tanpa menggunakan indeks pada atribut film(release\_year) dan film(length), query execution plan yang dipilih oleh PostgreSQL adalah sebagai berikut:



Dari keluaran tersebut, PostgreSQL akan melakukan sequential scan pada relasi film dengan estimasi total cost 74.00 dan estimasi start-up cost 0.00 (row pertama bisa dikeluarkan langsung). Estimasi cost merupakan kelipatan dari waktu untuk membaca satu page (8kB) secara sekuensial. Estimasi tuple yang akan dihasilkan oleh operasi sequential scan tersebut adalah 70 tuple dengan estimasi 21 bytes per tuplenya.

Operasi sequential scan tersebut setelah dicoba dieksekusi ternyata menghasilkan 68 tuple dan memakan waktu 0.366 ms.

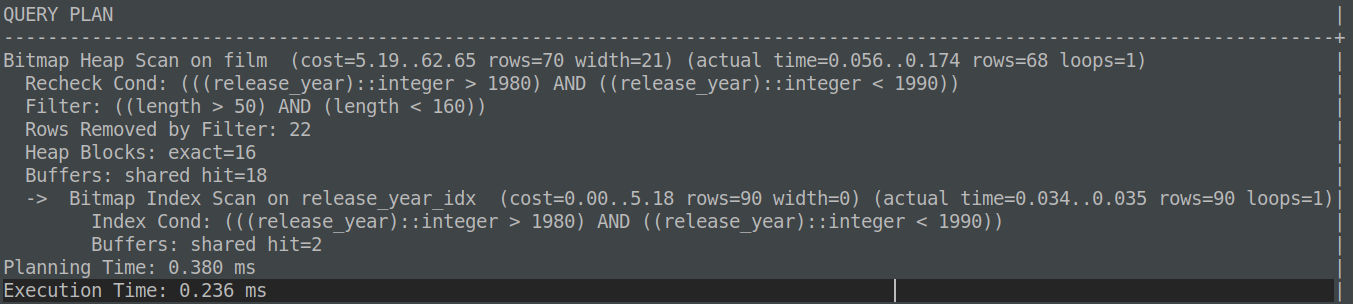
Cost operasi tersebut menjadi relatif mahal karena PostgreSQL melakukan sequential (linear) scan pada keseluruhan relasi film.

### Dengan Single Index pada release\_year dan length

Kita akan menambahkan single index pada kedua atribut yang terkait dengan query yaitu film(release\_year) dan film(length) menggunakan B+ Tree index. B+ Tree index dipilih karena query menggunakan komparator “>” dan “<”. SQL Statement yang digunakan untuk menambahkan index adalah:

| CREATE INDEX release\_year\_idx ON public.film USING btree (release\_year);  CREATE INDEX length\_idx ON public.film USING btree (length); |
| --- |

Setelah itu, didapatkan query plan sebagai berikut:



Terlihat bahwa query tersebut sudah diproses menggunakan index scan pada film(release\_year) dan difetch menggunakan heap scan. Indexing pada film(length) tidak digunakan karena hasil query dengan length > 50 dan length < 160 cukup banyak sehingga lebih efektif menggunakan filtering pada heap scan saja setelah index scan pada release\_year.

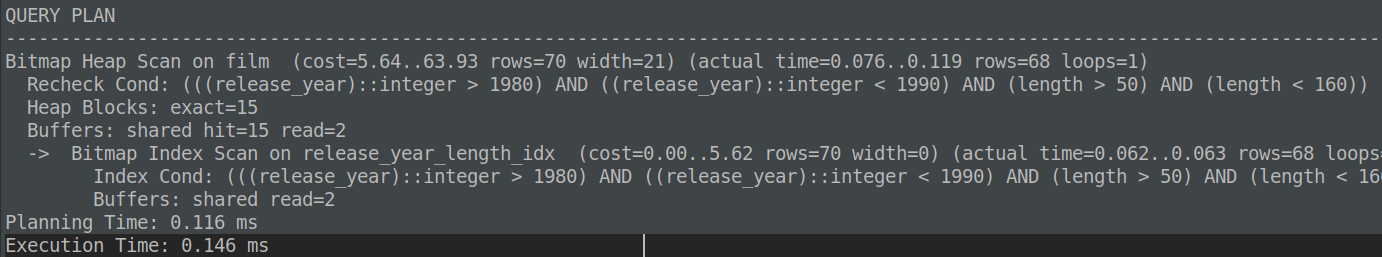
Total estimated cost yang didapatkan adalah 62.65 (sudah termasuk child node i.e. index scan). Cost masih relatif tinggi karena, meski pada query ini tidak diperlukan lagi checking 1980 < release\_year < 1990 pada proses heap scan, estimasi cost heap scan tetap memperhitungkan checking tersebut karena bisa jadi index scan mengirimkan lossy block (mengirimkan address page yang mengandung tuple query, bukan address tuplenya langsung) ke heap scan. Pengurangan cost hanya terjadi pada aspek jumlah row yang diproses. Jumlah estimasi disk page read tetap karena ini bukan clustered index.

### Dengan Composite Index pada film(release\_year, length)

Kita akan menambahkan composite index pada dua atribut yang menjadi kriteria query sesuai dengan urutannya yaitu film(release\_year, length). SQL Statement untuk index ini adalah sebagai berikut:

| CREATE INDEX release\_year\_length\_idx ON public.film USING btree (release\_year, length); |
| --- |

Setelah itu, didapatkan query plan sebagai berikut:



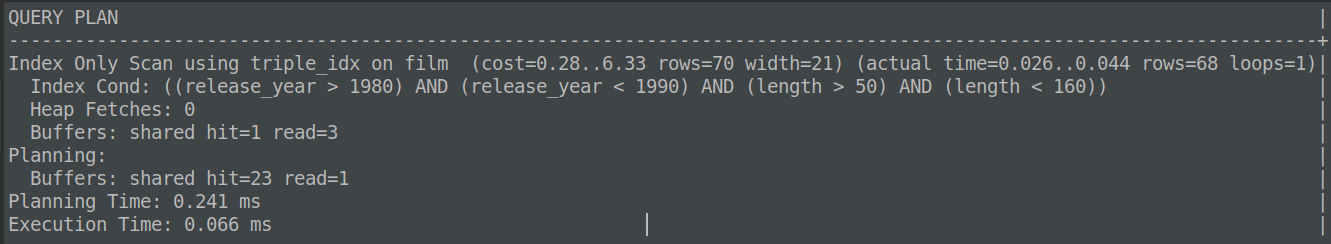
Total cost akhir dari query plan ini adalah 63.93. Jumlah tersebut tidak jauh berbeda dibandingkan dengan query plan yang dihasilkan pada 2.4.2. Hal ini karena proses dari index scan dari index film(release\_year) dan film(release\_year, length) menghasilkan estimasi jumlah tuple yang hampir sama (90 untuk single index dan 70 untuk composite index). Karena selisih dari estimasi tuple yang dikirim ke bitmap heap scan tidak terlalu besar, maka estimasi cost untuk bitmap heap scan pada kedua query plan juga mirip.

### Dengan Composite Index pada film(release\_year, length, title)

Kita akan menambahkan B+ Tree composite index pada atribut film(release\_year, length, title) dengan harapan dapat mengutilisasi Index-Only Search untuk pencarian data. Urutan atribut indexing dibuat menyesuaikan urutan kondisi selection. Atribut “title” ditaruh di akhir indeks karena hanya sebagai informasi untuk projection. SQL Statement untuk penambahan indeks ini adalah sebagai berikut:

| CREATE INDEX triple\_idx ON public.film USING btree (release\_year, length, title); |
| --- |

Setelah itu, didapatkan query plan sebagai berikut:



Terlihat, hanya ada satu operasi untuk query ini yaitu Index-Only Scan dengan estimasi cost yang memiliki selisih signifikan dibandingkan dengan alternatif di 2.4.1. - 2.4.3. Kita bisa menurunkan estimasi total cost dari 60-an menjadi hanya 6.33 saja. Karena informasi yang dibutuhkan oleh query sudah ada pada index, Index-Only Scan langsung mengeluarkan data yang sesuai dari index ke output tanpa harus melakukan heap access terlebih dahulu. Dengan begitu, kita dapat memangkas operasi random heap access yang mahal.

**Catatan:**

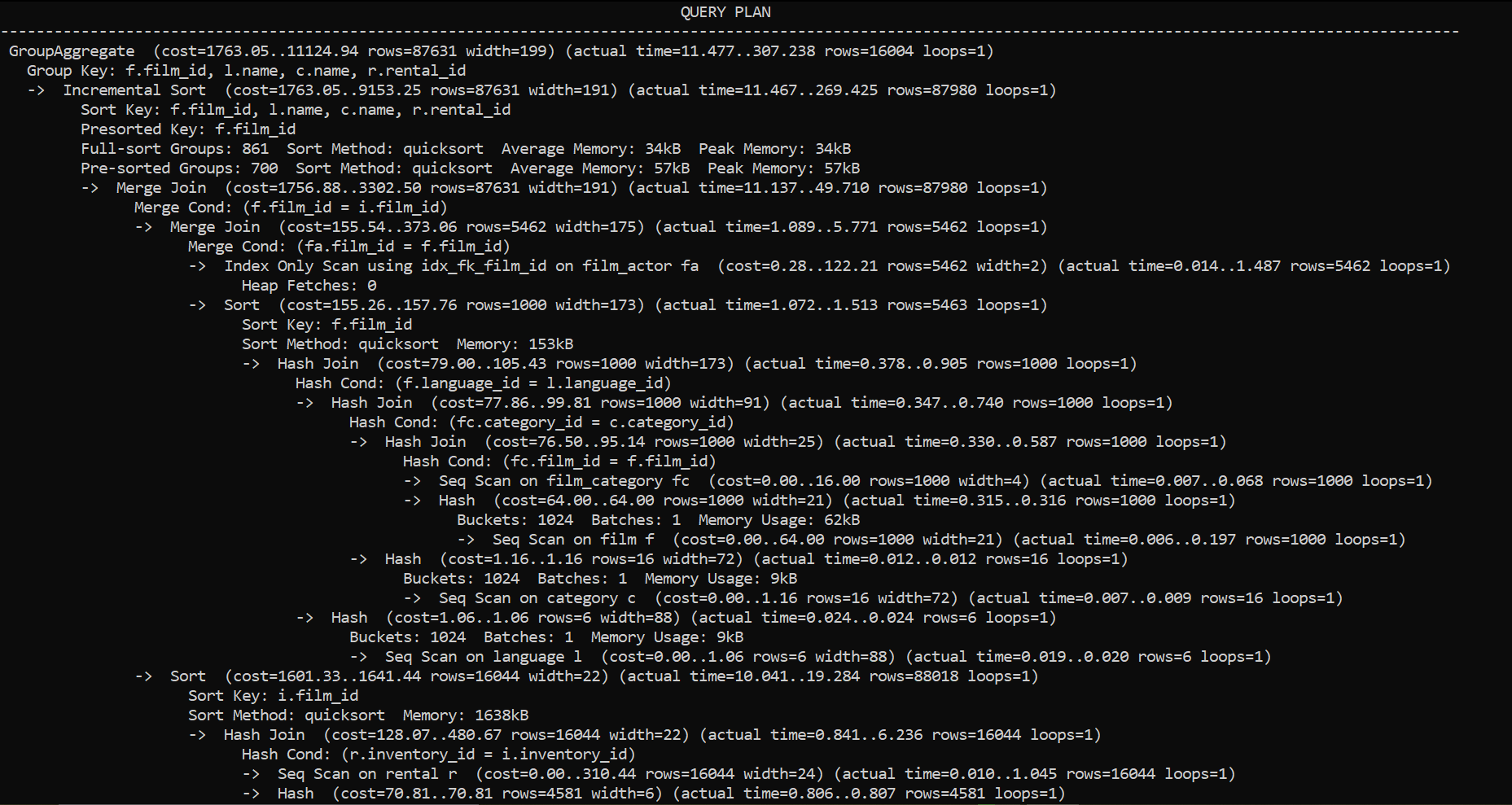
Jika PostgreSQL tidak menggunakan Index Only Scan setelah penambahan indeks ini, silakan eksekusi query VACUUM film; terlebih dahulu untuk me-refresh visibility map dari relasi film pada PostgreSQL.

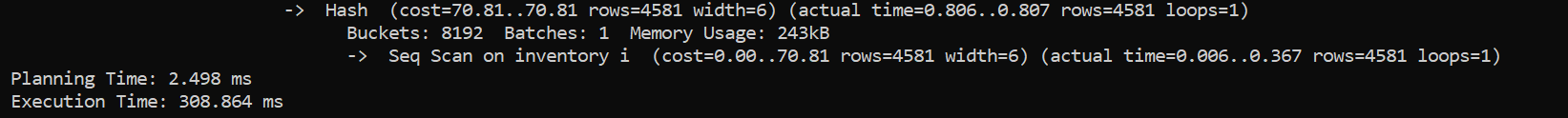
## 

## Observasi Perubahan Query Plan

Query 1:

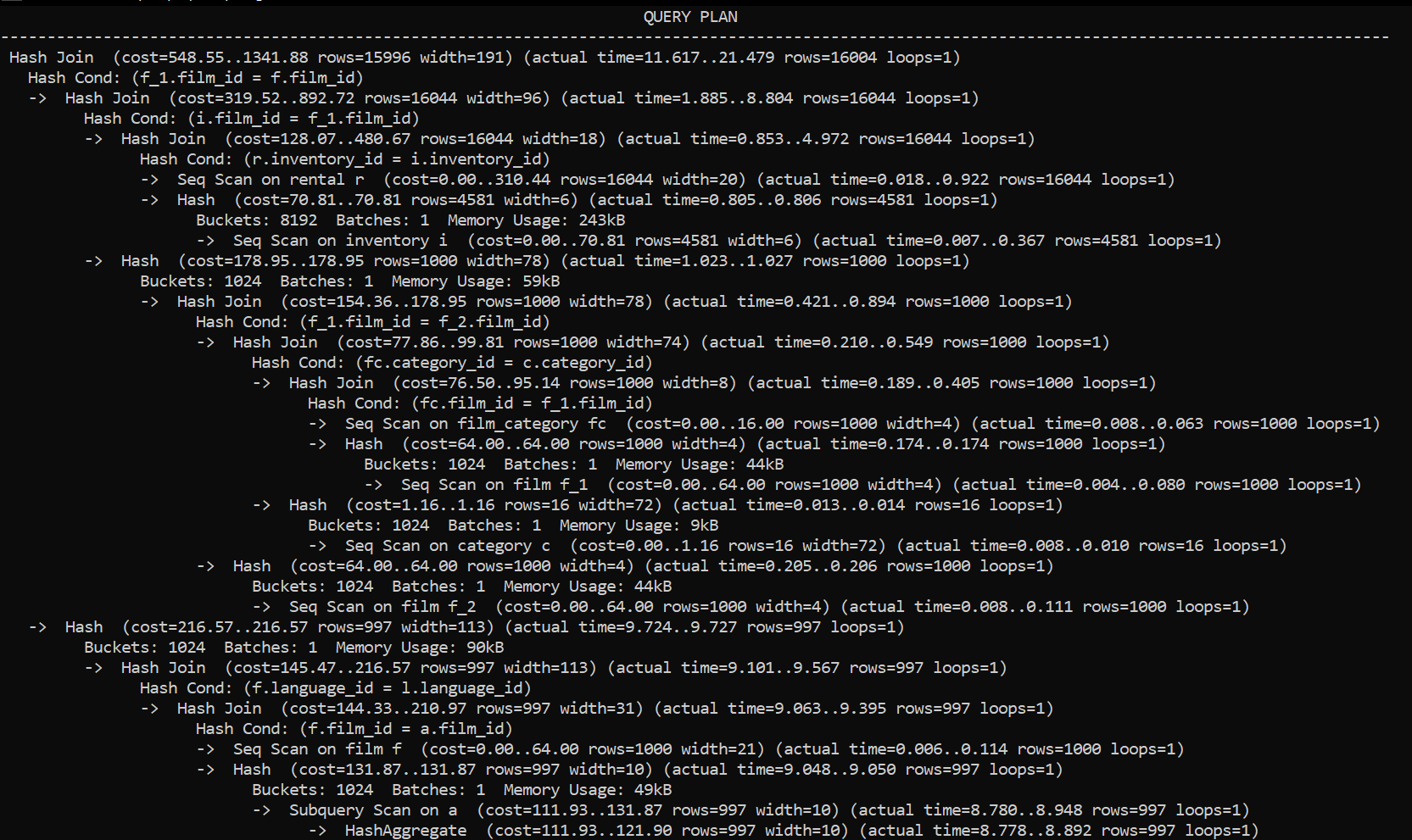
| SELECT f.title as judul\_film, l.name as bahasa\_film, c.name as kategori\_film, count(\*) as jumlah\_aktor, r.rental\_date, r.return\_date FROM film f  JOIN film\_category fc on f.film\_id = fc.film\_id  JOIN category c on c.category\_id = fc.category\_id  JOIN language l on l.language\_id = f.language\_id  JOIN film\_actor fa on f.film\_id = fa.film\_id  JOIN inventory i on i.film\_id = f.film\_id  JOIN rental r on r.inventory\_id = i.inventory\_id  GROUP BY f.film\_id, f.film\_id, f.title, l.name, c.name, r.rental\_date, r.return\_date, r.rental\_id; |
| --- |

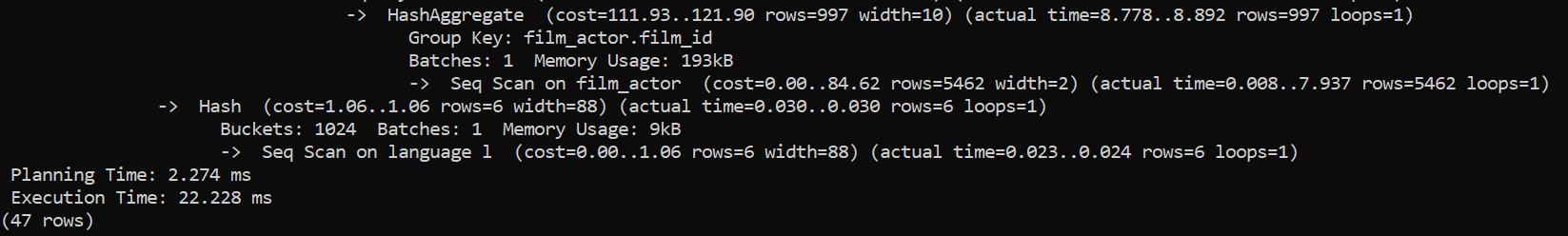




Query 2:

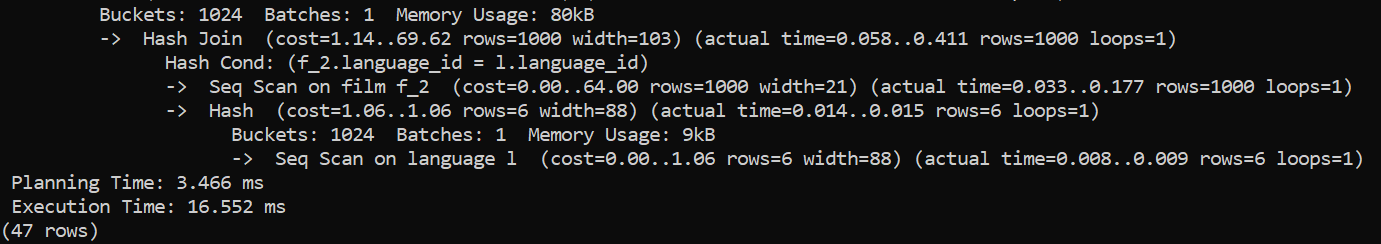
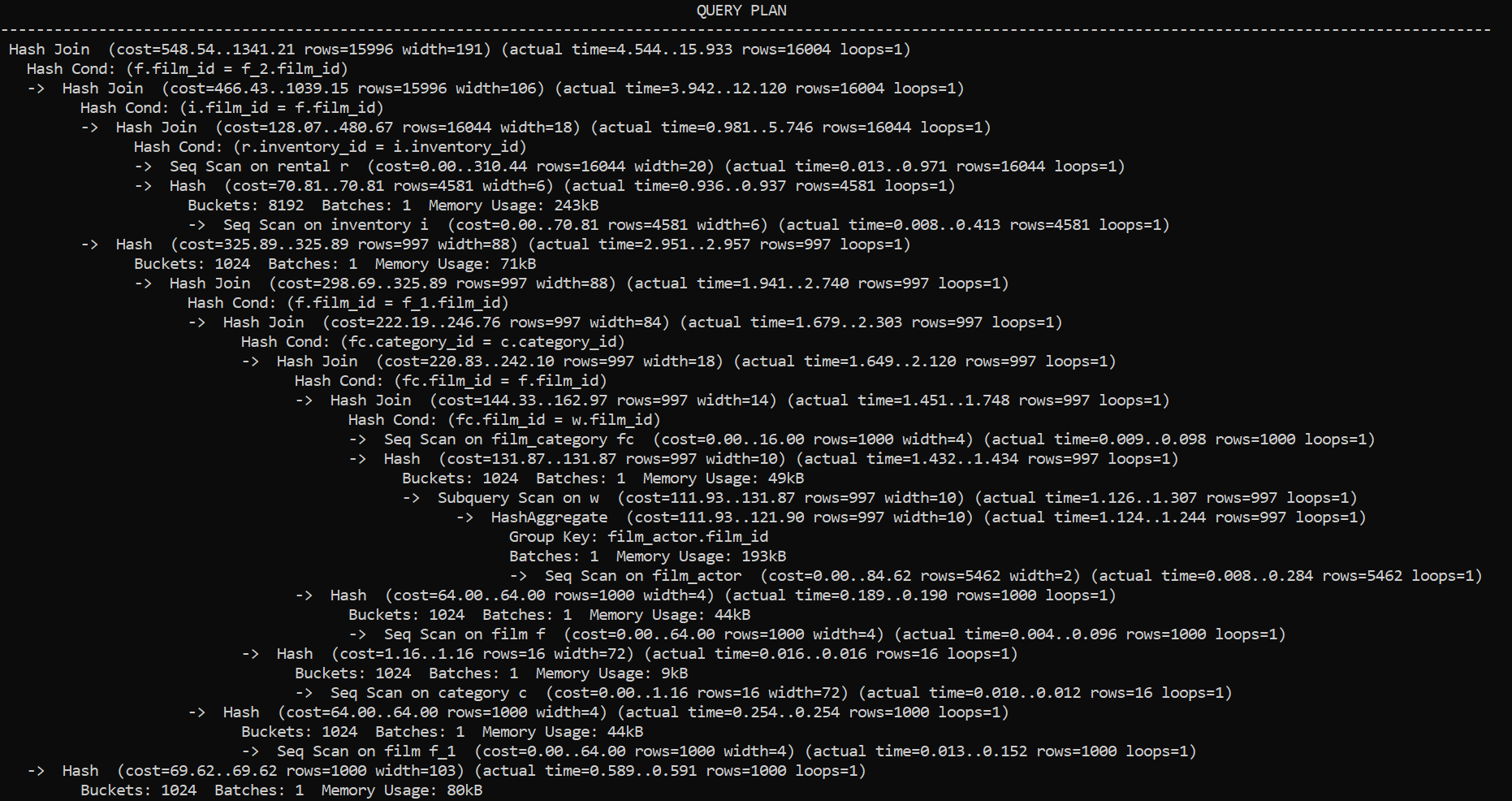
| SELECT title, language, category, jumlah\_aktor, rental\_date, return\_date  FROM  (SELECT a.film\_id, title, jumlah\_aktor, language  FROM  (SELECT film\_id, count(\*) as jumlah\_aktor  FROM film\_actor  GROUP BY film\_id) as a  JOIN  (SELECT f.film\_id, f.title, l.name as language  FROM film f  JOIN language l on l.language\_id = f.language\_id) as b on b.film\_id = a.film\_id) as e  JOIN  (SELECT c.film\_id, category, rental\_date, return\_date  FROM  (SELECT f.film\_id, c.name as category  FROM film f  JOIN film\_category fc on f.film\_id = fc.film\_id  JOIN category c on c.category\_id = fc.category\_id) as c  JOIN  (SELECT f.film\_id, r.rental\_date, r.return\_date  FROM film f  JOIN inventory i on i.film\_id = f.film\_id  JOIN rental r on r.inventory\_id = i.inventory\_id) as d on c.film\_id = d.film\_id) as f on e.film\_id = f.film\_id; |
| --- |





Query 3:

| WITH X as (SELECT f.film\_id, c.name as category  FROM film f  JOIN film\_category fc on f.film\_id = fc.film\_id  JOIN category c on c.category\_id = fc.category\_id),  Y as (SELECT f.film\_id, r.rental\_date, r.return\_date  FROM film f  JOIN inventory i on i.film\_id = f.film\_id  JOIN rental r on r.inventory\_id = i.inventory\_id),  W as (SELECT film\_id, count(\*) as jumlah\_aktor  FROM film\_actor  GROUP BY film\_id),  Z as (SELECT f.film\_id, f.title, l.name as bahasa  FROM film f  JOIN language l ON f.language\_id = l.language\_id)  SELECT title, bahasa, category, jumlah\_aktor, rental\_date, return\_date  FROM X  JOIN Y ON X.film\_id = Y.film\_id  JOIN W ON X.film\_id = W.film\_id  JOIN Z ON X.film\_id = Z.film\_id; |
| --- |

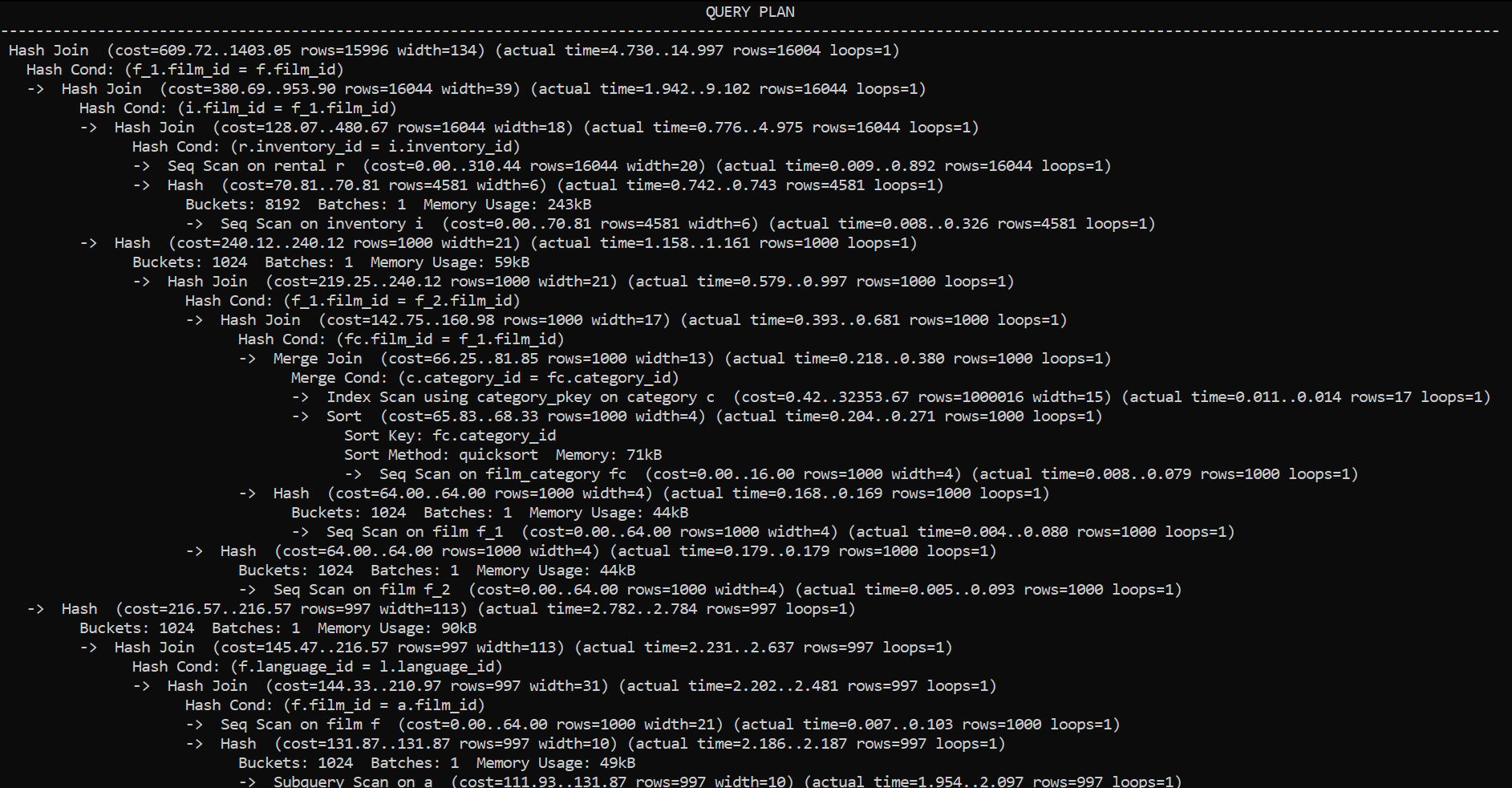


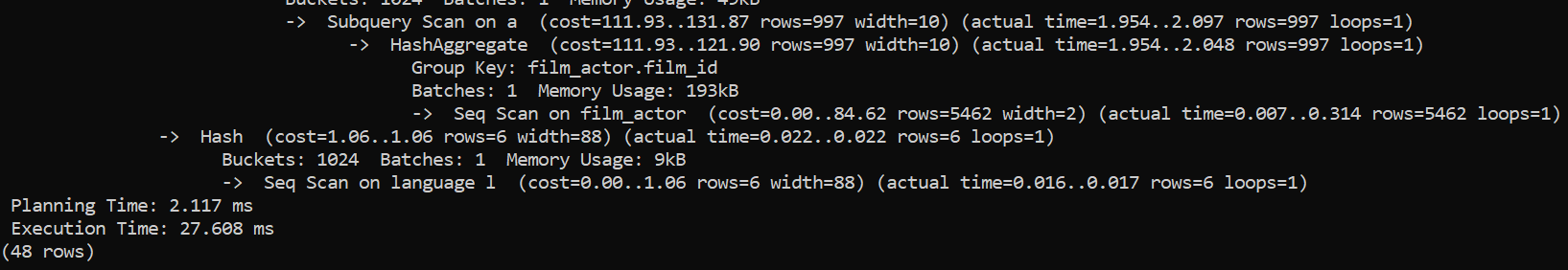
Dengan menambahkan 1.000.000 data kepada relasi *category*, maka dapat dilihat perbedaan query plan pada query alternatif yang kedua.

Proses yang dilakukan:

| START TRANSACTION;  INSERT INTO CATEGORY(name)  SELECT substr(md5(random()::text), 1, 10)  FROM generate\_series(1, 1000000); |
| --- |

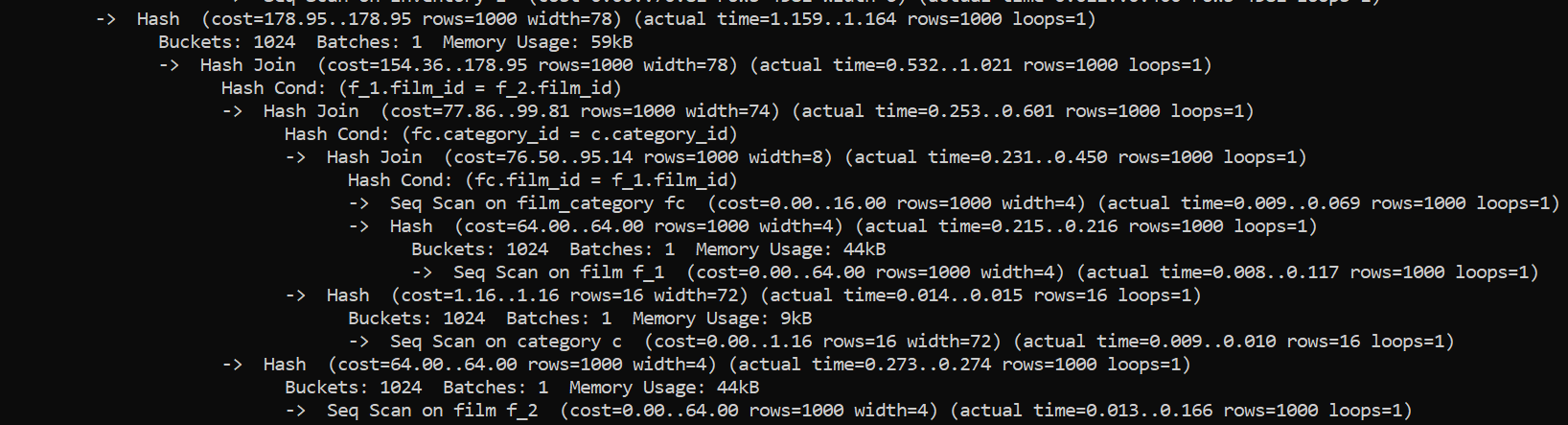
Hasil query plan alternatif 2 setelah penambahan data:



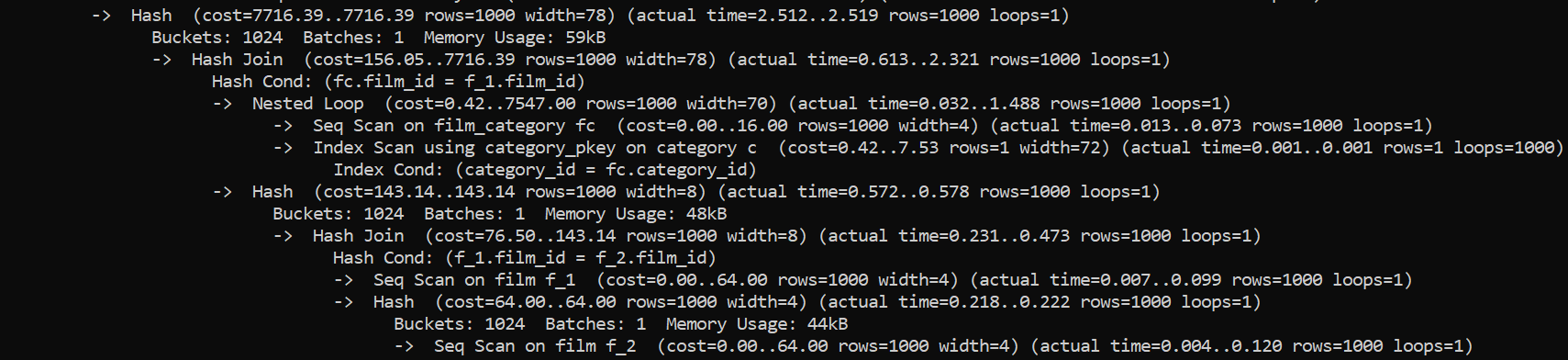


Dapat dilihat bahwa terdapat perbedaan pada bagian query plan yang bersangkutan dengan relasi category.

Sebelum proses:



Sesudah proses:



Dapat dilihat bahwa terdapat perbedaan query plan pada bagian yang telah ditampilkan. Perbedaan yang paling menonjol adalah operasi join antara relasi film\_category dan category. Sebelum ditambahkan 1.000.000 data baru ke relasi category, proses join dilakukan dengan menggunakan teknik Hash Join, akan tetapi setelah ditambahkan 1.000.000 data baru, pada proses join yang sama digunakan Nested Loop. Hal tersebut dikarenakan untuk melakukan Hash Join, database perlu membangun tabel hash terlebih dahulu. Hal tersebut merupakan hal yang cepat dan efisien untuk dilakukan apabila jumlah tuple pada relasi tidak banyak (pada kasus category, sebanyak 16 tuple). Akan tetapi, apabila terdapat jumlah tuple yang sangat banyak, pembangunan tabel hash menjadi tidak efisien (sesudah proses tuple menjadi 1.000.016). Alhasil dari hal tersebut, maka digunakannya Nested Loop join yang jauh lebih murah dan efisien bila dibandingkan dengan Hash Join pada kasus ini.

Sebagai respons dari perubahan jumlah data tersebut, maka terjadi perubahan urutan join pada query plan juga. Sesudah ditambahkan data, proses join untuk relasi selain *category* dilakukan terlebih dahulu karena diperkirakan akan menghasilkan jumlah tuple yang lebih sedikit. Hal tersebut berbeda dengan query plan sebelum proses dimana join yang berhubungan dengan relasi *category* dilakukan terlebih dahulu sebelum join dengan relasi yang lain.

## 

## Database Programming

Trigger akan dilakukan pada relasi rental dan payment sehingga setiap ada operasi INSERT, UPDATE, atau DELETE, entri pada tuning 2.3.1 dan 2.3.2 bisa menyesuaikan. Setelah itu, akan dilakukan refresh pada materialized view 2.3.3.

Trigger untuk relasi rental

| create or replace function adjust\_rental\_addition()  returns trigger  language plpgsql  as  $$ begin  /\* r\_id\_c\_id insert \*/  insert into r\_id\_c\_id values (new.rental\_id, new.customer\_id);  /\* +1 to customer(jml\_pinjam) \*/  update customer set jml\_pinjam = jml\_pinjam + 1 where customer\_id = new.customer\_id;  /\* refresh poin\_diskon\_customer \*/  refresh materialized view poin\_diskon\_customer;  return null;  end $$;  create or replace function adjust\_rental\_deletion()  returns trigger  language plpgsql  as  $$ begin  /\* r\_id\_c\_id delete \*/  delete from r\_id\_c\_id rc where rc.rental\_id = old.rental\_id;  /\* -1 to customer(jml\_pinjam) \*/  update customer set jml\_pinjam = jml\_pinjam - 1 where customer\_id = old.customer\_id;  /\* refresh poin\_diskon\_customer \*/  refresh materialized view poin\_diskon\_customer;  return null;  end $$;    create trigger rental\_trigger\_add  after insert or update on rental for each row  execute function adjust\_rental\_addition();  create trigger rental\_trigger\_del  after delete on rental for each row  execute function adjust\_rental\_deletion(); |
| --- |

Trigger untuk relasi payment

| create or replace function adjust\_payment\_addition()  returns trigger  language plpgsql  as  $$ begin  /\* p\_id\_c\_id insert \*/  insert into p\_id\_c\_id values (new.payment\_id, new.customer\_id);  /\* +1 to customer(jml\_pinjam) \*/  update customer set jml\_bayar = jml\_bayar + 1 where customer\_id = new.customer\_id;  /\* refresh poin\_diskon\_customer \*/  refresh materialized view poin\_diskon\_customer;  return null;  end $$;  create or replace function adjust\_payment\_deletion()  returns trigger  language plpgsql  as  $$ begin  /\* r\_id\_c\_id delete \*/  delete from p\_id\_c\_id pc where pc.payment\_id = old.payment\_id;  /\* -1 to customer(jml\_pinjam) \*/  update customer set jml\_bayar = jml\_bayar - 1 where customer\_id = old.customer\_id;  /\* refresh poin\_diskon\_customer \*/  refresh materialized view poin\_diskon\_customer;  return null;  end $$;    create trigger payment\_trigger\_add  after insert or update on payment for each row  execute function adjust\_payment\_addition();  create trigger payment\_trigger\_del  after delete on payment for each row  execute function adjust\_payment\_deletion(); |
| --- |

# Bab III Kesimpulan dan Saran

Bab III ini berisi kesimpulan dan saran berdasarkan pembahasan yang ada di dalam laporan serta implementasi Tugas Besar 1 IF3140 Manajemen basis data.

**3.1 Kesimpulan**

Telah Berhasil di implementasikan sebuah pengoptimalan basis data berupa langkah-langkah seperti pembuatan skema basis data, Query Processing, Database Design Tuning, Database index tuning, observasi perubahan Query Plan, dan pembuatan trigger sesuai dengan yang diminta dalam spesifikasi Tugas Besar 1 IF3140 Manajemen Basis data Semester 1 Tahun 2022/2023. Hal mengenai pengoptimalan yang berhasil diimplementasikan dalam basis data ini meliputi :

1. Konsep Query processing & optimization guna menganalisa query untuk menentukan sumber-sumber apa saja yang digunakan oleh query tersebut dan apakah dapat dioptimalkan kembali bentuknya tanpa merubah output.
2. Konsep Database Tuning sebagai peningkatan kinerja pada desain basis data secara fisikal yang mencakup relasi dan view sesuai dengan kebutuhan Manajer basis data.
3. Implementasi Trigger sebagai pemicu prosedur tersimpan secara otomatis pada saat ada perubahan/modifikasi data dalam tabel.
4. Eksplorasi Alternatif strategi yang berkaitan dengan pengoptimalan Basis Data guna mendapatkan sebuah database yang bekerja secara maksimal dan optimal

Pengimplementasian konsep di atas kemudian berhasil digunakan untuk menyelesaikan seluruh poin-poin pengoptimalan basis data yang ada di spesifikasi tugas besar 1 IF3140 Manajemen Basis data . Penggunaan konsep tersebut dilakukan secara maksimal dengan melihat aspek-aspek sekitar sehingga menghasilkan kinerja dan performa terbaik dari sebuah basis data.

Dengan demikian dapat disimpulkan bahwa basis data yang baik adalah basis data yang dilihat dari berbagai aspek komponen. Dalam penerapannya didalam industri maupun pribadi penggunaan basis data akan lebih maksimal apabila di optimalkan. Yang sebelumnya basis data dengan konsumsi memori dan kecepatan pembaca data terasa lambat namun dengan pengoptimalisasi seperti melakukan tuning terhadap query, index dan lain-lain dapat meningkatkan kinerja basisdata itu sendiri.

## 3.1 Saran

Tugas Besar 1 IF3140 Manajemen Basis Data Semester 1 Tahun 2022/2023 menjadi salah satu proses pembelajaran bagi kelompok dan tiap anggota dalam menerapkan ilmu-ilmu yang diajarkan atau didapat pada kuliah maupun hasil eksplorasi pembelajaran secara mandiri. Saran yang dapat diberikan setelah menyelesaikan Tugas Besar 1 ini. Yaitu

1. Teknik-teknik pengoptimalan yang dilakukan dapat diterapkan pada studi kasus yang berbeda dan menyesuaikan dengan modul-modul yang telah ada sehingga ilmu yang didapat dapat dikembangkan lebih jauh lagi
2. Melakukan perancangan suatu sistem basis data hendaknya memahami betul bagaimana sistem basis data tersebut berjalan. Hal itu dapat di implementasi dengan pembuatan ERD, skema, dll agar sistem yang dibuat bekerja dengan efektif dan efisien
3. Dalam pengoptimalan basis data dibutuhkan daya tingkat ketelitian yang tinggi serta pengetahuan akan pengimplemtasian teknik-teknik Optimalisasi guna memaksimalkan optimasi yang dapat dilakukan.
4. Untuk mencegah terjadinya kehilangan data, maka perlu diterapkan metode-metode keamanan seperti Backup data secara otomatis yang diatur pada waktu-waktu tertentu

# Bab IV Pembagian Kerja dalam Kelompok

| **No** | **NIM/Nama** | **Tugas** |
| --- | --- | --- |
| 1 | 10022191 / Muhammad Meiditya Putra | 1. Mengerjakan query pada pada bagian 2.3  2. Menuliskan Kesimpulan dan saran pada Bab III |
| 2 | 13520034 / Bryan Bernigen | 1. Mengerjakan bagian 2.1  2. Mengerjakan bagian 2.2  3. Mengerjakan tuning vertical dan redundant column (2.2.1 dan 2.2.2) 4. Membantu mencari solusi 2.5.2 |
| 3 | 13520133 / Jevant Jedidia Augustine | 1. Mengerjakan bagian 2.5  2. Menuliskan dasar teori  3. Membuat query pada bagian 2.3 |
| 4 | 13520157 / Thirafi Najwan Kurniatama | 1. Mengerjakan bagian 2.4  2. Mengerjakan tuning materialized view (2.3.3)  3. Mengerjakan database programming (2.6) |

# 

# Referensi

*Explain*. PostgreSQL Documentation. (2022, October 13). Retrieved October 19, 2022, from <https://www.postgresql.org/docs/current/sql-explain.html>

*11.9. index-only scans and covering indexes*. PostgreSQL Documentation. (2022, October 13). Retrieved October 19, 2022, from <https://www.postgresql.org/docs/current/indexes-index-only-scans.html>

*Statistics used by the planner*. PostgreSQL Documentation. (2012, January 1). Retrieved October 19, 2022, from <https://www.postgresql.org/docs/8.1/planner-stats.html>