Nama : Tersiqo Alfarezel

Kelas : SIB 2D

NIM : 244107060089

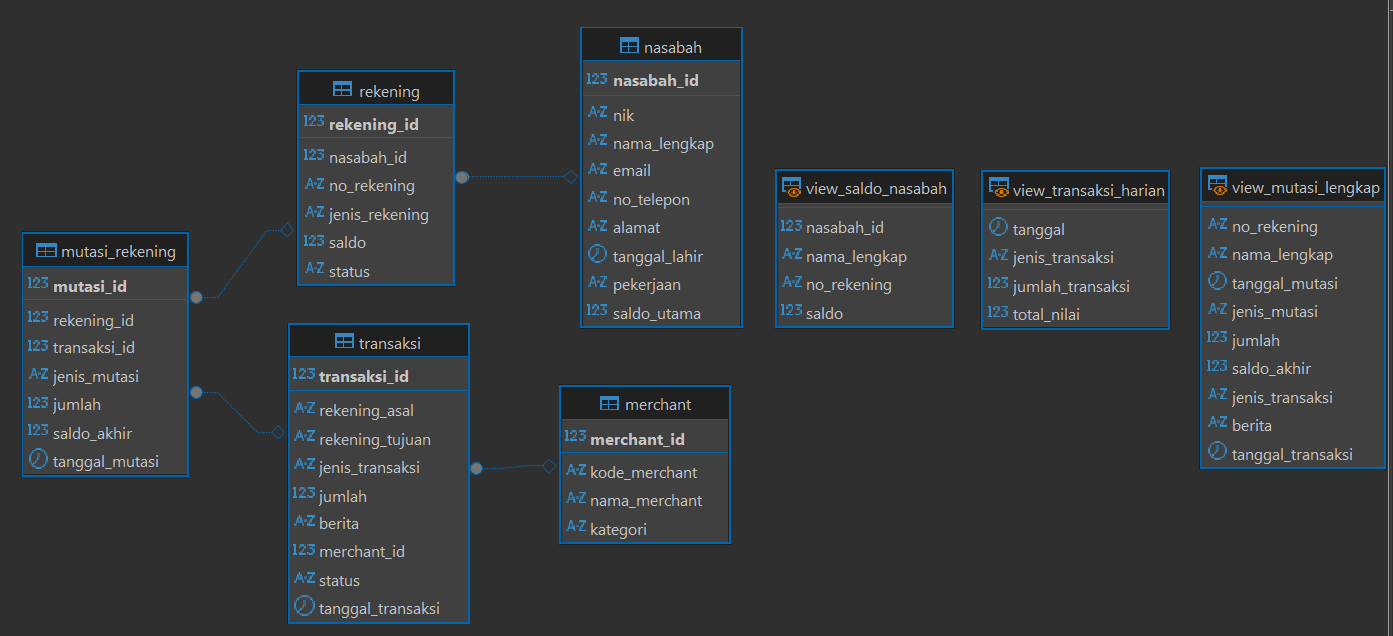
Absen : 20

|  |  |  |
| --- | --- | --- |
| Mata Kuliah | **:** | Basis Data Lanjut |
| Program Studi | **:** | D4 – Teknik Informatika **/** D4 – Sistem Informasi Bisnis |
| Semester | **:** | 3 (tiga) |
| Pertemuan ke- | **:** | 7 |

**JOBSHEET 07  
Transaction & Concurrenc**

**Praktikum 00 – Menyiapkan file Studi Kasus (db\_mobile\_banking)**

1. Download file db\_mobile\_banking.sql yang berisi **data dummy** pada contoh desain basis data transaksi mobile banking.
2. Buka aplikasi dBeaver, lalu buat database baru bernama db\_mobile\_banking
3. Kemudian klik kanan Tools 🡪 Execute Script 🡪
4. Kemudian arahkan ke file db\_akademik.sql yang sudah ada
5. Selanjutnya klik Next 🡪 Start
6. Jika sudah selesai, cek digram ERD melalui klik kanan Shemas/public 🡪 View Diagram
7. **Pertanyaan 1**
   1. Screenshot hasil diagram yang sudah ada, dan berikan hasil analisis kalian mengenai desain basis data pada db\_mobile\_banking



* Nasabah menyimpan data pelanggan yaitu id, nik, nama lengkap, kontak, alamat, tanggal lahir, pekerjaan, saldo utama)
* Rekening menyimpan data rekening yang dimiliki nasabah yaitu id, id nasabah, nomor rekening, jenis rekening, status
* Transaksi mencatat detail setiap perpindahan dana yaitu id, rekening asal, rekening tujuan, jenis transaksi, jumlah, berita, id merchant status, tanggal transaksi
* Mutase\_rekening mencatat perubahan saldo pada rekening yaitu id mutase, id rekening, id transaksi, jenis mutasi, saldo akhir, jumlah mutase
* Merchant menyimpan data pihak penerima atau penyedia barang yaitu id merchant, kode merchant, nama merchant, kategori
  1. jelaskan pula relasi yang ada
* nasabah >> rekening , satu nasabah dapat memiliki banyak rekening
* rekening >> mutase\_rekening , satu rekening dapat memiliki banyak catatan mutase\
* transaksi >> mutase\_rekening , satu transaksi dapat memicu banyak catatan mutase
* merchant >> transaksi , satu merchant dapat terlibat dalam banyak transaksi
* rekening >> transaksi , satu rekening dapat menjadi asal atau tujuan bagi banyak transaksi

**Praktikum 01 – Basic Transaction**

**Transaction** pada DBMS merupakan sekumpulan operasi database yang dianggap sebagai satu kesatuan (*atomic unit of work*). Tujuannya menjaga agar data tetap konsisten walaupun ada banyak operasi atau kegagalan. Transaction harus memenuhi prinsip **ACID (Atomicity, Consistency, Isolation, Durability)**. Perintah dasar transaction terdiri dari:

|  |  |
| --- | --- |
| **Perintah** | **Keterangan** |
| BEGIN | memulai transaksi, semua query setelah BEGIN akan dianggap satu kesatuan transaksi sampai ada COMMIT atau ROLLBACK. |
| COMMIT | Menyimpan semua perubahan transaksi ke database secara permanen. Setelah COMMIT, data tidak bisa dibatalkan dengan ROLLBACK. |
| ROLLBACK | Membatalkan semua perubahan sejak transaksi dimulai (kembali ke kondisi sebelum BEGIN). |
| SAVEPOINT | Membuat titik checkpoint dalam transaksi. Bisa melakukan ROLLBACK TO SAVEPOINT tanpa membatalkan seluruh transaksi. |

Sintaks dasar dari transaction adalah:

ACID merupakan sekumpulan karakteristik yang harus dimiliki sebuah transaksi basis data agar data tetap **benar, konsisten,** dan **andal,** meskipun ada transaksi yang berjalan bersamaan atau terjadi kegagalan.

1. *Atomicity*

Memastikan bahwa seluruh operasi dalam satu transaksi dianggap sebagai satu kesatuan yang utuh (***all – or – nothing*** ). Jika semua berhasil, maka transaksi COMMIT Jika ada yang gagal, maka semua perubahan dibatalkan (ROLLBACK).

1. *Consistency*

Transaksi harus membawa database dari satu **state valid** ke **state valid lainnya** sesuai aturan bisnis, constraint, dan integritas data. Mencegah data keluar dari aturan integritas (misalnya saldo tidak boleh negatif, constraint primary key harus unik).

1. *Isolation*

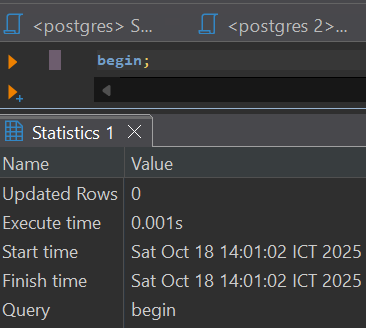
Transaksi berjalan **seolah-olah sendiri**, tanpa gangguan transaksi lain. Menghindari anomali concurrency: dirty read, non-repeatable read, phantom read, lost update.

1. *Durability*

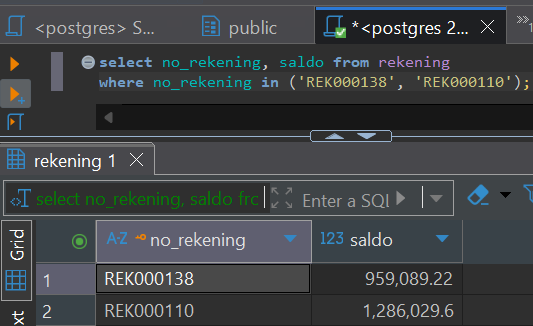
Setelah transaksi berhasil (commit), perubahan data akan tetap tersimpan meskipun ada kegagalan sistem. Menjamin data yang sudah di-commit tidak hilang.

Pada praktikum ini, kita akan mempraktikkan simulasi transaction:

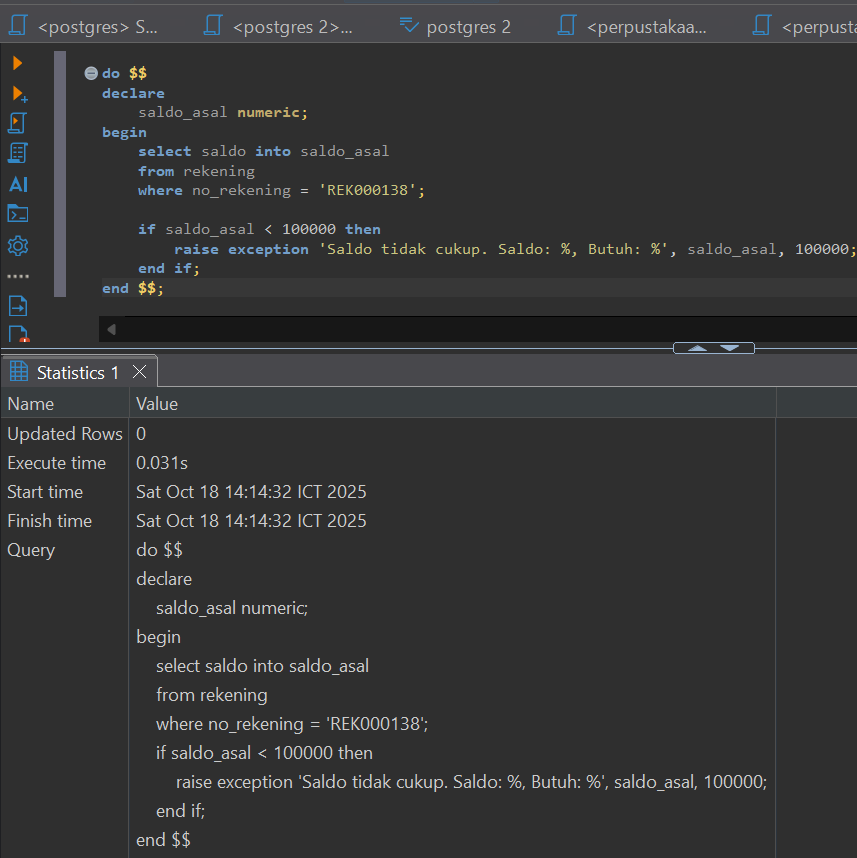
1. Kita buka SQL Editor dengan cara klik kanan Schemas/public 🡪 SQL Editor 🡪 New SQL Script
2. Langkah pertama adalah memulai transaksi dengan eksekusi perintah “BEGIN”



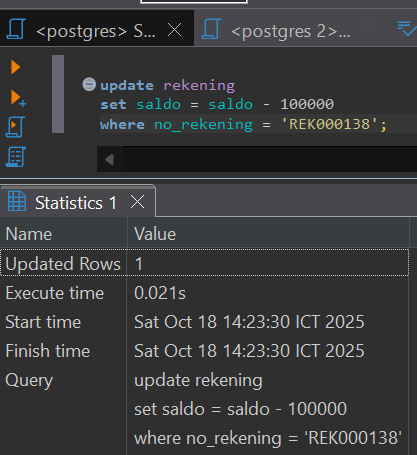
1. Eksekusi query SELECT berikut dan screenshot hasilnya



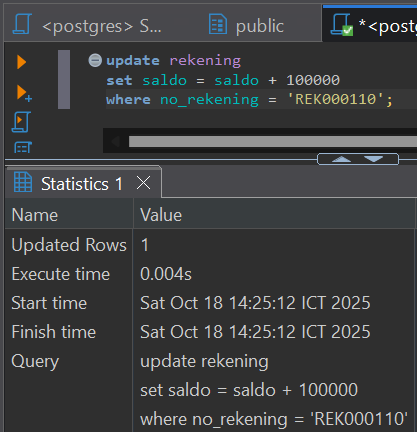
1. Eksekusi query berikut dan catat apa yang terjadi



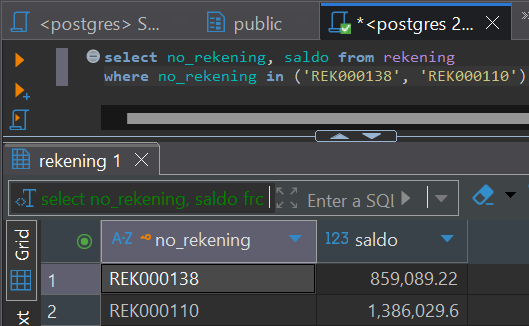
1. Eksekusi query UPDATE berikut untuk melakukan debit rekening asal



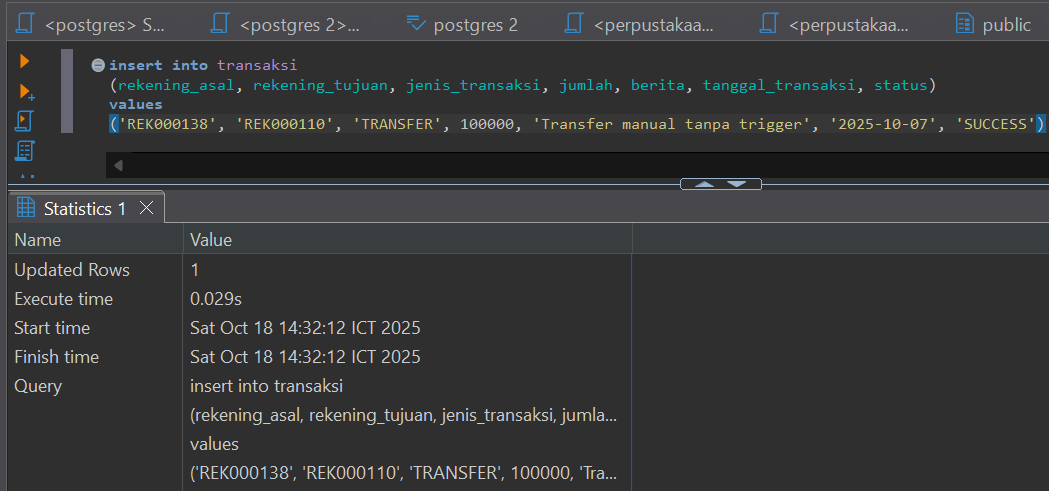
1. Eksekusi query UPDATE berikut untuk melakukan credit ke rekening tujuan



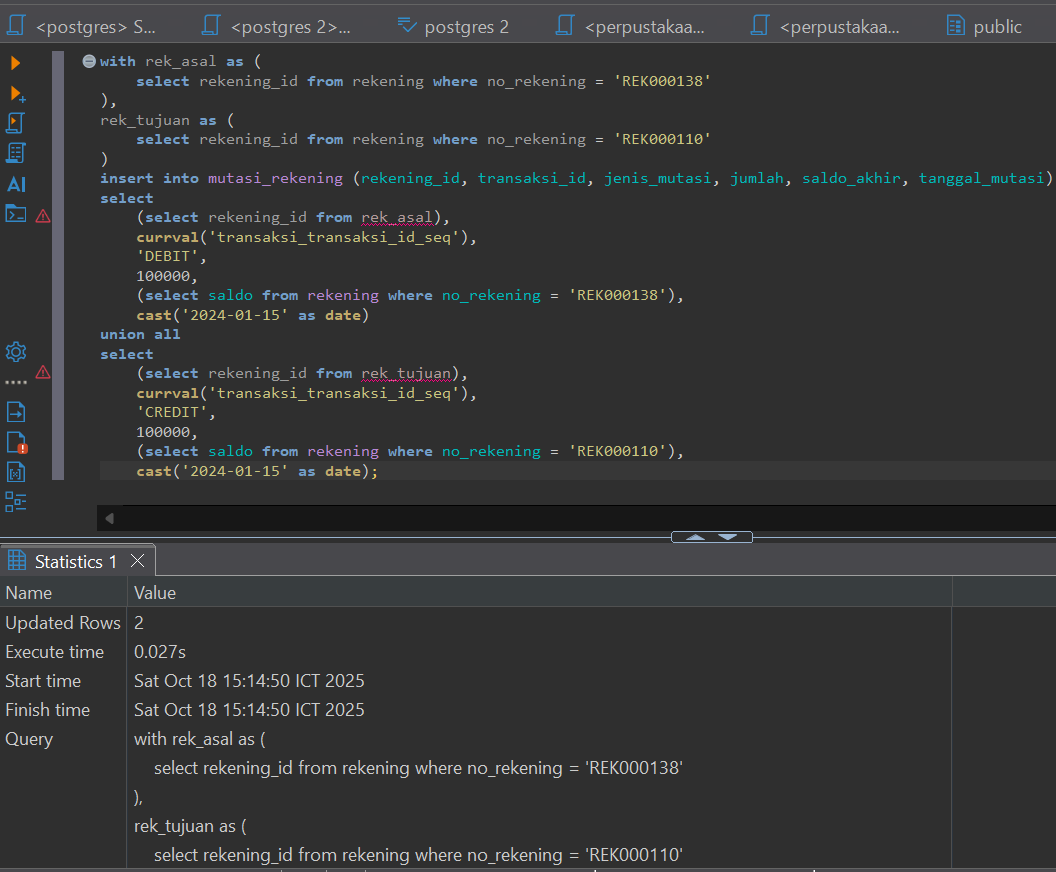
1. Cek saldo dalam transaksi dan screenshot hasilnya



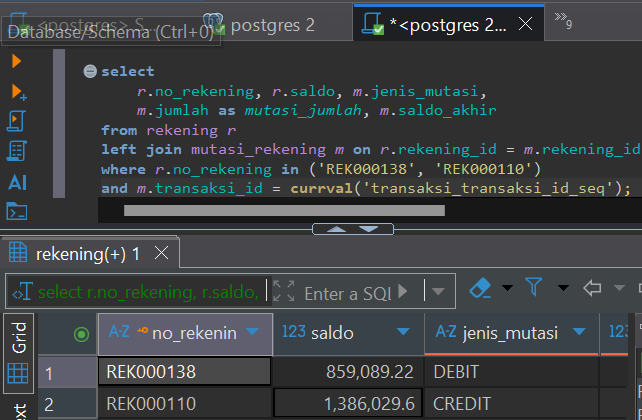
1. Catat transaksi ke dalam tabel transaksi menggunakan query INSERT berikut



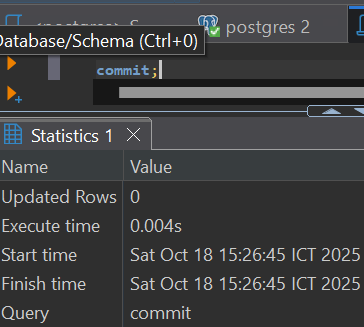
1. Dengan menggunakan CTE, buat catatan mutasi menggunakan query berikut:



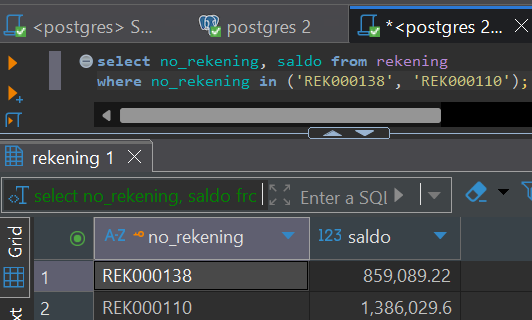
1. Cek hasil dalam transaksi dengan query berikut dan screenshot hasilnya



1. COMMIT transaction



1. Cek hasil setelah COMMIT dan screenshot hasilnya



1. **Hasil Pengamatan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Langkah** | **Operasi** | **Hasil Sebelum** | **Hasil Sesudah** | **Apa yang terjadi?** |
| 3 | Cek saldo awal | REK000138: 959.089  REK000110: 128.029 |  | Belum ada ada perubahan pada saldo kedua rekening |
| 7 | Cek dalam transaksi |  | REK000138: 859.089  REK000110: 1.386.029 | Saldo terupdate secara sementara |
| 10 | Cek mutasi |  | Debit: 859.089  Credit: 1.386.029 | Memverifikasi mutasi berhasil di insert |
| 12 | Cek setelah commit |  | REK000138: 859.089  REK000110: 1.386.029 | Seluruh perubahan saldo dan catatan mutasi disimpan secara permanen ke database |

1. **Pertanyaan Analisis**
   1. Apa yang terjadi jika berhenti di langkah 7 tanpa COMMIT?

* Transaksi pending, dan belum teerrsimpan di database
  1. Bagaimana jika terjadi error di langkah 4 atau 5?
* Transaksi akan dihentikan, perintah raise exception di langkah 4 atau error fatal di langkah 5 akan memicu rollback otomatis
  1. Sekarang eksekusi perintah ROLLBACK, Apa yang terjadi jika menjalankan ROLLBACK setelah COMMIT?
* Tidak terjadi apa apa pada data, karena perintah commit telah menyimpan perubahan secara permanen. Rollback hanya membatalkan transaksi yang belum di commit

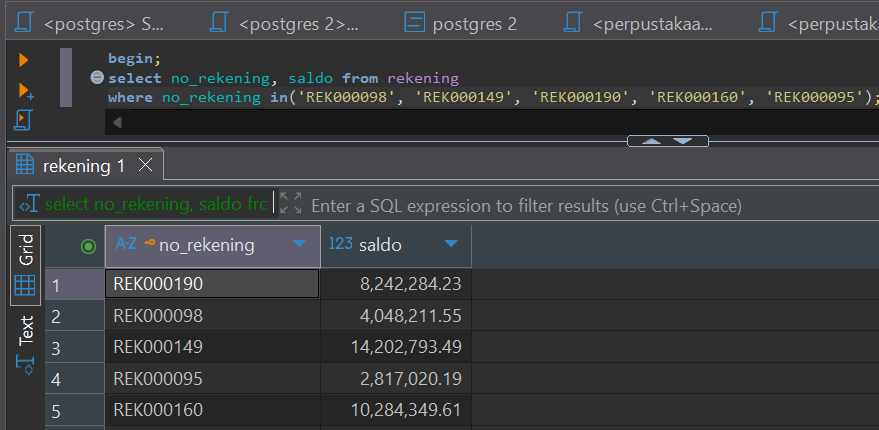
**Praktikum 02 – SAVEPOINT**

**SAVEPOINT** adalah titik penyimpanan sementara di dalam satu transaksi,  
yang memungkinkan kamu melakukan **rollback sebagian** tanpa membatalkan seluruh transaksi. Sintaks SAVEPOINT adalah sebagai berikut:

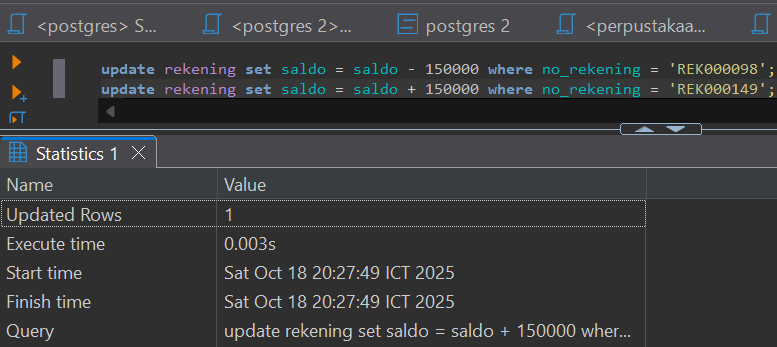
Pada praktikum ini, kita akan mempraktikkan simulasi transaction dengan SAVEPOINT:

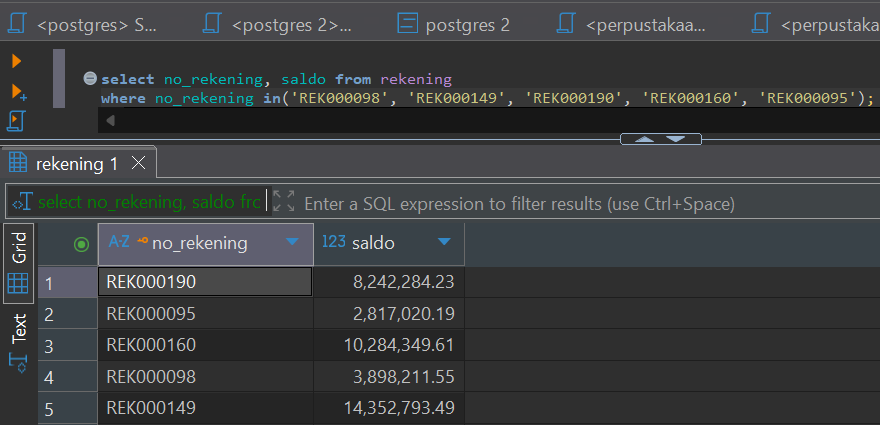
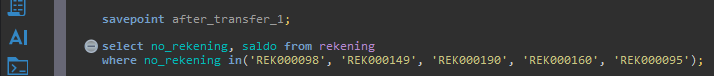
1. Kita buka SQL Editor klik kanan Schemas/public 🡪 SQL Editor 🡪 New SQL Script
2. Pastikan **menonaktifkan fitur auto-commit** pada **dBeaver** dengan cara berikut:

1. Jalankan query berikut untuk memulai transaksi dan mengecek saldo awal, screenshot hasilnya

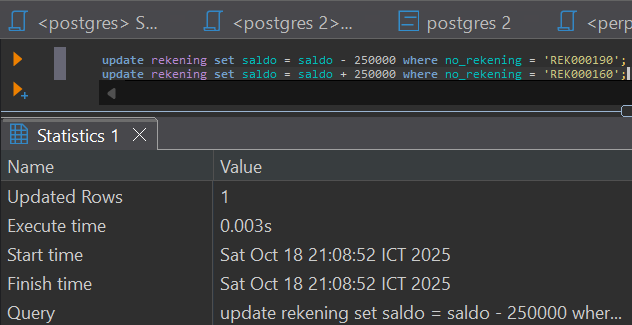


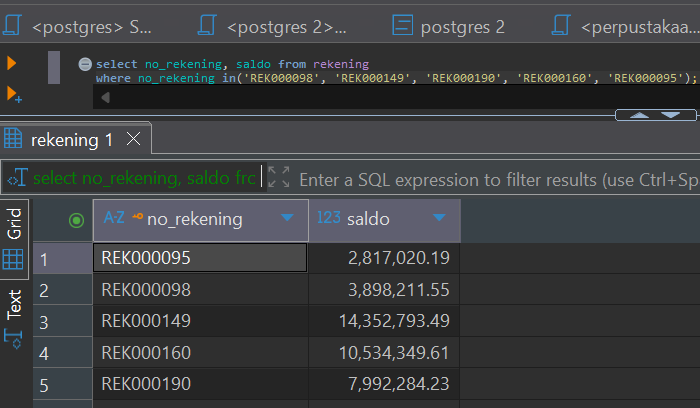
1. Jalankan query berikut untuk mengeksekusi operation 1, membuat savepoint after\_transfer\_1, dan Cek saldo ditengah transaksi savepoint after\_transfer\_1. Screenshot hasilnya



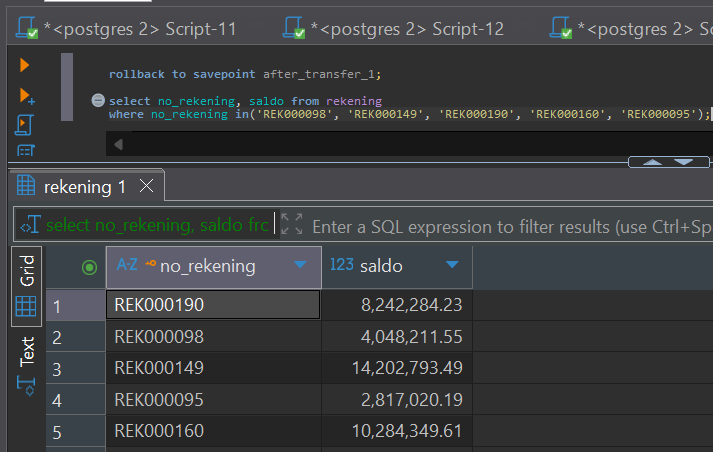


1. Jalankan query berikut untuk mengeksekusi operation 2, membuat savepoint after\_transfer\_2, dan Cek saldo ditengah transaksi savepoint after\_transfer\_2. Screenshot hasilnya

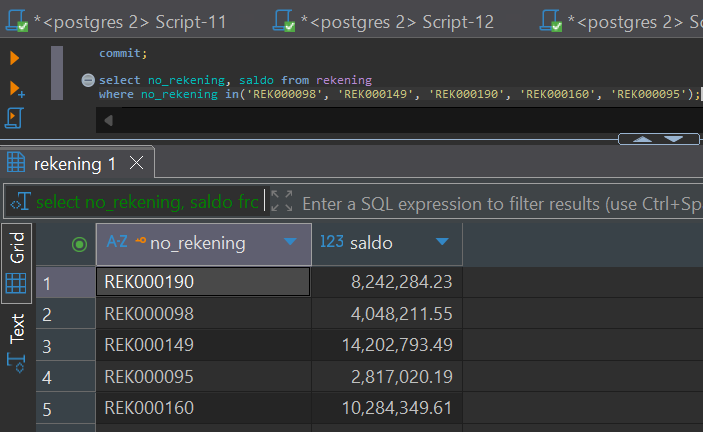




1. ROLLBACK transaksi ke SAVEPOINT after\_transfer\_1 dan cek saldo. Screenshot hasilnya



1. Lalu COMMIT transaction, dan cek kembali saldo akhir, catat perubahannya



1. **Hasil Pengamatan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Langkah** | **Operasi** | **Hasil Sebelum** | **Hasil Sesudah** | **Apa yang terjadi?** |
| 3 | Cek saldo awal | REK000098:8.242.284,23  REK000149: 14.202.793,49  REK000160: 10.284.349,61  REK000095: 2.817.020,19 |  | Tidak berubah karena hanya selct untuk menampilkan saldo awal |
| 4 | Cek setelah savepoint 1 |  | REK000098: 3.898.211,55  REK000149: 14.352.793,49  REK000160: 10.284.349,61  REK000095: 2.817.020,19 | Perubahan saldo berhasil dibuat, dan menandai savepoint pertama |
| 5 | Cek setelah operation 2 |  | REK000098: 3.898.211,55  REK000149: 14.352.793,49  REK000160: 10.534.349,61  REK000095: 2.817.020,19 | Perubahan saldo kedua berhasil dibuat |
| 6 | Cek setetelah rollback to savepoint 1 |  | REK000098: 3.898.211,55  REK000149: 14.352.793,49  REK000160: 10.284.349,61  REK000095: 2.817.020,19 | Membatalkan semua perubahan yang dibuat setelah savepoint 1,yaitu transfer kedua |
| 7 | Cek setelah commit |  | REK000098: 3.898.211,55  REK000149: 14.352.793,49  REK000160: 10.284.349,61  REK000095: 2.817.020,19 | commit menyimpan hasil akhir |

1. **Pertanyaan Analisis**
2. Bagaimana jika terjadi error di langkah 5?

* Posgre akan menandai transaksi sebagai failed, sehingga semua query berikutnya tidak akan bisa dijalankan sampai melakukan rollback, sehingga bisa rollback ke savepoint 1 di langkah 4 tanpa menghapus transaksi LANGKAH 4

1. Bagaimana jika kita menggunakan ROLLBACK tanpa menyebut savepoint?

* Seluruh langkah transaksi dari begin akan di batalkan

**Praktikum 03 – Concurrent Transaction**

Concurrency adalah kemampuan DBMS untuk menjalankan **banyak transaksi secara bersamaan (parallel).** Tujuannya memaksimalkan kinerja sistem menangani akses data oleh banyak user di saat bersamaan. Tantangannya mencegah masalah seperti: dirty read, lost update, phantom read.

1. *Dirty Read*

Transaksi A membaca data yang sedang diubah oleh Transaksi B, padahal B belum commit. Jika B rollback, maka A membaca data “kotor” (tidak valid).

|  |  |  |
| --- | --- | --- |
| **Time** | **Transaction A** | **Transaction B** |
| t1 | BEGIN | - |
| t2 | UPDATE accounts SET balance = 2000 WHERE id = 1; | SELECT balance FROM accounts WHERE id = 1  -- Output 2000 |
| t3 | ROLLBACK | - |
| t4 | Transaction A reads an uncommited value written by Transaction B | |

1. *Lost Update*

Terjadi ketika dua transaksi membaca data yang sama, lalu sama-sama mengubahnya, tetapi update terakhir menimpa hasil update pertama.

|  |  |  |
| --- | --- | --- |
| **Time** | **Transaction A** | **Transaction B** |
| t1 | BEGIN  SELECT balance FROM account WHERE id = 1;  -- Output: 1000 | BEGIN  SELECT balance FROM account WHERE id = 1;  -- Output: 1000 |
| t2 | UPDATE accounts SET balance = balance + 200 WHERE id= 1;  -- Output yang diharapkan 1200 | - |
| t3 | - | UPDATE accounts SET balance = balance-300 WHERE id= 1;  COMMIT;  -- Output 700 |
| t4 | SELECT balance FROM account WHERE id = 1;  -- Output: 700 |  |

1. Non repeatable read

Terjadi ketika dalam satu transaksi, baris data yang sama dibaca dua kali, tetapi hasilnya berbeda karena ada transaksi lain yang melakukan UPDATE atau DELETE dan COMMIT di antara dua pembacaan tersebut.

|  |  |  |
| --- | --- | --- |
| **Time** | **Transaction A** | **Transaction B** |
| t1 | BEGIN  SELECT balance FROM account WHERE id = 1;  -- Output: 1000 |  |
| t2 |  | UPDATE accounts SET balance = balance-300 WHERE id= 1;  COMMIT;  -- Output 700 |
| t3 | SELECT balance FROM account WHERE id = 1;  -- Output: 700 |  |

1. Phantom read

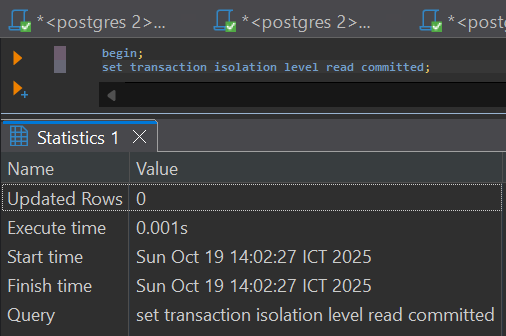
Terjadi ketika transaksi membaca **sekumpulan baris dengan kondisi tertentu**, lalu transaksi lain **menambah/menghapus** baris sehingga jumlah hasil query berubah saat dibaca ulang.

|  |  |  |
| --- | --- | --- |
| **Time** | **Transaction A** | **Transaction B** |
| t1 | BEGIN  SELECT \* FROM orders WHERE total > 500;  -- Output: Adi (600), Budi (800) | - |
| t2 | - | BEGIN  INSERT INTO orders VALUES (‘Chika’, 700);  COMMIT; |
| t3 | BEGIN  SELECT \* FROM orders WHERE total > 500;  -- Output: Adi (600), Budi (800), Chika (700) | - |

Langkah praktikum Isolation Level:

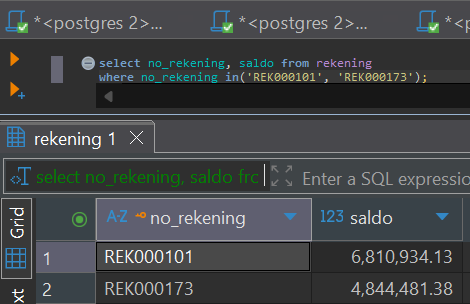
1. Buka 2 window SQL Editor
2. Pastikan **menonaktifkan fitur auto-commit** di kedua window dengan cara berikut:

1. Pada window 1, eksekusi query berikut untuk memulai transaksi dan set isolation level READ COMMITED, tetapi jangan COMMIT dulu

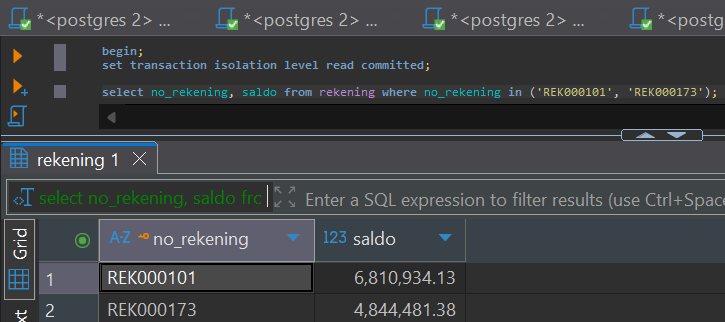


1. Pada window 1, eksekusi query untuk menampilkan saldo awal seperti berikut, screenshot hasilnya

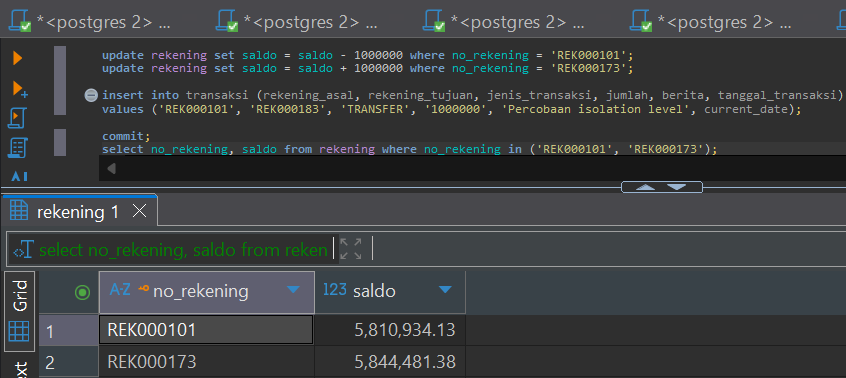
Transaksi pada window 1 **jangan** di-commit



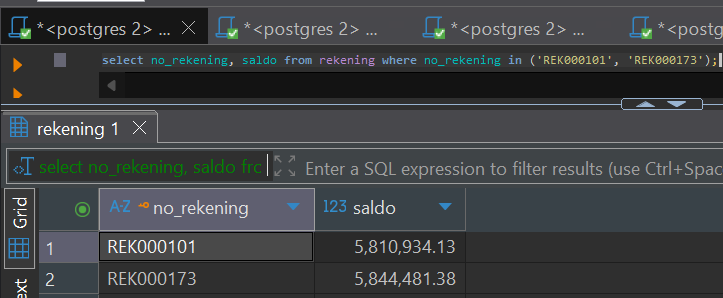
1. Pada window 2, eksekusi query berikut untuk memulai transaksi, set isolation level READ COMMITED, dan melihat saldo awal. screenshot hasilnya



1. Pada window 2, eksekusi berikut untuk update saldo dan insert mutasi. Screenshot hasilnya



1. Kembali ke window 1, cek ulang saldo dan catat hasilnya



1. **Hasil Pengamatan**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Langkah** | **Operasi** | **Hasil Sebelum** | **Hasil Sesudah** | **Apa yang terjadi?** |
| 4 | Cek saldo awal window 1 | REK000101: 6.810.934,13  REK000173: 4.844.481,38 |  | Membaca saldo awal |
| 5 | Cek saldo awal window 2 |  | REK000101: 6.810.934,13  REK000173: 4.844.481,38 | Membaca saldo awal |
| 6 | Cek saldo setelah commit window 1 |  | REK000101: 5.810.934,13  REK000173: 5.844.481,38 | Windo 2 melakukan update saldo dan commit |
| 7 | Cek saldo akhir window 1 |  | REK000101: 5.810.934,13  REK000173: 5.844.481.38 | Membaca data baru yang sudah di commit window 2 |

1. **Pertanyaan Analisis**
2. Bagaimana saldo awal dan akhir yang dilihat oleh transaksi pada window 1?

* Window 1 melihat data yang sudah di commit terbaru, saat pengecekan awal data yang terlihat adalah data awal yang belum dirubah, lalu saat pengecekan terakhir, maka menampilkan data yang sudah dirubah sebelumnya oleh window 2

1. Apa yang dapat kamu simpulkan dari isolation level READ COMMITED?

* Pada level read committed, setiap perintah select di dalam satu transaksi selalu membaca data terakhir yang sudah di commit oleh transaksi lain

**Praktikum 04 – Trigger**

Trigger adalah prosedur otomatis yang dijalankan oleh database saat event tertentu terjadi pada tabel atau view. Trigger di PostgreSQL biasanya terdiri dari:

* Event

kapan trigger berjalan (INSERT, UPDATE, DELETE, TRUNCATE).

* Timing

sebelum atau sesudah event terjadi (BEFORE, AFTER, INSTEAD OF).

Table/View

objek yang dipantau trigger.

* Function

kode (biasanya PL/pgSQL) yang dieksekusi.

Jenis trigger bervariasi bergantung pada waktu eksekusinya. Berikut ini adalah jenis-jenis trigger:

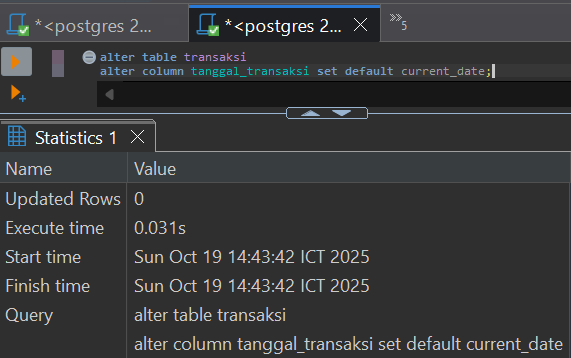
|  |  |
| --- | --- |
| **Jenis** | **Keterangan** |
| BEFORE | Dijalankan sebelum operasi terjadi, bisa validasi/modifikasi data |
| AFTER | Dijalankan setelah operasi terjadi, cocok untuk logging |
| INSTEAD OF | Digunakan pada view, mengganti aksi default |
| Row-level trigger | Aktif untuk setiap baris yang terpengaruh |
| Statement-level trigger | Aktif sekali untuk seluruh query, tidak peduli berapa banyak baris yang terpengaruhi |

Trigger juga memiliki aturan atau constrain. Berikut ini adalah constraint trigger:

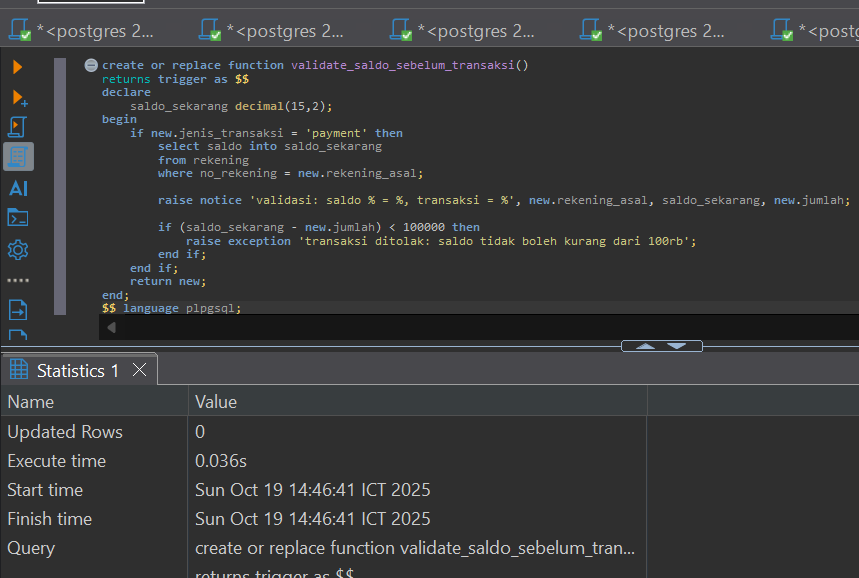
|  |  |
| --- | --- |
| **Constraint** | **Keterangan** |
| FOR EACH ROW | Trigger jalan per baris yang berubah |
| FOR EACH STATEMENT | Trigger jalan sekali per query |
| RETURN NEW | Data baru (INSERT/UPDATE) |
| RETURN OLD | Data lama (UPDATE/DELETE) |

Langkah praktikum trigger

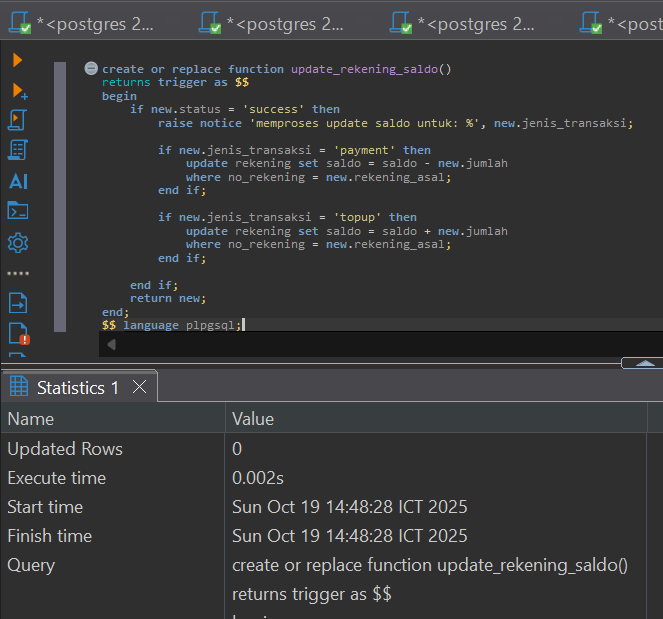
1. Buka SQL Editor
2. Eksekusi query berikut ini menambahkan constraint default pada tabel transaksi



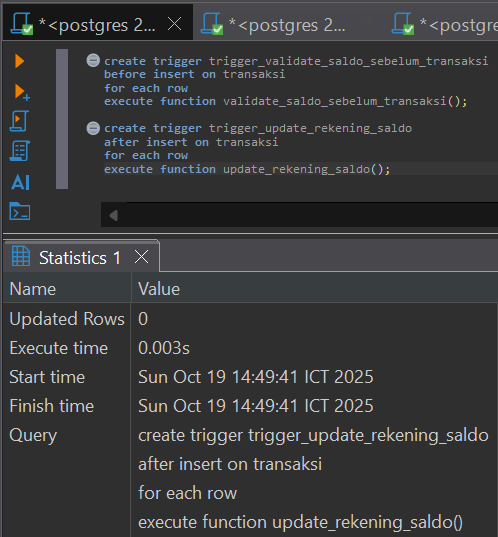
1. Buat function untuk validasi saldo sebelum transaksi



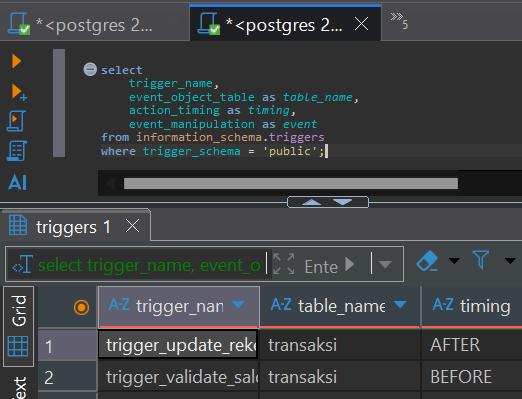
1. Buat Function untuk update saldo rekening



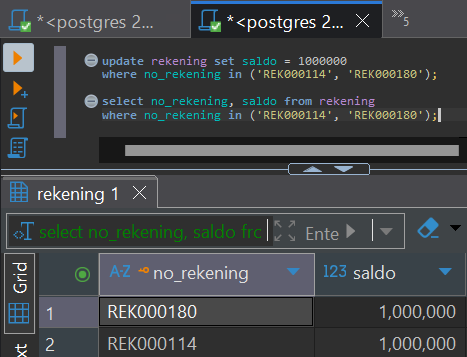
1. Buat trigger menggunakan query berikut



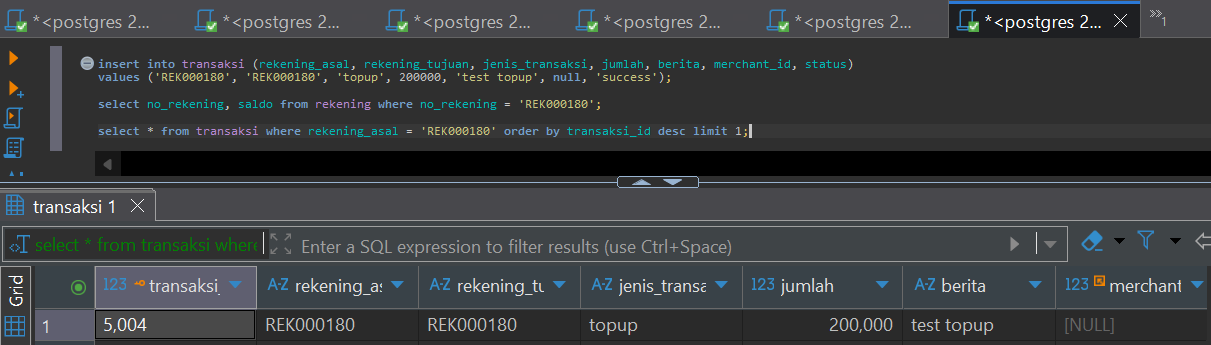
1. Pastikan trigger aktif dengan menjalankan query berikut



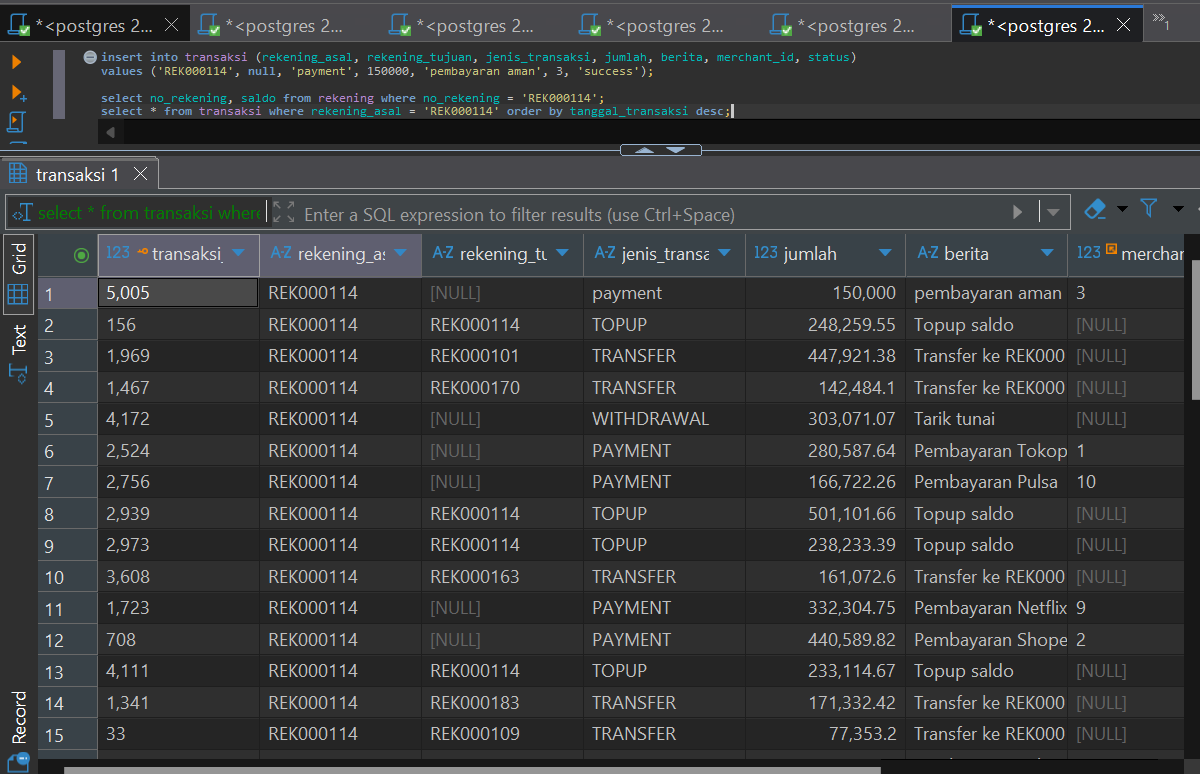
1. Eksekusi query berikut untuk mengatur saldo awal dan menampilkannya. Screenshot hasilnya



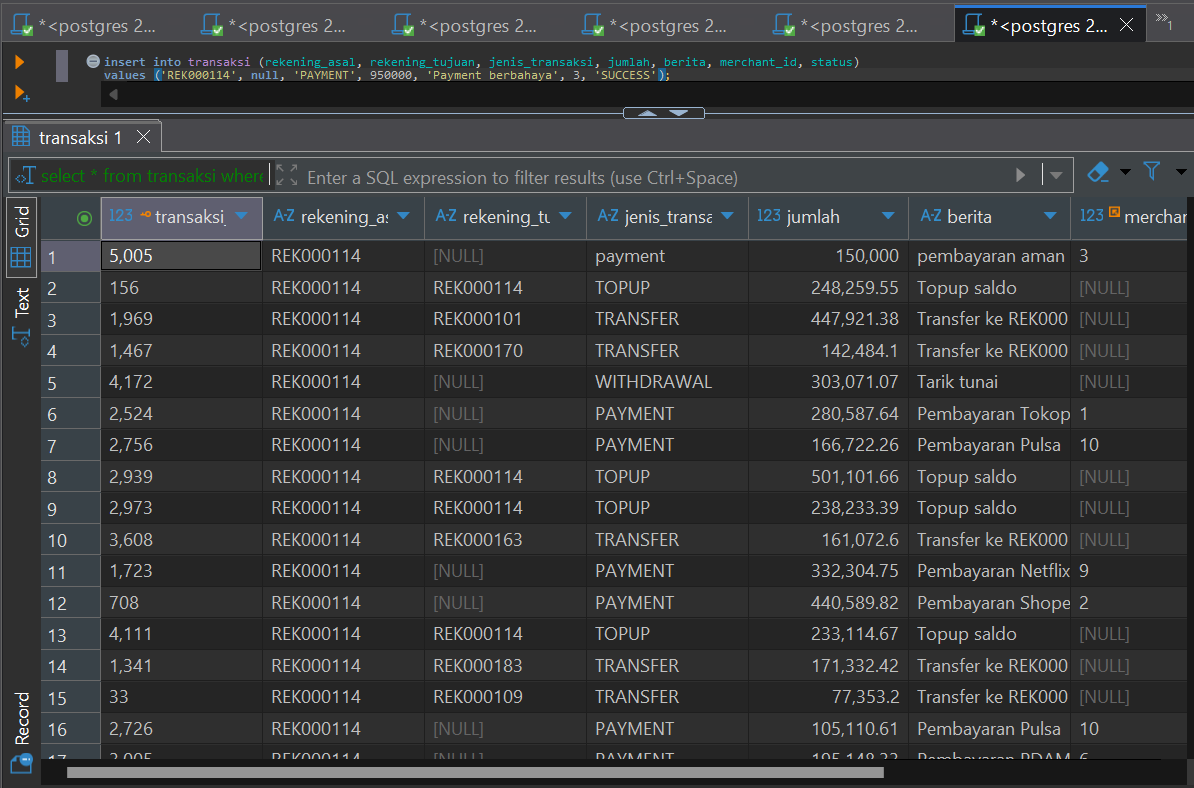
1. Eksekusi query berikut untuk simulasi topup saldo yang memenuhi kentutan (transaksi sukses), screenshot hasilnya

****

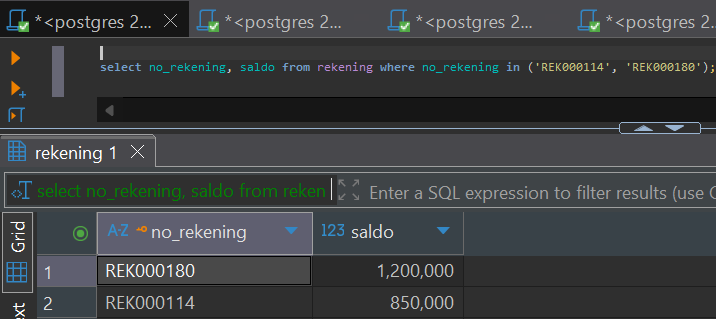
1. Eksekusi query berikut untuk simulasi payment saldo yang memenuhi kentutan (transaksi sukses), screenshot hasilnya

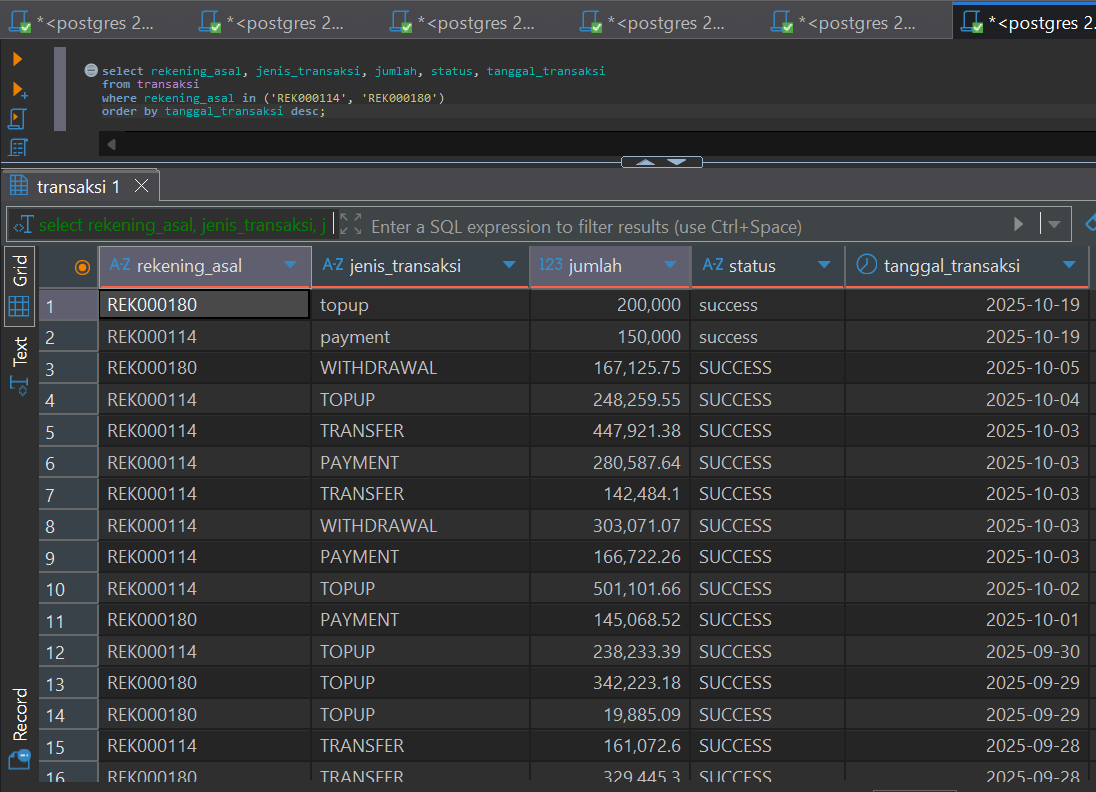
****

1. Eksekusi query berikut untuk simulasi payment saldo yang tidak memenuhi kentutan (transaksi gagal), screenshot hasilnya

****

1. Cek saldo akhir dan catatan transaksi terbaru dengan menjalankan query berikut





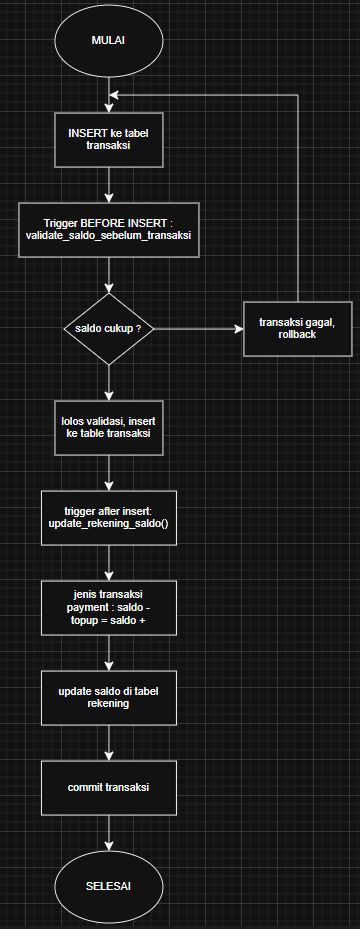
1. **Hasil Pengamatan**

| **Langkah** | **Operasi** | **Hasil Sebelum** | **Hasil Sesudah** | **Apa yang terjadi?** |
| --- | --- | --- | --- | --- |
| 6 | Cek trigger aktif |  |  | Setiap transaksi otomatis divalidasi dan saldo akan terupdate |
| 7 | Set saldo awal | REK000114: -  REK000180: - | REK000114: 1.000.000  REK000180: 1.000.000 | Saldo di set ke 1 juta pada rekening yang ditentukan |
| 8 | Cek saldo topup berhasil |  | REK000114: 1.000.000  REK000180: 1.200.000 | Saat top up sebesar 200.000, status success trigger after inser menambah saldo rekening asal, |
| Cek transaksi topup berhasil |  |  | transaksi juga tercatat di tabel transaksi |
| 9 | Cek saldo payment berhasil |  | REK000114: 850.000  REK000180: 1.200.000 | Transaksi payment sebesar 150.000 berhasil karena saldo >100.000. trigger after insert otomatis mengurangi saldo |
| Cek transaksi payment berhasil |  |  | Transaksi juga tercatat di tabel transaksi |
| 10 | Payment gagal |  |  | transaksi payment sebesar 950.000 ditolak oleh trigger validate\_saldo\_sebelum\_transaksi, dikarenakan saldo akhir akan <100.000, sehingga muncul erorr dan transaksi dibatalkan |
| 11 | Cek saldo akhir |  | REK000114: 850.000  REK000180: 1.200.000 | Hasil akhir menunjukkan top up dan payment pertama berhasil, payment kedua gagal |
| Cek transaksi akhir |  |  | Transaksi juga tercatat di tabel transaksi |

1. **Pertanyaan Analisis**
2. Apa perbedaan fundamental antara trigger BEFORE dan AFTER dalam konteks praktikum ini?

* Mengapa validasi saldo menggunakan BEFORE INSERT?
* Triger before cocok untuk validasi sebelum data masuk ke sistem
* Mengapa update saldo menggunakan AFTER INSERT?
* Trigger after cocok untuk aksi pasca transaksi, seperti update saldo

1. Buat flowchart yang menunjukkan execution flow dari insert transaksi hingga update saldo, termasuk semua trigger yang terlibat!



*\*\*\* Sekian, dan selamat belajar \*\*\**