

IF3140 MANAJEMEN BASIS DATA
MEKANISME CONCURRENCY CONTROL DAN RECOVERY



K01 Kelompok 11

Anggota :

Aulia Mey Diva Annandya	13521103
Jericho Russel Sebastian	13521107
Marcel Ryan Antony	13521127
Edia Zaki Naufal Ilman	13521141

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

Daftar Isi	1
1. Eksplorasi Transaction Isolation	2
a. Serializable	2
b. Repeatable Read	2
c. Read Committed	2
d. Read Uncommitted	2
2. Implementasi Concurrency Control Protocol	3
a. Two-Phase Locking (2PL)	3
b. Optimistic Concurrency Control (OCC)	3
c. Multiversion Timestamp Ordering Concurrency Control (MVCC)	3
3. Eksplorasi Recovery	4
a. Write-Ahead Log	4
b. Continuous Archiving	4
c. Point-in-Time Recovery	4
d. Simulasi Kegagalan pada PostgreSQL	4
4. Pembagian Kerja	5
Referensi	6

1. Eksplorasi Transaction Isolation

Level isolasi transaksi pada standar SQL dibagi menjadi empat, yaitu *serializable*, *repeatable read*, *read committed*, dan *read uncommitted*, terurut dari level isolasi transaksi yang paling ketat. Level-level tersebut didefinisikan oleh beberapa fenomena yang tidak boleh terjadi pada level-level isolasi transaksi tersebut. Fenomena-fenomena yang tidak diperbolehkan terjadi pada level-level tertentu tersebut antara lain :

1. *Dirty read*

Dirty read merupakan suatu fenomena ketika sebuah transaksi membaca data yang belum di-*commit*.

2. *Non-repeatable read*

Non-repeatable read merupakan suatu fenomena ketika sebuah transaksi membaca ulang data yang telah dibaca transaksi tersebut dan data tersebut telah dimodifikasi oleh transaksi lainnya yang telah di-*commit*.

3. *Phantom read*

Phantom read merupakan suatu fenomena ketika sebuah transaksi mengeksekusi ulang *query* yang mengembalikan baris-baris yang memenuhi kondisi tertentu, namun ketika dieksekusi ulang baris-baris yang dikembalikan berbeda dengan baris-baris sebelumnya karena transaksi lain yang melakukan *commit*.

4. *Serialization anomaly*

Serialization anomaly merupakan suatu fenomena ketika hasil dari sebuah grup transaksi yang berjalan secara konkuren tidak dapat dicapai apabila transaksi-transaksi tersebut dijalankan secara sekuensial dalam urutan apapun tanpa *overlap*.

Berikut tabel untuk memperlihatkan fenomena apa saja yang dapat terjadi pada level-level isolasi transaksi.

Tabel 1.1 Level Isolasi Transaksi

Level Isolasi	<i>Dirty read</i>	<i>Non-repeatable read</i>	<i>Phantom read</i>	<i>Serialization anomaly</i>
Serializable	Tidak dapat terjadi	Tidak dapat terjadi	Tidak dapat terjadi	Tidak dapat terjadi
Repeatable read	Tidak dapat terjadi	Tidak dapat terjadi	Dapat terjadi, namun tidak di PostgreSQL	Dapat terjadi
Read committed	Tidak dapat terjadi	Dapat terjadi	Dapat terjadi	Dapat terjadi
Read Uncommitted	Dapat terjadi, namun tidak di PostgreSQL	Dapat terjadi	Dapat terjadi	Dapat terjadi

a. Serializable

Level isolasi transaksi *serializable* merupakan level tertinggi dari isolasi transaksi dimana pada transaksi yang ada di level ini, transaksi seolah-olah dijalankan secara berurutan dan tidak bersamaan dengan transaksi lainnya. Pada level *serializable*, tidak dapat terjadi fenomena-fenomena yang telah disebutkan diatas, yaitu *dirty read*, *non-repeatable read*, *phantom read*, dan *serialization anomaly*. Karena pada level transaksi ini sangat *strict* maka performa yang diberikan akan berkurang karena banyaknya *overhead*.

Fenomena *dirty read*, tidak dapat terjadi karena data yang dibaca hanya data yang sudah dilakukan *commit*. Fenomena *non-repeatable read* juga tidak akan terjadi karena tidak akan ada transaksi yang mengubah data ketika sebuah transaksi sedang menggunakan data. Fenomena *phantom read* juga tidak dapat terjadi karena alasan yang sama yaitu ketika sebuah transaksi sedang berjalan, tidak ada transaksi yang dapat mengubah data sehingga *query* akan menghasilkan hasil yang sama. Terakhir, fenomena *serialization anomaly* juga tidak dapat terjadi karena sistem terus dipantau agar tidak ada fenomena yang dapat terjadi.

b. Repeatable Read

Level isolasi transaksi *repeatable read* akan memastikan bahwa sebuah transaksi akan membaca data yang sama, bahkan ketika transaksi lain melakukan *commit* untuk mengubah data. Pada level *repeatable read*, tidak dapat terjadi fenomena *dirty read* dan *non-repeatable read*, namun fenomena *phantom read* dan *serialization anomaly* masih dapat terjadi pada level isolasi ini. Fenomena *dirty read* dan *non-repeatable read* tidak dapat terjadi karena ketika sebuah transaksi membaca sebuah data, transaksi tersebut akan menerima *read lock* terhadap data tersebut dan transaksi lain tidak akan mendapatkan *write lock* pada data yang sama sampai transaksi yang sedang membaca data melepas *read lock*.

c. Read Committed

Level isolasi transaksi *read committed*, sebuah transaksi hanya dapat membaca data yang sudah di-*commit*, namun pada level ini tidak diberikan *read lock* ketika sebuah data dibaca. Level ini memberikan tingkat konkurensi yang lebih tinggi daripada level *serializable* dan *repeatable read*. Pada level ini, fenomena yang tidak dapat terjadi hanya fenomena *dirty read*. Karena pada level ini hanya data yang telah di-*commit* yang diperbolehkan untuk dibaca sehingga *dirty read* tidak dapat terjadi.

d. Read Uncommitted

Level isolasi transaksi *read uncommitted* dapat terjadi seluruh fenomena yang ada yaitu, *dirty read*, *non-repeatable read*, *phantom read*, dan *serialization anomaly*. Level ini biasanya digunakan ketika memprioritaskan konkurensi.

e. Simulasi Serializable, Repeatable Read, dan Read Committed

Untuk paham lebih dalam tentang derajat isolasi atau level isolasi yang ada, maka akan dilakukan simulasi terhadap setiap derajat isolasi pada PostgreSQL, kecuali pada level *read uncommitted* karena level isolasi *read uncommitted* tidak terdapat pada PostgreSQL. Simulasi ini akan menunjukkan perbedaan-perbedaan yang terdapat pada tiap derajat isolasi. Simulasi ini akan dilakukan pada dua terminal, dimana satu terminal untuk melakukan pengecekan isolasi dan satu terminal untuk menjalankan *query*.

1. Serializeable

Transaksi pada kedua terminal akan dimulai dengan melakukan inisiasi level isolasi transaksi ke *serializable* dengan *query* SET TRANSACTION ISOLATION LEVEL SERIALIZABLE. Transaksi pada kedua terminal menggunakan *database* pagila yang biasa digunakan saat praktikum.

```
pagila=# BEGIN;  
BEGIN  
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SET
```

Gambar 1.1.1 Awal transaksi

Kemudian, pada terminal pertama akan ditambahkan data pada kolom actor dengan *query* INSERT INTO actor(first_name, last_name) VALUES('JEFRI', 'NICHOL'). Kemudian, akan ditampilkan bahwa operasi INSERT telah berhasil dengan melakukan operasi SELECT.

```
pagila=# BEGIN;  
BEGIN  
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SET  
pagila=*# INSERT INTO actor(first_name, last_name) VALUES('JEFRI', 'NICHOL');  
INSERT 0 1  
pagila=*# SELECT FROM actor WHERE actor_id = 201;  
--  
(1 row)  
  
pagila=*# SELECT * FROM actor WHERE actor_id = 201;  
 actor_id | first_name | last_name | last_update  
-----+-----+-----+-----  
      201 | JEFRI      | NICHOL   | 2023-11-25 19:14:07.499971  
(1 row)
```

Gambar 1.1.2 Terminal 1: INSERT data pada tabel actor

Pada terminal kedua, akan dicoba untuk melakukan operasi INSERT untuk melihat apa yang terjadi pada terminal 2.

```
pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
pagila=*# INSERT INTO actor(actor_id, first_name, last_name) VALUES (201, 'JEFRI', 'NICHOL');
```

Gambar 1.1.3 Terminal 2 : INSERT data pada tabel actor

Dapat dilihat pada gambar diatas, operasi INSERT akan menunggu sampai transaksi pada terminal 1 melakukan *commit* sebelum melakukan operasi pada terminal 2.

```
pagila=*# COMMIT;
COMMIT
```

Gambar 1.1.4 Terminal 1 : COMMIT transaksi

Pada terminal pertama, transaksi di-*commit*.

```
pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
pagila=*# INSERT INTO actor(actor_id, first_name, last_name) VALUES (201, 'JEFRI', 'NICHOL');
ERROR: duplicate key value violates unique constraint "actor_pkey"
DETAIL: Key (actor_id)=(201) already exists.
pagila=!# COMMIT;
ROLLBACK
pagila=#
```

Gambar 1.1.5 Terminal 2: INSERT data pada tabel actor gagal

Pada terminal kedua, operasi INSERT langsung dijalankan ketika transaksi pada terminal pertama sudah melakukan *commit*, namun operasi INSERT gagal dikarenakan *value* yang ditambahkan ke tabel actor merupakan duplikat dari *value* yang ditambahkan pada terminal pertama. Sehingga transaksi akan melakukan *rollback* ketika transaksi dicoba untuk di-*commit*.

Hal ini tidak hanya berlaku dengan operasi INSERT namun juga operasi lainnya yaitu SELECT, UPDATE, dan DELETE karena transaksi pada terminal kedua menunggu transaksi pada terminal pertama untuk melakukan *commit*. Berikut contoh lain apabila terminal pertama melakukan update pada suatu data di tabel actor dan terminal kedua ingin melakukan update terhadap data yang sama di tabel actor.

```

pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
pagila=*# UPDATE actor SET first_name = 'DAVE' WHERE actor_id = 201;
UPDATE 1
pagila=*# UPDATE actor SET first_name = 'MICH' WHERE actor_id = 201;
UPDATE 1
pagila=*# |

```

Gambar 1.1.6 Terminal 1 : UPDATE data actor dengan id 201

```

pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
pagila=*# UPDATE actor SET first_name = 'DAVE' WHERE actor_id = 201;
|

```

Gambar 1.1.7 Terminal 2 : UPDATE data actor dengan id 201

```

pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
pagila=*# UPDATE actor SET first_name = 'DAVE' WHERE actor_id = 201;
ERROR:  could not serialize access due to concurrent update
pagila=!# COMMIT;
ROLLBACK
pagila=# |

```

Gambar 1.1.8 Terminal 2 : Setelah transaksi terminal pertama *commit*

Dapat dilihat bahwa transaksi pada terminal kedua akan menunggu transaksi pertama untuk melakukan *commit* karena kedua transaksi melakukan operasi UPDATE terhadap data yang sama pada tabel actor. Namun, operasi UPDATE pada transaksi di terminal kedua gagal karena transaksi pada terminal pertama dan kedua mencoba untuk melakukan operasi UPDATE pada data yang sama secara konkuren sehingga transaksi di terminal kedua akan melakukan *rollback*.

Dari simulasi diatas, dapat dilihat bahwa level isolasi transaksi *serializable* tidak memperbolehkan keempat fenomena *dirty read*, *non-repeatable read*, *phantom read*, dan *serialization anomaly*. Berikut tabel yang akan menggambarkan kedua transaksi

tersebut dimana transaksi pada terminal pertama diberi nama T1, dan transaksi pada terminal kedua diberi nama T2.

Tabel 1.1.1 Simulasi *Serializable*

T1	T2	Keterangan
BEGIN;		Memulai transaksi T1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;		Memilih level isolasi untuk T1
	BEGIN;	Memulai transaksi T2
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Memilih level isolasi untuk T2
UPDATE actor SET first_name = 'MICH' WHERE actor_id = 201;		Melakukan UPDATE pada transaksi T1 pada data dengan actor_id 201
	UPDATE actor SET first_name = 'DAVE' WHERE actor_id = 201;	Melakukan UPDATE pada transaksi T2 pada data dengan actor_id 201
COMMIT;		T1 melakukan <i>commit</i>
	ROLLBACK;	T2 melakukan <i>rollback</i> karena terjadi <i>error</i> pada transaksinya.

2. Repeatable read

Sama seperti pada simulasi sebelumnya, kedua transaksi dimulai dengan perintah BEGIN dan level isolasi transaksi akan diinisiasi dengan *query* SET TRANSACTION ISOLATION LEVEL REPEATABLE READ.

```

pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET

```

Gambar 1.2.1 Awal transaksi

```

pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
pagila=*# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |      last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
(1 row)

pagila=*# |

```

Gambar 1.2.2 Terminal 2 : SELECT actor

Pada terminal kedua akan dilakukan perintah SELECT untuk melihat data actor baru yang telah dibuat. Hasil SELECT menunjukkan baru ada 1 data yang ditambahkan pada actor.

```

pagila=# BEGIN;
BEGIN
pagila=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
pagila=# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |      last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
(1 row)

pagila=# INSERT INTO actor(actor_id, first_name, last_name) VALUES (202, 'MIKHA', 'ANGELO');
INSERT 0 1
pagila=# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |      last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
(2 rows)

pagila=# |

```

Gambar 1.2.3 Terminal 1 : INSERT data actor baru

Pada terminal pertama akan ditambahkan data actor baru dengan menggunakan operasi INSERT dan kemudian akan dilakukan operasi SELECT untuk memastikan data berhasil ditambah.

```

pagila=# BEGIN;
BEGIN
pagila=*# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
pagila=*# SELECT * FROM actor WHERE actor_id > 200;
  actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
(1 row)

pagila=*# SELECT * FROM actor WHERE actor_id > 200;
  actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
(1 row)

pagila=*# |

```

Gambar 1.2.4 Terminal 2 : SELECT data actor

Pada terminal kedua, akan dilakukan operasi SELECT ulang untuk melihat apakah terdapat baris baru yang telah ditambahkan pada terminal pertama. Hasil operasi SELECT ulang menunjukkan hanya terdapat data actor_id 201 dan data actor_id yang baru dibuat tidak terbaca.

```
pagila=*# COMMIT;
COMMIT
```

Gambar 1.2.5 Terminal 1 : COMMIT

```
pagila=*# SELECT * FROM actor WHERE actor_id > 200;
actor_id | first_name | last_name | last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
(1 row)

pagila=*# |
```

Gambar 1.2.6 Terminal 2 : SELECT setelah terminal 1 *commit*

Dapat dilihat meskipun terminal pertama telah melakukan *commit*, namun operasi SELECT pada terminal kedua masih menghasilkan baris yang sama dengan baris sebelumnya sehingga tidak terjadi fenomena *non-repeatable read*.

```
pagila=*# INSERT INTO actor(actor_id, first_name, last_name) VALUES (202, 'MIKHA', 'ANGELO');
ERROR: duplicate key value violates unique constraint "actor_pkey"
DETAIL: Key (actor_id)=(202) already exists.
pagila=!# COMMIT;
ROLLBACK
pagila=# |
```

Gambar 1.2.7 Terminal 2 : INSERT data yang sama dengan terminal 1

Dapat dilihat meskipun data yang ditambahkan pada terminal pertama belum terbaca pada operasi SELECT di terminal 2, apabila pada terminal 2 dilakukan operasi INSERT menggunakan data yang sama pada terminal 1 akan terjadi *error* dan transaksi akan dilakukan *rollback*.

Pada simulasi diatas dapat dilihat bahwa level isolasi transaksi *repeatable read* tidak dapat terjadi fenomena *dirty read*, *non-repeatable read*, dan *phantom read*, namun *serialization anomaly* dapat terjadi. Berikut tabel yang akan menggambarkan kedua transaksi tersebut dimana transaksi pada terminal pertama diberi nama T1, dan transaksi pada terminal kedua diberi nama T2.

Tabel 1.2.1 Simulasi *repeatable read*

T1	T2	Keterangan
----	----	------------

BEGIN;		Memulai transaksi T1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;		Memilih level isolasi untuk T1
	BEGIN;	Memulai transaksi T2
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Memilih level isolasi untuk T2
SELECT * FROM actor WHERE actor_id > 200;		Membaca <i>initial</i> data actor yang sudah ada pada T1
	SELECT * FROM actor WHERE actor_id > 200;	Membaca <i>initial</i> data actor yang sudah ada pada T1
INSERT INTO actor(actor_id, first_name, last_name) VALUES (202, 'MIKHA', 'ANGELO');		Menambahkan data actor baru pada T1
SELECT * FROM actor WHERE actor_id > 200;		Memastikan data actor baru dengan membaca data actor pada T1
	SELECT * FROM actor WHERE actor_id > 200;	Membaca data actor pada T2, hasil tetap tidak berubah dari <i>initial</i> data
COMMIT;		T1 melakukan <i>commit</i>
	SELECT * FROM actor WHERE actor_id > 200;	Membaca data actor pada T2, hasil tetap tidak berubah dari <i>initial</i> data
	INSERT INTO actor(actor_id, first_name, last_name) VALUES (202, 'MIKHA', 'ANGELO');	Menambahkan data actor yang sama pada T2
	ROLLBACK;	T2 <i>rollback</i> karena data sudah terdapat di actor sehingga terjadi <i>error</i> .

3. Read committed

Masih sama seperti simulasi-simulasi sebelumnya, kedua transaksi akan dimulai dengan *query* BEGIN dan kemudian kedua transaksi akan diinisiasi dengan level isolasi *read committed* dengan *query* SET TRANSACTION ISOLATION LEVEL READ COMMITTED.

```
pagila=# BEGIN;
BEGIN
pagila=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
pagila=# |
```

Gambar 1.3.1 Awal transaksi

```
pagila=# BEGIN;
BEGIN
pagila=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
pagila=# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
(2 rows)

pagila=# |
```

Gambar 1.3.2 Terminal 2 : SELECT actor

Pada terminal 2 dilakukan operasi SELECT untuk melihat *initial* data yang terdapat pada actor sebelum adanya operasi lain.

```

pagila=# BEGIN;
BEGIN
pagila=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
pagila=# INSERT INTO actor(actor_id, first_name, last_name) VALUES (203, 'MICHELLE', 'OBAMA');
INSERT 0 1
pagila=# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
      203 | MICHELLE  | OBAMA    | 2023-11-25 22:02:11.656922
(3 rows)

```

Gambar 1.3.3 Terminal 1 : INSERT data actor baru

Pada terminal 1 dilakukan operasi INSERT untuk menambahkan data actor baru dan kemudian dilakukan validasi apakah data baru sudah terdapat di tabel dengan melakukan operasi SELECT.

```

pagila=# BEGIN;
BEGIN
pagila=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
pagila=# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
(2 rows)

pagila=# SELECT * FROM actor WHERE actor_id > 200;
 actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
(2 rows)

```

Gambar 1.3.4 Terminal 2 : SELECT setelah terminal 1 INSERT

Dapat dilihat dari gambar diatas, meskipun terminal 1 melakukan INSERT data actor baru, operasi SELECT pada terminal 2 masih mengembalikan baris-baris yang sama dengan operasi SELECT sebelumnya sehingga fenomena *dirty read* tidak terjadi pada level isolasi transaksi ini.


```
pagila=# COMMIT;
COMMIT
```

Gambar 1.3.5 Terminal 1 : COMMIT

```
pagila=# SELECT * FROM actor WHERE actor_id > 200;
actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
      203 | MICHELLE  | OBAMA    | 2023-11-25 22:02:11.656922
(3 rows)
```

Gambar 1.3.6 Terminal 2 : SELECT setelah terminal 1 COMMIT

Kemudian, dapat dilihat pada gambar diatas bahwa setelah transaksi pada terminal 1 melakukan COMMIT, operasi SELECT langsung mengembalikan baris-baris yang berbeda dari sebelumnya yang menandakan bahwa fenomena *phantom read* dapat terjadi pada level isolasi transaksi ini.

```
pagila=# BEGIN;
BEGIN
pagila=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
pagila=# UPDATE actor SET first_name = 'BARRACK' WHERE actor_id = 203;
UPDATE 1
pagila=# COMMIT;
COMMIT
pagila=#
```

Gambar 1.3.7 Terminal 1 : UPDATE actor

```
pagila=# SELECT * FROM actor WHERE actor_id > 200;
actor_id | first_name | last_name |          last_update
-----+-----+-----+-----
      201 | MICH      | NICHOL   | 2023-11-25 19:50:12.642171
      202 | MIKHA     | ANGELO   | 2023-11-25 20:41:46.178482
      203 | BARRACK   | OBAMA    | 2023-11-25 22:11:44.892416
(3 rows)
```

Gambar 1.3.8 Terminal 2 : SELECT setelah terminal 1 UPDATE dan COMMIT

Kemudian juga, dapat dilihat pada gambar diatas bahwa setelah transaksi pada terminal 1 melakukan operasi UPDATE dan COMMIT, operasi SELECT langsung

mengembalikan baris yang berbeda dengan sebelumnya tepatnya pada baris dengan actor_id 203 yang sebelumnya first_name nya MICHELLE menjadi BARRACK. Hal ini menandakan fenomena *non-repeatable read* dapat terjadi pada level isolasi ini.

```
pagila=# INSERT INTO actor(actor_id, first_name, last_name) VALUES (203, 'MICHELLE', 'OBAMA');
ERROR:  duplicate key value violates unique constraint "actor_pkey"
DETAIL:  Key (actor_id)=(203) already exists.
pagila=# COMMIT;
ROLLBACK
```

Gambar 1.3.9 Terminal 2 : INSERT data actor baru dengan id sama pada terminal 1
Ketika terminal 2 mencoba untuk melakukan operasi INSERT dengan actor_id 203 yang sama dengan data yang di-insert dengan transaksi pada terminal 1, operasi INSERT pada terminal 2 mengalami *error* karena actor dengan id 203 sudah ada pada *database* sehingga transaksi pada terminal 2 akan *rollback*.

Dari simulasi yang telah dilakukan diatas, dapat dilihat pada level isolasi transaksi *read committed* tidak diperbolehkan fenomena *dirty read*, namun fenomena *non-repeatable read*, *phantom read*, dan *serialization anomaly* diperbolehkan. Berikut tabel yang akan menggambarkan kedua transaksi tersebut dimana transaksi pada terminal pertama diberi nama T1, dan transaksi pada terminal kedua diberi nama T2.

Tabel 1.3.1 Simulasi *read committed*

T1	T2	Keterangan
BEGIN;		Memulai transaksi T1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;		Memilih level isolasi untuk T1
	BEGIN;	Memulai transaksi T2
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Memilih level isolasi untuk T2
	SELECT * FROM actor WHERE actor_id > 200;	Membaca <i>initial</i> data actor yang sudah ada pada T2
INSERT INTO actor(actor_id, first_name, last_name) VALUES (203, 'MICHELLE', 'OBAMA');		Menambahkan data actor baru pada T1

SELECT * FROM actor WHERE actor_id > 200;		Memastikan data actor baru dengan membaca data actor pada T1
	SELECT * FROM actor WHERE actor_id > 200;	Membaca data actor pada T2, hasil tetap tidak berubah dari <i>initial</i> data
COMMIT;		T1 melakukan <i>commit</i>
	SELECT * FROM actor WHERE actor_id > 200;	Membaca data actor pada T2, baris yang ditambah pada T1 muncul pada hasil <i>query</i>
BEGIN;		Memulai transaksi T1
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;		Memilih level isolasi untuk T1
UPDATE actor SET first_name = 'BARRACK' WHERE actor_id = 203		Melakukan perubahan terhadap data actor dengan id 203
COMMIT;		T1 melakukan <i>commit</i> .
	SELECT * FROM actor WHERE actor_id > 200;	Membaca data actor pada T2, data actor dengan id 203 berbeda dengan <i>query</i> sebelumnya
	INSERT INTO actor(actor_id, first_name, last_name) VALUES (203, 'MICHELLE', 'OBAMA');	Menambahkan data actor baru pada T2 yang memiliki id actor yang sama dengan data yang ditambahkan pada T1
	ROLLBACK;	T2 <i>rollback</i> karena data actor dengan id 203 sudah ada sehingga menyebabkan <i>error</i> .

2. Implementasi Concurrency Control Protocol

a. Two-Phase Locking (2PL)

Two-Phase Locking (2PL) merupakan sebuah protokol *concurrency control* yang digunakan dalam sistem manajemen basis data untuk memastikan konsistensi transaksi yang dieksekusi secara bersamaan. Protokol ini membagi eksekusi transaksi menjadi dua fase utama, yaitu :

1. Fase Pertumbuhan (*Growing Phase*)

Pada fase pertumbuhan, suatu transaksi dapat mengambil kunci akses(*locks*) pada data tetapi tidak diperbolehkan melepaskan kunci tersebut. *Locks* yang diambil tidak boleh dilepaskan hingga transaksi mencapai titik *commit*-nya. Hal ini bertujuan untuk menghindari situasi pembacaan yang tidak konsisten atau “*dirty reads*” dimana transaksi lain dapat membaca item data yang telah diubah oleh transaksi lain yang belum diselesaikan.

2. Fase Penyusutan (*Shrinking Phase*)

Fase penyusutan merupakan langkah selanjutnya dimana transaksi diperbolehkan melepaskan kunci akses(*locks*) namun tidak dapat lagi mengambil kunci baru. Pada fase ini, transaksi memutuskan apakah akan melakukan *commit* atau *abort*. Setelah sukses di-*commit*, transaksi tidak lagi dapat memperoleh atau melepaskan kunci.

Two-Phase Locking diimplementasikan untuk memastikan *serializability*, artinya hasil dari sejumlah transaksi yang berjalan bersamaan ekuivalen dengan hasil eksekusi secara serial dari transaksi-transaksi tersebut. Ini membantu mencegah konflik seperti pembaharuan yang hilang dan memastikan bahwa basis data tidak mengalami inkonsistensi sementara. Meskipun efektif, *Two-Phase Locking* dapat menghadapi masalah *deadlock*, yaitu kondisi di mana dua atau lebih transaksi saling menunggu satu sama lain untuk melepaskan kunci, membentuk suatu siklus yang tidak dapat diselesaikan. Oleh karena itu, sistem implementasi seringkali dilengkapi dengan mekanisme pencegahan atau deteksi *deadlock*.

Mekanisme pencegahan atau deteksi deadlock yang signifikan ada 2 yang mana masing-masing menawarkan strategi untuk mengelola transaksi dan mencegah terbentuknya siklus antrian menunggu yang dapat mengakibatkan deadlock, yaitu :

1. *Wound Wait*

Mekanisme pencegahan *deadlock* menggunakan pendekatan *wound wait* menonjol dengan mempertimbangkan tingkat prioritas transaksi. Apabila transaksi yang lebih tua membutuhkan kunci yang dipegang oleh transaksi yang lebih muda, pendekatan ini memungkinkan transaksi lebih muda untuk terus berjalan sementara transaksi yang lebih tua akan 'melukai' transaksi yang lebih muda dengan memaksa pelepasan kunci. Saat transaksi yang lebih muda berhasil 'dilukai' maka transaksi tersebut akan melakukan rollback dan melepaskan semua *lock* yang dimilikinya.

2. *Wait Die*

Mekanisme pencegahan deadlock menggunakan pendekatan *wait die* memandang dari sudut pandang yang berbeda. Pendekatan ini mengutamakan penanganan berdasarkan transaksi dengan kebijakan 'menunggu mati' untuk transaksi yang lebih muda yang membutuhkan kunci yang dimiliki oleh transaksi yang lebih tua. Transaksi yang lebih muda akan menunggu hingga transaksi yang lebih tua selesai sebelum mendapatkan akses.

Berikut merupakan implementasi dari *Two-Phase Locking Concurrency Control* yang diawali dengan membuat class *LockManager* :

```
try:
    from typing import Dict, Tuple, Optional, List
except ImportError:
    from typing_extensions import Dict, Tuple
from enum import Enum
from structs.transaction import Transaction
```

```

class LockType(Enum):
    SHARED = 1
    EXCLUSIVE = 2

class _LockList:
    def __init__(self):
        self._first: Tuple[int, LockType] = None
        self._locks: Dict[int, LockType] = {}

    def add(self, transaction_id: int, lock: LockType) -> bool:
        if self._first is None or self._first[0] ==
transaction_id:
            self._first = (transaction_id, lock)
            self._locks[transaction_id] = lock
            return True

    def remove(self, transaction_id: int):
        del self._locks[transaction_id]
        if len(self._locks) == 0:
            self._first = None
        elif self._first[0] == transaction_id:
            self._first = list(self._locks.items())[0]

    def peek_id_except(self, except_id: int = None) -> int:
        if except_id is None: return self.peek_id
        ids = list(filter(lambda id: id != except_id,
self._locks.keys()))
        return ids[0] if len(ids) != 0 else None

    @property
    def peek_id(self) -> int:
        return self._first[0] if self._first is not None else
None

```

```

@property
def peek_lock(self) -> LockType:
    return self._first[1]

def __getitem__(self, key: int) -> LockType:
    return self._locks[key]

def __contains__(self, key: int) -> bool:
    return key in self._locks

def __len__(self) -> int:
    return len(self._locks)

class LockManager:
    def __init__(self):
        self.locks: Dict[str, _LockList] = {}

    def grant_lock(self, data_item: str, lock_type: LockType,
transaction_id: int):
        # If there is no lock list on this data item yet,
create a new lock list
        if data_item not in self.locks:
            self.locks[data_item] = _LockList()

        # If the transaction already has a sufficient lock, do
nothing
        if self.has_lock(data_item, lock_type, transaction_id):
            return True

        # If this data item has no locks, grant lock
        if len(self.locks[data_item]) == 0:
            return self.locks[data_item].add(transaction_id,
lock_type)

```

```

        # If this data item already has locks...
        if lock_type == LockType.SHARED:
            # If an S-lock is requested, grant only if this
            data item only has S-locks
            return \
                self.locks[data_item].peek_lock ==
LockType.SHARED and \
                self.locks[data_item].add(transaction_id,
lock_type)
            elif lock_type == LockType.EXCLUSIVE:
                # If an X-lock is requested, grant only if the
            requester already has an S-lock
                # and only the requester has a lock on the item
            return \

self.locks[data_item].peek_id_except(transaction_id) is None
and \
                self.locks[data_item].peek_lock ==
LockType.SHARED and \
                self.locks[data_item].add(transaction_id,
lock_type)
            else:
                # How the hell did you get here????
                raise RuntimeError('What?')

    def has_lock(self, data_item: str, lock_type: LockType,
transaction_id: int) -> bool:
        if data_item not in self.locks: return False
        if transaction_id not in self.locks[data_item]: return
False
        lock = self.locks[data_item][transaction_id]
        return lock == LockType.EXCLUSIVE or lock_type ==
LockType.SHARED

```



```

def release_lock(self, data_item: str, transaction_id:
int):
    if data_item in self.locks and transaction_id in
self.locks[data_item]:
        self.locks[data_item].remove(transaction_id)

    def peek_lock_holder(self, data_item: str,
except_transaction_id: int = None) -> int:
        return
self.locks[data_item].peek_id_except(except_transaction_id) if
data_item in self.locks else None

def release_locks(self, transaction_id: int):
    for data_item in self.locks:
        self.release_lock(data_item, transaction_id)

```

Implementasi *Lock Manager* ini mencakup dua kelas utama, yaitu ``LockType`` dan ``LockManager``, yang secara keseluruhan membentuk *Lock Manager* untuk mengelola penguncian transaksi terhadap data item dalam lingkungan sistem manajemen *database*. ``LockType`` adalah enumerasi yang menyediakan nilai untuk jenis-jenis kunci, seperti ``SHARED`` dan ``EXCLUSIVE``. Ini digunakan untuk menunjukkan apakah suatu kunci mengacu pada penguncian berbagi atau eksklusif. ``_LockList`` adalah kelas yang mengimplementasikan daftar kunci untuk suatu data item tertentu. Kelas ini menyimpan informasi tentang *lock* yang dimiliki oleh transaksi, jenis kunci yang dimiliki, dan transaksi pertama yang memegang kunci. Kemudian, ``LockManager`` adalah kelas utama yang *me-manage lock* untuk semua data item dalam *database*. Kelas ini memiliki metode-metode seperti ``grant_lock``, ``has_lock``, ``release_lock``, dan ``peek_lock_holder`` untuk memberikan, memeriksa, melepaskan, dan melihat pemegang kunci penguncian. Selama pemberian *lock* (``grant_lock``), kelas ini mempertimbangkan beberapa skenario seperti jika data item belum memiliki kunci penguncian, transaksi telah memiliki kunci yang memadai, atau jenis kunci yang diminta

sesuai dengan aturan penguncian berbagi atau eksklusif. Pemantauan status dan tindakan dalam kelas ini dilakukan melalui pesan pencetakan yang memungkinkan pemahaman rinci tentang setiap langkah operasional dalam manajemen *lock*.

```
try:
    from typing import List
except:
    from typing_extensions import List
from core.cc.strategy import CCStrategy, Schedule
from structs.schedule import Schedule
from core.lock import LockType, LockManager
from structs.transaction import Operation, OperationType

class TwoPhaseLockingCC(CCStrategy):
    def accept(self, schedule: Schedule) -> None:
        lm = LockManager()
        queue: List[Operation] = list()

        print('Beginning 2PL protocol (using wound-wait DP
strategy)...')
        i = 0
        sched_len = len(schedule.operations)
        while i < sched_len:
            op = schedule.operations[i]

            # Check the type of this operations
            if op.op_type == OperationType.COMMIT:
                # Release all locks of this transaction
                print(f' {op}: Commit transaction
T{op.transaction_id} and release all locks')
                lm.release_locks(op.transaction_id)

            # Retry queued operations
            j = i + 1
```

```

        while len(queue) > 0:
            schedule.operations.insert(j, queue.pop(0))
            j += 1
        else:
            # Do automatic lock acquisition:
            # Select the appropriate lock type for this
operation
            if op.op_type == OperationType.READ:
                req_lock = LockType.SHARED
            elif op.op_type == OperationType.WRITE:
                req_lock = LockType.EXCLUSIVE

            # Try to acquire lock for this operation
            # If lock acquisition fails, use the wound-wait
strategy to prevent deadlocks
            do_req = True
            while do_req:
                # Request a lock on the data item to be
accessed
                if lm.grant_lock(op.data_item, req_lock,
op.transaction_id):
                    # The lock has been granted
                    print(f'  {op}: Acquired
{req_lock.name} lock on {op.data_item}')
                    do_req = False
                else:
                    # Compare this transaction's TS with
the lock holder's
                    holder_id =
lm.peek_lock_holder(op.data_item, op.transaction_id)
                    holder_ts =
schedule.transactions[holder_id].timestamp
                    op_ts =
schedule.transactions[op.transaction_id].timestamp

```

```

        if op_ts < holder_ts:
            # Wound: Abort the lock holder and
request lock again

            print(f'  {op}: Wound transaction
T{holder_id} (incompatible lock on {op.data_item})')
            lm.release_locks(holder_id)

            # Gather executed operations
            pre_queue: List[Operation] = list()
            for j in range(i - 1, -1, -1):
                if
schedule.operations[j].transaction_id == holder_id:
                    pre_queue.insert(0,
schedule.operations.pop(j))

                    i -= 1

            # Gather future operations (will be
empty if waiting)

            post_queue: List[Operation] =
list()

            for j in
range(len(schedule.operations) - 1, i, -1):
                if
schedule.operations[j].transaction_id == holder_id:
                    post_queue.insert(0,
schedule.operations.pop(j))

            queue = pre_queue + queue +
post_queue

        else:
            # Wait: Enqueue this and subsequent
operations

            print(f'  {op}: Wait for

```

```

transaction T{holder_id} (incompatible lock on
{op.data_item})')

        pre_queue: List[Operation] = list()
        for j in
range(len(schedule.operations) - 1, i - 1, -1):
            if
schedule.operations[j].transaction_id == op.transaction_id:
                pre_queue.insert(0,
schedule.operations.pop(j))
            i -= 1
            queue.extend(pre_queue)
            do_req = False

        # Move to the next operation
        i += 1
    print('2PL protocol finished')

```

Implementasi *Two-Phase Locking Concurrency Control* (2PL) dengan *wound-wait deadlock prevention* memanfaatkan pendekatan penguncian dua tahap untuk mengelola akses transaksi terhadap data item dalam suatu schedule. Proses dimulai dengan inisialisasi objek 'LockManager' ('lm') yang bertanggung jawab untuk mengelola penguncian. Selain itu, dibuat pula antrian ('queue') untuk menangani operasi-operasi yang harus menunggu akibat penguncian.

Selama iterasi melalui operasi-operasi dalam schedule, setiap operasi diuji tipe operasinya. Jika operasi merupakan operasi COMMIT, maka semua kunci penguncian yang dipegang oleh transaksi tersebut dilepaskan, dan operasi-operasi yang sebelumnya masuk dalam *queue* akan diulang. Pada operasi READ atau WRITE, sistem melakukan usaha penguncian otomatis dengan menentukan tipe kunci yang sesuai ('SHARED' untuk operasi READ dan 'EXCLUSIVE' untuk operasi WRITE).

Jika usaha penguncian gagal, sistem menggunakan strategi *wound-wait* untuk mencegah *deadlock*. Proses ini melibatkan perbandingan antara *timestamp* transaksi

saat ini (`op_ts`) dengan *timestamp* transaksi pemegang kunci saat ini (`holder_ts`). Jika *timestamp* transaksi saat ini lebih kecil daripada *timestamp* pemegang kunci, maka transaksi saat ini "melukai" pemegang kunci, yang mengakibatkan pemegang kunci tersebut di-*rollback*, dan transaksi saat ini dapat mencoba penguncian lagi. Sebaliknya, jika *timestamp* transaksi saat ini lebih besar, transaksi akan "menunggu" dan operasi-operasi yang terkait akan dimasukkan ke dalam antrian.

Selama proses ini, pesan pencetakan digunakan untuk memantau setiap langkah operasional 2PL dengan *wound-wait*. Pada akhirnya, pencetakan menunjukkan bahwa protokol 2PL telah selesai. Implementasi ini memberikan gambaran detail tentang bagaimana 2PL dengan *wound-wait* dapat mengelola penguncian dalam lingkungan konkurensi untuk mencegah *deadlock*.

Berikut merupakan beberapa contoh dari hasil penggunaan algoritma *Two-Phase Locking Concurrency Control*:

Hasil	Input	Deskripsi
<pre> a@lameydyvaannandya@Aulias-MacBook-Pro concurrency-protocol % python3 src/main.py -s 2pl "R1(A); R2(B); W1(B); W2(A); C1; C2" Input schedule: T1 T2 R1(A) R2(B) W1(B) W2(A) C1 C2 Beginning 2PL protocol (using wound-wait DP strategy)... R1(A): Acquired SHARED lock on A R2(B): Acquired SHARED lock on B W1(B): Wound transaction T2 (incompatible lock on B) W1(B): Acquired EXCLUSIVE lock on B C1: Commit transaction T1 and release all locks R2(B): Acquired SHARED lock on B W2(A): Acquired EXCLUSIVE lock on A C2: Commit transaction T2 and release all locks 2PL protocol finished New schedule (using CC strategy "TwoPhaseLockingCC"): T1 T2 R1(A) R2(B) W1(B) W2(A) C1 C2 </pre>	<p>"R1(A); R2(B); W1(B); W2(A); C1; C2"</p>	<p>Schedule di samping merupakan contoh dari schedule yang mengalami deadlock. Dimana transaksi 1 menunggu transaksi 2 untuk melepaskan lock pada data item B dan juga transaksi 2 menunggu transaksi 1 untuk melepaskan lock pada data item A.</p>
<pre> a@lameydyvaannandya@Aulias-MacBook-Pro concurrency-protocol % python3 src/main.py -s 2pl "R1(A); W2(A); W2(B); C1; R3(B); W3(B); C3; R2(B); C2; W1(B); C1; R2(A); C2; R3(A); C3" Input schedule: T1 T2 T3 R1(A) W2(A) R3(B) C1 W2(B) W3(B) C3 R2(B) C2 R2(A) C2 R3(A) C3 Beginning 2PL protocol (using wound-wait DP strategy)... R1(A): Acquired SHARED lock on A W2(A): Wait for transaction T1 (incompatible lock on A) C1: Commit transaction T1 and release all locks W2(A): Acquired EXCLUSIVE lock on A W2(B): Acquired EXCLUSIVE lock on B R2(B): Acquired SHARED lock on B C2: Commit transaction T2 and release all locks R2(A): Acquired SHARED lock on A C2: Commit transaction T2 and release all locks R3(B): Acquired EXCLUSIVE lock on B W3(B): Acquired EXCLUSIVE lock on B C3: Commit transaction T3 and release all locks W1(B): Acquired EXCLUSIVE lock on B C1: Commit transaction T1 and release all locks R3(A): Acquired SHARED lock on A C3: Commit transaction T3 and release all locks 2PL protocol finished New schedule (using CC strategy "TwoPhaseLockingCC"): </pre>	<p>"R1(A); W2(A); W2(B); C1; R3(B); W3(B); C3; R2(B); C2; W1(B); C1; R2(A); C2; R3(A); C3"</p>	<p>Schedule di samping merupakan contoh dari schedule yang mengalami tidak mengalami deadlock.</p>

b. Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control atau disebut juga *Validation Based Protocol* merupakan sebuah protokol *concurrency control* pada DBMS yang menggunakan asumsi bahwa transaksi yang dieksekusi sepenuhnya akan berjalan lancar selama validasi. Pada protokol ini, setiap transaksi akan memiliki 3 *timestamp*, yaitu:

- *StartTS*: Waktu ketika transaksi memulai eksekusi operasi pertamanya
- *ValidationTS*: Waktu ketika transaksi memasuki fase validasi
- *FinishTS*: Waktu ketika transaksi telah menyelesaikan fase tulis

Timestamp ini akan digunakan pada fase validasi untuk menentukan valid atau tidaknya transaksi pada suatu jadwal. *Optimistic Concurrency Control* memiliki 3 tahap atau fase, yaitu:

1. Read Phase

Pada fase ini akan dieksekusi seluruh operasi *read* dan *write* pada transaksi. Namun, perubahan dan penambahan yang dilakukan dengan operasi *write* akan dilakukan pada variabel lokal terlebih dahulu dan akan diaplikasikan pada database di fase setelah validasi.

2. Validation Phase

Pada fase ini dilakukan validasi terhadap transaksi untuk menentukan apakah variabel lokal yang telah ditulis dapat dieksekusi tanpa melanggar *serializability*. Sebuah transaksi dikatakan tervalidasi apabila memenuhi salah satu aturan berikut:

- $FinishTS(T_i) < StartTS(T_j)$ atau waktu selesainya transaksi-transaksi sebelumnya lebih awal dari waktu mulai transaksi yang sedang divalidasi sehingga operasi yang dilakukan pada database akan dieksekusi menggunakan data paling terbaru.
- $StartTS(T_j) < FinishTS(T_i) < ValidationTS(T_j)$ atau waktu selesainya transaksi-transaksi sebelumnya lebih awal dari waktu validasi transaksi meskipun waktu mulai transaksi berada lebih awal. Namun, pada aturan

ini perlu dipastikan juga bahwa tidak ada operasi antara transaksi yang saling berpotongan.

3. Write Phase

Pada fase ini, apabila sebuah transaksi sudah tervalidasi, penulisan yang dilakukan pada variabel lokal di tahap sebelumnya akan diaplikasikan pada database. Namun apabila transaksi tidak tervalidasi, maka transaksi tersebut akan dilakukan *roll back* atau diulang secara keseluruhan.

Dengan menggunakan *Optimistic Concurrency Control*, akan didapatkan kinerja yang lebih baik karena tidak adanya *cascading rollback* serta tidak adanya penguncian atau *locking* yang diperlukan selama fase membaca. Namun, hal ini juga dapat menyebabkan lebih banyak pengulangan transaksi jika konflik sering terjadi yang dapat berakhir pada *starvation*.

Berikut merupakan implementasi dari *Optimistic Concurrency Control*:

```
class _State:
    def __init__(self, startTS: int, validationTS: int, finishTS: int,
isFirstTr: bool = False) -> None:
        self.isFirstTr = isFirstTr
        self.startTS = startTS
        self.validationTS = validationTS
        self.finishTS = finishTS
        self.read_set: List[Operation] = []
        self.write_set: List[Operation] = []

class OptimisticCC(CCStrategy):
    def accept(self, schedule: Schedule) -> None:
        transaction_ts: Dict[int, _State] = dict()
        preExecutedOp: List[Operation] = schedule.operations
        executedTr: List[Transaction] = []

        # Setting initial/default timestamps
        for tr in schedule.transactions.keys():
            transaction_ts[tr] =
```



```

_State(schedule.operations.index(schedule.transactions[tr].operations[0]
), 0, 0)

    # Marking first transaction
    transaction_ts[schedule.operations[0].transaction_id].isFirstTr
= True

    # Read Phase
    for op in schedule.operations:
        if op.op_type == OperationType.READ:
            transaction_ts[op.transaction_id].read_set.append(op)
        elif op.op_type == OperationType.WRITE:
            transaction_ts[op.transaction_id].write_set.append(op)

    # Validation phase
    for tr in schedule.transactions.keys():
        print(f"Validating transaction {tr}...")

        # Set validationTS in the time commit
        transaction_ts[tr].validationTS =
preExecutedOp.index(schedule.transactions[tr].operations[-1])

        # If first transaction, automatically validated
        if transaction_ts[tr].isFirstTr:
            transaction_ts[tr].finishTS =
transaction_ts[tr].validationTS
            executedTr.append(schedule.transactions[tr])

        else:
            success = False

            # Keep rollingback everytime aborted
            while not success:

                # Check if data used was updated in previous
transaction
                for executed in executedTr:
                    if transaction_ts[executed.id].finishTS <
transaction_ts[tr].startTS:

```

```

        success = True

        elif transaction_ts[tr].startTS <
transaction_ts[executed.id].finishTS < transaction_ts[tr].validationTS:
            for execWrite in
transaction_ts[executed.id].write_set:
                if execWrite.data_item in [op.data_item
for op in schedule.transactions[tr].operations]:
                    print(f"Operation intersect detected
with {execWrite} from transaction {executed.id}")
                    success = False
                    break
                success = True
            else:
                success = False

        if not success:
            break

        # Rollback
        if not success:
            print(f"Abort, restarting transaction {tr} to
after transaction {executed.id} at timestamp
{transaction_ts[executed.id].finishTS + 1}")
            transaction_ts[tr].startTS =
transaction_ts[executed.id].finishTS + 1
            transaction_ts[tr].validationTS +=
(transaction_ts[executed.id].finishTS + 1 - transaction_ts[tr].startTS)

            # Move operations to the front
            for op in schedule.transactions[tr].operations:
                restart: Operation =
preExecutedOp.pop(preExecutedOp.index(op))
                preExecutedOp.append(restart)

            # Write Phase
        else:
            transaction_ts[tr].finishTS =
transaction_ts[tr].validationTS
            executedTr.append(schedule.transactions[tr])

```

```

        print(f"Transaction {tr} has been validated, adding to
schedule\n")
        schedule.operations = preExecutedOp

```

Pada implementasi, dibuat sebuah class state yang digunakan untuk menyimpan detail versi sebuah transaksi berisi *timestamp*, *read set*, dan *write set*-nya. Algoritma dimulai dengan melakukan inisiasi terhadap transaksi-transaksi pada schedule yang meliputi inisiasi *timestamp*, transaksi pertama, dan variabel lainnya. Selanjutnya dilakukan *read phase* di mana operasi pada *schedule* akan dimasukkan ke dalam *read* dan *write set* masing-masing transaksi. Setelah *read phase*, setiap transaksi pada jadwal akan masuk *validation phase* yang mengecek validasi transaksi menggunakan aturan yang sudah telah dijelaskan sebelumnya. Apabila transaksi tidak berhasil validasi, maka transaksi akan di *rollback* dengan mengubah *timestamp* pada transaksi menjadi setelah *finish timestamp* transaksi sebelumnya, memindahkan urutan operasi ke depan jadwal, dan melakukan kembali validasi. Namun apabila transaksi berhasil validasi, akan masuk ke *write phase* di mana akan di aplikasikan perubahan-perubahan yang dilakukan pada jadwal secara langsung.

Berikut merupakan beberapa contoh dari hasil penggunaan algoritma *Optimistic Concurrency Control*:

Hasil	Masukkan	Deskripsi
-------	----------	-----------

<pre> Input schedule: T1 T2 ----- ----- R1(A) W1(A) R2(B) W2(B) C1 C2 Validating transaction 1... Transaction 1 has been validated, adding to schedule Validating transaction 2... Transaction 2 has been validated, adding to schedule New schedule (using CC strategy "OptimisticCC"): T1 T2 ----- ----- R1(A) W1(A) R2(B) W2(B) C1 C2 </pre>	<p>"R1(A); R2(B); W1(A); W2(B); C1; C2"</p>	<p>Jadwal atau urutan operasi yang dimasukkan tidak melanggar aturan sehingga output menghasilkan jadwal yang sama</p>
<pre> Input schedule: T1 T2 T3 ----- ----- ----- R1(X) W2(X) W2(Y) W3(Y) W1(Y) C2 C1 C3 Validating transaction 1... Transaction 1 has been validated, adding to schedule Validating transaction 2... Operation intersect detected with W1(Y) from transaction 1 Abort, restarting transaction 2 to after transaction 1 at timestamp 6 Transaction 2 has been validated, adding to schedule Validating transaction 3... Abort, restarting transaction 3 to after transaction 1 at timestamp 6 Abort, restarting transaction 3 to after transaction 2 at timestamp 7 Transaction 3 has been validated, adding to schedule New schedule (using CC strategy "OptimisticCC"): T1 T2 T3 ----- ----- ----- R1(X) W2(X) W1(Y) W2(Y) W3(Y) C1 C2 C3 </pre>	<p>"R1(X); W2(X); W2(Y); W3(Y); W1(Y); C1; C2; C3"</p>	<p>Terdapat konflik pada T2 di mana dilakukan penulisan pada item Y, namun di T1 dilakukan penulisan pada item yang sama sehingga terjadi <i>intersect</i> dan T2 perlu di <i>rollback</i>. Hal yang sama dilakukan pada T3.</p>

c. Multiversion Timestamp Ordering Concurrency Control (MVCC)

Multiversion Timestamp Ordering merupakan sebuah protokol concurrency control dalam sistem manajemen basis data yang bertujuan untuk mengelola akses

bersama pada data dengan cara yang efisien dan konsisten. Protokol ini dapat menjadi sebuah solusi untuk masalah pembacaan yang tidak konsisten dan menangani konflik akses antar transaksi secara bersamaan.

Pertama-tama, MVCC memungkinkan adanya versi terbaru dari data yang berbeda-beda, setiap versi ditandai dengan timestamp yang merefleksikan saat terjadinya perubahan pada data tersebut. Saat suatu transaksi ingin membaca atau menulis data, MVCC akan memastikan bahwa transaksi tersebut melibatkan versi data yang sesuai dengan timestampnya. Oleh karena itu, pembacaan transaksi tidak akan melibatkan data yang sedang dalam proses penulisan oleh transaksi lain, mencegah terjadinya pembacaan yang tidak konsisten. MVCC juga menyediakan mekanisme untuk menangani write-write conflicts. Jika suatu transaksi ingin menulis data yang sudah diakses oleh transaksi lain, MVTO memeriksa cap waktu dari kedua transaksi tersebut. Jika transaksi yang ingin menulis memiliki cap waktu yang lebih rendah, maka transaksi yang ingin menulis akan di-rollback untuk memastikan konsistensi data dan integritas transaksi.

Berikut merupakan implementasi dari *Multiversion Timestamp Ordering Concurrency Control* :

```
class _Version:
    def __init__(self, data_item: str, version: int, read_t:
int, write_t: int) -> None:
        self.data_item = data_item
        self.version = version
        self.read_t = read_t
        self.write_t = write_t

    def __repr__(self) -> str:
        return f'<{self.data_item}{self.version}, RTS:
{self.read_t}, WTS: {self.write_t}>'
```

Protokol MVCC dimulai dengan menginisialisasi versi awal untuk setiap data item yang ada dalam schedule. Setiap versi dilabeli dengan nomor versi (version) dan

memiliki dua timestamp, yaitu read timestamp (RTS) dan write timestamp (WTS). Timestamp ini penting untuk menentukan urutan akses dan untuk menangani konflik antar transaksi.

```
class MultiversionTimestampCC(CCStrategy):
    def accept(self, schedule: Schedule) -> None:
        versions: Dict[str, List[_Version]] = dict()

        # Get initial timestamp as one unit before the smallest
        transaction timestamp
        init_t = min([t for t in list(map(lambda t:
        t.timestamp, schedule.transactions.values()))]) - 1

        # Add all initial versions of each data item
        for op in schedule.operations:
            if op.data_item is not None and op.data_item not in
versions:
                versions[op.data_item] =
[_Version(op.data_item, 0, init_t, init_t)]

        print('Initial versions:')
        print('\n'.join([f'{dat}: ' + repr(ver) for dat, ver in
versions.items()])))

        print('Beginning MVCC protocol...')
        i = 0
        sched_len = len(schedule.operations)
        while i < sched_len:
            op = schedule.operations[i]

            # Check the type of this operation
            if op.op_type == OperationType.COMMIT:
                # Commit this transaction
                print(f'    {op}: Commit transaction
```

```

T{op.transaction_id}')

        else:
            # Find the latest version with less or equal
            WTS to this transaction's TS
            ts =
            schedule.transactions[op.transaction_id].timestamp
            ver = max(
                filter(
                    lambda v: v.write_t <= ts,
                    versions[op.data_item]
                ),
                key=lambda v: v.write_t
            )

            if op.op_type == OperationType.READ:
                # Update the RTS of this version to this
                transaction's TS
                if ver.read_t < ts:
                    print(f'    {op}: Read from and update
                    RTS of version {ver} to {ts}')
                    ver.read_t = ts
                else:
                    print(f'    {op}: Read from version
                    {ver}')

            elif op.op_type == OperationType.WRITE:
                # Compare the timestamps of this version to
                the transaction's TS
                if ver.read_t > ts:
                    # This version already has a newer
                    transaction reading its value:
                    # Rollback the current transaction
                    print(f'    {op}: Rollback transaction
                    T{op.transaction_id}')

```

```

        # Remove this transaction's operations
from the schedule

        to_abort: List[Operation] = list()
        for j in range(len(schedule.operations) -
1, -1, -1):

            if
schedule.operations[j].transaction_id == op.transaction_id:
                to_abort.insert(0,
schedule.operations.pop(j))

                if j <= i: i -= 1

        # Append the operations at the end of
the schedule and update the
        # corresponding transaction's timestamp
        schedule.operations += to_abort
        new_ts = max(list(map(lambda t:
t.timestamp, schedule.transactions.values())) + 1

schedule.transactions[op.transaction_id].timestamp = new_ts

        elif ver.write_t == ts:
            # Overwrite the content of this version
            print(f'    {op}: Overwrite data at
{ver}')

        else:
            # Create a new version of this data
item with its timestamps set
            # to the current transactions' TS
            new_ver = _Version(op.data_item,
len(versions[op.data_item]), ts, ts)
            versions[op.data_item].append(new_ver)
            print(f'    {op}: Add new version
{new_ver}')

```



```

        # Move to the next operation
        i += 1
        print('MVCC protocol finished')

        print('Final versions:')
        print('\n'.join([f'{dat}: ' + repr(ver) for dat, ver in
versions.items()])))

```

Class MultiversionTimestampCC dimulai dengan inialisasi variabel, termasuk `versions` yang berfungsi sebagai wadah untuk menyimpan versi-versi data item. Timestamp awal (`init_t`) dihitung sebagai satu unit sebelum timestamp transaksi terkecil dalam *schedule*. Selama iterasi pertama melalui operasi-operasi dalam *schedule*, versi awal dari setiap data item ditambahkan ke `versions`.

Selanjutnya, setiap operasi dalam *schedule* diproses secara berurutan. Jika operasi merupakan operasi COMMIT, transaksi dianggap selesai, dan pesan commit dicetak. Pada operasi READ atau WRITE, MVCC mencari versi terbaru dari data item yang memiliki *Write Timestamp* (WTS) kurang dari atau sama dengan *timestamp* transaksi saat ini (`ts`). Jika versi tersebut ditemukan, *Read Timestamp* (RTS) dari versi tersebut diperbarui ke timestamp transaksi, menunjukkan bahwa transaksi telah membaca nilai dari versi tersebut.

Pada operasi WRITE, MVXX melakukan perbandingan antara *timestamp* transaksi dengan WTS dari versi yang ditemukan. Jika WTS lebih kecil dari *timestamp* transaksi, menandakan bahwa versi tersebut sudah dibaca oleh transaksi lain yang lebih baru sehingga transaksi saat ini di-*rollback*. Jika WTS sama dengan *timestamp* transaksi, transaksi tersebut akan memiliki hak akses untuk menulis ke versi tersebut, dan nilai dari versi tersebut ditulis kembali. Jika WTS lebih besar dari *timestamp* transaksi, MVCC membuat versi baru dari data item dengan RTS dan WTS yang di set ke *timestamp* transaksi saat ini.

Berikut merupakan beberapa contoh dari hasil penggunaan algoritma *Multiversion Timestamp Ordering Concurrency Control*:

Hasil	Input	Deskripsi
<pre> mulamerydianandya@Julias-MacBook-Pro concurrency-protocol % python3 src/main.py --s mvcc "R1(A); W1(A); C1; R2(B); W2(A); C2; R3(A); R3(B); C3;" Input schedule: T1 T2 T3 ----- R1(A) W1(A) C1 R2(B) W2(A) C2 R3(A) R3(B) C3 Initial versions: A: [-<0>, RTS: 0, WTS: 0-] B: [-<0>, RTS: 0, WTS: 0-] Beginning MVCC protocol... R1(A): Read from and update RTS of version <0>, RTS: 0, WTS: 0- to 1 W1(A): Add new version <1>, RTS: 1, WTS: 1- C1: Commit transaction T1 R2(B): Read from and update RTS of version <0>, RTS: 0, WTS: 0- to 2 W2(A): Add new version <2>, RTS: 2, WTS: 2- C2: Commit transaction T2 R3(A): Read from and update RTS of version <2>, RTS: 2, WTS: 2- to 3 R3(B): Read from and update RTS of version <0>, RTS: 2, WTS: 0- to 3 C3: Commit transaction T3 MVCC protocol finished Final versions: A: [-<0>, RTS: 1, WTS: 0-], <1>, RTS: 1, WTS: 1-], <2>, RTS: 3, WTS: 2-] B: [-<0>, RTS: 3, WTS: 0-] New schedule (using CC strategy "MultiversionTimestampCC"): T1 T2 T3 ----- R1(A) W1(A) C1 R2(B) W2(A) C2 R3(A) R3(B) C3 </pre>	<p>"R1(A); W1(A); C1; R2(B); W2(A); C2; R3(A); R3(B); C3"</p>	<p>Transaksi 3 membaca data item A dan B secara bersamaan dengan T1 dan T2 karena melakukan pembacaan dari versi sebelum commit.</p>
<pre> mulamerydianandya@Julias-MacBook-Pro concurrency-protocol % python3 src/main.py --s mvcc "R1(A); W1(A); C1; R2(B); C2; W3(A); C3;" Input schedule: T1 T2 T3 ----- R1(A) W1(A) C1 R2(B) C2 W3(A) C3 Initial versions: A: [-<0>, RTS: 0, WTS: 0-] B: [-<0>, RTS: 0, WTS: 0-] Beginning MVCC protocol... R1(A): Read from and update RTS of version <0>, RTS: 0, WTS: 0- to 1 W1(A): Add new version <1>, RTS: 1, WTS: 1- C1: Commit transaction T1 R2(B): Read from and update RTS of version <0>, RTS: 0, WTS: 0- to 2 C2: Commit transaction T2 W3(A): Add new version <2>, RTS: 3, WTS: 3- C3: Commit transaction T3 MVCC protocol finished Final versions: A: [-<0>, RTS: 1, WTS: 0-], <1>, RTS: 1, WTS: 1-], <2>, RTS: 3, WTS: 3-] B: [-<0>, RTS: 2, WTS: 0-] New schedule (using CC strategy "MultiversionTimestampCC"): T1 T2 T3 ----- R1(A) W1(A) C1 R2(B) C2 W3(A) C3 </pre>	<p>"R1(A); W1(A); C1; R2(B); C2; W3(A); C3"</p>	<p>Transaksi 3 dapat menulis ke versi baru dari data item A tanpa menunggu commit transaksi 1.</p>

3. Eksplorasi Recovery

a. *Write-Ahead Log*

Write-Ahead Log merupakan sebuah metode *recovery* pada DBMS PostgreSQL yang beroperasi dengan menggunakan konsep melakukan *logging* terhadap data yang ditulis ke dalam tempat penyimpanan yang aman sebelum perubahan disimpan di dalam *database*. Dengan *write-ahead log* apabila terjadi *crash* ketika suatu operasi sedang dijalankan, maka *recovery* dapat dilakukan dengan melihat *log entry* ketika sistem mengalami *crash* dan menggunakan *log* tersebut untuk melakukan *recovery*. Selain itu, karena penyimpanan *log* dilakukan secara sekuensial, maka *atomicity* data akan tetap bisa di-*maintain*.

Beberapa kelebihan dari *write-ahead log* adalah mengurangi *write* ke *disk*, mengurangi biaya sinkronisasi, operasi *redo* yang cepat, dan data *backup* yang *accessible*.

b. *Continuous Archiving*

Continuous archiving merupakan ekstensi tambahan dari metode *write-ahead log* dimana dengan *continuous archiving file write-ahead log* akan disimpan juga pada tempat penyimpanan eksternal yang lebih stabil. Hal ini akan menambah proteksi data karena data juga disimpan di *archive* eksternal.

Terdapat konfigurasi yang harus dilakukan untuk *WAL Archiving*, yaitu dengan melakukan konfigurasi *wal_level* menjadi *replica* atau lebih tinggi, mengaktifkan *archive_mode*, dan menspesifikasikan perintah shell yang digunakan di dalam parameter konfigurasi *archive_command*.

c. *Point-in-Time Recovery*

Point-in-time recovery merupakan sebuah metode yang dapat dilakukan oleh *administrator database* untuk melakukan *restore* atau *recover* pada *database* pada suatu waktu spesifik menggunakan *WAL* yang sudah di-*archive*. PITR ini biasa digunakan ketika seseorang secara tidak sengaja melakukan penghapusan terhadap sebuah tabel atau *record* atau ketika terjadi *corrupt* pada *database*.

d. Konfigurasi pada PostgreSQL

Proses konfigurasi dilakukan pada sistem operasi WSL dan PostgreSQL 14.

1. Melakukan konfigurasi *continuous archiving* pada klaster basis data

Pertama-tama akan dibuat direktori baru untuk menyimpan *archive file WAL*

```
$ mkdir database_archive
```

Kemudian, berikan *default user* PostgreSQL, yaitu **postgres** akses untuk melakukan penulisan pada direktori tersebut

```
$ sudo chown postgres:postgres database_archive
```

Kemudian, lakukan konfigurasi pada file **postgresql.conf** untuk memperbolehkan *archiving*

```
$ sudo nano /etc/postgresql/14/main/postgresql.conf
```

Setelah file dibuka, cari variabel **archive_mode** dan *uncomment* variabel tersebut dengan menghapus # pada awal *line* dan ubah value dari variabel tersebut menjadi **on**

```
...  
archive_mode = on  
...
```

Kemudian ubah juga variabel **archive_command** sesuai perintah dibawah

```
...  
archive_command = 'test ! -f /path/to/database_archive/%f && cp %p  
/path/to/database_archive/%f'  
...
```

Perintah diatas digunakan untuk melakukan pengecekan apakah *file WAL* sudah ada di *archive* atau tidak, apabila *file WAL* belum ada maka *file WAL* akan di-copy ke *archive*. Untuk perintah di atas gantila **/path/to/database_archive** sesuai dengan path *database_archive* yang telah dibuat sebelumnya

Terakhir, lakukan konfigurasi terhadap variabel **wal_level** menjadi **replica**. Variabel **wal_level** digunakan untuk menentukan seberapa banyak informasi yang ditulis ke *log* oleh PostgreSQL

```
...  
wal_level = replica  
...
```

Simpan perubahan yang telah dibuat pada file konfigurasi, dan untuk mengimplementasikan perubahan terhadap file konfigurasi yang telah dibuat, lakukan *restart* terhadap klaster basis data dengan perintah berikut:

```
$ sudo systemctl restart postgresql@14-main
```

Setelah PostgreSQL berhasil di-*restart*, klaster basis data akan meng-*archive* tiap *file WAL* ketika file tersebut sudah penuh. Apabila ingin melakukan *archive* sebuah transaksi secara langsung, bisa dengan memaksa klaster basis data untuk mengubah dan meng-*archive WAL file* saat itu dengan menjalankan perintah berikut:

```
$ sudo -u postgres psql -c "SELECT pg_switch_wal();"
```

Dengan klaster basis data yang sudah berhasil melakukan *copy* ke *file WAL* pada *archive*, dapat dilakukan *physical backup* kepada file-file klaster basis data.

2. Melakukan *physical backup* terhadap klaster PostgreSQL

Melakukan *backup* secara regular merupakan hal penting untuk mencegah hal-hal yang tidak diinginkan terjadi seperti kehilangan data. PostgreSQL memperbolehkan untuk melakukan *backup* secara *logical* dan *physical*. Untuk melakukan *PITR* dibutuhkan *backup* data secara *physical* yaitu dengan meng-*copy* seluruh file basis data di direktori data PostgreSQL. Secara *default*, lokasi direktori data PostgreSQL terletak di **/var/lib/postgresql/14/main/**.

Apabila ingin melakukan pengecekan terhadap direktori data PostgreSQL dapat dilakukan dengan perintah berikut.

```
$ sudo -u postgres psql -c "SHOW_data_directory;"
```

Pada tahap sebelumnya, sudah dibuat direktori *database_archive* untuk menyimpan seluruh file *WAL*. Untuk menyimpan data *backup* secara *physical* dibutuhkan untuk membuat direktori baru, yaitu ***database_backup***.

```
$ mkdir database_backup
```

Kemudian, pastikan user ***postgres*** memiliki akses untuk melakukan penulisan data ke direktori tersebut dengan perintah berikut.

```
$ sudo chown postgres:postgres database_backup
```

Kemudian, akan dilakukan *physical backup* pada PostgreSQL dengan menggunakan *pg_basebackup* yang merupakan perintah *built-in* PostgreSQL dengan perintah berikut.

```
$ sudo -u postgres pg_basebackup -D /path/to/database_backup
```

Ubah ***/path/to/database_backup*** dengan path direktori yang telah dibuat sebelumnya untuk menyimpan data *backup* secara *physical*. Dengan ini, *point-in-time recovery* telah dapat dilakukan.

3. Melakukan *point-in-time recovery* pada klaster basis data

Dengan adanya *physical backup* dari basis data dan di-*archive* nya file *WAL*, dapat dilakukan *point-in-time recovery* apabila diperlukan *rollback* basis data ke *state* tertentu.

Pertama, apabila basis data sedang berjalan, matikan basis data dengan perintah berikut.

```
$ sudo systemctl stop postgresql@14-main
```

Kemudian, setelah basis data berhasil dimatikan, hapus semua file pada direktori data PostgreSQL. Namun, sebelum itu pindahkan direktori ***pg_wal*** ke direktori yang

berbeda karena **pg_wal** bisa saja menyimpan file *WAL* yang belum ter-*archive* yang penting untuk proses *recovery*. Gunakan perintah **mv** untuk memindahkan **pg_wal** dengan perintah berikut.

```
$ sudo mv /var/lib/postgresql/14/main/pg_wal ~/
```

Setelah **pg_wal** dipindahkan, hapus direktori **/var/lib/postgresql/14/main** dan buat ulang direktori tersebut dengan perintah-perintah berikut.

```
$ sudo rm -rf /var/lib/postgresql/14/main
```

Dilanjutkan dengan perintah:

```
$ sudo mkdir /var/lib/postgresql/14/main
```

Kemudian, salin kembali seluruh data dari *physical backup* yang telah dibuat sebelumnya ke direktori data PostgreSQL baru dengan perintah berikut.

```
$ sudo cp -a /path/to/database_backup/. /var/lib/postgresql/14/main
```

Jangan lupa untuk mengubah **/path/to/database_backup** dengan *path* direktori *physical backup* yang telah dibuat sebelumnya. Kemudian, pastikan user **postgres** merupakan pemilik dari direktori data PostgreSQL dan memiliki akses terhadap direktori tersebut. Hal ini dapat dilakukan dengan perintah berikut

```
$ sudo chown postgres:postgres /var/lib/postgresql/14/main
```

Kemudian, lakukan *update* terhadap aksesnya.

```
$ sudo chmod 700 /var/lib/postgresql/14/main
```

File *WAL* pada direktori **pg_wal** yang disalin dari *physical backup* sudah *outdated* dan tidak berguna sehingga gantikan file *WAL* pada **pg_wal** dengan file *WAL* pada

pg_wal yang disalin sebelumnya ketika mengosongkan direktori data PostgreSQL. Hapus **pg_wal** pada direktori `/var/lib/postgresql/14/main` dengan perintah berikut.

```
$ sudo rm -rf /var/lib/postgresql/14/main/pg_wal
```

Sekarang, salin file-file dari **pg_wal** yang sebelumnya telah disalin dengan perintah berikut.

```
$ sudo cp -a ~/pg_wal /var/lib/postgresql/14/main/pg_wal
```

Dengan direktori data yang sudah pulih, perlu dilakukan konfigurasi terhadap *recovery settings* untuk memastikan basis data me-recover file WAL yang telah di-archive dengan benar. *Recovery settings* dapat ditemukan pada file **postgresql.conf** yang berada pada direktori `/etc/postgresql/14/main/`. Buka file konfigurasi dengan perintah berikut.

```
$ sudo nano /etc/postgresql/14/main/postgresql.conf
```

Ketika file telah berhasil dibuka, cari variabel dengan nama **restore_command** dan *uncomment* variabel tersebut dengan menghapus `#` pada awal *line*. Ubah *value* pada variabel tersebut dengan *value* berikut.

```
...
restore_command = 'cp /path/to/database_archive/%f %p'
...
```

Jangan lupa untuk mengubah `/path/to/database_archive/` dengan path direktori *archive* yang sebelumnya telah dibuat. Selanjutnya, lakukan konfigurasi untuk menspesifikasikan *recovery target*. *Recovery target* dapat berupa *timestamp*, *transaction ID*, *log sequence number*, dan nama dari *restore point* yang dibentuk oleh perintah **pg_create_restore_point()**. *Recovery target* dispesifikasikan agar klaster basis data tidak membaca seluruh *log* dari file-file WAL di *archive*. Untuk

opsi-opsi yang tersedia terdapat pada *recovery target* dapat dilihat di [PostgreSQL recovery target](#).

Apabila variabel ***restore_command*** dan ***recovery_target*** sudah dikonfigurasi, save dan keluar dari file ***postgresql.conf***. Sebelum, melakukan *restart* terhadap klaster basis data, beritahukan kepada PostgreSQL untuk mulai dengan mode *recovery*. Hal ini dapat dilakukan dengan membuat file kosong pada direktori klaster data, yaitu ***recovery.signal*** dengan menggunakan perintah *touch*.

```
$ sudo touch /var/lib/postgresql/14/main/recovery.signal
```

Kemudian, lakukan *restart* terhadap klaster basis data dengan perintah berikut.

```
$ sudo systemctl start postgresql@14-main
```

Apabila basis data berhasil dijalankan, maka basis data akan masuk ke mode *recovery* dan ketika *recovery target* telah dicapai maka file ***recovery.signal*** akan secara otomatis dihapus.

Untuk melihat *transaction log* setelah proses *recovery* selesai dilakukan, dapat dilakukan dengan menjalankan perintah berikut.

```
$ sudo less /var/log/postgresql/postgresql-14-main.log
```

e. Simulasi Kegagalan pada PostgreSQL

Untuk simulasi kali ini, basis data yang digunakan adalah *pagila.sql* yang merupakan basis data yang digunakan selama praktikum.

Data tabel awal :

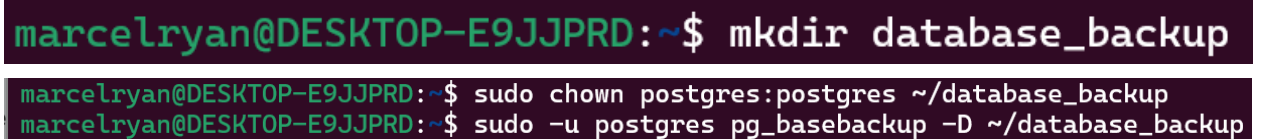


```
postgres=# \d
              List of relations
Schema | Name          | Type  | Owner
-----+-----+-----+-----
public | actor         | table | postgres
public | address       | table | postgres
public | category      | table | postgres
public | city          | table | postgres
public | country       | table | postgres
public | customer      | table | postgres
public | film          | table | postgres
public | film_actor    | table | postgres
public | film_category | table | postgres
public | inventory     | table | postgres
public | language      | table | postgres
public | payment       | table | postgres
public | payment_p2007_01 | table | postgres
public | payment_p2007_02 | table | postgres
public | payment_p2007_03 | table | postgres
public | payment_p2007_04 | table | postgres
public | payment_p2007_05 | table | postgres
public | payment_p2007_06 | table | postgres
public | rental        | table | postgres
public | staff         | table | postgres
public | store         | table | postgres
(21 rows)

postgres=#
```

Gambar 3.5.1 Daftar tabel pada *database* pagila.

Mempersiapkan *physical backup* untuk klaster basis data dengan perintah berikut.



```
marcelryan@DESKTOP-E9JJPRD:~$ mkdir database_backup
marcelryan@DESKTOP-E9JJPRD:~$ sudo chown postgres:postgres ~/database_backup
marcelryan@DESKTOP-E9JJPRD:~$ sudo -u postgres pg_basebackup -D ~/database_backup
```

Gambar 3.5.2 Konfigurasi *physical backup*

Kemudian, terdapat seorang *user* yang tidak sengaja melakukan delete terhadap suatu tabel *language* pada database.

```

DROP TABLE
postgres=# DROP TABLE language CASCADE;
NOTICE: drop cascades to 2 other objects
DETAIL: drop cascades to constraint film_language_id_fkey on table film
drop cascades to constraint film_original_language_id_fkey on table film
DROP TABLE
postgres=#

```

Gambar 3.5.3 Tabel *language* dihapus

```

postgres=# \d
List of relations

```

Schema	Name	Type	Owner
public	actor	table	postgres
public	address	table	postgres
public	category	table	postgres
public	city	table	postgres
public	country	table	postgres
public	customer	table	postgres
public	film	table	postgres
public	film_actor	table	postgres
public	film_category	table	postgres
public	inventory	table	postgres
public	payment	table	postgres
public	payment_p2007_01	table	postgres
public	payment_p2007_02	table	postgres
public	payment_p2007_03	table	postgres
public	payment_p2007_04	table	postgres
public	payment_p2007_05	table	postgres
public	payment_p2007_06	table	postgres
public	rental	table	postgres
public	staff	table	postgres
public	store	table	postgres

```

(20 rows)

postgres=#

```

Gambar 3.5.4 Daftar tabel setelah penghapusan

Melakukan proses pemulihan basis data dari file-file yang sudah di *backup* dengan perintah-perintah sebagai berikut.

```

marcelryan@DESKTOP-E9JJPRD:~$ sudo systemctl stop postgresql@14-main
marcelryan@DESKTOP-E9JJPRD:~$ sudo mv /var/lib/postgresql/14/main/pg_wal ~
marcelryan@DESKTOP-E9JJPRD:~$ sudo rm -rf /var/lib/postgresql/14/main
marcelryan@DESKTOP-E9JJPRD:~$ sudo mkdir /var/lib/postgresql/14/main
marcelryan@DESKTOP-E9JJPRD:~$ sudo cp -a ~/database_backup/. /var/lib/postgresql/14/main/
marcelryan@DESKTOP-E9JJPRD:~$ sudo chown postgres:postgres /var/lib/postgresql/14/main
marcelryan@DESKTOP-E9JJPRD:~$ sudo chmod 700 /var/lib/postgresql/14/main
marcelryan@DESKTOP-E9JJPRD:~$ sudo rm -rf /var/lib/postgresql/14/main/pg_wal
marcelryan@DESKTOP-E9JJPRD:~$ sudo cp -a ~/pg_wal /var/lib/postgresql/14/main/pg_wal
marcelryan@DESKTOP-E9JJPRD:~$ sudo nano /etc/postgresql/14/main/postgresql.conf

```

Gambar 3.5.5 Konfigurasi *pitr*

```

restore_command = 'cp ~/database_archive/%f %p'
# placeholders: %f = file name, %p = path
# e.g. 'cp /mnt/secondary/%f %p'
#archive_cleanup_command = '' # command to execute after archiving
#recovery_end_command = '' # command to execute after recovery

```

Gambar 3.5.6 Konfigurasi variabel *restore_command*

Berikut adalah isi dari *recovery target* dimana hanya variabel *recovery_target_time* yang diisi dikarenakan pemulihan basis data ingin dilakukan pada saat data basis data belum mengalami kerusakan di waktu tertentu.

```
# - Recovery Target -

# Set these only when performing a targeted recovery.

#recovery_target = ''           # 'immediate' to end recovery as soon as a
                                # consistent state is reached
                                # (change requires restart)
#recovery_target_name = ''      # the named restore point to which recovery will proceed
                                # (change requires restart)
recovery_target_time = '2023-11-28 21:29:00.000000' # the time stamp up to which recovery will proceed
                                # (change requires restart)
#recovery_target_xid = ''       # the transaction ID up to which recovery will proceed
                                # (change requires restart)
#recovery_target_lsn = ''       # the WAL LSN up to which recovery will proceed
                                # (change requires restart)
#recovery_target_inclusive = on # Specifies whether to stop:
                                # just after the specified recovery target (on)
                                # just before the recovery target (off)
                                # (change requires restart)
#recovery_target_timeline = 'latest' # 'current', 'latest', or timeline ID
                                # (change requires restart)
#recovery_target_action = 'pause' # 'pause', 'promote', 'shutdown'
                                # (change requires restart)
```

Gambar 3.5.7 Konfigurasi variabel *recovery_target_time*

```
marcelryan@DESKTOP-E9JJPRD:~$ sudo touch /var/lib/postgresql/14/main/recovery.signal
marcelryan@DESKTOP-E9JJPRD:~$ sudo systemctl start postgresql@14-main
```

Gambar 3.5.8 Recover database

```
2023-11-28 21:31:49.521 +07 [3186] LOG:  starting PostgreSQL 14.9 (Ubuntu 14.9-0ubuntu0.22.04.1) on x86_64-pc-li
nux-gnu, compiled by gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, 64-bit
2023-11-28 21:31:49.521 +07 [3186] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2023-11-28 21:31:49.525 +07 [3186] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-11-28 21:31:49.535 +07 [3187] LOG:  database system was interrupted; last known up at 2023-11-28 21:28:42 +
07
cp: cannot stat '~/database_archive/00000002.history': No such file or directory
2023-11-28 21:31:50.263 +07 [3187] LOG:  starting point-in-time recovery to 2023-11-28 21:29:00+07
cp: cannot stat '~/database_archive/00000001000000000000000000000004': No such file or directory
2023-11-28 21:31:50.274 +07 [3187] LOG:  redo starts at 0/4000028
2023-11-28 21:31:50.278 +07 [3187] LOG:  consistent recovery state reached at 0/4000138
2023-11-28 21:31:50.279 +07 [3186] LOG:  database system is ready to accept read-only connections
cp: cannot stat '~/database_archive/00000001000000000000000000000005': No such file or directory
2023-11-28 21:31:50.280 +07 [3187] LOG:  recovery stopping before commit of transaction 1411, time 2023-11-28 21
:29:16.755837+07
2023-11-28 21:31:50.280 +07 [3187] LOG:  pausing at the end of recovery
2023-11-28 21:31:50.280 +07 [3187] HINT:  Execute pg_wal_replay_resume() to promote.
(END)
```

Gambar 3.5.9 Transaction *log* saat *recovery*

List of relations			
Schema	Name	Type	Owner
public	actor	table	postgres
public	address	table	postgres
public	category	table	postgres
public	city	table	postgres
public	country	table	postgres
public	customer	table	postgres
public	film	table	postgres
public	film_actor	table	postgres
public	film_category	table	postgres
public	inventory	table	postgres
public	language	table	postgres
public	payment	table	postgres
public	payment_p2007_01	table	postgres
public	payment_p2007_02	table	postgres
public	payment_p2007_03	table	postgres
public	payment_p2007_04	table	postgres
public	payment_p2007_05	table	postgres
public	payment_p2007_06	table	postgres
public	rental	table	postgres
public	staff	table	postgres
public	store	table	postgres
(21 rows)			

Gambar 3.5.10 Daftar tabel setelah *recover* berhasil

Dapat dilihat bahwa proses *recover* menggunakan metode *point-in-time recovery* telah berhasil dimana pada *transaction log* proses *redo* dimulai pada 0/4000028 dan proses *recovery* berhasil mencapai *consistent recovery state* pada 0/4000138.

4. Pembagian Kerja

NIM	Nama	Bagian
13521103	Aulia Mey Diva Annandya	- Implementasi <i>Two Phase Locking</i>
13521107	Jericho Russel Sebastian	- Implementasi MVCC - Implementasi <i>Two Phase Locking</i>
13521127	Marcel Ryan Antony	- Eksplorasi <i>transaction isolation</i> - Eksplorasi <i>recovery</i>
13521141	Edia Zaki Naufal Ilman	- Implementasi OCC

Referensi

<https://www.digitalocean.com/community/tutorials/how-to-set-up-continuous-archiving-and-perform-point-in-time-recovery-with-postgresql-12-on-ubuntu-20-04>

<https://www.postgresql.org/docs/14/continuous-archiving.html>

<https://www.postgresql.org/docs/14/wal-intro.html>