

**IF3140 MANAJEMEN BASIS DATA**

**MEKANISME CONCURRENCY CONTROL DAN RECOVERY**



**K03 Kelompok 06**

Anggota :

Angger Ilham Amanullah	13521001
Haikal Ardzi Shofiyyurrohman	13521012
Eunice Sarah Siregar	13521013
Hidayatullah Wildan Ghaly Buchary	13521015

**Teknik Informatika**

**Sekolah Teknik Elektro dan Informatika**

**Institut Teknologi Bandung**

**2023**

## Daftar Isi

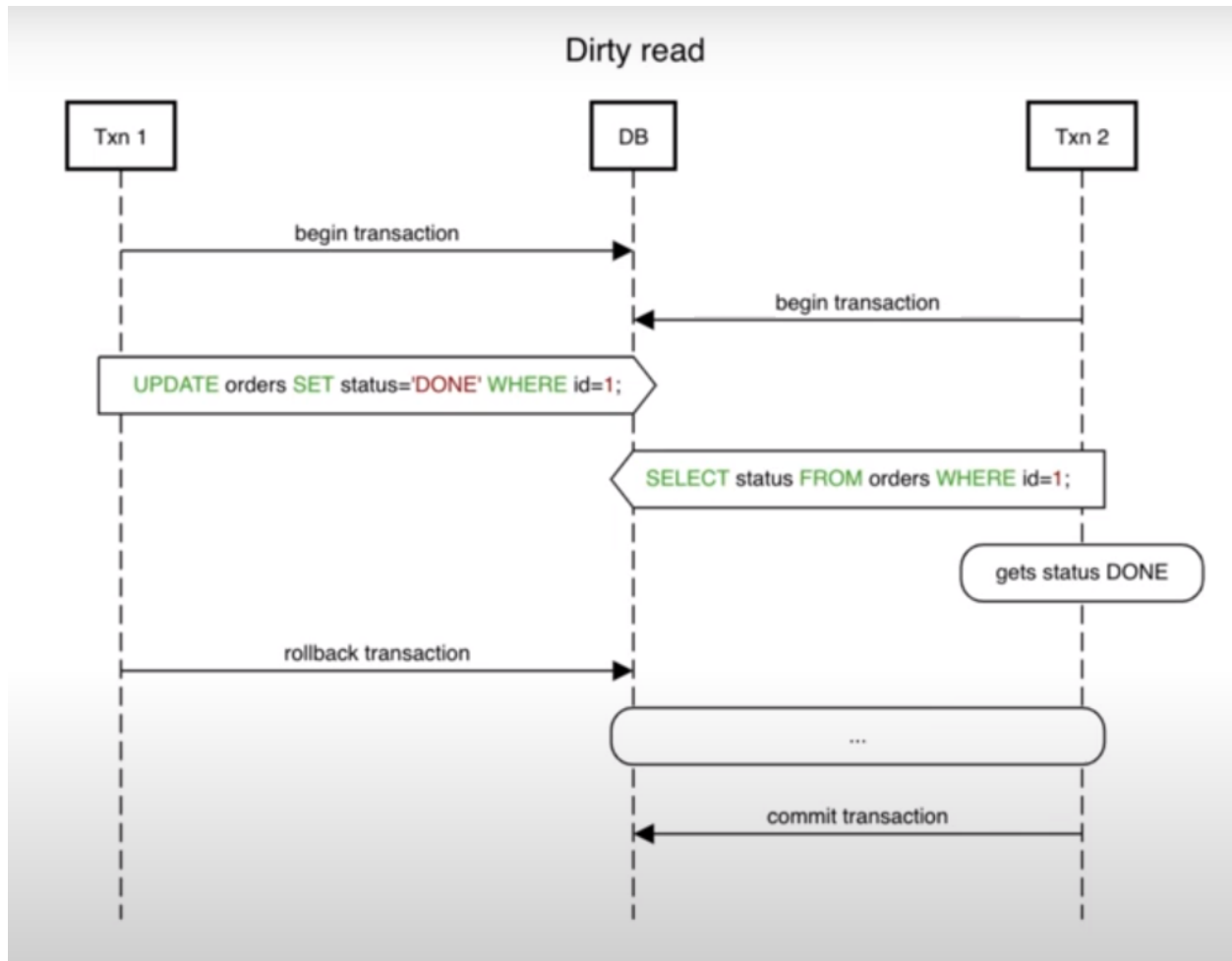
<b>Daftar Isi</b>	<b>1</b>
<b>1. Eksplorasi Transaction Isolation</b>	<b>2</b>
a. Serializable	5
b. Repeatable Read	9
c. Read Committed	12
d. Read Uncommitted	15
<b>2. Implementasi Concurrency Control Protocol</b>	<b>16</b>
a. Two-Phase Locking (2PL)	16
b. Optimistic Concurrency Control (OCC)	26
c. Multiversion Timestamp Ordering Concurrency Control (MVCC)	36
<b>3. Eksplorasi Recovery</b>	<b>42</b>
a. Write-Ahead Log	42
b. Continuous Archiving	42
c. Point-in-Time Recovery	43
d. Simulasi Kegagalan pada PostgreSQL	43
<b>4. Pembagian Kerja</b>	<b>50</b>
<b>Referensi</b>	<b>51</b>

# 1. Eksplorasi Transaction Isolation

## a. Dirty Read

*Dirty read* adalah situasi dalam database transaction di mana suatu transaksi membaca data yang telah dimodifikasi oleh transaksi lain, tetapi perubahan tersebut belum di-commit. Ini dapat menyebabkan ketidakseimbangan data dan biasanya dianggap sebagai perilaku yang tidak diinginkan dalam sistem database.

Contohnya ketika transaksi 1 mengupdate data A, lalu transaksi 2 mengambil data A, namun, transaksi 2 mengambil data yang sudah diupdate oleh transaksi 1. Ketika transaksi 1 ternyata rollback, maka seharusnya data A tidak terjadi update, akan tetapi transaksi 2 tetap memproses data A yang sudah terupdate sebelumnya oleh transaksi 1



### **b. Lost Update**

*Lost update* adalah sebuah kondisi dalam database transactions di mana dua transaksi atau lebih bersaing untuk mengupdate nilai suatu data yang sama, dan satu atau lebih dari perubahan tersebut "hilang" atau tidak diterapkan. Kondisi ini bisa menyebabkan kehilangan informasi yang seharusnya disimpan dalam database. Contoh skenario dari lost update

1. Data Awal:
  - $A = 10$
2. Transaksi T1:
  - T1 membaca nilai data ( $A = 10$ ).
  - T1 menghitung dan menambahkan 1 ( $A = 11$ ).
  - T1 belum di-commit.
3. Transaksi T2:
  - T2 membaca nilai data ( $A = 10$ ).
  - T2 menghitung dan menambahkan 1 ( $A = 11$ ).
  - T2 di-commit, mengubah nilai A menjadi 11.

Dalam skenario ini, nilai A yang diharapkan adalah 12 ( $10 + 1 + 1$ ), tetapi akhirnya nilainya menjadi 11.

### **c. Non-Repeatable Read**

*Non-repeatable read* adalah kondisi yang terjadi ketika satu transaksi membaca data, kemudian transaksi lain mengubah data tersebut, dan ketika transaksi pertama membaca data lagi, nilainya sudah berubah. Contoh Non-Repeatable Read:

1. Data Awal:
  - $A = 10$
2. Transaksi T1:
  - T1 membaca nilai data A dan mendapatkan nilai 10.
  - Sebelum T1 di-commit, transaksi T2 melakukan perubahan pada nilai A.
3. Transaksi T2:
  - T2 mengubah nilai A menjadi 20 dan di-commit.
4. Transaksi T1:
  - T1 kembali dan mencoba membaca nilai data A lagi.
  - Kali ini, nilai yang dibaca adalah 20, bukan 10 seperti sebelumnya.

Dari proses tersebut didapatkan bahwa Transaksi T1 mengalami hasil yang berbeda saat membaca data yang sama lebih dari sekali dan perubahan nilai A oleh T2 telah menyebabkan hasil yang tidak konsisten pada pembacaan kedua oleh T1.

#### **d. Phantom Read**

*Phantom Read* adalah peristiwa ketika sebuah transaksi melihat baris atau tuple yang baru muncul atau menghilang selama eksekusinya, menciptakan ilusi hantu atau *phantom*. Contoh untuk memahami konsep *phantom read*:

1. Data Awal:
  - Misalkan kita memiliki tabel dengan dua baris (A=10 dan B=20).
2. Transaksi T1:
  - T1 membaca semua baris yang memenuhi suatu kondisi (misalnya, semua baris dengan nilai lebih dari 5).
3. Transaksi T2:
  - Saat T1 berlanjut, T2 menyisipkan baris baru (C=15) yang memenuhi kondisi T1.
4. Transaksi T1:
  - T1 membaca kembali semua baris yang memenuhi kondisi yang sama.
  - Kali ini, T1 melihat baris "hantu" yang sebelumnya tidak ada, yaitu baris dengan nilai C=15.

Akibat dari proses transaksi tersebut terjadi fenomena *phantom* karena T1 melihat baris yang muncul setelah pembacaan pertama, menciptakan ilusi bahwa baris tersebut adalah hantu.

#### **e. Serialization Anomaly**

*Serialization anomaly* adalah hasil dari keberhasilan melakukan sekelompok transaksi tidak konsisten dengan semua urutan yang mungkin untuk menjalankan transaksi tersebut satu per satu. Sebagai contoh, satu transaksi membaca jumlah total field untuk pesanan dengan status 'BARU' dan transaksi lainnya melakukan hal yang sama tetapi untuk status 'GAGAL'. Setelah itu, setiap transaksi akan membuat record baru dengan status yang berlawanan, tetapi dengan total field yang sama dengan jumlah yang terbaca. Setelah melakukan keduanya, tidak ada kemungkinan urutan berurutan dari transaksi-transaksi ini yang mengarah pada hasil yang dilakukan karena transaksi-transaksi tersebut bersifat *cyclic dependent* pada hasil satu sama lain.

Isolation Level	Dirty Read	Non-Repeatable Read	Lost Update	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Not Possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not Possible	Not possible	Not possible

Not possible maksudnya tidak diperbolehkan ada

### a. Serializable

Serializable merupakan salah satu isolasi transaksi yang paling ketat. Isolasi transaksi ini mengemulasi eksekusi transaksi serial dan pada kenyataannya isolasi transaksi ini mirip dengan cara kerja repeatable read. Dengan melakukan serializable dapat menghindari dari anomali, sangat penting untuk membaca data dari permanen user table sampai transaksi yang akan dibaca berhasil melakukan *commit*.

Untuk menjamin kebenaran dari isolasi transaksi ini, PostgreSQL menggunakan *predicate locking* dengan mempertahankan *lock* sampai diizinkan untuk melakukan perubahan yang memiliki pengaruh pada hasil akhir transaksi. Namun, predicate locking tidak akan menyebabkan *deadlock* karena tidak melakukan blocking.

Dengan menggunakan serializable akan memudahkan dalam pengembangan. Transaksi yang berhasil di commit akan memiliki efek yang sama ketika menjalankan

satu transaksi dalam waktu yang sama. Untuk read/ write akan membutuhkan cost seperti melakukan transaksi ulang ketika kegagalan serialisasi.

Dalam mengoptimalkan kinerja serializable transaction, dapat mempertimbangkan hal-hal berikut:

- Mendeklarasikan transaksi sebagai READ ONLY
- Mengontrol jumlah koneksi yang aktif menggunakan connection pool
- Jangan menggunakan single transaction secara berlebihan
- Jangan membiarkan koneksi "idle in transaction" lebih lama dari yang diperlukan
- Eliminasi eksplisit lock, SELECT FOR UPDATE, dan SELECT FOR SHARE ketika sudah tidak dibutuhkan

T1	T2	Keterangan
<pre>mbd=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN mbd=# select * from sayur;  id   sayur   price -----+-----+-----  2   Bayam   3000  3   Tomat   7000  4   Kangkung   2500  5   Brokoli   6000  1   Wortel   3000 (5 rows)</pre>	<pre>mbd=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN mbd=# select * from sayur;  id   sayur   price -----+-----+-----  2   Bayam   3000  3   Tomat   7000  4   Kangkung   2500  5   Brokoli   6000  1   Wortel   3000 (5 rows)</pre>	Transaksi 1 dan 2 memulai dengan isolation level serializable dan diikuti read untuk masing-masing transaksi
<pre>mbd=# update sayur set price = 7000 WHERE id = 1; UPDATE 1</pre>		Transaksi 1 melakukan UPDATE untuk tabel sayur pada id = 1, tanpa melakukan commit.
	<pre>mbd=# update sayur set price = 8000 where id = 1; ERROR:  could not serialize access due to concurrent update mbd=# mbd=#</pre>	Transaksi 2 melakukan UPDATE untuk tabel sayur pada id = 1, tetapi terjadi deadlock sehingga transaksi 2 menunggu transaksi 1 selesai
<pre>mbd=# commit; COMMIT</pre>		Transaksi 1 melakukan commit untuk mengakhiri transaksi
	<pre>ERROR:  could not serialize access due to concurrent update mbd=# mbd=#</pre>	Transaksi 2 terjadi error karena tidak ekuivalen dengan

		transaksi serialize
--	--	---------------------

Pengujian non repeatable read:

T1		
<pre>if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN if3140_k03_g06=# SELECT * FROM fruit; id   name   price ---+-----+----- 1   apple   10000 3   avocado   20000 4   manggo   15000 5   coconut   40000 2   banana   3000 (5 rows)  if3140_k03_g06=# SELECT * FROM fruit; id   name   price ---+-----+----- 1   apple   10000 3   avocado   20000 4   manggo   15000 5   coconut   40000 2   banana   3000 (5 rows)</pre>		
T2		



```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
if3140_k03_g06=*# SELECT * FROM fruit;
  id |  name  | price
-----+-----+-----
   1 | apple  | 10000
   3 | avocado | 20000
   4 | manggo | 15000
   5 | coconut | 40000
   2 | banana |  3000
(5 rows)

if3140_k03_g06=*# UPDATE fruit SET price = 8000 WHERE id = 2;
UPDATE 1
if3140_k03_g06=*# end;
COMMIT
if3140_k03_g06=#

```

Pengujian phantom read:

T1
<pre> if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE; BEGIN if3140_k03_g06=*# SELECT COUNT(*) FROM fruit;  count -----       7 (1 row)  if3140_k03_g06=*# SELECT COUNT(*) FROM fruit;  count -----       7 (1 row)  if3140_k03_g06=*# END; COMMIT if3140_k03_g06=#   </pre>
T2

```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
if3140_k03_g06=## SELECT COUNT(*) FROM fruit;
count
-----
      7
(1 row)

if3140_k03_g06=## INSERT INTO fruit (id, name, price) VALUES (8, 'melon', 35
000);
INSERT 0 1
if3140_k03_g06=## END;
COMMIT
if3140_k03_g06=# |

```

## b. Repeatable Read

Repeatable read isolation level hanya menunjukkan data yang sudah di commit sebelum transaksi di mulai, perubahan yang belum di commit tidak akan diperlihatkan. Ketika melakukan operasi UPDATE, DELETE, MERGE, SELECT FOR UPDATE, dan SELECT FOR SHARE, dengan repeatable read akan mencari target yang sudah dilakukan commit ketika transaksi dimulai. Namun, ketika target sudah dilakukan perubahan oleh transaksi lain, maka repeatable read akan menunggu sampai transaksi yang pertama kali melakukan perubahan pada baris tersebut melakukan commit atau rollback. Ketika transaksi pertama melakukan rollback, maka perubahan yang dilakukan akan dibatalkan dan transaksi repeatable read dapat melanjutkan operasinya pada baris yang ditemukan pada awal transaksi.

T1	T2	Keterangan
<pre> mbd=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN mbd==# select * from sayur; id   sayur   price ---+---+---  2   Bayam   3000  3   Tomat   7000  4   Kangkung   2500  5   Brokoli   6000  1   Wortel   7000 (5 rows) </pre>	<pre> mbd=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN mbd==# select * from sayur; id   sayur   price ---+---+---  2   Bayam   3000  3   Tomat   7000  4   Kangkung   2500  5   Brokoli   6000  1   Wortel   7000 (5 rows) </pre>	Transaksi 1 dan 2 memulai transaksi isolation level repeatable read dengan diikuti read untuk masing-masing transaksi
<pre> mbd==# update sayur set price = 1000 WHERE id = 2; UPDATE 1 </pre>		Transaksi 1 melakukan update pada tabel sayur dengan id = 2

	<code>mbd=# update sayur set price = 5000 WHERE id = 2;</code>	Transaksi 2 melakukan update pada tabel sayur dengan id = 2, tetapi terjadi deadlock sehingga transaksi 2 menunggu transaksi 1 selesai
<code>mbd=*# end; COMMIT</code>		Transaksi 1 melakukan commit untuk mengakhiri transaksi
	<code>ERROR: could not serialize access due to concurrent update mbd=#</code>	Pada transaksi 2 terjadi error karena terdapat update yang bersifat concurrent

Pengujian non repeatable read:

T1
<pre>if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN if3140_k03_g06=*# SELECT * FROM fruit;  id   name     price -----+-----+-----   1   apple    10000   3   avocado   20000   4   manggo   15000   5   coconut   40000   2   banana    8000 (5 rows)  if3140_k03_g06=*# SELECT * FROM fruit;  id   name     price -----+-----+-----   1   apple    10000   3   avocado   20000   4   manggo   15000   5   coconut   40000   2   banana    8000 (5 rows)</pre>
T2

```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
if3140_k03_g06=# SELECT * FROM fruit;
  id | name   | price
-----+-----+-----
   1 | apple  | 10000
   3 | avocado | 20000
   4 | manggo | 15000
   5 | coconut | 40000
   2 | banana |  8000
(5 rows)

if3140_k03_g06=# UPDATE fruit SET price = 3000 WHERE id = 2;
UPDATE 1
if3140_k03_g06=# end;
COMMIT
if3140_k03_g06=# |

```

Hasil dari transaksi berbeda karena adanya ketergantungan terhadap urutan query yang akan digunakan, meskipun transaksi 1 duluan datang. Hal inilah yang menyebabkan terjadinya masalah terhadap konsistensi data

Pengujian phantom read:

T1
<pre> if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN if3140_k03_g06=# SELECT COUNT(*) FROM fruit;   count -----       6 (1 row)  if3140_k03_g06=# SELECT COUNT(*) FROM fruit;   count -----       6 (1 row)  if3140_k03_g06=# END; COMMIT if3140_k03_g06=#   </pre>
T2

```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
if3140_k03_g06=# SELECT COUNT(*) FROM fruit;
count
-----
      6
(1 row)

if3140_k03_g06=# INSERT INTO fruit (id, name, price) VALUES (7, 'anas', 5000);
INSERT 0 1
if3140_k03_g06=# END;
COMMIT
if3140_k03_g06=# |

```

### c. Read Committed

Read committed merupakan default isolation dalam PostgreSQL. Ketika sebuah transaksi menerapkan read committed, SELECT query akan memperlihatkan data yang sudah di commit sebelum query dimulai. SELECT query akan memperlihatkan snapshot dari database, tetapi operasi ini juga memperlihatkan efek dari update sebelumnya yang sudah dieksekusi oleh transaksi.

UPDATE, DELETE, SELECT FOR UPDATE, dan SELECT FOR SHARE memiliki perilaku yang sama dengan SELECT dalam mencari baris target dengan menargetkan baris yang sudah di commit saat command start time. Namun, ada kemungkinan target row sudah di update oleh transaksi lain ketika sudah ditemukan. Dalam kasus ini, akan terjadi updater yang akan menunggu update pertama transaksi untuk di commit atau rollback. Ketika update pertama melakukan rollback, maka efeknya akan dinegasikan dan updater kedua dapat melanjutkan dengan memperbarui baris yang ditemukan semula. Namun, ketika updater pertama melakukan commit, updater kedua akan menghiraukan baris jika update pertama menghapusnya. Dalam read committed, setiap baris akan diajukan untuk insersi baik insert atau update. Jika sebuah konflik dengan transaksi lain yang menyebabkan tidak dapat melakukan INSERT, operasi UPDATE akan mempengaruhi baris tersebut walaupun mungkin tidak ada versi baris tersebut yang terlihat secara konvensional oleh perintah.

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

<pre>mbd=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED; BEGIN mbd=# select * from sayur; id   sayur   price ---+-----+----- 3   Tomat   7000 5   Brokoli   6000 1   Wortel   7000 2   Bayam   1000 (4 rows)</pre>	<pre>mbd=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED; BEGIN mbd=# select * from sayur; id   sayur   price ---+-----+----- 3   Tomat   7000 5   Brokoli   6000 1   Wortel   7000 2   Bayam   1000 (4 rows)</pre>	Transaksi 1 dan 2 memulai transaksi isolation level read committed dengan diikuti read untuk masing-masing transaksi
<pre>mbd=# update sayur set price = 8000 WHERE id = 3; UPDATE 1</pre>		Transaksi 1 melakukan update pada tabel sayur dengan id = 3, tanpa melakukan commit
	<pre>mbd=# delete from sayur where id = 3; DELETE 1</pre>	Transaksi 2 melakukan delete pada tabel sayur dengan id = 3
<pre>mbd=# commit; COMMIT mbd=# select * from sayur; id   sayur   price ---+-----+----- 5   Brokoli   6000 1   Wortel   7000 2   Bayam   1000 3   Tomat   8000 (4 rows)</pre>		Transaksi 1 melakukan commit. Terlihat pada gambar, adanya perubahan pada tabel sayur dengan id = 3
	<pre>mbd=# commit; COMMIT mbd=# select * from sayur; id   sayur   price ---+-----+----- 5   Brokoli   6000 1   Wortel   7000 2   Bayam   1000 (3 rows)</pre>	Transaksi 2 melakukan commit. Terlihat pada gambar, adanya perubahan tabel sayur dengan penghapusan id = 3.
<pre>mbd=# select * from sayur; id   sayur   price ---+-----+----- 5   Brokoli   6000 1   Wortel   7000 2   Bayam   1000 (3 rows)</pre>		

Pengujian non repeatable read:

T1

```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
if3140_k03_g06=## SELECT * FROM fruit;
  id |  name  | price
-----+-----+-----
   1 | apple  | 10000
   2 | banana |  5000
   3 | avocado | 20000
   4 | manggo | 15000
   5 | coconut | 40000
(5 rows)

if3140_k03_g06=## SELECT * FROM fruit;
  id |  name  | price
-----+-----+-----
   1 | apple  | 10000
   3 | avocado | 20000
   4 | manggo | 15000
   5 | coconut | 40000
   2 | banana |  8000
(5 rows)

if3140_k03_g06=## end;
COMMIT
if3140_k03_g06=# |

```

**T2**

```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
if3140_k03_g06=## SELECT * FROM fruit;
  id |  name  | price
-----+-----+-----
   1 | apple  | 10000
   2 | banana |  5000
   3 | avocado | 20000
   4 | manggo | 15000
   5 | coconut | 40000
(5 rows)

if3140_k03_g06=## UPDATE fruit SET price = 8000 WHERE id = 2;
UPDATE 1
if3140_k03_g06=## end;
COMMIT
if3140_k03_g06=#

```

Pengujian phantom read:

**T1**

```

if3140_k03_g06=#
if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
if3140_k03_g06=# SELECT COUNT(*) FROM fruit;
count
-----
      5
(1 row)

if3140_k03_g06=# SELECT COUNT(*) FROM fruit;
count
-----
      6
(1 row)

if3140_k03_g06=# END;
COMMIT
if3140_k03_g06=# |

```

T2

```

if3140_k03_g06=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
if3140_k03_g06=# SELECT COUNT(*) FROM fruit;
count
-----
      5
(1 row)

if3140_k03_g06=# INSERT INTO fruit (id, name, price) VALUES (6, 'peer', 550
00);
INSERT 0 1
if3140_k03_g06=# END;
COMMIT
if3140_k03_g06=# |

```

#### d. Read Uncommitted

Read uncommitted adalah derajat isolasi level 0 dengan mengizinkan read untuk perubahan walaupun belum dilakukan commit. Namun, dengan adanya read uncommitted ini akan terjadi dirty read, yaitu membaca transaksi yang belum di commit. Pada PostgreSQL juga tidak mengimplementasikan derajat isolasi ini.



## 2. Implementasi Concurrency Control Protocol

### a. Two-Phase Locking (2PL)

Locking adalah sebuah prosedur yang digunakan untuk mengendalikan akses bersamaan ke data. Ketika sebuah transaksi sedang mengakses database, sebuah lock mungkin menolak akses ke transaksi lain untuk mencegah hasil yang salah. Ada dua macam lock, yaitu shared lock dan exclusive lock yang harus digunakan sebelum melakukan akses membaca ataupun menulis terhadap database. Penggunaan lock ini adalah untuk menjaga konsistensi data didalam database. Jika sebuah transaksi mempunyai sebuah shared lock pada sebuah item data, transaksi tersebut dapat membaca item tapi tidak dapat mengubah datanya. Jika sebuah transaksi mempunyai sebuah exclusive lock pada sebuah item data, transaksi tersebut dapat membaca dan mengubah item data. Lock digunakan dengan cara sebagai berikut:

- Transaksi apapun yang membutuhkan akses pada sebuah item data harus melakukan *lock* terhadap item tersebut, meminta shared lock untuk akses membaca saja atau sebuah exclusive lock untuk akses membaca dan menulis.
- Jika item belum dikunci oleh transaksi lain, lock tersebut akan dikabulkan
- Jika item sedang dikunci, DBMS menentukan apakah permintaan ini compatible dengan lock saat ini. Jika diminta shared lock pada sebuah item yang sudah mempunyai shared lock terpasang padanya, permintaan itu akan dikabulkan. Selain itu, transaksi harus menunggu sampai lock yang ada terlepas.
- Sebuah transaksi lanjut memegang lock sampai transaksi tersebut melepasnya baik pada waktu eksekusi ataupun pada waktu transaksi tersebut berakhir (abort atau commit). Efek operasi tulis akan terlihat pada transaksi lain hanya pada waktu exclusive lock telah dilepas.

Two Phase Locking adalah sebuah transaksi yang mengikuti protokol two-phase locking jika semua operasi locking mendahului operasi unlock pertama pada transaksi. Aturan-aturannya adalah sebagai berikut :

- Sebuah transaksi harus mendapatkan sebuah lock pada item sebelum beroperasi pada item tersebut. Lock tersebut bisa berupa baca atau tulis, tergantung dari tipe akses yang dibutuhkan.

- Sebelum transaksi melepaskan sebuah *lock*, transaksi tersebut tidak akan pernah mendapatkan lock baru lainnya.

Program *Two Phase Locking* telah diimplementasikan untuk melakukan simulasi kontrol konkurensi dalam sistem basis data. Program ini menerima urutan operasi kontrol konkurensi sebagai input, yang mencakup operasi pembacaan (R), penulisan (W), dan komit (C) dari transaksi. Setiap operasi direpresentasikan dalam bentuk objek dengan informasi seperti jenis operasi, ID transaksi, dan nama tabel.

Program menggunakan mekanisme penguncian dua fase (Two-Phase Locking) yang melibatkan penguncian bersama (SL) dan penguncian eksklusif (XL). Selama jalannya program, setiap operasi dicek untuk memastikan keabsahan, seperti nama tabel yang valid dan komit yang sesuai dengan operasi pembacaan/tulisan.

Selama eksekusi, program mengelola tabel penguncian bersama (SL\_table) dan penguncian eksklusif (XL\_table), serta antrian transaksi yang menunggu penguncian. Jika sebuah transaksi dapat memperoleh penguncian, operasi berhasil dieksekusi dan hasilnya direkam.

Setelah selesai, program menghasilkan output berupa urutan operasi yang berhasil dieksekusi. Hasil ini dapat digunakan untuk memahami bagaimana kontrol konkurensi diimplementasikan dalam sistem basis data untuk memastikan konsistensi dan integritas data. Selain itu, program juga menyediakan riwayat transaksi dalam bentuk teks atau format JSON untuk memudahkan analisis dan pemahaman. Berikut adalah kode programnya dalam bahasa python.

#### 1. Inisiasi kelas

```
class TwoPhaseLocking:
    def __init__(self, input_seq: str) -> None:
        self.SL_table = {}
        self.XL_table = {}
        self.seq = []
        self.timestamp = []
        self.transaction_history = []
        self.result = []
        self.queue = []
```

```
self.process_input_sequence(input_seq)
```

## 2. Proses menerima input

```
def process_input_sequence(self, input_seq: str):
    if input_seq.endswith(';'):
        input_seq = input_seq[:-1]

    input_seq = input_seq.split(';')
    for input in input_seq:
        input = input.strip()
        self.validate_and_store_operation(input)

    self.verify_commit_operations()
    self.verify_table_names()

def validate_and_store_operation(self, input: str):
    operation = input[0]
    if operation in ('R', 'W'):
        self.store_read_write_operation(input, operation)
    elif operation == 'C':
        self.store_commit_operation(input)
    else:
        raise ValueError("Invalid operation detected")

def store_read_write_operation(self, input: str, operation: str):
    transaction_id = int(input[1])
    table_name = input[3]
    self.seq.append({"operation": operation, "transaction": transaction_id, "table":
table_name})

    if transaction_id not in self.timestamp:
        self.timestamp.append(transaction_id)

def store_commit_operation(self, input: str):
    transaction_id = int(input[1])
```

```

self.seq.append({"operation": 'C', "transaction": transaction_id})

if transaction_id not in self.timestamp:
    raise ValueError("Transaction has no read or write operation")

def verify_commit_operations(self):
    if len([x for x in self.seq if x["operation"] == 'C']) != len(set(self.timestamp)):
        raise ValueError("Missing commit operation")

def verify_table_names(self):
    invalid_tables = [x for x in self.seq if x["operation"] in ('R', 'W') and (len(x["table"]) != 1
or not x["table"].isalpha())]
    if invalid_tables:
        raise ValueError("Invalid table name")

```

### 3. Proses eksklusif lock

```

def XL(self, transaction: int, table: str) -> bool:
    if table in self.SL_table:
        if transaction in self.SL_table[table] and len(self.SL_table[table]) == 1:
            # remove the shared lock
            self.SL_table = {
                k: v for k, v in self.SL_table.items() if v != transaction}
            self.XL_table[table] = transaction
            self.result.append(
                {"operation": "UPL", "transaction": transaction, "table": table})
            self.transaction_history.append({"transaction": transaction, "table": table,
"operation": "UPL", "status": "Success"})
            return True
        else:
            return False
    else:
        if table in self.XL_table:
            if self.XL_table[table] == transaction:
                return True
            else:
                return False

```

```

else:
    self.XL_table[table] = transaction
    self.result.append(
        {"operation": "XL", "transaction": transaction, "table": table})
    self.transaction_history.append({"transaction" : transaction, "table": table,
"operation": "XL", "status": "Success"})
    return True

```

#### 4. Proses share lock

```

def SL(self, transaction: int, table: str) -> bool:
    if table in self.XL_table:
        if self.XL_table[table] == transaction:
            return True
        else:
            return False
    else:
        if table in self.SL_table and transaction in self.SL_table[table]:
            return True
        else: # Check if the table is locked by another shared lock
            # Add the current transaction to the shared lock table
            if table not in self.SL_table:
                self.SL_table[table] = []
            self.SL_table[table].append(transaction)
            self.result.append(
                {"operation": "SL", "transaction": transaction, "table": table})
            self.transaction_history.append({"transaction" : transaction, "table": table,
"operation": "SL", "status": "Success"})
            return True

```

#### 5. Proses membersihkan lock

```

def clear_XL(self, current: dict) -> None:
    if current["transaction"] in self.XL_table.values():
        table = [
            k for k, v in self.XL_table.items() if v == current["transaction"]]

```

```

    for t in table:
        self.result.append(
            {"operation": "UL", "transaction": current["transaction"], "table": t})
        self.transaction_history.append({"transaction" : current["transaction"], "table": t,
"operation": "UL", "status": "Success"})
        self.XL_table = {
            k: v for k, v in self.XL_table.items() if v != current["transaction"]}

def clear_SL(self, current: dict) -> None:
    table = [
        k for k, v in self.SL_table.items() if v == current["transaction"]]
    for t in table:
        self.result.append(
            {"operation": "UL", "transaction": current["transaction"], "table": t})
        self.transaction_history.append({"transaction" : current["transaction"], "table": t,
"operation": "UL", "status": "Success"})
    for k, v in self.SL_table.items():
        if current["transaction"] in v:
            v.remove(current["transaction"])
    self.SL_table = {
        k: v for k, v in self.SL_table.items() if v != []}

```

#### 6. Proses deadlock prevention menggunakan wait-die

```

def run_queue(self) -> None:
    while self.queue:
        transaction = self.queue.pop(0)
        # Check if the table is locked
        if self.XL(transaction["transaction"], transaction["table"]):
            self.result.append(transaction)
            self.transaction_history.append({"transaction" : transaction["transaction"],
"table": transaction["table"], "operation": transaction["operation"], "status": "Success"})
        else:
            self.queue.insert(0, transaction)
            break

def commit(self, current: dict) -> None:

```

```

if current["transaction"] in [x["transaction"] for x in self.queue]:
    self.seq.insert(1, current)
else:
    self.clear_SL(current)
    self.clear_XL(current)
    self.result.append(current)
    self.transaction_history.append({"transaction": current["transaction"], "table": "-",
"operation": "Commit", "status": "Commit"})

def abort(self, current: dict) -> None:
    self.transaction_history.append({"transaction": current["transaction"], "table":
current["table"], "operation": "Abort", "status": "Abort"})
    curr = [x for x in self.result if x["transaction"] == current["transaction"] and (
        x["operation"] == 'R' or x["operation"] == 'W')]
    self.result = [
        x for x in self.result if x["transaction"] != current["transaction"]]
    seq = [x for x in self.seq if x["transaction"] == current["transaction"]]
    self.seq = [
        x for x in self.seq if x["transaction"] != current["transaction"]]
    if current["transaction"] in self.XL_table.values():
        self.XL_table = {
            k: v for k, v in self.XL_table.items() if v != current["transaction"]}
    if current["transaction"] in [x for v in self.SL_table.values() for x in v]:
        for k, v in self.SL_table.items():
            if current["transaction"] in v:
                v.remove(current["transaction"])
    self.SL_table = {
        k: v for k, v in self.SL_table.items() if v != []}

    self.seq.extend(curr)
    self.seq.append(current)
    self.seq.extend(seq)

def wait_die(self, current: dict) -> None:
    if ((current["table"] in self.XL_table and self.timestamp.index(current["transaction"])
< self.timestamp.index(self.XL_table[current["table"]])) or
        (current["table"] in self.SL_table and
all(self.timestamp.index(current["transaction"]) < self.timestamp.index(t) for t in

```

```

self.SL_table[current["table"]] if t != current["transaction"])))
    self.queue.append(current)
    self.transaction_history.append({"transaction": current["transaction"], "table":
current["table"], "operation": current["operation"], "status": "Queue"})
    else:
        self.abort(current)

```

## 7. Proses run dan debugging

```

def run(self) -> None:
    while self.seq:
        self.run_queue()
        index = next((i for i, x in enumerate(self.seq) if x["transaction"] not in [
            y["transaction"] for y in self.queue]), None)
        current = self.seq.pop(index)

        if current["operation"] == 'C':
            self.commit(current)
        elif current["operation"] == 'R' and self.SL(current["transaction"], current["table"]):
            self.result.append(current)
            self.transaction_history.append({"transaction": current["transaction"], "table":
current["table"], "operation": current["operation"], "status": "Success"})
        elif current["operation"] == 'W' and self.XL(current["transaction"], current["table"]):
            self.result.append(current)
            self.transaction_history.append({"transaction": current["transaction"], "table":
current["table"], "operation": current["operation"], "status": "Success"})
        else:
            self.wait_die(current)

    def result_string(self) -> None:
        res = ""
        for r in self.result:
            if r["operation"] == 'C':
                res += f"{r['operation']}{r['transaction']};"
            else:
                res += f"{r['operation']}{r['transaction']}{r['table']};"
        if res[-1] == ';':

```



```

        res = res[:-1]
    return res

def history_string(self):
    str = ""
    for t in self.transaction_history:
        str += f'{t["operation"]} {t["transaction"]} {t["table"] if "table" in t else ""}\n'
    return str

def history_json(self):
    res = []
    for t in self.transaction_history:
        res.append({t["transaction"]: f'{t["operation"]}({t["table"]})'})
    return res

```

#### 8. Main program

```

if __name__ == "__main__":
    try:
        lock = TwoPhaseLocking(input("Enter Concurrency Control Sequence: "))
        lock.run()
        print(lock.result_string())

    except (ValueError, IndexError) as e:
        print("Error: ", e)
        exit(1)

```

#### 9. Contoh masukan dan keluaran

### Concurrency Control Protocol

R1(A)R2(B)W1(A)R1(B)W3(A)W1(B)W2(B)R1(C)C1C2C3C4

Process

☒ Two Phase Locking  
☐ Optimistic Concurrency Control

T1	T2	T3	T4
SL1(A)			
R1(A)			
L1(A)			
W1(A)			
SL1(B)			
R1(B)			
SL1(C)			
R1(C)			
UL1(A)			
C1			
		XL3(A)	
		W3(A)	
		UL3(A)	
		C3	
			XL4(B)
			W4(B)
			UL4(B)
			C4
	SL2(B)		
	R2(B)		
	L2(B)		
	W2(B)		
	UL2(B)		
	C2		

Final Schedule: SL1(A)R1(A)UPL1(A)W1(A)SL1(B)R1(B)SL1(C)R1(C)UL1(A)C1XL3(A)W3(A)UL3(A)C3XL4(B)W4(B)UL4(B)C4SL2(B)R2(B)UL2(B)W2(B)UL2(B)C2

### Concurrency Control Protocol

R1(X)R2(X)W1(X)C1W2(X)C2

Process

☒ Two Phase Locking  
☐ Optimistic Concurrency Control  
☐ Multiversion Timestamp Ordering Concurrency Control

T1	T2
SL1(X)	
R1(X)	
L1(X)	
W1(X)	
UL1(X)	
C1	
	SL2(X)
	R2(X)
	L2(X)
	W2(X)
	UL2(X)
	C2

Final Schedule: SL1(X)R1(X)UPL1(X)W1(X)UL1(X)C1SL2(X)R2(X)UPL2(X)W2(X)UL2(X)C2

### Concurrency Control Protocol

R1(X);R2(X);W1(X);W2(X);W3(X);C1;C2;C3

Process

☒ Two Phase Locking  
☐ Optimistic Concurrency Control  
☐ Multiversion Timestamp Ordering Concurrency Control

T1	T2	T3
SL1(X)		
R1(X)		
L1(X)		
W1(X)		
UL1(X)		
C1		
	SL2(X)	
	R2(X)	
	L2(X)	
	W2(X)	
	UL2(X)	
	C2	
		XL3(X)
		W3(X)
		UL3(X)
		C3

Final Schedule: SL1(X);R1(X);UPL1(X);W1(X);UL1(X);C1;SL2(X);R2(X);UPL2(X);W2(X);UL2(X);C2;XL3(X);W3(X);UL3(X);C3

## b. Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control (OCC) adalah teknik untuk mengelola akses bersamaan ke data dalam sistem manajemen basis data relasional (RDBMS). Hal ini memungkinkan beberapa transaksi untuk membaca dan mengubah data yang sama tanpa mengunci atau memblokir satu sama lain, selama tidak bertentangan. OCC bekerja dengan menetapkan nomor versi atau stempel waktu untuk setiap item data yang dibaca atau dimodifikasi oleh transaksi. Ketika suatu transaksi ingin melakukan perubahannya, ia memeriksa apakah ada transaksi lain yang telah memperbarui item data yang sama sejak membacanya. Jika tidak, komit berhasil dan nomor versi atau stempel waktu bertambah. Jika ya, komit gagal dan transaksi harus dibatalkan dan dicoba lagi.

OCC memiliki beberapa manfaat untuk kinerja dan skalabilitas RDBMS. Pertama, ini mengurangi overhead penguncian dan pemblokiran, yang dapat menyebabkan penundaan, kebuntuan, dan perselisihan. Kedua, meningkatkan throughput dan daya tanggap sistem, karena transaksi dapat dilanjutkan tanpa menunggu kunci atau sumber daya. Ketiga, mendukung tingkat konkurensi dan isolasi yang tinggi, karena transaksi dapat bekerja pada salinan data yang berbeda tanpa mengganggu satu sama lain. Keempat, memungkinkan penyetelan sistem yang lebih

fleksibel dan adaptif, karena tingkat deteksi dan resolusi konflik dapat disesuaikan sesuai dengan beban kerja dan persyaratan aplikasi.

Untuk melakukan simulasi terhadap OCC, dibuat suatu program dalam bahasa python. Program menerima urutan operasi kontrol konkurensi sebagai input, yang mencakup operasi pembacaan (R), penulisan (W), dan komit (C) dari transaksi. Setiap operasi direpresentasikan dalam bentuk objek dengan informasi seperti jenis operasi, ID transaksi, dan nama tabel.

Selama eksekusi program, setiap operasi dibaca dan diproses sesuai dengan protokol kontrol konkurensi OCC. Transaksi-transaksi dibuat dan dilacak, sementara operasi-operasi pembacaan dan penulisan dicatat dalam setiap transaksi. Program juga memvalidasi urutan operasi dan memastikan komit hanya dilakukan setelah semua operasi pembacaan dan penulisan selesai. Selain itu, program mengelola timestamp untuk setiap transaksi dan melakukan validasi terhadap transaksi-transaksi lainnya. Jika sebuah transaksi dapat di-commit, hasilnya direkam dan ditampilkan dalam bentuk urutan operasi yang berhasil dieksekusi. Jika terjadi konflik, transaksi tersebut dianggap gagal (aborted) dan dilakukan rollback.

Hasil akhir dari eksekusi program ditampilkan dalam format yang jelas, mencakup operasi-operasi yang berhasil, status komit, dan transaksi yang gagal (aborted). Warna ditambahkan untuk mempermudah pemahaman hasil eksekusi. Program ini memberikan pemahaman yang baik tentang bagaimana kontrol konkurensi beroperasi dalam lingkungan basis data dan bagaimana OCC mengelola transaksi untuk mencapai konsistensi data. Berikut adalah kode program yang mengimplementasikan OCC.

#### 1. Inisiasi program

```
import math
from Color import*

class Transaction:
    def __init__(self, tx_id):
        self.tx_id = tx_id
        self.reads = []
```

```

self.writes = []
self.timestamps = {
    "start": math.inf,
    "validation": math.inf,
    "finish": math.inf
}

def __str__(self):
    read_set_str = ", ".join(self.reads)
    write_set_str = ", ".join(self.writes)
    color_reset = "\033[0m"
    color_bold = "\033[1m"
    color_green = "\033[92m"
    color_cyan = "\033[96m"

    return (
        f"{color_bold}Transaction {self.tx_num}:{color_reset}\n"
        f"\t{color_cyan}Read Set:{color_reset}"
        f"{color_green}{read_set_str}{color_reset}\n"
        f"\t{color_cyan}Write Set:{color_reset}"
        f"{color_green}{write_set_str}{color_reset}\n"
        f"\t{color_cyan}Timestamps:{color_reset}"
        f"{color_green}{self.timestamps}{color_reset}"
    )

class OCC:
    COMMIT_OPERATION = 'C'
    READ_OPERATION = 'R'
    WRITE_OPERATION = 'W'

    def __init__(self, input_sequence: str) -> None:
        self.current_timestamp = 0
        self.timestamp = []
        self.sequence = []
        self.transactions = {}
        self.result = []
        self.history_transaction = []
        self.rollback_transactions = []

```

```

try:
    if (input_sequence.endswith(';')):
        input_sequence = input_sequence[:-1]

    self.parse_input_sequence(input_sequence)
    self.validate_operations()

except ValueError as e:
    raise ValueError(str(e))

def parse_input_sequence(self, input_sequence: str) -> None:
    for operation_str in input_sequence.split(';'):
        operation_str = operation_str.strip()
        self.parse_operation(operation_str)

def parse_operation(self, operation_str: str) -> None:
    operation = operation_str[0]
    if (operation in {self.READ_OPERATION, self.WRITE_OPERATION}):
        self.handle_read_write_operation(operation, operation_str)
    elif (operation == self.COMMIT_OPERATION):
        self.handle_commit_operation(operation_str)
    else:
        raise ValueError("Invalid operation detected")

def handle_read_write_operation(self, operation: str, operation_str: str) -> None:
    transaction_id = int(operation_str[1])
    table_name = operation_str[3]

    self.sequence.append({"operation": operation, "transaction": transaction_id, "table":
table_name})

    if (transaction_id not in self.timestamp):
        self.timestamp.append(transaction_id)

def handle_commit_operation(self, operation_str: str) -> None:
    transaction_id = int(operation_str[1])
    self.sequence.append({"operation": self.COMMIT_OPERATION, "transaction":

```

```

transaction_id})

def validate_operations(self) -> None:
    if (len([x for x in self.sequence if x["operation"] == self.COMMIT_OPERATION]) !=
len(set(self.timestamp))):
        raise ValueError("Missing commit operation")

    if (any(len(x["table"]) != 1 or not x["table"].isalpha() for x in self.sequence if
x["operation"] in {self.READ_OPERATION, self.WRITE_OPERATION})):
        raise ValueError("Invalid table name")

```

## 2. Proses OCC

```

def read(self, cmd) -> None:
    self.current_timestamp += 1
    transaction_id = cmd["transaction"]

    if (cmd["table"] not in self.transactions[transaction_id].reads):
        self.transactions[transaction_id].reads.append(cmd["table"])

    self.history_transaction.append(
        {"operation": cmd["operation"], "transaction": transaction_id, "table": cmd["table"],
"status": "success"}
    )

def tempwrite(self, cmd) -> None:
    self.current_timestamp += 1
    transaction_id = cmd["transaction"]

    if (cmd["table"] not in self.transactions[transaction_id].writes):
        self.transactions[transaction_id].writes.append(cmd["table"])

    self.history_transaction.append(
        {"operation": cmd["operation"], "transaction": transaction_id, "table": cmd["table"],
"status": "success"}
    )

```

```

def validate(self, cmd) -> None:
    self.current_timestamp += 1
    transaction_id = cmd['transaction']
    self.transactions[transaction_id].timestamps['validation'] = self.current_timestamp
    valid = True

    for other_tx_id in self.transactions.keys():
        if (other_tx_id != transaction_id):
            validation_timestamp_ti =
self.transactions[other_tx_id].timestamps['validation']
            finish_timestamp_ti = self.transactions[other_tx_id].timestamps['finish']
            start_timestamp_tj = self.transactions[transaction_id].timestamps['start']
            validation_timestamp_tj =
self.transactions[transaction_id].timestamps['validation']
            if (validation_timestamp_ti < validation_timestamp_tj):
                if (finish_timestamp_ti < start_timestamp_tj):
                    pass
                elif (finish_timestamp_ti != math.inf and (start_timestamp_tj <
finish_timestamp_ti and finish_timestamp_ti < validation_timestamp_tj)):
                    write_set_ti = self.transactions[other_tx_id].writes
                    read_set_tj = self.transactions[transaction_id].reads
                    is_element_intersect = False
                    for v in write_set_ti:
                        if v in read_set_tj:
                            is_element_intersect = True
                            break
                    if is_element_intersect:
                        valid = False
                        break
                else:
                    valid = False
                    break
    if valid:
        self.commit(cmd)
    else:
        self.handle_aborted_transaction(cmd, transaction_id)

def handle_aborted_transaction(self, cmd, transaction_id) -> None:

```



```

        print(f"{color_bold}Transaction {transaction_id} {color_red} is aborted
{color_reset}")
        self.rollback_transactions.append(transaction_id)
        self.history_transaction.append(
            {"operation": cmd["operation"], "transaction": transaction_id, "status": "aborted"}
        )

def commit(self, cmd) -> None:
    self.current_timestamp += 1
    transaction_id = cmd["transaction"]
    self.transactions[transaction_id].timestamps["finish"] = self.current_timestamp

    for cmds in self.sequence:
        if cmds["transaction"] == transaction_id:
            self.result.append(cmds)

    self.history_transaction.append(
        {"operation": cmd["operation"], "transaction": transaction_id, "status": "commit"}
    )

    self.result.append(
        {"operation": cmd["operation"], "transaction": transaction_id}
    )

def run_rollback(self) -> None:
    while self.rollback_transactions:
        self.current_timestamp += 1
        tx_id = self.rollback_transactions.pop(0)
        self.reset_transaction_attr(tx_id)
        self.replay_transaction_commands(tx_id)

def reset_transaction_attr(self, tx_id) -> None:
    cmd = self.transactions[tx_id]
    cmd.reads = []
    cmd.writes = []
    cmd.timestamps = {"start": self.current_timestamp, "validation": math.inf, "finish":
math.inf}

```

```

def replay_transaction_commands(self, tx_id) -> None:
    cmd_sequence = [cmds for cmds in self.sequence if cmds["transaction"] == tx_id]

    for cmds in cmd_sequence:
        if (cmds["operation"] == self.READ_OPERATION):
            self.read(cmds)
        elif (cmds["operation"] == self.WRITE_OPERATION):
            self.tempwrite(cmds)
        elif (cmds["operation"] == self.COMMIT_OPERATION):
            self.validate(cmds)

    self.current_timestamp += 1

    self.current_timestamp += 1

def run(self) -> None:
    for cmd in self.sequence:
        self.create_transaction(cmd)

        if (cmd["operation"] == self.READ_OPERATION):
            self.read(cmd)
        elif (cmd["operation"] == self.WRITE_OPERATION):
            self.tempwrite(cmd)
        elif (cmd["operation"] == self.COMMIT_OPERATION):
            self.validate(cmd)

    self.current_timestamp += 1

    self.run_rollbacks()

def create_transaction(self, cmd) -> None:
    transaction_id = cmd["transaction"]
    if (transaction_id not in self.transactions):
        self.transactions[transaction_id] = Transaction(transaction_id)
        self.transactions[transaction_id].timestamps["start"] = self.current_timestamp

def __str__(self):

```

```

    res = ""
    for cmd in self.history_transaction:
        if cmd['status'] == 'success':
            res +=
f"{color_bold}{cmd['operation']}{cmd['transaction']}{cmd['table']}{color_reset}\n"
            elif cmd['status'] == 'commit':
                res += f"{color_bold}{cmd['operation']}{cmd['transaction']} -
{color_green}commit{color_reset}\n"
            elif cmd['status'] == 'aborted':
                res += f"{color_bold}{cmd['operation']}{cmd['transaction']} -
{color_red}aborted{color_reset}\n"
    return res

```

### 3. Main program

```

if __name__ == '__main__':
    try:
        occ = OCC(input("Enter Concurrency Control Sequence: "))
        occ.run()
        print(occ)
    except Exception as e:
        print("Error: ", e)

```

### 4. Contoh input output

## Concurrency Control Protocol

R1(A);R2(B);W1(A);R1(B);W3(A);W4(B);W2(B);R1(C);C1;C2;C3;C4

Process

☐ Two Phase Locking  
☒ Optimistic Concurrency Control  
☐ Multiversion Timestamp Ordering Concurrency Control

T1	T2	T3	T4
R1(A)			
	R2(B)		
W1(A)			
R1(B)			
		W3(A)	
			W4(B)
	W2(B)		
R1(C)			
C1			
	C2		
		C3	
			C4

Final Schedule: R1(A) R2(B) W1(A) R1(B) W3(A) W4(B) W2(B) R1(C) C1 - commit C2 - commit C3 - commit C4 - commit

## Concurrency Control Protocol

R1(A);W1(A);R2(A);C1;W2(A);C2;

Process

☐ Two Phase Locking  
☒ Optimistic Concurrency Control  
☐ Multiversion Timestamp Ordering Concurrency Control

T1	T2
R1(A)	
W1(A)	
	R2(A)
C1	
	W2(A)
	C2
	R2(A)
	W2(A)
	C2

Final Schedule: R1(A) W1(A) R2(A) C1 - commit W2(A) C2 - aborted R2(A) W2(A) C2 - commit

## Concurrency Control Protocol

R1(X);R2(X);W1(X);C1;W2(X);C2;

Process

☐ Two Phase Locking  
☒ Optimistic Concurrency Control  
☐ Multiversion Timestamp Ordering Concurrency Control

T1	T2
R1(X)	
	R2(X)
W1(X)	
C1	
	W2(X)
	C2
	R2(X)
	W2(X)
	C2

Final Schedule: R1(X) R2(X) W1(X) C1 - commit W2(X) C2 - aborted R2(X) W2(X) C2 - commit

### c. Multiversion Timestamp Ordering Concurrency Control (MVCC)

Dalam sistem database modern, paralelisme penting untuk memastikan konsistensi dan kinerja data dalam lingkungan di mana banyak transaksi terjadi secara bersamaan. Teknik yang efisien untuk mengelola konkurensi dalam database relasional adalah Multiversion Concurrency Control (MVCC). Pendekatan ini mempertahankan versi catatan yang terpisah, memungkinkan pembaca mengakses snapshot database yang konsisten tanpa diblokir oleh penulisan yang sedang berlangsung.

MVCC menyelesaikan konflik dengan membuat beberapa versi rekaman tanpa menggunakan kunci eksplisit, mengurangi pertikaian kunci, dan meningkatkan kinerja. Hal ini sangat penting terutama dalam lingkungan transaksi tinggi dan konkurensi tinggi di mana perubahan data sering terjadi. MVCC memastikan konsistensi database dengan memastikan bahwa suatu transaksi hanya dapat mengakses versi record yang valid pada saat transaksi, tanpa mempengaruhi pandangan transaksi bersamaan lainnya.

Untuk melakukan simulasi MVCC dibuat program dengan bahasa python. Program dibuat untuk simulasi mekanisme kontrol konkurensi yang memungkinkan transaksi membaca versi data yang sesuai dengan waktu transaksi dimulai. Program menerima urutan operasi kontrol konkurensi sebagai input, termasuk operasi pembacaan (read) dan penulisan (write) dari transaksi. Setiap operasi direpresentasikan dalam bentuk objek dengan informasi seperti ID transaksi, elemen data yang diakses, tindakan (read/write), timestamp, dan versi data.

Selama eksekusi, program mengelola tabel versi untuk setiap elemen data yang diakses oleh transaksi. Jika elemen data belum pernah diakses, program membuat entri baru dalam tabel versi. Jika sudah ada versi sebelumnya, program memeriksa timestamp dan versi untuk menentukan aksi selanjutnya. Program juga dapat melakukan rollback jika terjadi konflik.

Hasil eksekusi program ditampilkan dalam bentuk urutan operasi yang berhasil dieksekusi, mencakup informasi tentang operasi read/write, versi data yang diakses, dan timestamp. Program memberikan pemahaman yang baik tentang bagaimana MVCC bekerja dalam konteks kontrol konkurensi untuk mendukung transaksi bersamaan dan menjaga konsistensi data. Berikut adalah kode program simulasi MVCC dalam python.

```

from collections import deque

COMMIT_OPERATION = "commit"
READ_OPERATION   = "read"
WRITE_OPERATION  = "write"

class MVCC:
    def __init__(self, input_sequence):
        self.counter          = 0
        self.version_table    = {}
        self.sequence         = deque([])
        self.input_sequence   = deque(input_sequence)
        self.transaction_counter = [i for i in range(10)]

    def get_max_version_index_by_write(self, item):
        max_w_timestamp = self.version_table[item][0]["timestamp"][1]
        max_index = 0
        for i in range(len(self.version_table[item])):
            if self.version_table[item][i]["timestamp"][1] > max_w_timestamp:
                max_w_timestamp = self.version_table[item][i]["timestamp"][1]
                max_index = i
        return max_index

    def write(self, tx, item):
        if item not in self.version_table.keys():
            self.version_table[item] = []
            self.version_table[item].append({'tx': tx, 'timestamp': (
                self.transaction_counter[tx], self.transaction_counter[tx]), 'version':
self.transaction_counter[tx]})
            self.sequence.append({'tx': tx, 'item': item, 'action': 'write', 'timestamp': (
                self.transaction_counter[tx], self.transaction_counter[tx]), 'version':
self.transaction_counter[tx]})
            print(f"T{tx}: W({item}) at version {self.transaction_counter[tx]}. Timestamp({item}):
({self.transaction_counter[tx]}, {self.transaction_counter[tx]}).")
            self.counter += 1
        else:
            max_index = self.get_max_version_index_by_write(item)
            max_w_timestamp = self.version_table[item][max_index]["timestamp"][1]

```

```

max_r_timestamp = self.version_table[item][max_index]['timestamp'][0]
max_version = self.version_table[item][max_index]['version']

if self.transaction_counter[tx] < max_r_timestamp:
    self.sequence.append({'tx': tx, 'item': item, 'action': 'write', 'timestamp': (
        max_r_timestamp, self.transaction_counter[tx]), 'version': max_version})
    self.rollback(tx)
elif self.transaction_counter[tx] == max_w_timestamp:
    self.version_table[item][max_index]['timestamp'] = (
        max_r_timestamp, self.transaction_counter[tx])
    self.sequence.append({'tx': tx, 'item': item, 'action': 'write', 'timestamp': (
        max_r_timestamp, self.transaction_counter[tx]), 'version': max_version})
    self.counter += 1
else:
    self.version_table[item].append({'tx': tx, 'timestamp': (
        max_r_timestamp, self.transaction_counter[tx]), 'version':
self.transaction_counter[tx]})
    print(f"T{tx}: W({item}) at version {self.transaction_counter[tx]}.
Timestamp({item}): ({max_r_timestamp}, {self.transaction_counter[tx]}).")
    self.counter += 1

def read(self, tx, item):
    if item not in self.version_table.keys():
        if ('tx', tx) not in self.version_table.items():
            self.version_table[item] = []
            self.version_table[item].append({'tx': tx, 'timestamp': (
                self.transaction_counter[tx], 0), 'version': 0})
            self.sequence.append({'tx': tx, 'item': item, 'action': 'read', 'timestamp': (
                self.transaction_counter[tx], 0), 'version': 0})
            print(f"T{tx}: R({item}) at version 0. Timestamp({item}):
({self.transaction_counter[tx]}, 0).")
            self.counter += 1
        else:
            max_index = self.get_max_version_index_by_write(item)
            max_w_timestamp = self.version_table[item][max_index]['timestamp'][1]
            max_r_timestamp = self.version_table[item][max_index]['timestamp'][0]
            max_version = self.version_table[item][max_index]['version']

```

```

        if self.transaction_counter[tx] > max_r_timestamp:
            self.version_table[item][max_index]['timestamp'] = (
                self.transaction_counter[tx], max_w_timestamp)
            print(f"T{tx}: R({item}) at version {max_version}. Timestamp({item}):
{self.version_table[item][max_index]['timestamp']}")
            self.counter += 1
        else:
            max_index = self.get_max_version_index_by_write(item)
            max_w_timestamp = self.version_table[item][max_index]['timestamp'][1]
            max_r_timestamp = self.version_table[item][max_index]['timestamp'][0]
            max_version = self.version_table[item][max_index]['version']

            if self.transaction_counter[tx] > max_r_timestamp:
                self.version_table[item][max_index]['timestamp'] = (
                    self.transaction_counter[tx], max_w_timestamp)
                print(f"T{tx}: R({item}) at version {max_version}. Timestamp({item}):
{self.version_table[item][max_index]['timestamp']}")
                self.counter += 1

    def rollback(self, tx):
        tx_sequence = []
        for i in range(len(self.sequence)):
            if self.sequence[i]['tx'] == tx and self.sequence[i]['action'] != 'aborted':
                tx_sequence.append(
                    {'tx': self.sequence[i]['tx'], 'item': self.sequence[i]['item'], 'action':
self.sequence[i]['action']})
        for i in range(len(self.input_sequence)):
            if self.input_sequence[i]['tx'] == tx:
                tx_sequence.append(self.input_sequence[i])
                self.input_sequence.remove(self.input_sequence[i])
        for i in range(len(tx_sequence)):
            self.input_sequence.append(tx_sequence[i])
        self.sequence.append({'tx': tx, 'item': None, 'action': 'rollback'})
        self.transaction_counter[tx] = self.counter
        print(f"T{tx}: rolled back. Assigned new timestamp: {self.transaction_counter[tx]}")

    def print_sequence(self):

```



```

for i in range(len(self.sequence)):
    if (self.sequence[i]['action'] == 'rollback'):
        print(f"T{self.sequence[i]['tx']}: rolled back.")
    elif (self.sequence[i]['action'] != 'aborted'):
        print(self.sequence[i]['item'], self.sequence[i]['tx'],
              self.sequence[i]['timestamp'], self.sequence[i]['version'])

def run(self):
    while len(self.input_sequence) > 0:
        current = self.input_sequence.popleft()
        if current['action'] == READ_OPERATION:
            self.read(current['tx'], current['item'])
        elif current['action'] == WRITE_OPERATION:
            self.write(current['tx'], current['item'])
        else:
            print("Invalid action.")

def parse_input(input_string):
    input_list = input_string.split(";")
    sequence = []

    for input_item in input_list:
        input_item = input_item.strip()
        if not input_item:
            continue

        try:
            action_type = input_item[0]
            tx = int(input_item[1])
            item = input_item[3]

            if action_type == "R":
                sequence.append({"action": READ_OPERATION, "tx": tx, "item": item})
            elif action_type == "W":
                sequence.append({"action": WRITE_OPERATION, "tx": tx, "item": item})
        except:
            print("Invalid input string")
            exit()

```

```

return sequence

def main():
    input_string = input("Enter Concurrency Control Sequence: ")
    sequence = parse_input(input_string)

    mvcc = MVCC(sequence)
    mvcc.run()

if __name__ == "__main__":
    main()

```

Contoh input output:

### Concurrency Control Protocol

R5(A);R2(B);R1(B);W3(B);W3(C);R5(C);R2(C);R1(A);R4(D);W3(D);W5(B);W5(C)

Process

☐ Two Phase Locking  
☐ Optimistic Concurrency Control  
☒ Multiversion Timestamp Ordering Concurrency Control

**Final Schedule:** T5: R(A) at version 0. Timestamp(A): (5, 0); T2: R(B) at version 0. Timestamp(B): (2, 0); T1: R(B) at version 0. Timestamp(B): (2, 0); T3: W(B) at version 3. Timestamp(B): (2, 3); T3: W(C) at version 3. Timestamp(C): (3, 3); T5: R(C) at version 3. Timestamp(C): (5, 3); T2: R(C) at version 3. Timestamp(C): (5, 3); T1: R(A) at version 0. Timestamp(A): (5, 0); T4: R(D) at version 0. Timestamp(D): (4, 0); T3: rolled back. Assigned new timestamp: 9; T5: W(B) at version 5. Timestamp(B): (2, 5); T5: W(C) at version 5. Timestamp(C): (5, 5); T3: W(C) at version 9. Timestamp(C): (5, 9); T3: W(D) at version 9. Timestamp(D): (4, 9);

### Concurrency Control Protocol

R1(X);R2(X);W1(X);W2(X)

Process

☐ Two Phase Locking  
☐ Optimistic Concurrency Control  
☒ Multiversion Timestamp Ordering Concurrency Control

**Final Schedule:** T1: R(X) at version 0. Timestamp(X): (1, 0); T2: R(X) at version 0. Timestamp(X): (2, 0); T1: rolled back. Assigned new timestamp: 2; W(X) at version 2. Timestamp(X): (2, 2); T1: R(X) at version 2. Timestamp(X): (2, 2);

...

### 3. Eksplorasi Recovery

#### a. *Write-Ahead Log*

Write-Ahead Log (WAL) adalah sebuah mekanisme yang digunakan dalam sistem database untuk memastikan *reliability* dan *recovery* data setelah kegagalan sistem atau kehilangan daya. Sistem database menggunakan log ini untuk mencatat perubahan yang akan diterapkan ke database sebelum perubahan tersebut benar-benar terjadi. Prinsip utama dari WAL adalah menulis (write) informasi log sebelum melakukan perubahan ke database. WAL berisi catatan perubahan yang terjadi pada database. Setiap log menyimpan informasi-informasi penting pada WAL, informasi tersebut berupa id transaksi, jenis operasi (*Insert, Update, Delete*), nilai data yang berubah pada transaksi, lokasi file, *timestamp*, dan beberapa informasi tambahan lainnya. Isi WAL sangat penting untuk memastikan pemulihan (*recovery*) yang konsisten setelah terjadinya kegagalan sistem. Pastikan untuk selalu mendahulukan update data log pada perangkat yang menyimpan log sebelum data tersebut disimpan pada perangkat yang menyimpan database. Pada WAL *undo information* dipastikan bersifat *atomicity* dan *redo information* dipastikan bersifat *durability*.

Oleh karena itu, WAL memiliki keuntungan yang dapat memastikan bahwa data yang dimiliki akan selalu konsisten bahkan ketika terdapat kegagalan sistem dikarenakan write ahead logging memungkinkan sistem untuk memulihkan data sehingga selalu konsisten. Sistem undo redonya membuat data selalu ter-*recover* karena memungkinkan untuk balik atau kembali sesuai apa yang ada di log.

#### b. *Continuous Archiving*

Write ahead log yang sudah di generate biasanya tersimpan pada pg\_wal dengan PG database cluster. Konfigurasi parameter tersebut (*max\_wal\_size* dan *min\_wal\_size*) mengontrol seberapa banyak write along head yang akan disimpan pada pg\_wal. Checkpoints akan secara berkala membersihkan file WAL lama dan menyisakan file yang terbaru. Oleh sebab itu, diperlukannya continuous archiving agar WAL files yang terhapus sudah tersip.

### c. *Point-in-Time Recovery*

Point in time recovery atau yang biasa disebut *incremental database backup* merupakan salah satu metode recovery yang menggunakan *history record* yang sudah tersimpan pada WAL log file. Metode ini melakukan perubahan *roll-forward* yang sudah dibuat sejak database yang terakhir di *backup*. Memiliki semua segmen WAL yang sudah di backup akan meningkatkan kemampuan untuk *recovery database*. Jika melakukan kesalahan yang major dan butuh untuk memulai lagi dari waktu sebelumnya dapat dilakukan dengan *recover*. Keuntungan menggunakan point in time recovery salah satunya adalah *zero down time*. Pengulangan database backup sangat penting untuk sistem yang bersifat ciritacal yang membutuhkan down time besar. Dengan point in time recovery, database backup downtime akan tereliminasi karena mekanisme ini melakukan database backup sama dengan akses sistem. Selain itu, *save storage size* juga merupakan keuntungan dari metode ini. Dengan pengulangan database backup akan tersimpan *archive log* terakhir setelah backup terakhir daripada backup semua database setiap hari.

### d. Simulasi Kegagalan pada PostgreSQL

#### Setup postgre

1. Membuat direktori baru untuk archive

```
C:\Program Files\PostgreSQL\15\data\pg_archive_wal>
```

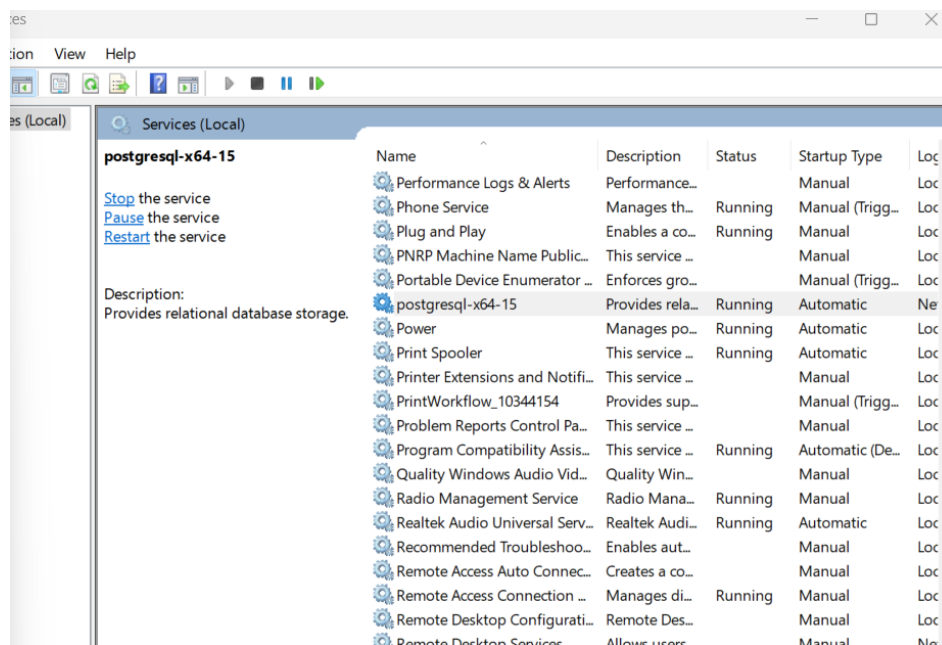
2. Mengubah konfigurasi yang terdapat pada PostgreSQL untuk mengaktifkan Continuous Archiving Wal pada direktori 'pg\_archive\_wal'

```

postgres=# alter system set wal_level = replica;
ALTER SYSTEM
postgres=# alter system set max_wal_size = '1 GB'
postgres=# ;
ALTER SYSTEM
postgres=# alter system set archive_mode = on;
ALTER SYSTEM
postgres=# alter system set archive_command = 'copy %p "C:\Program Files\PostgreSQL\15\data\pg_archive_wal\archive_wal_%f"';
ALTER SYSTEM
postgres=# alter system set archive_timeout '1 h';
ERROR: syntax error at or near "1 h"
LINE 1: alter system set archive_timeout '1 h';
                                           ^
postgres=# alter system set archive_timeout = '1 h';
ALTER SYSTEM
postgres=# alter system set restore_command = 'copy "C:\Program Files\PostgreSQL\15\data\pg_archive_wal\archive_wal_%f"%p';
ALTER SYSTEM

```

### 3. Melakukan *restart* pada *service* PostgreSQL



## Backup dan simulasi kegagalan

### 1. Membuat basis data dan tabel baru

```

postgres=# create database contoh;
CREATE DATABASE
postgres=# \c contoh;
You are now connected to database "contoh" as user "postgres".

```

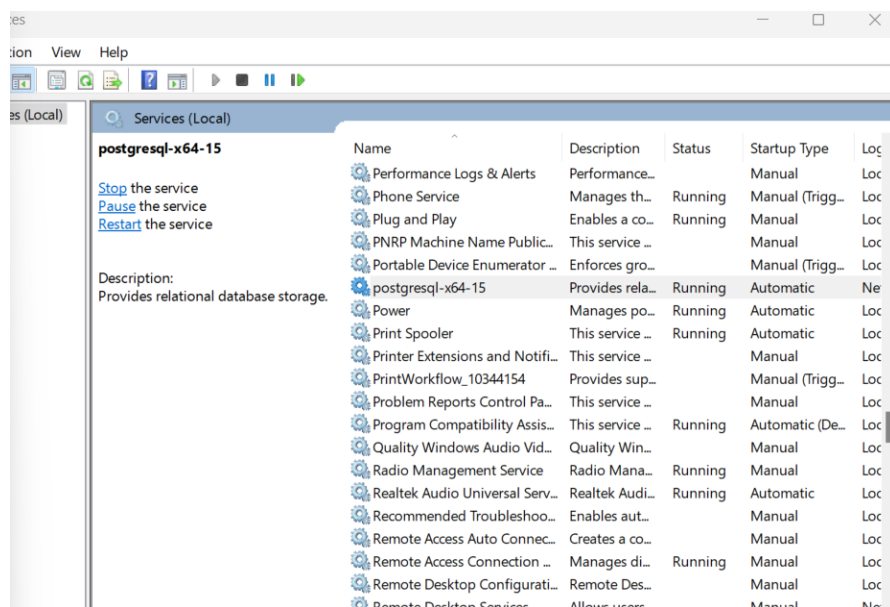
```
contoh=# create table contoh_table (
contoh(# id INTEGER,
contoh(# name Character Varying(69)
contoh(# );
CREATE TABLE
```

```
contoh=# insert into contoh_table(id,name) values
contoh-# (1, 'angger'),
contoh-# (2, 'ilham'),
contoh-# (3, 'amanullah');
INSERT 0 3
contoh=# select * from contoh_table;
 id |  name
----+-----
  1 | angger
  2 | ilham
  3 | amanullah
(3 rows)
```

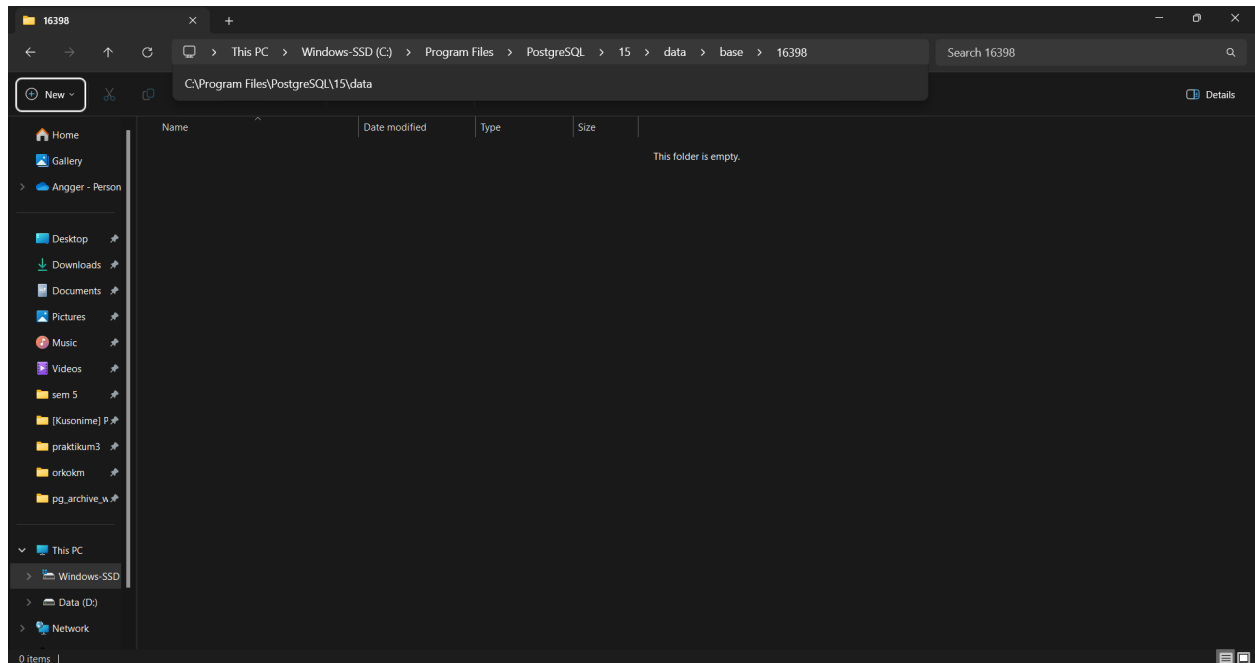
## 2. Melakukan backup cluster basis data

```
C:\Program Files\PostgreSQL\15\data>pg_basebackup -U postgres -Ft -D "C:\Program Files\PostgreSQL\15\data\pg_backup"
Password:
C:\Program Files\PostgreSQL\15\data>
```

## 3. Mematikan service PostgreSQL

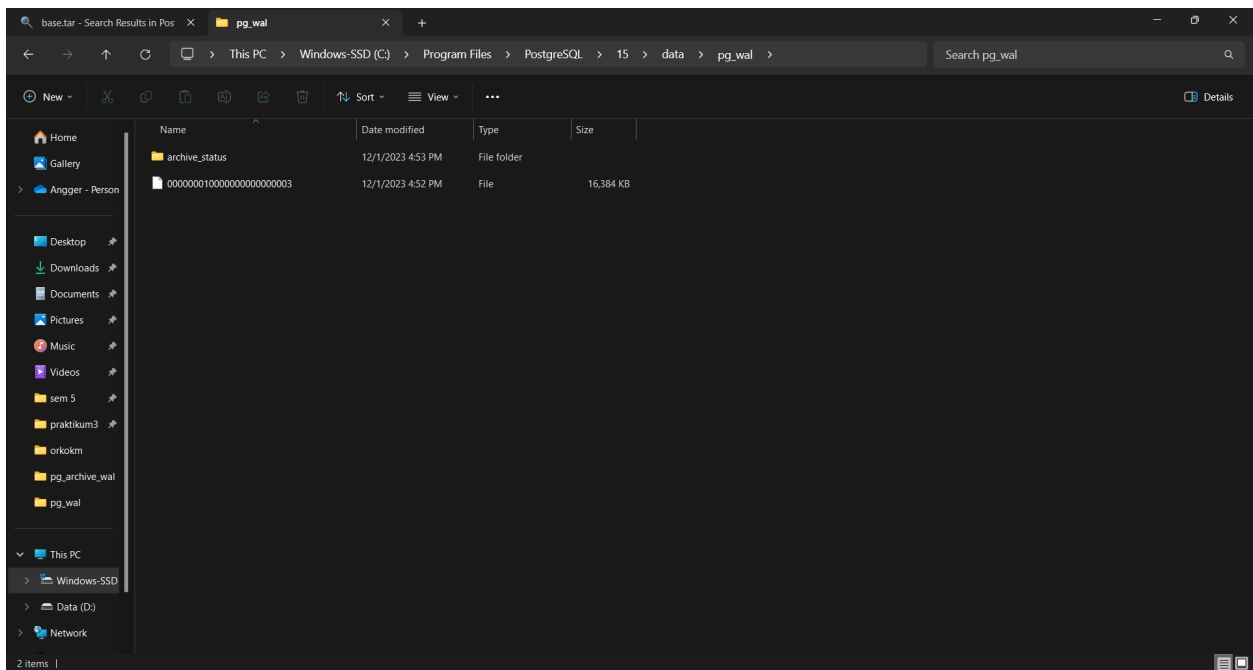
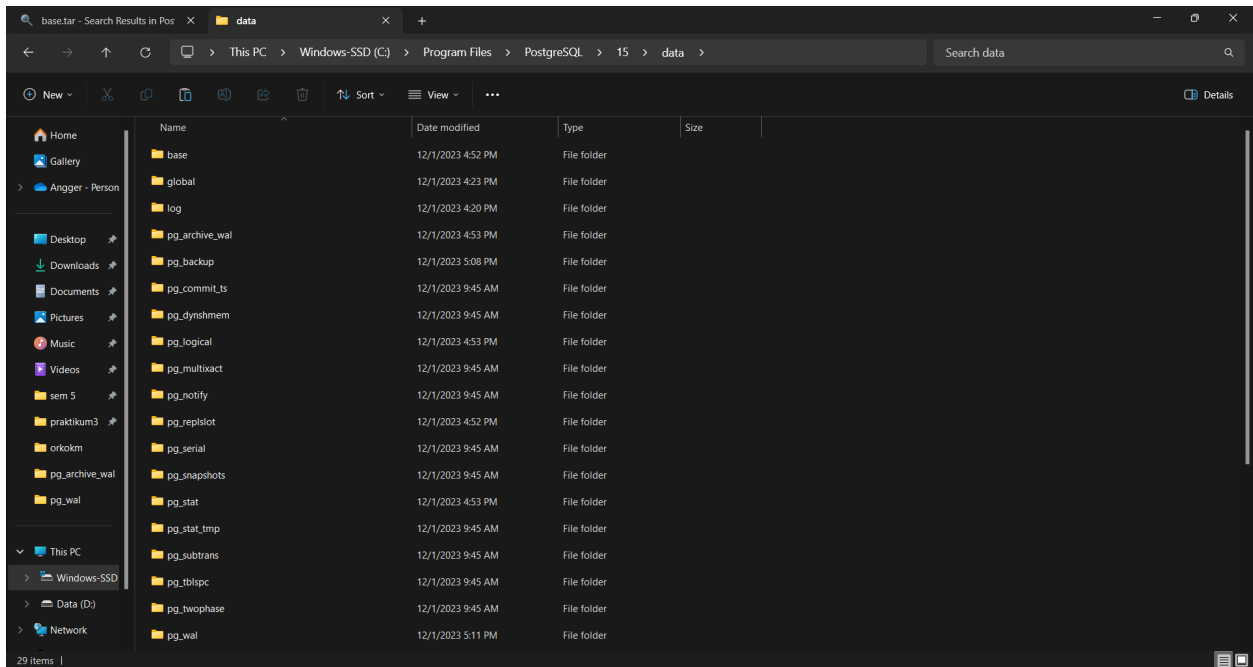


#### 4. Menghapus data



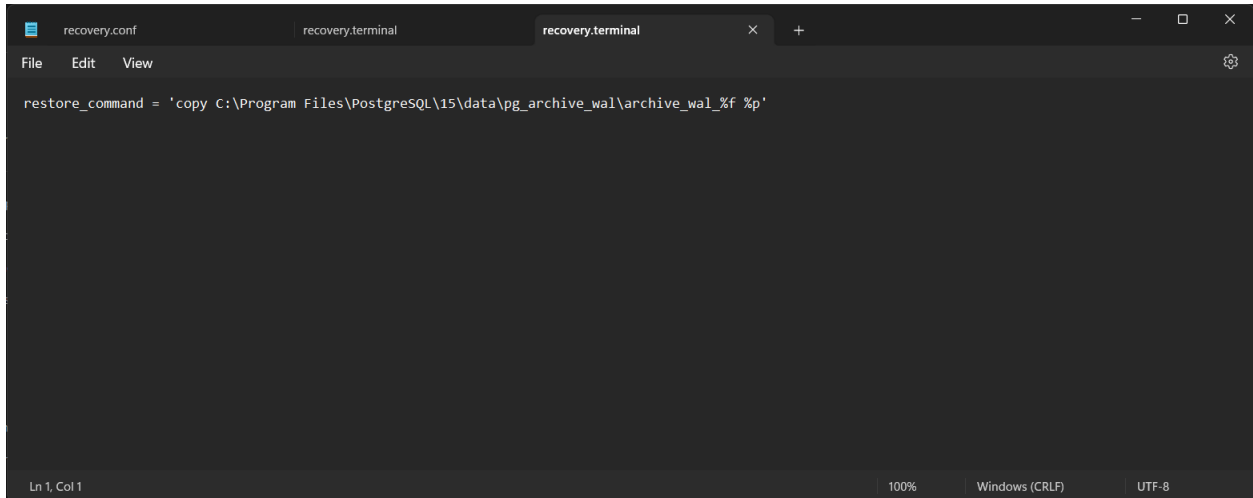
### Proses recovery

1. Unzip file base.tar dan pg\_wal.tar pada file backup yang telah dibuat. Untuk file yang terdapat pada base digunakan untuk mereplace data pada folder data. Untuk file yang terdapat pada pg\_wal.tar digunakan untuk mereplace file pada folder pg\_wal.



2. Buat file recovery.terminal pada folder data



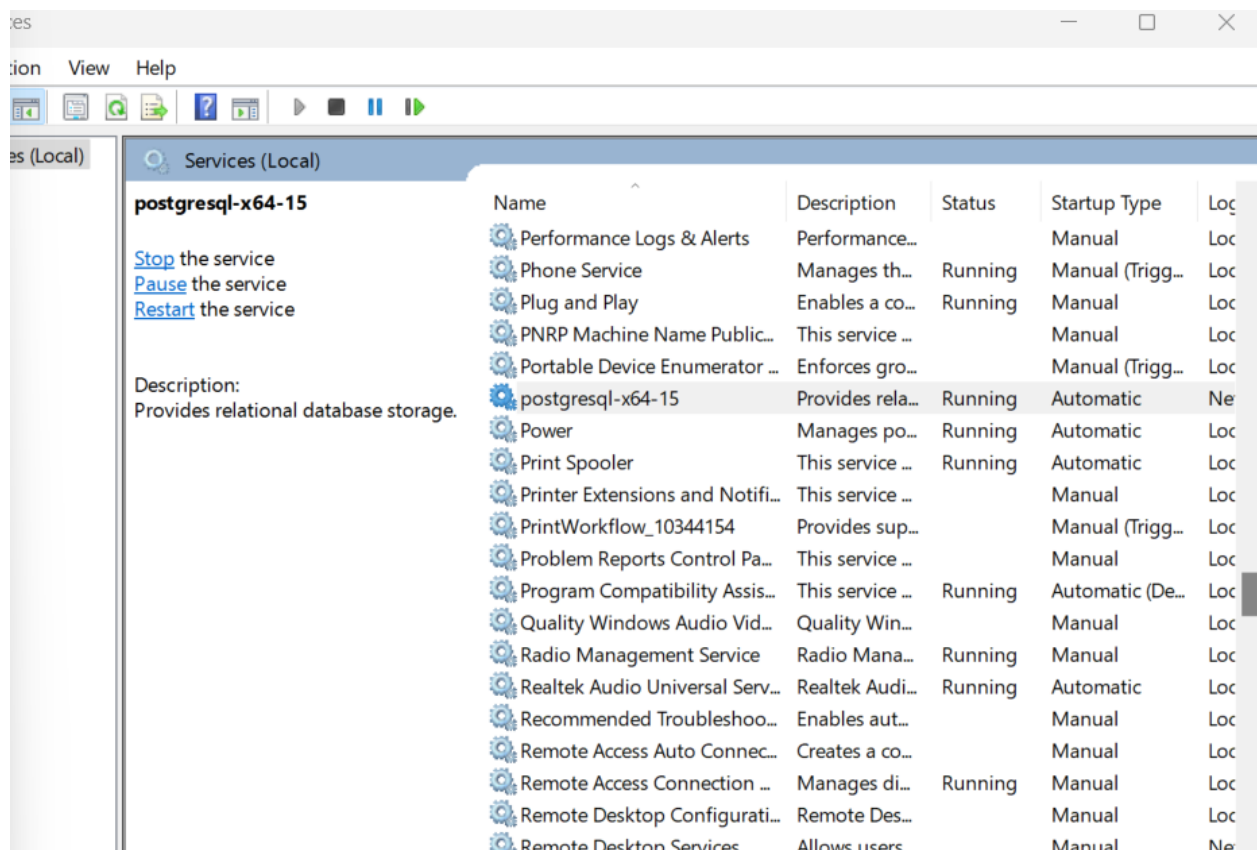


```
File Edit View

restore_command = 'copy C:\Program Files\PostgreSQL\15\data\pg_archive_wal\archive_wal_%f %p'

Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

### 3. Menyalakan kembali service PostgreSQL



### 4. Melihat hasil *recovery* yang telah dilakukan

Isi log

```
recovery.conf  recovery.terminal  recovery.terminal  recovery.signal  recovery.signal  postgresql-2023-12-1
File Edit View
2023-12-01 18:13:46.542 +07 [19576] LOG: database system was shut down at 2023-12-01 18:13:43 +07
2023-12-01 18:13:46.559 +07 [19576] LOG: starting archive recovery
2023-12-01 18:13:46.580 +07 [19576] LOG: consistent recovery state reached at 0/60000A0
2023-12-01 18:13:46.580 +07 [19576] LOG: invalid record length at 0/60000A0: wanted 24, got 0
2023-12-01 18:13:46.581 +07 [19576] LOG: redo is not required
2023-12-01 18:13:46.606 +07 [19576] LOG: selected new timeline ID: 2
2023-12-01 18:13:46.662 +07 [19576] LOG: archive recovery complete
2023-12-01 18:13:46.665 +07 [16388] LOG: checkpoint starting: end-of-recovery immediate wait
2023-12-01 18:13:46.678 +07 [16388] LOG: checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.004 s, sync=
0.002 s, total=0.014 s; sync files=2, longest=0.002 s, average=0.001 s; distance=0 kB, estimate=0 kB

Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

You are now connected to database "contoh" as user "postgres".

```
contoh=# show tables;
```

ERROR: unrecognized configuration parameter "tables"

```
contoh=# \dt
```

```
          List of relations
Schema |      Name      | Type  | Owner
-----+-----+-----+-----
public | contoh_table   | table | postgres
(1 row)
```

```
contoh=# select * from contoh_table;
```

```
 id | name
----+-----
  1 | angger
  2 | ilham
  3 | amanullah
(3 rows)
```

#### 4. Pembagian Kerja

NIM	Nama	Bagian
13521001	Angger Ilham Amanullah	Laporan, Mensimulasikan kegagalan pada postgreSQL
13521012	Haikal Ardzi Shofiyyurrohman	Laporan
13521013	Eunice Sarah Siregar	Laporan, Mensimulasikan perbedaan dari setiap derajat <i>Transaction isolation</i>
13521015	Hidayatullah Wildan Ghaly Buchary	Laporan, Implementasi <i>Concurrency Control</i>

## Referensi

- [1] [PostgreSQL: Documentation: 16: 13.2. Transaction Isolation](#)
- [2] [PostgreSQL: Documentation: 16: SET TRANSACTION](#)
- [3] [Cara Kerja MVCC di Database Relasional | AppMaster](#)
- [4] <https://mkdev.me/posts/transaction-isolation-levels-with-postgresql-as-an-example>