IF3140 MANAJEMEN BASIS DATA

MEKANISME CONCURRENCY CONTROL DAN RECOVERY

K02 Kelompok 02

Anggota :

Muhammad Fahkry Malta	13519032
Michael Leon Putra Widhi	13521108
Ulung Adi Putra	13521122
Nathan Tenka	13521172

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

1. Eksplorasi *Transaction Isolation*

Ada beberapa tingkat isolasi transaksi yang didefinisikan oleh standar SQL, yang dikenal sebagai tingkat isolasi. Tingkat-tingkat ini menentukan sejauh mana suatu transaksi harus diisolasi dari efek transaksi konkuren lainnya. Empat tingkat isolasi standar yang didefinisikan oleh SQL adalah *Serializable*, *Repeatable Read*, *Read Committed*, dan *Read Uncommitted*.

Dalam konteks isolasi transaksi, terdapat beberapa fenomena yang dapat terjadi sebagai hasil dari eksekusi transaksi secara konkuren (berbarengan). Fenomena-fenomena ini dapat mempengaruhi konsistensi dan integritas data dalam sistem basis data. Berikut adalah beberapa fenomena yang umum terkait dengan isolasi transaksi :

1. ***Dirty Read*** **:** Terjadi ketika suatu transaksi membaca data yang telah dimodifikasi oleh transaksi lain, tetapi transaksi tersebut belum di *commit*. Jika transaksi yang melakukan modifikasi tersebut dibatalkan (*rolled back*), maka transaksi yang melakukan *dirty read* akan mendapatkan data yang tidak valid.
2. ***Non-Repeatable Read*** **:** Terjadi ketika suatu transaksi membaca kembali data yang sudah dibaca sebelumnya dan nilai yang didapat berbeda dengan hasil pembacaan sebelumnya yang dikarenakan data tersebut diubah oleh transaksi lain sebelum transaksi pertama selesai. Ini dapat menyebabkan ketidaksesuaian atau inkonsistensi dalam hasil baca data.
3. ***Phantom Read*** **:** Fenomena ini terjadi ketika suatu transaksi melakukan *query* terhadap suatu rentang data, kemudian transaksi lain memasukkan atau menghapus data dalam rentang tersebut sebelum transaksi pertama selesai. Hal ini dapat mengakibatkan transaksi pertama melihat "data hantu" yang sebenarnya tidak ada pada awalnya.
4. ***Serialization Anomaly*** **:** Fenomena ini mencakup beberapa situasi yang melibatkan transaksi yang dieksekusi secara konkuren yang menghasilkan hasil yang tidak sesuai dengan hasil yang diharapkan jika transaksi tersebut dieksekusi secara serial (berurutan).

Tingkat isolasi transaksi standar SQL dan PostgreSQL dijelaskan dalam tabel di bawah :

Tabel 1. Tingkat Isolasi pada PostgreSQL.

Isolation Level	***Dirty Read***	***Non-Repeatable Read***	***Phantom Read***	***Serialization Anomaly***
Read Uncommitted	*Allowed, but not in PG*	*Possible*	*Possible*	*Possible*
Read Committed	*Not Possible*	*Possible*	*Possible*	*Possible*
Repeatable Read	*Not Possible*	*Not Possible*	*Allowed, but not in PG*	*Possible*
Serializable	*Not Possible*	*Not Possible*	*Not Possible*	*Not Possible*

Berikut merupakan penjelasan lebih lanjut beserta simulasi dari empat tingkat isolasi standar yang didefinisikan oleh SQL:

a. *Serializable*

Serializability adalah kontrol konkurensi dengan tingkat isolasi paling ketat di mana tidak mungkin mendapatkan *dirty read*, *non repeatable read*, *phantom read*, dan yang terpenting tidak ada anomali serialisasi. Ketiga fenomena ini (selain *serialization anomaly*) telah diatasi oleh level isolasi lain, tetapi hanya kemampuan serialisasi yang mengatasi *serialization anomaly*.

Untuk mendemonstrasikan, kita menyediakan sebuah tabel *instructor* berisi instructor_id, name, department, dan salary sebagai berikut :

```
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             1 | Fahkry | Comp Sci   | 10000000.00
             2 | Leon   | Physics    | 25000000.00
             3 | Ulung  | Physics    |  5000000.00
             4 | Nathan | Comp Sci   | 18500000.00
```

Gambar 1.1. Ilustrasi isi tabel *instructor*.

Simulasi

1. *Dirty Read*

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM instructor WHERE instructor_id = 4;
SELECT * FROM instructor WHERE instructor_id = 4;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE instructor SET salary = 19000000 WHERE
instructor_id = 4;
```

Urutan Eksekusi

- Transaksi 1 (*Read* pertama)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.2. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Update*)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# UPDATE instructor SET salary = 19000000 WHERE instructor_id = 4;
UPDATE 1
```

Gambar 1.3. *Query* dan hasil *update*.

- Transaksi 1 (*Read* Kedua)

```
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.3. *Query* dan hasil *update*.

Pada simulasi diatas dapat diperhatikan transaksi 1 masih membaca nilai yang sama dari *salary* walaupun sudah di-*update* tanpa *commit* oleh transaksi 2. Jadi, berdasarkan simulasi diatas dapat dipastikan bahwa derajat isolasi *serializable* tidak memungkinkan terjadinya *dirty read*.

2. *Non-Repeatable Read*

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM instructor WHERE instructor_id = 4;
SELECT * FROM instructor WHERE instructor_id = 4;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE instructor SET salary = 19000000 WHERE
instructor_id = 4;
COMMIT;
```

Urutan Eksekusi

- Transaksi 1 (*Read* Pertama)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.4. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Update* dan *commit*)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# UPDATE instructor SET salary = 19000000 WHERE instructor_id = 4;
COMMIT;
UPDATE 1
COMMIT
```

Gambar 1.5. *Query* dan hasil *update* dan *commit*.

- Transaksi 1 (*Read* Kedua)

```
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.6. *Query* dan hasil *read* kedua.

Pada simulasi diatas dapat diperhatikan transaksi 1 masih membaca nilai yang sama dari *salary* walaupun sudah di-*update* dan di-*commit* oleh transaksi 2. Jadi, berdasarkan simulasi diatas dapat dipastikan bahwa derajat isolasi *serializable* tidak memungkinkan terjadinya *non-repeatable read*.

3. *Phantom Read*

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT * FROM instructor WHERE salary > 15000000;
SELECT * FROM instructor WHERE salary > 15000000;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (5, 'Toni', 'Comp Sci', 17000000);
COMMIT;
```

Urutan Eksekusi

- Transaksi 1 (*Read* Pertama)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# SELECT * FROM instructor WHERE salary > 15000000;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             2 | Leon   | Physics    | 25000000.00
             4 | Nathan | Comp Sci   | 18500000.00
(2 rows)
```

Gambar 1.7. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Insert*)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# INSERT INTO instructor(instructor_id, name, department, salary) VALUES (5, 'Toni', 'Comp Sci', 17000000);
COMMIT;
INSERT 0 1
COMMIT
```

Gambar 1.8. *Query* dan hasil *insert*.

- Transaksi 1 (*Read* Kedua)

```
tubes=*# SELECT * FROM instructor WHERE salary > 15000000;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             2 | Leon   | Physics    | 25000000.00
             4 | Nathan | Comp Sci   | 18500000.00
(2 rows)
```

Gambar 1.9. *Query* dan hasil *read* kedua.

Pada simulasi diatas dapat diperhatikan transaksi 1 masih membaca data dan nilai yang sama walaupun sudah di-*insert* data baru yang memenuhi syarat dari *read* oleh transaksi 2. Jadi, berdasarkan simulasi diatas dapat dipastikan bahwa derajat isolasi *serializable* tidak memungkinkan terjadinya *phantom read*.

4. *Serialization Anomaly*

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT AVG(salary) FROM instructor WHERE department =
'Comp Sci';
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (5, 'Andi', 'Physics', 14250000);
COMMIT;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT AVG(salary) FROM instructor WHERE department =
'Physics';
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (6, 'Toni', 'Comp Sci', 15000000);
COMMIT;
```

Urutan Eksekusi

- Transaksi 1

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# SELECT AVG(salary) FROM instructor WHERE department = 'Comp Sci';
          avg
-----------------------
 14250000.000000000000
(1 row)

tubes=*# INSERT INTO instructor(instructor_id, name, department, salary) VALUES (5, 'Andi', 'Physics', 14250000);
INSERT 0 1
```

Gambar 1.10. *Query* dan hasil transaksi pertama.

- Transaksi 2

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
SET
tubes=*# SELECT AVG(salary) FROM instructor WHERE department = 'Physics';
          avg
-----------------------
 15000000.000000000000
(1 row)

tubes=*# INSERT INTO instructor(instructor_id, name, department, salary) VALUES (6, 'Toni', 'Comp Sci', 15000000);
INSERT 0 1
```

Gambar 1.11. *Query* dan hasil transaksi kedua.

- Transaksi 1 (*Commit*)

```
tubes=*# COMMIT;
COMMIT
```

Gambar 1.12. *Query* dan hasil *commit* transaksi pertama.

- Transaksi 2 (*Commit*)

```
tubes=*# COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Gambar 1.13. *Query* dan hasil *commit* transaksi kedua.

Pada simulasi diatas dapat diperhatikan bahwa kedua transaksi tidak akan memiliki hasil yang konsisten dengan semua kemungkinan hasil dari transaksi apabila dilakukan secara serial *(serialization anomaly*). Kemudian berdasarkan simulasi diatas dapat diperhatikan juga bahwa derajat isolasi *serializable* tidak memungkinkan terjadinya *serialization anomaly* yang terbukti ketika hanya salah satu transaksi (transaksi 1) yang diizinkan *commit* dan transaksi 2 harus melakukan *rollback*.

b. *Repeatable Read*

Pada tingkat ini, sebuah transaksi memastikan bahwa begitu suatu data dibaca, data tersebut tidak akan dimodifikasi atau dihapus oleh transaksi lain sampai transaksi saat ini selesai. Mencegah *dirty reads* dan *non-repeatable reads* tetapi masih dapat memungkinkan *phantom reads*, tapi tidak dimungkinkan di PostgreSQL.

Untuk mendemonstrasikan, kita menyediakan sebuah tabel *instructor* yang sama dengan sebelumnya.

Simulasi

1. Dirty Read

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM instructor WHERE instructor_id = 4;
SELECT * FROM instructor WHERE instructor_id = 4;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE instructor SET salary = 19000000 WHERE
instructor_id = 4;
```

Urutan Eksekusi

- Transaksi 1 (*Read* pertama)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
SET
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
[            4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.14. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Update*)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
SET
tubes=*# UPDATE instructor SET salary = 19000000 WHERE instructor_id = 4;
UPDATE 1
```

Gambar 1.15. *Query* dan hasil *update*.

- Transaksi 1 (Read Kedua)

```
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.16. *Query* dan hasil *read* kedua.

Pada simulasi diatas dapat diperhatikan transaksi 1 masih membaca nilai yang sama dari *salary* walaupun sudah di-*update* tanpa *commit* oleh transaksi 2. Jadi, berdasarkan simulasi diatas dapat dipastikan bahwa derajat isolasi *repeatable read* tidak memungkinkan terjadinya *dirty read*.

2. *Non-Repeatable Read*

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM instructor WHERE instructor_id = 4;
SELECT * FROM instructor WHERE instructor_id = 4;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE instructor SET salary = 19000000 WHERE
instructor_id = 4;
COMMIT;
```

Urutan Eksekusi

- Transaksi 1 (*Read* Pertama)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
SET
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.17. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Update* dan *commit*)

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
SET
tubes=*# UPDATE instructor SET salary = 19000000 WHERE instructor_id = 4;
COMMIT;
UPDATE 1
COMMIT
```

Gambar 1.18. *Query* dan hasil *update* dan *commit*.

- Transaksi 1 (Read Kedua)

```
tubes=*# SELECT * FROM instructor WHERE instructor_id = 4;
 instructor_id |  name  | department |   salary
---------------+--------+------------+-------------
             4 | Nathan | Comp Sci   | 18500000.00
(1 row)
```

Gambar 1.19. *Query* dan hasil *read* kedua.

Pada simulasi diatas dapat diperhatikan transaksi 1 masih membaca nilai yang sama dari *salary* walaupun sudah di-*update* dan di-*commit* oleh transaksi 2. Jadi, berdasarkan simulasi diatas dapat dipastikan bahwa derajat isolasi *repeatable read* tidak memungkinkan terjadinya *non-repeatable read*.

3. *Phantom Read*

Query dari Kedua Transaksi

Transaksi pertama

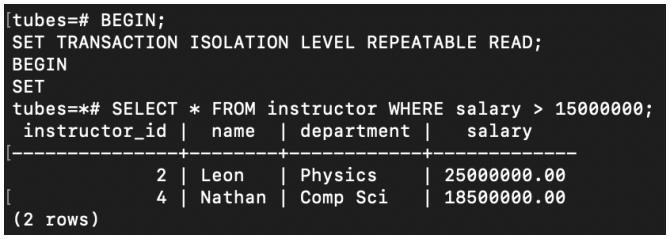
```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT * FROM instructor WHERE salary > 15000000;
SELECT * FROM instructor WHERE salary > 15000000;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (5, 'Toni', 'Comp Sci', 17000000);
COMMIT;
```

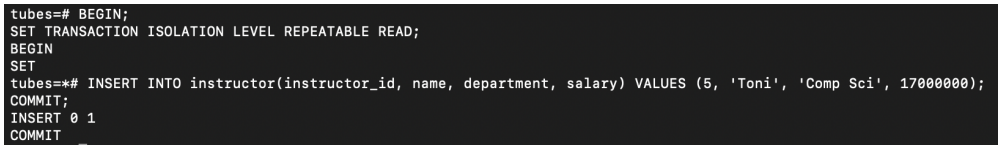
Urutan Eksekusi

- Transaksi 1 (*Read* Pertama)



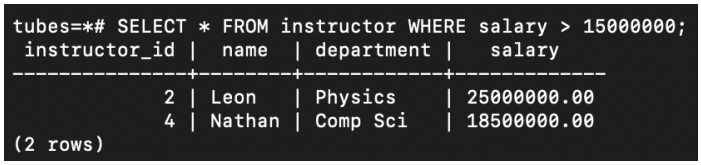
Gambar 1.20. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Insert*)



Gambar 1.21. *Query* dan hasil *insert*.

- Transaksi 1 (*Read* Kedua)



Gambar 1.22. *Query* dan hasil *read* kedua.

Pada simulasi diatas dapat diperhatikan transaksi 1 masih membaca data dan nilai yang sama walaupun sudah di-*insert* data baru yang memenuhi syarat dari *read* oleh transaksi 2. Sebenarnya derajat isolasi *repeatable read* memungkinkan terjadinya *phantom read* akan tetapi tidak dimungkinkan pada PostgreSQL seperti yang sudah disimulasikan diatas.

4. *Serialization Anomaly*

Query dari Kedua Transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT AVG(salary) FROM instructor WHERE department =
'Comp Sci';
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (5, 'Andi', 'Physics', 14250000);
COMMIT;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT AVG(salary) FROM instructor WHERE department =
'Physics';
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (6, 'Toni', 'Comp Sci', 15000000);
COMMIT;
```

Urutan Eksekusi

- Transaksi 1

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
SET
tubes=*# SELECT AVG(salary) FROM instructor WHERE department = 'Comp Sci';
          avg
-----------------------
 14250000.000000000000
(1 row)

tubes=*# INSERT INTO instructor(instructor_id, name, department, salary) VALUES (5, 'Andi', 'Physics', 14250000);
INSERT 0 1
```

Gambar 1.23. *Query* dan hasil transaksi pertama.

- Transaksi 2

```
tubes=# BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
SET
tubes=*# SELECT AVG(salary) FROM instructor WHERE department = 'Physics';
          avg
-----------------------
 15000000.000000000000
(1 row)

tubes=*# INSERT INTO instructor(instructor_id, name, department, salary) VALUES (6, 'Toni', 'Comp Sci', 15000000);
INSERT 0 1
```

Gambar 1.24. *Query* dan hasil transaksi kedua.

- Transaksi 1 (*Commit*)

```
tubes=*# COMMIT;
COMMIT
```

Gambar 1.25. *Query* dan hasil *commit* transaksi pertama.

- Transaksi 2 (Commit)

```
tubes=*# COMMIT;
COMMIT
```

Gambar 1.26. *Query* dan hasil *commit* transaksi kedua.

Pada simulasi diatas dapat diperhatikan bahwa kedua transaksi tidak akan memiliki hasil yang konsisten dengan semua kemungkinan hasil dari transaksi apabila dilakukan secara serial (*serialization anomaly*). Namun berdasarkan simulasi diatas kedua transaksi diizinkan dan berhasil *commit* sehingga dapat dipastikan bahwa derajat isolasi *repeatable read* memungkinkan terjadinya *serialization anomaly*.

c. *Read Committed*

Isolasi *Read Committed* dalam PostgreSQL adalah tingkat isolasi yang memastikan bahwa setiap transaksi hanya dapat melihat perubahan yang sudah di-*commit* oleh transaksi lain. Ini berarti bahwa ketika sebuah transaksi membaca data, transaksi tersebut hanya akan melihat perubahan yang sudah di-*commit* oleh transaksi lain, bukan perubahan yang masih dalam proses (belum di-*commit*). Isolasi *Read Committed* juga menjadi tingkat isolasi *default* pada PostgreSQL.

Untuk mendemonstrasikan, kita menyediakan sebuah tabel *instructor* yang sama dengan sebelumnya.

Simulasi

1. ***Dirty Read***

   ***Query* dari kedua transaksi**

   Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM instructor WHERE instructor_id = 4;
```

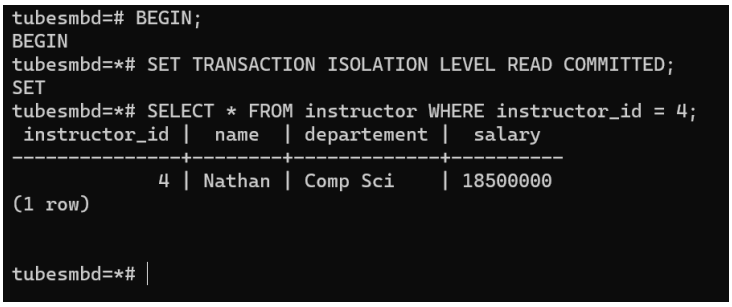
```
SELECT * FROM instructor WHERE instructor_id = 4;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE instructor SET salary = 19000000 WHERE
instructor_id = 4;
```

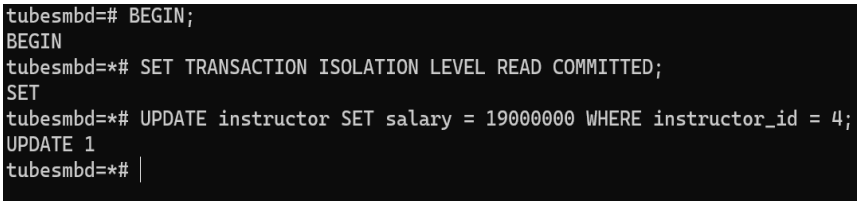
Urutan Eksekusi

- Transaksi 1 (*Read* pertama)



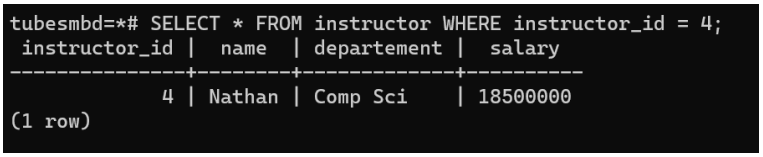
Gambar 1.27. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Update*)



Gambar 1.28. *Query* dan hasil *update*.

- Transaksi 1 (*Read* Kedua)



Gambar 1.29. *Query* dan hasil *read* kedua.

Pada simulasi diatas dapat dilihat bahwa transaksi 1 masih membaca nilai yang sama walaupun transaksi 2 sudah melakukan *update* tanpa *commit*. Oleh karena itu, berdasarkan simulasi dapat disimpulkan bahwa derajat isolasi *Read Committed* tidak memungkinkan terjadinya *dirty read*.

2. *Non-Repeatable Read*

Query dari kedua transaksi

Transaksi pertama

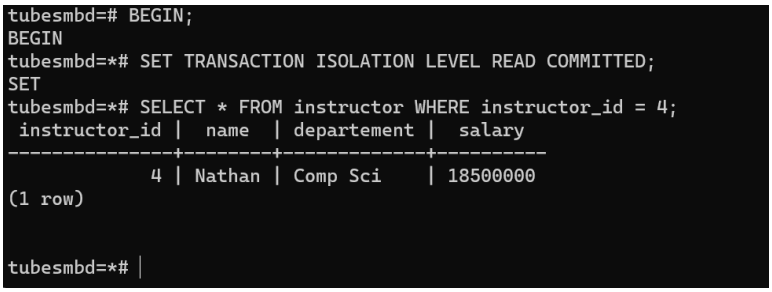
```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM instructor WHERE instructor_id = 4;
SELECT * FROM instructor WHERE instructor_id = 4;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
UPDATE instructor SET salary = 19000000 WHERE
instructor_id = 4;
COMMIT;
```

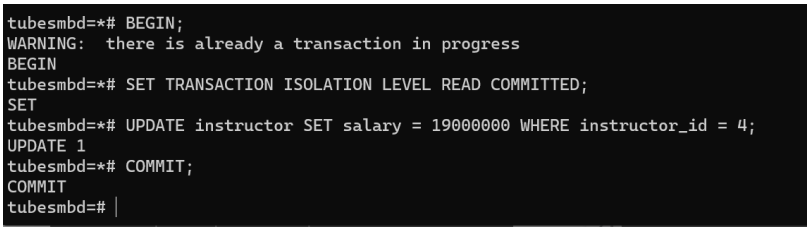
Urutan Eksekusi

- Transaksi 1 (*Read* pertama)



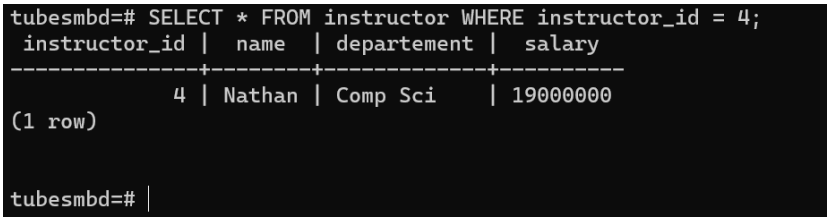
Gambar 1.30. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Update* dan *commit*)



Gambar 1.31. *Query* dan hasil *update* dan *commit*.

- Transaksi 1 (*Read* Kedua)



Gambar 1.32. *Query* dan hasil *read* kedua.

Pada simulasi diatas terlihat bahwa *read* kedua dari transaksi 1 sudah membaca nilai hasil *update* dari transaksi 2. Hal ini dikarenakan transaksi 2 sudah melakukan *commit* sebelum nilainya dibaca oleh transaksi 1. Oleh karena itu, dapat disimpulkan bahwa derajat isolasi *Read Committed* memungkinkan untuk terjadinya *non-repeatable read.*

3. *Phantom Read*

Query dari kedua transaksi

Transaksi pertama

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT * FROM instructor WHERE salary > 15000000;
SELECT * FROM instructor WHERE salary > 15000000;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (5, 'Toni', 'Comp Sci', 17000000);
COMMIT;
```

Urutan Eksekusi

- Transaksi 1 (*Read* pertama)

```
tubesmbd=# BEGIN;
BEGIN
tubesmbd=*# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
tubesmbd=*# SELECT * FROM instructor WHERE salary > 15000000;
 instructor_id |  name  | departement |  salary
---------------+--------+-------------+----------
             2 | Leon   | Physics     | 25000000
             4 | Nathan | Comp Sci    | 19000000
(2 rows)


tubesmbd=*#
```

Gambar 1.33. *Query* dan hasil *read* pertama.

- Transaksi 2 (*Insert* dan *commit*)

```
postgres=# \c tubesmbd
You are now connected to database "tubesmbd" as user "postgres".
tubesmbd=# BEGIN;
BEGIN
tubesmbd=*# INSERT INTO instructor(instructor_id, name, departement, salary) VALUES (5, 'Toni', 'Comp Sci', 17000000);
INSERT 0 1
tubesmbd=*# COMMIT;
COMMIT
tubesmbd=#
```

Gambar 1.34. *Query* dan hasil *insert* dan *commit*.

- Transaksi 1 (*Read* Kedua)

```
tubesmbd=*# SELECT * FROM instructor WHERE salary > 15000000;
 instructor_id |  name  | departement |  salary
---------------+--------+-------------+----------
             2 | Leon   | Physics     | 25000000
             4 | Nathan | Comp Sci    | 19000000
             5 | Toni   | Comp Sci    | 17000000
(3 rows)


tubesmbd=*#
```

Gambar 1.35. *Query* dan hasil *read* kedua.

Pada simulasi diatas terlihat bahwa *read* kedua dari transaksi 1 sudah membaca nilai hasil *insert* dari transaksi 2. Hal ini dikarenakan transaksi 2 sudah melakukan *commit* sebelum nilainya dibaca oleh transaksi 1. Oleh karena itu, dapat disimpulkan bahwa derajat isolasi *Read Committed* memungkinkan untuk terjadinya *phantom read*.

4. ***Serialization Anomaly***

Query dari kedua transaksi

Transaksi pertama

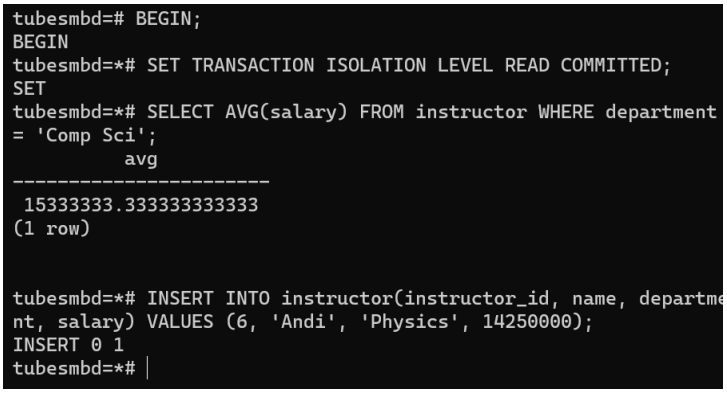
```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT AVG(salary) FROM instructor WHERE department = 'Comp
Sci';
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (6, 'Andi', 'Physics', 14250000);
COMMIT;
```

Transaksi kedua

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT AVG(salary) FROM instructor WHERE department =
'Physics';
INSERT INTO instructor(instructor_id, name, department,
salary) VALUES (7, 'Toni', 'Comp Sci', 15000000);
COMMIT;
```

Urutan Eksekusi

- Transaksi 1



Gambar 1.36. *Query* dan hasil transaksi pertama.

- Transaksi 2

```
tubesmbd=# BEGIN;
BEGIN
tubesmbd=*# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
tubesmbd=*# SELECT AVG(salary) FROM instructor WHERE department
= 'Physics';
          avg
-----------------------
 15000000.000000000000
(1 row)


tubesmbd=*# INSERT INTO instructor(instructor_id, name, departme
nt, salary) VALUES (7, 'Toni', 'Comp Sci', 15000000);
INSERT 0 1
tubesmbd=*#
```

Gambar 1.37. *Query* dan hasil transaksi kedua.

- Transaksi 1 (*Commit*)

```
tubesmbd=*# COMMIT;
COMMIT
tubesmbd=#
```

Gambar 1.38. *Query* dan hasil *commit* transaksi pertama.

- Transaksi 2 (*Commit)*

```
tubesmbd=*# COMMIT;
COMMIT
```

Gambar 1.39. *Query* dan hasil *commit* transaksi kedua.

Pada simulasi diatas dapat diperhatikan bahwa kedua transaksi tidak akan memiliki hasil yang konsisten dengan semua kemungkinan hasil dari transaksi apabila dilakukan secara serial (*serialization anomaly*). Namun berdasarkan simulasi diatas kedua transaksi diizinkan dan berhasil *commit* sehingga dapat dipastikan bahwa derajat isolasi *Read Committed* memungkinkan terjadinya *serialization anomaly*.

d. *Read Uncommitted*

Read Uncommitted adalah tingkat isolasi yang paling rendah pada PostgreSQL. Tingkat isolasi ini memungkinkan sebuah transaksi membaca hasil transaksi lain yang belum dilakukan *commit.* Hal ini berarti transaksi yang sedang berjalan dapat melihat perubahan yang dilakukan oleh transaksi lain bahkan jika transaksi tersebut belum

terkonfirmasi selesai. Tingkat isolasi ini dapat meningkatkan ketersediaan data untuk diakses oleh transaksi lain. Namun, tingkat isolasi ini juga akan menyebabkan masalah konsistensi. Masalah konsistensi ini disebabkan adanya transaksi *uncommitted* yang dibatalkan, namun nilainya sudah terlanjur diakses oleh transaksi lain. Oleh karena itu, perlu dipertimbangakan terlebih dahulu sebelum mengganti *level* isolasi transaksi menjadi *Read Uncommitted.* Biasanya tingkat isolasi ini digunakan dalam kondisi bila konsistensi data tidak menjadi prioritas utama, dan ketersediaan data lebih diutamakan.

.

2. Implementasi *Concurrency Control Protocol*

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

Two-Phase Locking (2PL) adalah protokol kontrol konkurensi yang menggunakan *lock* atau kunci untuk mencegah eksekusi operasi yang melanggar ACID maupun *serializability* dalam eksekusi konkuren. Dalam protokol ini digunakan dua jenis kunci, yaitu *shared lock* dan *exclusive lock*. *Shared lock* diperlukan jika transaksi ingin membaca suatu *item data* dan bisa dipegang beberapa transaksi sekaligus, sedangkan *exclusive lock* diperlukan untuk melakukan penulisan *item data* dan hanya boleh dipegang oleh satu transaksi.

Sebelum melakukan suatu operasi, transaksi yang bersangkutan harus memiliki *lock* terhadap *item data* yang ingin dipakai. Jika *lock* tidak bisa diakuisisi, maka transaksi perlu menunggu sampai transaksi lain yang menggunakan *lock* tersebut melepas *lock*. Akuisisi dan pelepasan *lock* dalam *Two-Phase Locking* perlu mengikuti fase sebagai berikut :

1. ***Growing phase*** **:** dalam fase ini, transaksi boleh mengakuisisi *lock* yang diperlukan, namun tidak boleh melepas *lock*.
2. ***Shrinking phase*** **:** dalam fase ini, transaksi boleh melepas *lock* namun tidak boleh mengakuisisi *lock* lagi.

Two-Phase Locking memastikan bahwa jadwal eksekusi yang dihasilkan *serializable* (ekuivalen dengan sebuah jadwal *serial* / dieksekusi berurutan satu per satu) sesuai dengan urutan mendapat *lock* terakhir.

Walau memastikan *serializability*, protokol ini tidak menjamin tidak adanya *deadlock* (dua atau lebih transaksi menunggu selesainya eksekusi satu sama lain, sehingga tidak ada yang bisa lanjut), *recoverability*, maupun mencegah *cascading rollback*. *Recoverability* dan pencegahan *cascading rollback* bisa dilakukan dengan menggunakan variasi *Two-Phase Locking* sebagai berikut :

1. ***Strict Two-Phase Locking*** : transaksi harus memegang *exclusive lock* sampai telah *commit/abort*.
2. ***Rigorous Two-Phase Locking*** : transaksi harus memegang semua *lock* sampai telah *commit/abort*.

Untuk masalah *deadlock*, bisa digunakan strategi pencegahan *deadlock* seperti *wait-die, wound-wait,* dan *timeout-based schemes*. Berikut penjelasan ketiganya :

1. ***Wait-die*** : transaksi yang lebih dulu bisa menunggu transaksi yang lebih baru, namun transaksi baru akan langsung *rollback* jika meminta *lock* dari transaksi yang lebih dulu.
2. ***Wound-wait*** : transaksi yang lebih dulu akan memaksa transaksi yang lebih baru untuk *rollback* jika memerlukan *lock* yang sedang dipegang transaksi baru. Transaksi baru bisa menunggu transaksi yang lebih dulu.
3. ***Timeout-based schemes*** : transaksi hanya menunggu sampai lama waktu tertentu sebelum melakukan *rollback* pada diri sendiri.

Dalam implementasi kali ini, akan digunakan *Rigorous Two-Phase Locking with Automatic Acquisition of Locks* (mengambil kunci hanya saat operasi akan dilakukan) dengan strategi pencegahan *deadlock wound-wait*.

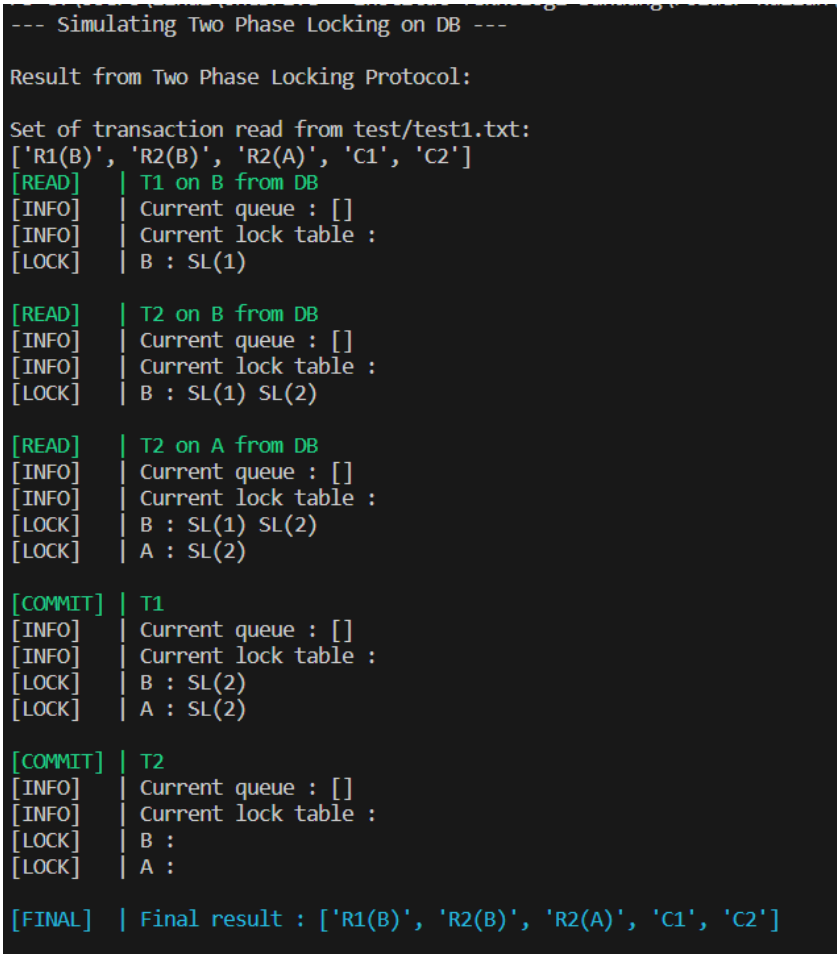
Implementasi dari Protokol 2PL terdapat pada tautan berikut. Adapun untuk menjalankannya, perlu dilakukan pemindahan direktori ke direktori 2pl, kemudian ketikkan perintah berikut.

```
python3 main.py test/<nama test case>
```

Pengujian Hasil Implementasi

Uji Kasus 1

Kasus Uji - `test1.txt`
R1(B); R2(B); R2(A); C1; C2
Hasil Pengujian



Dari hasil program bisa dilihat bahwa seluruh transaksi berhasil dijalankan tanpa masalah. Seluruh operasi yang dilakukan hanya berupa *read* sehingga tidak ada masalah saat akuisisi *lock* (semua hanya memerlukan *shared lock*).

Uji Kasus 2

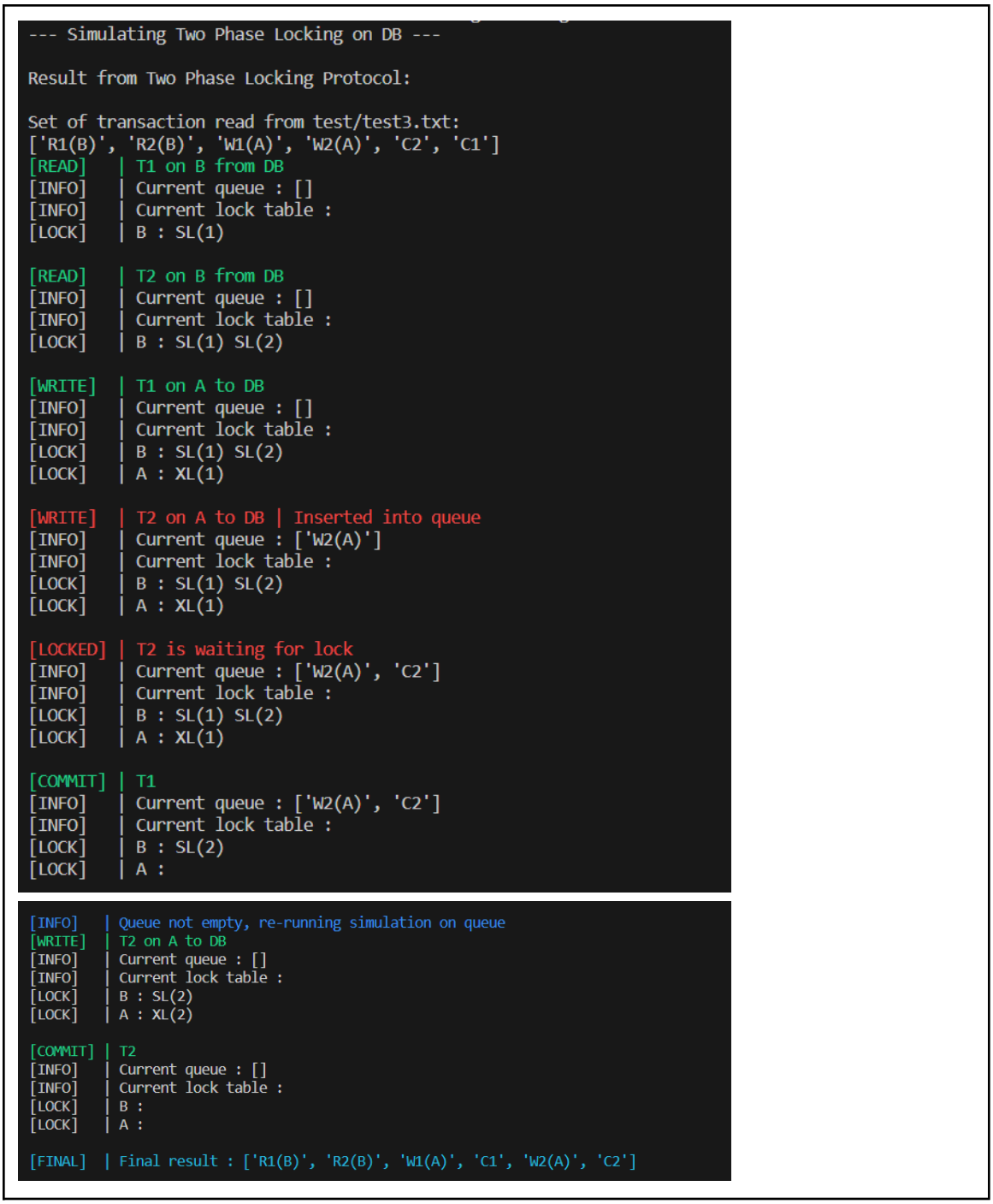
Kasus Uji - `test2.txt`
R1(B); R2(B); W1(A); R2(A); C1; C2
Hasil Pengujian

```
Result from Two Phase Locking Protocol:

Set of transaction read from test/test2.txt:
['R1(B)', 'R2(B)', 'W1(A)', 'R2(A)', 'C1', 'C2']
[READ]   | T1 on B from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | B : SL(1)

[READ]   | T2 on B from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | B : SL(1) SL(2)

[WRITE]  | T1 on A to DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | B : SL(1) SL(2)
[LOCK]   | A : XL(1)

[READ]   | T2 on A from DB | Inserted into queue
[INFO]   | Current queue : ['R2(A)']
[INFO]   | Current lock table :
[LOCK]   | B : SL(1) SL(2)
[LOCK]   | A : XL(1)

[COMMIT] | T1
[INFO]   | Current queue : ['R2(A)']
[INFO]   | Current lock table :
[LOCK]   | B : SL(2)
[LOCK]   | A :

[READ]   | T2 on A from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | B : SL(2)
[LOCK]   | A : SL(2)

[COMMIT] | T2
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | B :
[LOCK]   | A :

[FINAL]  | Final result : ['R1(B)', 'R2(B)', 'W1(A)', 'C1', 'R2(A)', 'C2']
```

Melalui hasil program dapat diamati bahwa terdapat masalah akuisisi *lock* untuk operasi R2(A). Hal ini karena transaksi 1 sudah memegang *exclusive lock* pada *item data* A lebih dulu, sehingga transaksi 2 harus menunggu sampai transaksi 1 *commit* dan melepas *lock* yang dimilikinya.

Uji Kasus 3

Kasus Uji - `test3.txt`
R1(B); R2(B); W1(A); W2(A); C2; C1
Hasil Pengujian



Melalui hasil program dapat diamati bahwa terdapat masalah dalam akuisisi *lock* untuk operasi W2(A) sehingga operasi tersebut dimasukkan ke *queue* lebih dulu. Hal ini disebabkan transaksi 1 sudah memegang *exclusive lock* terhadap *item data* A lebih dulu melalui operasi W1(A) dan transaksi 2 lebih baru dibanding transaksi 1. Operasi C2

juga dimasukkan ke *queue* karena transaksi 2 masih menunggu *exclusive lock* untuk *item data* A.

Uji Kasus 4

Kasus Uji - `test4.txt`
R1(E); R2(G); R1(F); R2(F); R1(G); R2(E); W1(F); W2(E); W1(G); C2; C1
Hasil Pengujian

```
--- Simulating Two Phase Locking on DB ---

Result from Two Phase Locking Protocol:

Set of transaction read from test/test4.txt:
['R1(E)', 'R2(G)', 'R1(F)', 'R2(F)', 'R1(G)', 'R2(E)', 'W1(F)', 'W2(E)', 'W1(G)', 'C2', 'C1']
[READ]   | T1 on E from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)

[READ]   | T2 on G from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : SL(2)

[READ]   | T1 on F from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : SL(2)
[LOCK]   | F : SL(1)

[READ]   | T2 on F from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : SL(2)
[LOCK]   | F : SL(1) SL(2)
```

```
[READ]   | T1 on G from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : SL(2) SL(1)
[LOCK]   | F : SL(1) SL(2)

[READ]   | T2 on E from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(1) SL(2)
[LOCK]   | G : SL(2) SL(1)
[LOCK]   | F : SL(1) SL(2)

[WRITE]  | T1 on F to DB | Lock held by younger transaction
[ABORT]  | T2 rolled back
[WRITE]  | T1 on F to DB
[INFO]   | Current queue : ['R2(G)', 'R2(F)', 'R2(E)']
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : SL(1)
[LOCK]   | F : XL(1)
```

```
[LOCKED] | T2 is waiting for lock
[INFO]   | Current queue : ['R2(G)', 'R2(F)', 'R2(E)', 'W2(E)']
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : SL(1)
[LOCK]   | F : XL(1)

[WRITE]  | T1 on G to DB
[INFO]   | Current queue : ['R2(G)', 'R2(F)', 'R2(E)', 'W2(E)']
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : XL(1)
[LOCK]   | F : XL(1)

[LOCKED] | T2 is waiting for lock
[INFO]   | Current queue : ['R2(G)', 'R2(F)', 'R2(E)', 'W2(E)', 'C2']
[INFO]   | Current lock table :
[LOCK]   | E : SL(1)
[LOCK]   | G : XL(1)
[LOCK]   | F : XL(1)

[COMMIT] | T1
[INFO]   | Current queue : ['R2(G)', 'R2(F)', 'R2(E)', 'W2(E)', 'C2']
[INFO]   | Current lock table :
[LOCK]   | E :
[LOCK]   | G :
[LOCK]   | F :
```

```
[INFO]   | Queue not empty, re-running simulation on queue
[READ]   | T2 on G from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E :
[LOCK]   | G : SL(2)
[LOCK]   | F :

[READ]   | T2 on F from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E :
[LOCK]   | G : SL(2)
[LOCK]   | F : SL(2)

[READ]   | T2 on E from DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : SL(2)
[LOCK]   | G : SL(2)
[LOCK]   | F : SL(2)

[WRITE]  | T2 on E to DB
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E : XL(2)
[LOCK]   | G : SL(2)
[LOCK]   | F : SL(2)
```

```
[COMMIT] | T2
[INFO]   | Current queue : []
[INFO]   | Current lock table :
[LOCK]   | E :
[LOCK]   | G :
[LOCK]   | F :

[FINAL]  | Final result : ['R1(E)', 'R1(F)', 'R1(G)', 'W1(F)', 'W1(G)', 'C1', 'R2(G)', 'R2(F)', 'R2(E)', 'W2(E)', 'C2']
```

Melalui hasil program dapat diamati bahwa terjadi *rollback* pada transaksi 2 saat eksekusi operasi W1(F). Hal ini karena transaksi 2 sedang memegang *shared lock item data* F dan transaksi 1 dijalankan lebih dulu dibanding transaksi 2. Sesuai dengan strategi *wound-wait*, transaksi 1 memaksa transaksi 2 untuk melakukan *rollback* dan transaksi 2 perlu menunggu sampai transaksi 1 selesai. Jadwal yang didapat merupakan jadwal *serial* dengan urutan transaksi 1 diikuti transaksi 2.

Desain dan Implementasi Algoritma

`2pl/twoPL.py`

Pada *file* ini diimplementasikan kelas TwoPL untuk mensimulasikan transaksi menggunakan *Two-Phase Locking*. Untuk menjalankan simulasi, diperlukan daftar operasi yang akan dilakukan. Operasi yang ada menggunakan aturan penulisan seperti yang terdapat pada salindia kuliah, yaitu :

a. Rx(A) menyatakan operasi *read* pada *item data* A oleh transaksi Tx.
b. Wx(A) menyatakan operasi *write* pada *item data* B oleh transaksi Tx.
c. Cx menyatakan operasi *commit* oleh transaksi Tx.

Kelas ini menyimpan berbagai informasi tentang transaksi, operasi, dan variabel lokal. Berikut adalah metode yang terdapat pada kelas OCC :

a. `_rollback(num)`: Melakukan *rollback* pada transaksi dengan nomor num.
b. `_acquire_lock(op, num, item)`: Mencoba mengakuisisi *lock* untuk operasi op oleh transaksi num pada *item data* item.
c. `_release_locks(num)`: Melepas seluruh *lock* yang sedang dipegang transaksi num.
d. `_execute(trans)`: Metode untuk memproses eksekusi operasi trans, mulai dari mengakuisisi *lock*, memasukkannya ke hasil akhir/*queue*, maupun melepas *lock*.
e. `simulate(transaction)`: Mensimulasikan seluruh operasi yang ada dalam transaction dan mendapatkan jadwal akhir.

`2pl/FileHandler.py`

Terdapat kelas FileHandler yang memungkinkan untuk membantu proses pembacaan *file*, mulai dari inisiasi hingga seluruh karakter pada *file* terbaca dan dapat diuraikan sesuai dengan kebutuhan.

`2pl/main.py`

Pada *file* ini terdapat fungsi `main()` yang mengeksekusi program utama dengan membaca argumen berupa nama *file*. Fungsi ini juga akan menginisialisasi FileHandler dan 2PL, lalu menjalankan protokol 2PL yang telah disebutkan diatas. Selain itu juga terdapat main program yang menjalankan program utama jika dijalankan sebagai *script*.

Proses implementasi algoritma berlangsung dengan baik dan fungsionalitas protokol berhasil untuk dibuat dan berjalan sebagaimana mestinya.

b. *Optimistic Concurrency Control* (OCC)

Optimistic Concurrency Control atau OCC merupakan salah satu protokol yang digunakan oleh DBMS untuk mengatasi permasalahan konkurensi pada sebuah transaksi. Protokol ini dikategorikan sebagai "optimistik" dikarenakan protokol mengasumsikan seluruh transaksi dapat berjalan secara konkuren ketika transaksi dijalankan. Dalam protokol ini, tidak dilakukan pengecekan saat transaksi dijalankan.

Pembaruan dalam transaksi (*write* atau *update*) tidak diterapkan langsung ke basis data hingga akhir transaksi tercapai (melakukan *commit*). Semua pembaruan diterapkan ke salinan lokal *item data* yang disimpan untuk transaksi. Di akhir eksekusi transaksi, terdapat fase validasi yang memeriksa apakah terdapat pembaruan transaksi yang melanggar *serializability*. Jika tidak ada pelanggaran, maka transaksi akan dieksekusi dan memperbaharui nilai pada basis data. Jika tidak, maka akan dilakukan *rollback* terhadap seluruh operasi yang terdapat dalam transaksi dan dijalankan kembali tepat setelah *state aborted* tercapai.

Terdapat beberapa fase (*phase*) pada *Optimistic Concurrency Control* (OCC), yaitu sebagai berikut.

a. ***Begin*** : Tahap dilakukannya pencatatan *timestamp* yang menjadi tanda dimulainya suatu transaksi.
b. ***Read and Execution (Modify) phase*** : Tahap dilakukannya pembacaan data pada basis data dan penulisan data hanya pada salinan lokal data saja.
c. ***Validation phase*** : Tahap dimana transaksi telah menjalankan semua operasi yang terdapat didalamnya dan melakukan ”*validation test*” untuk menentukan apakah salinan lokal data dapat dituliskan pada basis data tanpa melanggar *serializability*. Pada ”*validation test*” akan diperiksa apakah terdapat transaksi lain yang berjalan konkuren dengan transaksi tersebut, tetapi telah melakukan modifikasi data pada basis data untuk salinan lokal data tersebut.

Adapun terdapat 3 buah *timestamp* pada transaksi yang berada pada *Serial Optimistic Concurrency Control* (OCC), yaitu sebagai berikut.

a. **StartTS(Ti)** : Waktu ketika Transaksi Ti memulai *Read* and *Execution* (*Modify*) *phase*.
b. **ValidationTS(Ti)** : Waktu ketika Transaksi Ti memulai proses validasi (memasuki *validation phase*).
c. **FinishTS(Ti)** : Waktu ketika Transaksi Ti selesai melakukan penulisan data pada basis data (waktu *write phase* berakhir).

Untuk proses validasi pada validation phase, terdapat kondisi yang menyatakan bahwa Transaksi sukses melakukan validasi dimana untuk semua Ti dengan TS(Ti) < TS(Tj) harus memenuhi salah satu kondisi berikut.

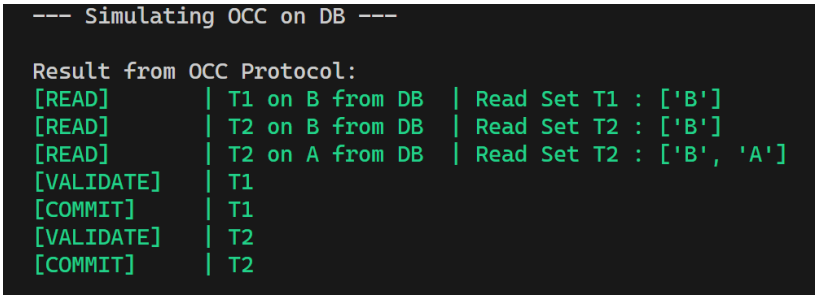
a. finishTS(Ti) < startTS(Tj).
b. startTS(Tj) < finishTS(Ti) < validationTS(Tj) dan *Write Set* pada Ti tidak beririsan dengan *Read Se*t dan *Write Set* dari Tj .

Implementasi dari Protokol OCC terdapat pada tautan berikut. Adapun untuk menjalankannya, perlu dilakukan pemindahan direktori ke direktori occ, kemudian ketikkan perintah berikut.

```
python3 main.py test/<nama test case>
```

Pengujian Hasil Implementasi

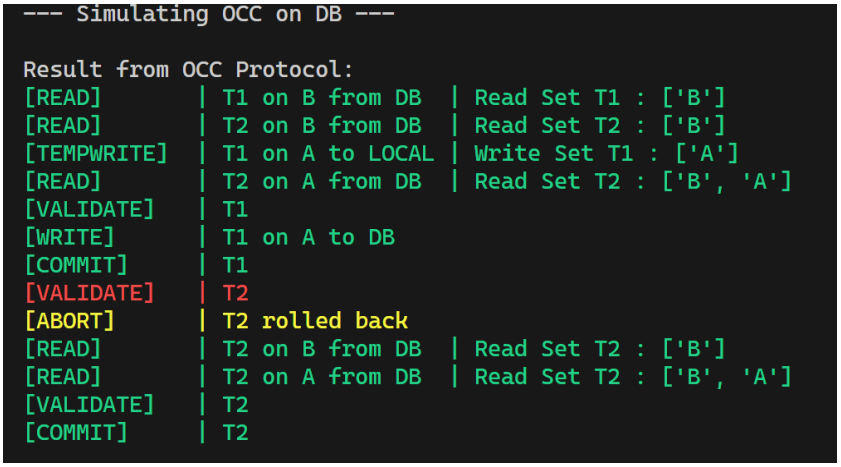
Uji Kasus 1

Kasus Uji - `test1.txt`
R1(B); R2(B); R2(A); C1; C2
Hasil Pengujian


```
--- Simulating OCC on DB ---

Result from OCC Protocol:
[READ]       | T1 on B from DB  | Read Set T1 : ['B']
[READ]       | T2 on B from DB  | Read Set T2 : ['B']
[READ]       | T2 on A from DB  | Read Set T2 : ['B', 'A']
[VALIDATE]   | T1
[COMMIT]     | T1
[VALIDATE]   | T2
[COMMIT]     | T2
```

Melalui hasil program dapat diamati bahwa seluruh proses validasi dapat berjalan dengan baik, ditandai dengan warna hijau pada terminal. Operasi yang terjadi pada kedua transaksi hanya merupakan operasi *read* sehingga tidak melakukan perubahan pada setiap *item data* yang ada.

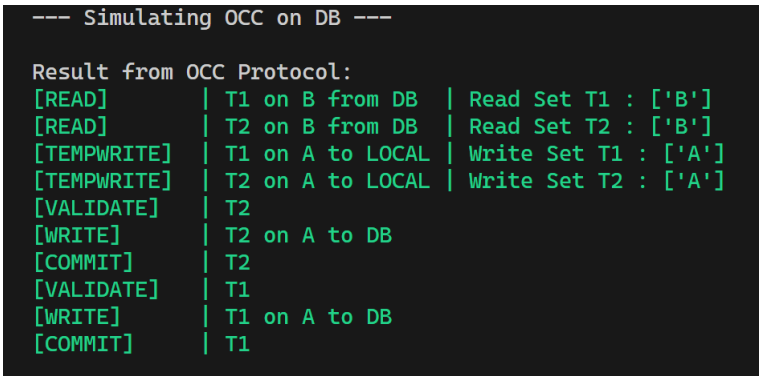
Uji Kasus 2

Kasus Uji - `test2.txt`
R1(B); R2(B); W1(A); R2(A); C1; C2
Hasil Pengujian


```
--- Simulating OCC on DB ---

Result from OCC Protocol:
[READ]       | T1 on B from DB  | Read Set T1 : ['B']
[READ]       | T2 on B from DB  | Read Set T2 : ['B']
[TEMPWRITE]  | T1 on A to LOCAL | Write Set T1 : ['A']
[READ]       | T2 on A from DB  | Read Set T2 : ['B', 'A']
[VALIDATE]   | T1
[WRITE]      | T1 on A to DB
[COMMIT]     | T1
[VALIDATE]   | T2
[ABORT]      | T2 rolled back
[READ]       | T2 on B from DB  | Read Set T2 : ['B']
[READ]       | T2 on A from DB  | Read Set T2 : ['B', 'A']
[VALIDATE]   | T2
[COMMIT]     | T2
```

Melalui hasil program dapat diamati bahwa seluruh proses validasi T2 tidak berlangsung dengan baik sehingga harus melakukan *rollback*. Jika diamati dengan baik, T1 adalah transaksi yang memiliki *timestamp* lebih kecil dari T2. Lebih lanjut, finishTS(T1) berada di antara startTS(T2) dan validationTS(T2). Karena T2 membaca nilai A hasil modifikasi dari T1, artinya terdapat irisan *item data* pada *read set* dari T2 dengan *write set* dari T1, sehingga hasil dari *validation test* pada T2 bernilai *fail* dan harus melakukan *rollback*.

Uji Kasus 3

Kasus Uji - `test3.txt`
R1(B); R2(B); W1(A); W2(A); C2; C1
Hasil Pengujian


```
--- Simulating OCC on DB ---

Result from OCC Protocol:
[READ]       | T1 on B from DB  | Read Set T1 : ['B']
[READ]       | T2 on B from DB  | Read Set T2 : ['B']
[TEMPWRITE]  | T1 on A to LOCAL | Write Set T1 : ['A']
[TEMPWRITE]  | T2 on A to LOCAL | Write Set T2 : ['A']
[VALIDATE]   | T2
[WRITE]      | T2 on A to DB
[COMMIT]     | T2
[VALIDATE]   | T1
[WRITE]      | T1 on A to DB
[COMMIT]     | T1
```

Melalui hasil program dapat diamati bahwa seluruh proses validasi berlangsung dengan baik. Jika diamati dengan baik, T1 adalah transaksi yang memiliki *timestamp* lebih kecil dari T2. Lebih lanjut, finishTS(T1) berada di antara startTS(T2) dan validationTS(T2). Karena tidak terdapat irisan *item data* pada *read set* dari T2 dengan *write set* dari T1, maka memenuhi kondisi *validation test* ke 2, sehingga hasil dari *validation test* pada T2 bernilai valid.

Uji Kasus 4

Kasus Uji - `test4.txt`
R1(E); R2(G); R1(F); R2(F); R1(G); R2(E); W1(F); W2(E); W1(G); C2; C1

Hasil Pengujian

```
--- Simulating OCC on DB ---

Result from OCC Protocol:
[READ]       | T1 on E from DB  | Read Set T1 : ['E']
[READ]       | T2 on G from DB  | Read Set T2 : ['G']
[READ]       | T1 on F from DB  | Read Set T1 : ['E', 'F']
[READ]       | T2 on F from DB  | Read Set T2 : ['G', 'F']
[READ]       | T1 on G from DB  | Read Set T1 : ['E', 'F', 'G']
[READ]       | T2 on E from DB  | Read Set T2 : ['G', 'F', 'E']
[TEMPWRITE]  | T1 on F to LOCAL | Write Set T1 : ['F']
[TEMPWRITE]  | T2 on E to LOCAL | Write Set T2 : ['E']
[TEMPWRITE]  | T1 on G to LOCAL | Write Set T1 : ['F', 'G']
[VALIDATE]   | T2
[WRITE]      | T2 on E to DB
[COMMIT]     | T2
[VALIDATE]   | T1
[ABORT]      | T1 rolled back
[READ]       | T1 on E from DB  | Read Set T1 : ['E']
[READ]       | T1 on F from DB  | Read Set T1 : ['E', 'F']
[READ]       | T1 on G from DB  | Read Set T1 : ['E', 'F', 'G']
[TEMPWRITE]  | T1 on F to LOCAL | Write Set T1 : ['F']
[TEMPWRITE]  | T1 on G to LOCAL | Write Set T1 : ['F', 'G']
[VALIDATE]   | T1
[WRITE]      | T1 on F to DB
[WRITE]      | T1 on G to DB
[COMMIT]     | T1
```

Melalui hasil program dapat diamati bahwa seluruh proses validasi T1 tidak berlangsung dengan baik sehingga harus melakukan *rollback*. Jika diamati dengan baik, T1 adalah transaksi yang memiliki *timestamp* lebih kecil dari T2. Lebih lanjut, finishTS(T1) tidak berada di antara startTS(T2) dan validationTS(T2), sehingga T1 gagal memenuhi kondisi *validation test* ke 2 dan harus melakukan *rollback*.

Desain dan Implementasi Algoritma

`occ/Lib.py`

Terdapat kelas Operation yang mewakili operasi pada suatu transaksi, seperti membaca (R) atau menulis (W) pada suatu item. Operasi yang ada menggunakan aturan penulisan seperti yang terdapat pada salindia kuliah, yaitu :

f. Rx(A) menyatakan operasi *read* pada *item data* A oleh transaksi Tx.
g. Wx(A) menyatakan operasi *write* pada *item data* B oleh transaksi Tx.
h. Cx menyatakan operasi *commit* oleh transaksi Tx.

Selain itu kelas ini juga menyimpan informasi seperti nomor transaksi (t_num) dan detail *item data* yang terlibat.

Pada *file* ini juga terdapat kelas OCCTransaction yang mewakili komponen dari transaksi yang menggunakan protokol OCC. Kelas ini menyimpan informasi seperti *read set*, *write set*, dan *timestamp* dari transaksi tersebut.

occ/Occ.py

Pada *file* ini diimplementasikan kelas utama yaitu kelas OCC yang mewakili fungsionalitas protokol *Optimistic Concurrency Control*. Kelas ini menyimpan berbagai informasi tentang transaksi, operasi, dan variabel lokal. Berikut adalah metode yang terdapat pada kelas OCC :

d. `find_transaction(t_num)`: Mencari transaksi berdasarkan nomor transaksi.
e. `prepare(op)`: Menyiapkan kondisi awal untuk protokol OCC.
f. `process(op)`: Mengeksekusi operasi tertentu (baca, tulis, validasi, *commit*).
i. `process_read(op), process_tempwrite(op), process_commit(op), process_validate(op)`: Metode untuk memproses masing-masing jenis operasi *read*, *write*, *commit*, dan *validate* secara berturut-turut.
j. `write(op)`: Menyimulasikan penulisan transaksi ke basis data.
k. `handle_abort(t_num)`: Menangani *abort* dan melakukan *rollback* penuh.
l. `run_operation(op)`: Menjalankan setiap operasi pada setiap transaksi.
m. `run()`: Menjalankan protokol OCC pada seluruh rangkaian operasi.

occ/FileHandler.py

Terdapat kelas FileHandler yang memungkinkan untuk membantu proses pembacaan *file*, mulai dari inisiasi hingga seluruh karakter pada *file* terbaca dan dapat diuraikan sesuai dengan kebutuhan.

occ/main.py

Pada *file* ini terdapat fungsi `main()` yang mengeksekusi program utama dengan membaca argumen berupa nama *file*. Fungsi ini juga akan menginisialisasi FileHandler dan OCC, lalu menjalankan protokol OCC yang telah disebutkan diatas. Selain itu juga terdapat main program yang menjalankan program utama jika dijalankan sebagai *script*.

Proses implementasi algoritma berlangsung dengan baik dan fungsionalitas protokol berhasil untuk dibuat dan berjalan sebagaimana mestinya.

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

Multiversion Concurrency Control atau MVCC merupakan salah satu protokol yang digunakan oleh DBMS untuk mengatasi permasalahan konkurensi pada transaksi. Skema *multiversion* memiliki arti bahwa basis data akan menyimpan seluruh versi dari seluruh data yang tercatat pada basis data untuk diakses oleh transaksi yang sedang berjalan dalam rangka meningkatkan *degree of concurrency* antar transaksi yang terlibat.

Setiap operasi *write* yang berhasil ditambahkan pada basis data akan membentuk versi baru dari data tersebut. Selain itu, setiap versi dari data yang tercatat di dalam basis data akan diberikan *write timestamp* dan *read timestamp* untuk mencatat versi mana yang boleh dibaca oleh transaksi terkait. Ketika suatu operasi *read* membaca suatu data, maka versi yang disediakan oleh *concurrency controller* haruslah sesuai dengan *timestamp* dari mulainya transaksi tersebut agar konsistensi dan integritas basis data tetap terjaga dengan baik.

Pada *Multiversion Timestamp Ordering*, setiap *item data Q* akan memiliki sebuah sekuens versi <$Q1$, $Q2$, …, Qm> yang setiap versinya (Qk) menyimpan tiga buah informasi, yaitu

a. **Content** : Nilai dari versi Qk.
b. **W-TS(Qk)** : *Timestamp* dari transaksi yang membuat versi Qk.
c. **R-TS(Qk)** : *Timestamp* terbesar dari transaksi yang berhasil membaca versi Qk.

Dengan menggunakan seluruh informasi tersebut, maka dapat dijabarkan protokol *Multiversion Timestamp Ordering*. Misalkan terdapat Qk yang menyatakan versi dari Q yang memiliki *write timestamp* lebih kecil atau sama dengan TS(Ti).

Jika operasi yang akan dijalankan adalah operasi ***read*(Q)**, maka :

a. Nilai yang dikembalikan adalah *content* dari versi Qk.
b. Jika R-TS(Qk) < TS(Ti), maka ubah nilai R-TS(Qk) = TS(Ti).

Jika operasi yang akan dijalankan adalah operasi ***write*(Q)**, maka :

a. Jika TS(Ti) < R-TS(Qk), maka transaksi Ti harus *rollback*.
b. Jika TS(Ti) = W-TS(Qk), maka nilai *content* dari Qk akan ditimpa.

c. Selain dua kondisi tersebut, versi baru dari *Q*, *Q*i, akan dibuat dengan nilai R-TS(*Q*i) dan W-TS(*Q*i) akan diinisialisasi dengan TS(Ti).

Berdasarkan penjelasan protokol tersebut, dapat disimpulkan bahwa operasi *read* pasti akan sukses karena akan selalu dapat menemukan versi yang sesuai secepat mungkin. Akan tetapi, operasi *write* dari transaksi Ti akan ditolak jika beberapa dari transaksi lain, Tj, yang harus membaca nilai hasil *write* dari Ti sudah membaca versi yang dibuat oleh transaksi yang lebih tua dari Ti.

Implementasi dari protokol MVCC terdapat pada tautan berikut. Adapun untuk menjalankannya, perlu dilakukan pemindahan direktori ke direktori occ, kemudian ketikkan perintah berikut.

```
python3 main.py test/<nama test case>
```

Pengujian Hasil Implementasi

Uji Kasus

Kasus Uji - `test.txt`
R5(X); R2(Y); R1(Y); W3(Y); W3(Z); R5(Z); R2(Z); R1(X); R4(W); W3(W); W5(Y); W5(Z); C1; C2; C3; C4; C5
Hasil Pengujian

```
--- Simulating MVCC on DB ---

Result from MVCC Timestamp Ordering Protocol:
[READ]       | T5 on X from DB | Version X : X0(5,0);
[READ]       | T2 on Y from DB | Version Y : Y0(2,0);
[READ]       | T1 on Y from DB | Version Y : Y0(2,0);
[WRITE]      | T3 on Y to DB   | Version Y : Y0(2,0); Y3(3,3);
[WRITE]      | T3 on Z to DB   | Version Z : Z0(0,0); Z3(3,3);
[READ]       | T5 on Z from DB | Version Z : Z0(0,0); Z3(5,3);
[READ]       | T2 on Z from DB | Version Z : Z0(2,0); Z3(5,3);
[READ]       | T1 on X from DB | Version X : X0(5,0);
[READ]       | T4 on W from DB | Version W : W0(4,0);
[WRITE]      | T3 on W to DB   | Version W : W0(4,0);
[ABORT]      | T3 rolled back
[ABORT]      | T5 rolled back
[WRITE]      | T3 on Y to DB   | Version Y : Y0(2,0); Y6(6,6);
[WRITE]      | T3 on Z to DB   | Version Z : Z0(2,0); Z6(6,6);
[WRITE]      | T3 on W to DB   | Version W : W0(4,0); W6(6,6);
[READ]       | T5 on X from DB | Version X : X0(7,0);
[READ]       | T5 on Z from DB | Version Z : Z0(2,0); Z6(7,6);
[WRITE]      | T5 on Y to DB   | Version Y : Y0(2,0); Y6(6,6); Y7(7,7);
[WRITE]      | T5 on Z to DB   | Version Z : Z0(2,0); Z6(7,6); Z7(7,7);
[COMMIT]     | T1
[COMMIT]     | T2
[COMMIT]     | T3
[COMMIT]     | T4
[COMMIT]     | T5
```

Adapun berikut adalah tabel urutan rangkaian transaksi awal

T1	T2	T3	T4	T5
				R5(X)
	R2(Y)			
R1(Y)				
		W3(Y)		
		W3(Z)		
				R5(Z)
	R2(Z)			
R1(X)				
			R4(W)	
		W3(W)		
		W5(Y)		
		W5(Z)		

C1				
	C2			
		C3		
			C4	
				C5

Melalui hasil yang diperoleh dari program dapat diamati bahwa W3(Y) dan W3(Z) akan membuat versi baru dari Y dan Z yaitu berturut-turut Y3(3,3) dan Z3(3,3). Karena hasil tersebut, maka R5(Z) akan membaca hasil Z3(3,3) akibat sudah ada nilai Z dengan *timestamp* Z yang lebih besar, tetapi lebih kecil dari TS(T5). Proses ini akan membuat memperbaharui versi Z3 menjadi Z3(5,3).

Berbeda dengan *Timestamp Ordering* yang biasa, R2(Z) tidak gagal dan harus melakukan *rollback*, melainkan akan membaca versi Z0(0,0) yang masih dapat diakses. Proses ini akan membuat memperbaharui versi Z0 menjadi Z3(2,0).

Jika diperhatikan, R3(W) akan mendapatkan versi Z0(0,0) karena belum ada versi yang lain. Permasalahan timbul karena nilai R-TS(W0) adalah 4, padahal nilai TS(T3) adalah 3, sehingga R3(W) tidak berhasil menuliskan nilai baru karena sudah terlanjur dibaca oleh T4. Karena memenuhi kondisi TS(T4) < R-TS(W0), maka T3 harus mengalami *rollback*. Lebih lanjut, proses *rollback* ini membuat semua pembacaan (*read*) yang berbasis nilai versi yang dihasilkan dari T3 juga harus dibatalkan, konsep ini disebut sebagai *cascading rollback*. Dalam rangkaian transaksi ini, R5(Z) membaca versi Z yang dihasilkan dari T3, sehingga T5 juga harus mengalami *rollback*. Setelah transaksi dinyatakan *abort*, maka T3 dan T5 harus dijalankan kembali dari awal.

Desain dan Implementasi Algoritma

`mvcc/Lib.py`

Pada *file* ini terdapat kelas TransactionItem yang merepresentasikan *item data* transaksi dengan versi yang berbeda pada basis data. Kelas ini menyimpan informasi seperti label item, versi, *timestamp* pembacaan (read_ts), *timestamp* penulisan (write_ts),

dan dependen (item yang telah dibaca atau ditulis) untuk mendukung kemudahan dalam melakukan *cascading rollback*.

Terdapat kelas Operation yang mewakili operasi pada suatu transaksi, seperti membaca (R) atau menulis (W) pada suatu item. Operasi yang ada menggunakan aturan penulisan seperti yang terdapat pada salindia kuliah, yaitu :

a. Rx(A) menyatakan operasi *read* pada *item data* A oleh transaksi Tx.
b. Wx(A) menyatakan operasi *write* pada *item data* B oleh transaksi Tx.
c. Cx menyatakan operasi *commit* oleh transaksi Tx.

Selain itu kelas ini juga menyimpan informasi seperti nomor transaksi (t_num) dan detail *item data* yang terlibat.

Pada *file* ini juga terdapat kelas MVCCTransaction yang mewakili komponen dari transaksi yang menggunakan protokol MVCC. Kelas ini menyimpan informasi seperti *read set*, *write set*, dan *timestamp* dari transaksi tersebut.

Dalam rangka membantu proses *version controlling* dari setiap versi *item data* pada basis data, dibuatlah kelas VersionControl yang bertanggung jawab atas pengelolaan versi dari setiap *item data* dalam basis data. Kelas ini menyediakan metode untuk menambahkan versi baru dan mendapatkan semua versi dari suatu *item data* yang telah disimpan.

`mvcc/Mvcc.py`

Pada *file* ini diimplementasikan kelas utama yaitu kelas MVCC yang mewakili fungsionalitas protokol *Multiversion Timestamp Ordering Concurrency Control*. Kelas ini menyimpan berbagai informasi tentang transaksi, operasi, dan kontrol versi. Berikut adalah metode yang terdapat pada kelas MVCC :

a. `find_transaction(t_num)`: Mencari transaksi berdasarkan nomor transaksi.
b. `prepare_item(op)`: Menyiapkan kondisi awal untuk MVCC *Timestamp Ordering* dengan menambahkan item ke kontrol versi jika belum ada.
c. `prepare(op)`: Menyiapkan kondisi awal setiap kali operasi perlu dijalankan.
g. `process(op)`: Mengeksekusi operasi tertentu (baca, tulis, validasi, *commit*).

h. `process_read(op), process_tempwrite(op), process_commit(_)`: Metode untuk memproses masing-masing jenis operasi *read*, *write*, dan *commit* secara berturut-turut.
i. `handle_abort(t_num)`: Menangani *abort* dan melakukan *rollback* penuh, termasuk *cascading rollback*, jika transaksi dibatalkan
j. `run_operation(op)`: Menjalankan setiap operasi pada setiap transaksi.
k. `run()`: Menjalankan protokol MVCC pada seluruh rangkaian operasi.

`mvcc/FileHandler.py`

Terdapat kelas FileHandler yang memungkinkan untuk membantu proses pembacaan *file*, mulai dari inisiasi hingga seluruh karakter pada *file* terbaca dan dapat diuraikan sesuai dengan kebutuhan.

`mvcc/main.py`

Pada *file* ini terdapat fungsi `main()` yang mengeksekusi program utama dengan membaca argumen berupa nama *file*. Fungsi ini juga akan menginisialisasi FileHandler dan MVCC, lalu menjalankan protokol MVCC yang telah disebutkan diatas. Selain itu juga terdapat main program yang menjalankan program utama jika dijalankan sebagai *script*.

3. Eksplorasi *Recovery*

a. *Write-Ahead Log*

Write-Ahead Log adalah sebuah metode yang dimana *log record* akan melakukan *update* informasi dari suatu data ke dalam *log* sebelum data tersebut dimasukkan ke dalam *secondary storage,* jadi metode ini akan menjamin bahwa data akan dituliskan terlebih dahulu ke *log* sebelum ke *secondary storage* setiap saat. hal ini akan menjamin *Atomicity* pada transaksi. Kemudian metode WAL ini juga harus memastikan bahwa seluruh perubahan yang terjadi pada suatu transaksi termasuk *commit* sudah masuk ke *log records* pada *stable storage* sebelum transaksi tadi di *return* ke pengguna. Hal ini akan menjamin *Durability* pada transaksi.

Dengan menggunakan metode WAL ini, jumlah penulisan *disk* akan berkurang secara signifikan, hal ini dikarenakan hanya *file log* saja yang perlu di-*flush* ke *disk* untuk menjamin bahwa transaksi dilakukan, bukan setiap *file* data yang diubah oleh transaksi. *File log* ditulis secara berurutan sehingga biaya sinkronisasi *log* jauh lebih murah daripada biaya pembersihan halaman data. Hal ini terutama berlaku untuk *server* yang menangani banyak transaksi kecil yang menyentuh berbagai bagian penyimpanan data. Selanjutnya, ketika *server* memproses banyak transaksi kecil bersamaan, satu ‘fsync’ dari *file log* mungkin cukup untuk melakukan banyak transaksi.

WAL juga mendukung pencadangan *online* dan pemulihan tepat waktu. Dengan mengarsipkan data WAL, DBMS dapat mendukung pengembalian ke waktu instan yang dicakup oleh data WAL yang tersedia. Pada implementasinya nanti cukup menginstal cadangan fisik *database* sebelumnya, dan memutar ulang *log* WAL sejauh waktu yang diinginkan. Terlebih lagi, pencadangan fisik tidak harus berupa *snapshot* instan dari status *database* jika dibuat selama beberapa periode waktu, memutar ulang log WAL untuk periode tersebut akan memperbaiki inkonsistensi internal.

b. *Continuous Archiving*

Continuous Archiving adalah proses menyalin file WAL (*Write-Ahead Log*) ke *secondary storage*. Keuntungan dari *Continuous Archiving* ini adalah *cluster database backup* dapat menggunakan *file* WAL yang disimpan tersebut untuk digunakan nanti sebagai data landasan yang direplikasi maupun di-*recovery* dengan menggunakan metode

Point-in-Time Recovery (PITR). Namun, saat mengimplementasikan *continuous archiving* di PostgreSQL, ada beberapa batasan terkait proses *continuous archiving* yang dapat dilakukan. Batasan pertama adalah bahwa operasi pemasangan *hash index* tidak akan tersimpan dalam file WAL, sehingga ketika kita melakukan operasi modifikasi pada tabel (*insert, update, delete*) menggunakan *hash index* maka hasilnya akan salah. Batasan kedua adalah jika perintah *CREATE DATABASE* sedang dijalankan selagi *backup* berjalan dan terdapat modifikasi pada basis data *template* ketika proses *backup*, maka hal tersebut memungkinkan modifikasi baru tersebut untuk ikut terbawa dalam *file recovery*. Batasan ketiga adalah perintah *CREATE TABLESPACE* disimpan pada *file* WAL dengan lokasi yang absolut, sehingga ketika file WAL dijalankan pada mesin yang berbeda banyak hal yang tidak diharapkan akan muncul dan dapat menyebabkan masalah yang sama pada mesin, seperti tidak sengaja melakukan *overwrite* terhadap berkas aslinya.

c. *Point-in-Time Recovery*

Point-in-Time Recovery (PITR) adalah teknik pemulihan pada sistem basis data yang memungkinkan pengguna untuk mengembalikan database ke keadaan tertentu pada titik waktu tertentu di masa lalu. Dengan kata lain, PITR memungkinkan untuk memulihkan data hingga suatu waktu tertentu sebelum terjadinya kegagalan atau kesalahan. Untuk melakukan ini pada *PostgreSQL* perlu dibuat salinan penuh dari *database* yang disebut sebagai *base backup. Base backup* ini mencakup seluruh *database* pada saat *backup* dibuat. Setelah itu, setiap transaksi pada *database, PostgreSQL* akan mencatat setiap perubahan yang dilakukan pada data ke dalam *file* log transaksi yang disebut *Write Ahead Logs* (WAL). *Log* ini berisi urutan perubahan yang telah dilakukan pada *database*. Jika terjadi kegagalan, *base backup* yang sudah dibuat dan *log* transaksi dapat digunakan untuk melakukan pemulihan hingga titik waktu yang diinginkan. UNtuk melakukan PITR perlu dibuat sebuah file bernama *recovery.conf* pada directory *PostgreSQL . file* ini berisi konfigurasi pemulian seperti waktu atau nomor log transaksi yang diinginkan. Setelah konfigurasi dibuat pemulihan akan langsung dilakukan ketika server *PostgreSQL* dimulai ulang.

d. Simulasi Kegagalan pada PostgreSQL

Sebelum melakukan simulasi kegagalan, akan disiapkan terlebih dahulu *directory* untuk *archive logs*. Hal ini bisa dilakukan dengan cara mengetikan

```
sudo -H -u postgres mkdir /var/lib/postgresql/pg_log_archive
```

pada terminal. Setelah itu akan dilakukan konfigurasi pada *postgresql*. Konfigurasi berada pada path `/etc/postgresql/10/main/postgresql.conf`. Konfigurasi yang perlu diubah adalah sebagai berikut:

```
wal_level = replica
archive_mode = on # (change requires restart)
Archive_command = 'test ! -f
/var/lib/postgresql/pg_log_archive/%f && cp %p
/var/lib/postgresql/pg_log_archive/%f'
```

Setelah mengatur konfigurasi, akan dibuat *database* yang akan dilakukan uji coba.

```
postgres@LAPTOP-CIIB8VGQ:~$ psql -c "create database recover;"
CREATE DATABASE
```

Gambar 3.1. Inisiasi *database*.

```
postgres@LAPTOP-CIIB8VGQ:~$ psql recover -c "
create table indeks(
id integer primary key,
nama varchar(255),
indeks varchar(8));
"
CREATE TABLE
postgres@LAPTOP-CIIB8VGQ:~$
```

Gambar 3.2. Membuat tabel.

Insert data pada *database*

```
postgres@LAPTOP-CIIB8VGQ:~$ psql recover -c "select * from indeks;"
 id |  nama  | indeks
----+--------+--------
  1 | Fahkry | A
  2 | Leon   | A
  3 | Ulung  | A
  4 | Nathan | A
(4 rows)
```

Gambar 3.3. Nilai awal *database*.

Switch WAL

```
postgres@LAPTOP-CIIB8VGQ:~$ psql -c "select pg_switch_wal();"
 pg_switch_wal
---------------
 0/40004A0
(1 row)
```

Gambar 3.4. Menyimpan *log*.

Buat *basebackup*

```
postgres@LAPTOP-CIIB8VGQ:~$ pg_basebackup -Ft -D /var/lib/postgresql/db_file_backup
postgres@LAPTOP-CIIB8VGQ:~$ ls
10  db_file_backup  pg_log_archive
postgres@LAPTOP-CIIB8VGQ:~$ ls db_file_backup/
base.tar  pg_wal.tar
```

Gambar 3.5. Membuat *basebackup*.

Stop PostgreSQL

```
ulung@LAPTOP-CIIB8VGQ:~$ sudo service postgresql stop
 * Stopping PostgreSQL 10 database server
ulung@LAPTOP-CIIB8VGQ:~$ sudo service postgresql status
10/main (port 5432): down
ulung@LAPTOP-CIIB8VGQ:~$
```

Gambar 3.6. Matikan server *PostgreSQL*.

Hapus `/var/lib/postgresql/10/main/*` untuk menghilangkan data pada *database*

```
ulung@LAPTOP-CIIB8VGQ:~$ ls /var/lib/postgresql/10/main/
ls: cannot access '/var/lib/postgresql/10/main/': No such file or directory
ulung@LAPTOP-CIIB8VGQ:~$
```

Gambar 3.7. Hasil penghapusan *database*.

Setelah dihapus terlihat tidak ada *file* apapun di *directory* tersebut.

Saat terjadi kegagalan

```
postgres@LAPTOP-CIIB8VGQ:~$ psql recover -c "select * from indeks;"
psql: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: No such file or directory
        Is the server running locally and accepting connections on that socket?
postgres@LAPTOP-CIIB8VGQ:~$
```

Gambar 3.8. Eksekusi *query* saat server mati.

Setelah itu akan dilakukan *recovery* pada database.

- masukkan *command* :

```
sudo tar xvf/var/lib/postgresql/db_file_backup/base.tar
-C /var/lib/postgresql/10/main/
```

```
sudo tar xvf
/var/lib/postgresql/db_file_backup/pg_wal.tar -C
/var/lib/postgresql/10/main/pg_wal/
```

- Tambahkan *file* recovery.conf pada *directory* `var/lib/postgresql/10/main/recovery.conf`

  Isi file tersebut dengan

```
restore_command=                                        'cp
/var/lib/postgresql/pg_log_archive/%f %p'
recovery_target_time = '2023-12-01 16:20:00'
```

  Pada `recovery_target_time` dimasukkan waktu spesifik ke kondisi database ingin di-*restore*.

- Hidupkan kembali *database*

```
ulung@LAPTOP-CIIB8VGQ:~$ sudo service postgresql start
 * Starting PostgreSQL 10 database server
ulung@LAPTOP-CIIB8VGQ:~$ sudo service postgresql status
10/main (port 5432): online
ulung@LAPTOP-CIIB8VGQ:~$
```

Gambar 3.9. Menghidupkan kembali PostgreSQL.

- Data setelah *recovery*

```
postgres@LAPTOP-CIIB8VGQ:~$ psql recover -c "select * from indeks;"
 id |  nama  | indeks
----+--------+--------
  1 | Fahkry | A
  2 | Leon   | A
  3 | Ulung  | A
  4 | Nathan | A
(4 rows)
```

Gambar 3.10. Nilai pada *database* setelah *recovery* dilakukan.

Terlihat data setelah *recovery* sama seperti data sebelum terjadinya kegagalan, hal ini menunjukan *recovery* berhasil dilakukan. Recovery dilakukan dimulai dari *base backup* untuk dimulainya recovery. Kemudian dilakukan pembacaan log secara *roll-forward* dan transaksi yang ada pada log yang dibaca akan dieksekusi terhadap database. Hingga akhirnya ditemukan waktu pada log yang sesuai dengan target waktu yang diinputkan oleh admin.

4. Pembagian Kerja

NIM	Nama	Bagian
13519032	Muhammad Fahkry Malta	Eksplorasi *Transaction Isolation* (*Serializable*, *Repeatable Read*), Eksplorasi *Recovery* (*Write-Ahead Log*, *Continuous Archiving*).
13521108	Michael Leon Putra Widhi	Implementasi *concurrency control* (*Framework Python*, OCC, MVCC).
13521122	Ulung Adi Putra	Implementasi *concurrency control* (*Two-Phase Locking*).
13521172	Nathan Tenka	Eksplorasi *Transaction Isolation* (*Read Committed*, *Read Uncommitted*), Eksplorasi *Recovery* (*Point-in-Time Recovery*, Simulasi Kegagalan pada PostgreSQL).

Referensi

26.3. Continuous Archiving and Point-in-Time Recovery (PITR). (2023, November 9). PostgreSQL Documentation. https://www.postgresql.org/docs/current/continuous-archiving.html

PostgreSQL Backup & Point-In-Time recovery. (n.d.). Scaling PostgreSQL. https://www.scalingpostgres.com/tutorials/postgresql-backup-point-in-time-recovery/

13.2. Transaction isolation. (2023, November 9). PostgreSQL Documentation. https://www.postgresql.org/docs/current/transaction-iso.html