IF3140 MANAJEMEN BASIS DATA

MEKANISME CONCURRENCY CONTROL DAN RECOVERY

K03 Kelompok 02

Anggota :

Jason Rivalino	13521008
Muhamad Salman Hakim Alfarisi	13521010
Ahmad Nadil	13521024
Kartini Copa	13521026

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

1. Eksplorasi Transaction Isolation

Dalam PostgreSQL, terdapat empat jenis tingkatan isolasi transaksi yang dapat dipergunakan untuk mengamankan data yang ada . Tingkatan tersebut sifatnya beragam berdasarkan dengan larangan fenomena pembacaan yang dapat terjadi pada tiap tingkatan isolasi. Beberapa larangan fenomena pembacaan yang mungkin terjadi adalah sebagai berikut:

1. *Dirty read* -> melakukan pembacaan data yang ditulis oleh transaksi lain sebelum *commit*
2. *Nonrepeatable read* -> pembacaan ulang data yang telah dibaca dan ditemukan bahwa data telah diubah oleh transaksi lain yang sudah *commit*
3. *Phantom read* -> eksekusi ulang query pada transaksi yang menghasilkan baris berbeda karena perubahan transaksi
4. *Serialization anomaly* -> tidak konsistennya sekelompok grup transaksi yang *commit* bersamaan dengan kemungkinan urutan transaksi yang dijalankan

Adapun untuk tingkatan isolasi transaksi yang mungkin terjadi antara lain mulai dari Serializable, Repeatable Read, Read Committed, dan Read Uncommitted. Untuk pembagian antara tingkatan isolasi dengan fenomena yang mungkin terjadi dalam tiap tingkatannya adalah sebagai berikut:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Mungkin, namun untuk PostgreSQL tidak	Mungkin	Mungkin	Mungkin
Read committed	Tidak mungkin	Mungkin	Mungkin	Mungkin
Repeatable read	Tidak mungkin	Tidak mungkin	Mungkin, namun untuk PostgreSQL tidak	Mungkin
Serializable	Tidak mungkin	Tidak mungkin	Tidak mungkin	Tidak mungkin

a. Serializable

Untuk isolasi transaksi dengan tipe *serializable*, isolasi ini memiliki derajat yang paling tinggi dengan tingkat isolasi yang paling ketat jika dibandingkan dengan berbagai jenis isolasi transaksi yang lainnya. Dalam tingkat isolasi ini, isolasi memastikan agar semua transaksi yang terjadi seolah-olah terjadi secara serial dan tidak bersamaan sehingga transaksi yang ada tidak akan saling mempengaruhi antara yang satu dengan yang lainnya atau dengan kata lain, untuk satu sesi hanya bisa melakukan eksekusi untuk satu kelompok transaksi saja dan untuk kelompok transaksi lainnya hanya bisa melakukan *read* dan tidak bisa transaksi (di-*block*).

Pada tingkat isolasi *serializable*, semua larangan fenomena tidak akan mungkin terjadi mulai dari *dirty read* karena pembacaan data hanya bisa untuk data yang sudah *commit*, *nonrepeatable read* dan *phantom read* karena tidak ada perubahan data yang dilakukan oleh transaksi lain saat transaksi pertama berjalan sehingga eksekusi ulang *query* akan menghasilkan hasil yang sama, hingga tidak diperbolehkannya *serialization anomaly*.

Adapun proses implementasi dari isolasi transaksi untuk tipe *Serializable* adalah sebagai berikut:

1. Menjalankan *query* untuk mengatur *transaction isolation* pada kedua terminal transaksi dengan level *serializable* sebagai berikut:

T1	T2

nubes=# BEGIN;
BEGIN
nubes=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
nubes=*# SHOW TRANSACTION ISOLATION LEVEL;
transaction_isolation
--- 	
serializable
(1 row)
	nubes=# BEGIN;
	BEGIN
	nubes=*# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
	SET
	nubes=*# SHOW TRANSACTION ISOLATION LEVEL;
	transaction_isolation
	--- 	
	serializable
	(1 row)

Pada implementasi disini, proses dilakukan dengan menjalankan *query* BEGIN untuk masuk dalam *transaction*, setelahnya dapat dilakukan pengaturan untuk mengatur level isolasi transaksi menjadi serializable dengan *query* SET TRANSACTION ISOLATION LEVEL SERIALIZABLE. Untuk pengecekan apakah level isolasi

transaksi sudah benar berada pada level SERIALIZABLE, bisa dengan menjalankan SHOW TRANSACTION ISOLATION LEVEL.

2. Berikutnya, akan dilakukan proses *inserting* untuk menambahkan data baru pada transaksi T1. Disini untuk proses penambahan data pada transaksi T1 berhasil untuk dieksekusi.

T1

```
nubes=*# INSERT INTO benda(id,barang,jumlah) VALUES(8,'Meja',5);
INSERT 0 1
```

3. Setelah itu, akan dilakukan proses *inserting* untuk menambahkan data baru pada transaksi T2. Disini untuk proses penambahan data pada transaksi T2 harus menunggu T1 melakukan *commit* terlebih dahulu (tidak bisa melakukan penambahan data karena masih satu sesi yang sama).

T2

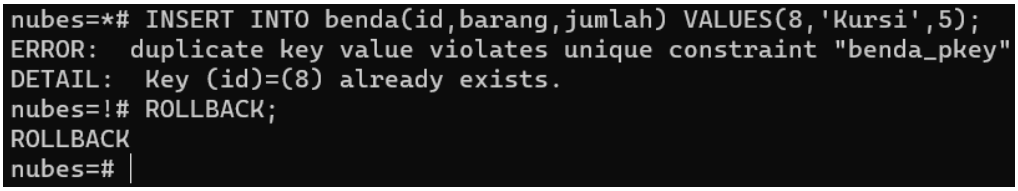
```
nubes=*# INSERT INTO benda(id,barang,jumlah) VALUES(8,'Kursi',5);
```

4. Kemudian pada terminal T1 akan dilakukan *commit*

T1

```
nubes=*# COMMIT;
COMMIT
nubes=#
```

5. Setelahnya, eksekusi *inserting* yang dilakukan pada terminal T2 akan berjalan, namun mengalami error yang terjadi dikarenakan adanya kesamaan nilai pada id karena proses query *inserting* sudah dilakukan sebelumnya sehingga selanjutnya ketika dilakukan *commit* akan langsung mengalami proses *rollback*.

T2


Berdasarkan proses ini, dapat dilihat semua larangan fenomena tidak mungkin terjadi sam sekali mulai dari *serialization anomaly*, *phantom read*, *nonrepeatable read*, hingga *dirty read*.

Berdasarkan proses yang telah dilakukan, urutan proses transaksi yang terjadi untuk pembuktian isolasi *repeatable read* yaitu sebagai berikut.

T1	**T2**	**Keterangan**
BEGIN;		Memulai transaksi T1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;		Mengatur transaction isolation agar menjadi serialization untuk transaksi T1
SHOW TRANSACTION ISOLATION LEVEL;		Melakukan pengecekan level isolasi transaksi T1
	BEGIN;	Memulai transaksi T2
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Mengatur transaction isolation agar menjadi serialization untuk transaksi T2
	SHOW TRANSACTION ISOLATION LEVEL;	Melakukan pengecekan level isolasi transaksi T2

INSERT INTO benda(id, barang, jumlah) VALUES(8,'Meja',5);		Melakukan penambahan data pada tabel benda untuk transaksi T1 (berhasil)
	INSERT INTO benda(id, barang, jumlah) VALUES(8,'Kursi',5);	Melakukan penambahan data pada tabel benda untuk transaksi T2 (penambahan gagal dan error, tetapi error baru dimunculkan setelah COMMIT pada T1 karena loading sebagai efek dari isolasi transaksi *serializable*)
COMMIT;		Melakukan *commit* untuk T1
	COMMIT;	Melakukan *rollback* untuk T2

b. Repeatable Read

Untuk isolasi transaksi dengan tipe *repeatable read*, isolasi yang ada tidak dapat melakukan pembacaan data yang belum mengalami *commit* ataupun perubahan *commit* selama eksekusi transaksi. Pembacaan data hanya dapat dilakukan untuk data yang telah mengalami *commit* sebelumnya.

Untuk isolasi dengan tipe *repeatable read*, derajat isolasi yang ada pada transaksi ini memungkinkan untuk terjadinya larangan fenomena yaitu *serialization anomaly* dan dalam beberapa jenis DBMS lain selain PostgreSQL, kondisi larangan fenomena *phantom read* juga mungkin terjadi. Sedangkan, untuk fenomena *dirty read* dan *nonrepeatable read* tidak akan mungkin terjadi karena pembacaan data yang dilakukan secara berulang tetap akan menerapkan *Snapshot Isolation* untuk menjamin hasil yang sama tanpa ada perubahan.

Adapun proses implementasi dari isolasi transaksi untuk tipe Repeatable Read adalah sebagai berikut:

1. Menjalankan *query* untuk mengatur transaction isolation pada kedua terminal transaksi dengan level *repeatable read* sebagai berikut:

T1	T2

nubes=# BEGIN;
BEGIN
nubes=*# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
nubes=*# SHOW TRANSACTION ISOLATION LEVEL;
transaction_isolation
--- 	
repeatable read
(1 row)
	nubes=# BEGIN;
	BEGIN
	nubes=*# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
	SET
	nubes=*# SHOW TRANSACTION ISOLATION LEVEL;
	transaction_isolation
	--- 	
	repeatable read
	(1 row)

Pada implementasi disini, proses dilakukan dengan menjalankan *query* BEGIN untuk masuk dalam *transaction*, setelahnya dapat dilakukan pengaturan untuk mengatur level isolasi transaksi menjadi *serializable* dengan *query* SET TRANSACTION ISOLATION LEVEL REPEATABLE READ. Untuk pengecekan apakah level isolasi transaksi sudah benar berada pada level REPEATABLE READ, bisa dengan menjalankan SHOW TRANSACTION ISOLATION LEVEL.

2. Setelah pengecekan level isolasi transaksi, akan dilakukan pengecekan untuk data awal yang ada. Pengecekan dilakukan dengan menjalankan *query* SELECT * from benda dari kedua terminal transaksi untuk mengecek keseluruhan isi tabel yang ada di awal.

T1	T2
nubes=*# SELECT * from benda; id | barang | jumlah ----+-----------+-------- 1 | Buku | 10 2 | Pensil | 5 3 | Penghapus | 10 4 | Penggaris | 2 5 | Batu | 1 6 | Kertas | 5 7 | Gunting | 10 (7 rows)	nubes=*# SELECT * from benda; id | barang | jumlah ----+-----------+-------- 1 | Buku | 10 2 | Pensil | 5 3 | Penghapus | 10 4 | Penggaris | 2 5 | Batu | 1 6 | Kertas | 5 7 | Gunting | 10 (7 rows)

3. Berikutnya, pada transaksi T1 akan dilakukan pengubahan berupa *update* perubahan data dan juga penambahan data (*inserting*). (*Note*: untuk perubahan data yang dapat dilakukan bisa juga dengan penghapusan data (*deleting*)). Adapun perubahan data *update* dilakukan untuk mengecek larangan fenomena *nonrepeatable read* dan *insert-delete* dilakukan untuk mengecek larangan fenomena *phantom read*. Setelahnya kembali dilakukan proses *query* SELECT * from benda untuk mengecek kondisi tabel pada transaksi T1 terbaru yang telah mengalami perubahan

T1

nubes=*# UPDATE benda SET jumlah=2+100 WHERE id=4;
UPDATE 1
nubes=*# INSERT INTO benda(id, barang, jumlah) VALUES(8,'Meja',5);
INSERT 0 1
nubes=*# SELECT * from benda ORDER BY id;
 id | barang | jumlah
----+-----------+--------
 1 | Buku | 10
 2 | Pensil | 5
 3 | Penghapus | 10
 4 | Penggaris | 102
 5 | Batu | 1
 6 | Kertas | 5
 7 | Gunting | 10
 8 | Meja | 5
(8 rows)

4. Setelah pembacaan, kemudian pada terminal T1 akan dilakukan *commit*.

T1

```
nubes=*# COMMIT;
COMMIT
nubes=#
```

5. Setelah *commit* pada T1 telah dijalankan, akan dilakukan pengecekan kembali pada kondisi tabel yang ada di terminal transaksi T2 dengan menjalankan *query* SELECT * from benda.

T2

```
nubes=*# SELECT * from benda;
 id |  barang   | jumlah
----+-----------+--------
  1 | Buku      |     10
  2 | Pensil    |      5
  3 | Penghapus |     10
  4 | Penggaris |      2
  5 | Batu      |      1
  6 | Kertas    |      5
  7 | Gunting   |     10
(7 rows)
```

Setelah dilakukan pembacaan ulang, ditemukan bahwa tabel yang ada pada tabel transaksi T2 masih tidak mengalami perubahan, padahal telah dilakukan *commit* pada terminal transaksi T1. Hal ini menunjukkan bahwa larangan fenomena *nonrepeatable read* dan *phantom read* tidak diperbolehkan dalam isolasi transaksi repeatable read karena pembacaan data yang dilakukan menghasilkan data yang tetap seperti awal meskipun telah terjadi perubahan data dan commit pada T1. Jika larangan fenomena terjadi, pada *nonrepeatable read*, data akan mengalami perubahan akibat *update* dan pada *phantom read*, data akan mengalami perubahan akibat *inserting*.

6. Jika pada transaksi T2 akan dijalankan *query* yang sama untuk melakukan *update* sama sepert pada tahapan nomor 3. Transaksi tidak akan melakukan *update* dan akan langsung *error* sehingga ketika dilakukan *commit* akan langsung mengalami proses *rollback*.

T2


Adapun *error* seperti ini dapat terjadi dikarenakan terjadi karena query ini berada pada antrian dibelakang query lain yang sudah dieksekusi sebelumnya.

Dari proses yang ada ini, dapat dilihat bahwa hampir semua jenis larangan fenomena (kecuali *serialization anomaly*) masih tidak mungkin terjadi pada level isolasi transaksi *repeatable read*.

Berdasarkan proses yang telah dilakukan, urutan proses transaksi yang terjadi untuk pembuktian isolasi repeatable read yaitu sebagai berikut.

T1	**T2**	**Keterangan**
BEGIN;		Memulai transaksi T1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		Mengatur transaction isolation agar menjadi repeatable read untuk transaksi T1
SHOW TRANSACTION ISOLATION LEVEL;		Melakukan pengecekan level isolasi transaksi T1
	BEGIN;	Memulai transaksi T2

	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	Mengatur transaction isolation agar menjadi repeatable read untuk transaksi T2
	SHOW TRANSACTION ISOLATION LEVEL;	Melakukan pengecekan level isolasi transaksi T2
SELECT * FROM benda;		Melihat keseluruhan data yang ada pada tabel benda pada transaksi T1 (data yang ditampilkan yaitu data awal)
	SELECT * FROM benda;	Melihat keseluruhan data yang ada pada tabel benda pada transaksi T2 (data yang ditampilkan yaitu data awal)
UPDATE benda SET jumlah = 2+100 WHERE id = 4;		Melakukan *update* untuk data jumlah pada baris dengan id keempat pada tabel benda untuk transaksi T1 (untuk mengecek *nonrepeatable read*)
INSERT INTO benda(id, barang, jumlah) VALUES(8,'Meja',5);		Melakukan penambahan data pada tabel benda untuk transaksi T1 (untuk mengecek *phantom read*)
SELECT * FROM benda;		Melihat keseluruhan data yang ada pada tabel benda pada transaksi T1 (data yang ditampilkan yaitu data yang sudah berubah akibat proses *inserting* dan *updating*)
COMMIT;		Melakukan *commit* untuk T1
	SELECT * FROM benda;	Melihat keseluruhan data yang ada pada tabel benda pada transaksi T2 (data yang ditampilkan masih

		data awal dan tidak mengalami perubahan padahal sudah commit -> menunjukkan bahwa larangan fenomena *nonrepeatable read* dan *phantom read* tidak terjadi
	UPDATE benda SET jumlah = 2+100 WHERE id = 4;	Error dan gagal melakukan perubahan
	COMMIT;	Melakukan *rollback* untuk T2

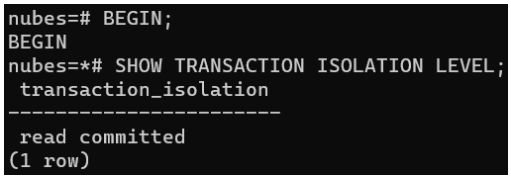
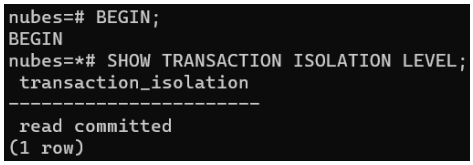
c. Read Committed

Untuk isolasi transaksi dengan tipe read committed, pembacaan data hanya dapat dilakukan setelah proses *commit*. Hal yang membedakan antara isolasi transaksi read committed dengan isolasi repeatable read adalah pada read committed hanya akan melakukan pembatasan untuk proses setelah *commit*, tetapi tidak menjamin bahwa jika transaksi melakukan pembacaan kembali, akan menghasilkan data yang sama (bisa terjadi perubahan. Sedangkan pada tingkat isolasi *repeatable read*, pembacaan berulang akan tetap menghasilkan hasil data yang sama. Pada PostgreSQL, tingkatan isolasi ini yang menjadi isolasi *default* untuk keseluruhan aplikasi dikarenakan tingkat isolasi ini tidak terlalu ketat jika dibandingkan dengan tingkatan lainnya. Dengan tingkat isolasi yang tidak terlalu ketat, konkurensi dan pembacaan yang dilakukan dapat lebih cepat karena tidak perlu proses menunggu dari transaksi lain serta mengurangi terjadinya *locking*. Namun, terdapat resiko yang terjadi pada tingkat isolasi yang tidak terlalu ketat yaitu kesalahan dalam pembacaan data yang disebabkan oleh larangan fenomena yang mungkin terjadi.

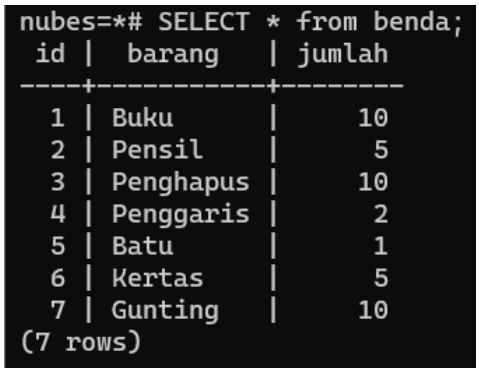
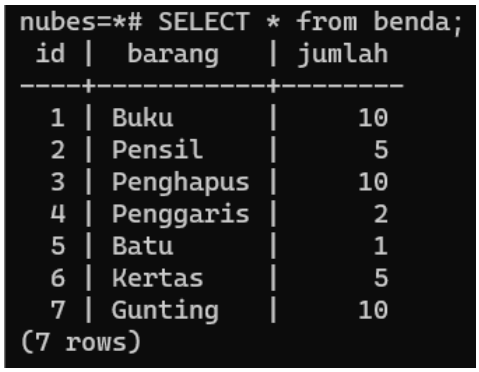
Berdasarkan hal ini, maka untuk derajat isolasi *read committed*, larangan fenomena yang tidak mungkin terjadi yaitu *dirty read* saja dikarenakan data harus dilakukan pembacaan setelah *commit*. Sedangkan, untuk larangan fenomena lainnya seperti *nonrepeatable read*, *phantom read,* dan *serialization anomaly* masih mungkin terjadi dikarenakan pembacaan yang memungkinkan terjadinya perubahan data dan eksekusi ulang *query* bisa menghasilkan baris data yang berbeda.

Adapun proses implementasi dari isolasi transaksi untuk tipe Read Committed adalah sebagai berikut:

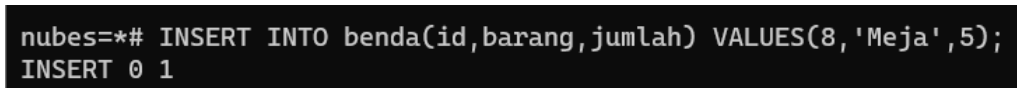
1. Untuk *query read committed*, *query* ini seharusnya sudah menjadi isolasi transaksi *default* dari aplikasi PostgreSQL sehingga tidak perlu dilakukan pengaturan pada kedua terminal transaksi, untuk pembuktian pengecekan bisa langsung dengan menjalankan SHOW TRANSACTION ISOLATION LEVEL setelah *query* BEGIN

T1	T2
	

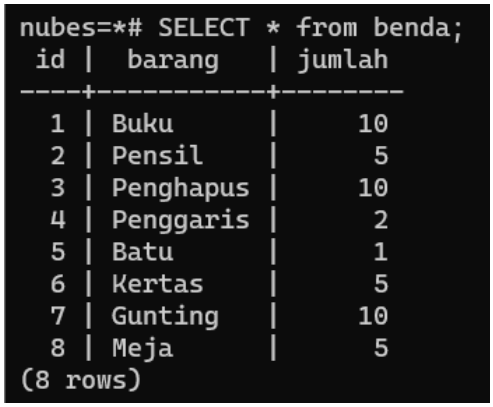
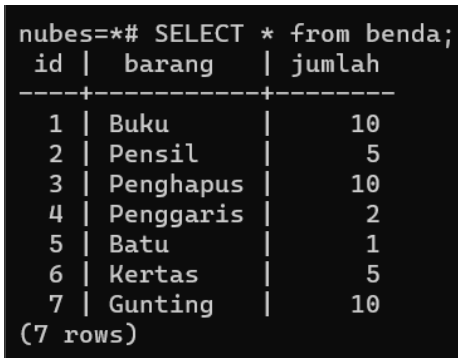
2. Setelah pengecekan level isolasi transaksi, akan dilakukan pengecekan untuk data awal yang ada. Pengecekan dilakukan dengan menjalankan *query* SELECT * from benda dari kedua terminal transaksi untuk mengecek keseluruhan isi tabel yang ada di awal.

T1	T2
	

3. Berikutnya, pada transaksi T1 akan dilakukan pengubahan berupa penambahan data (*inserting*). (Note: untuk perubahan data yang dapat dilakukan bisa juga dengan pengubahan data (*updating*) ataupun penghapusan data (*deleting*))

T1


4. Setelah penambahan data pada transaksi T1, dilakukan kembali pembacaan data yang ada pada kedua terminal transaksi melalui *query* SELECT * from benda..

T1	T2
	

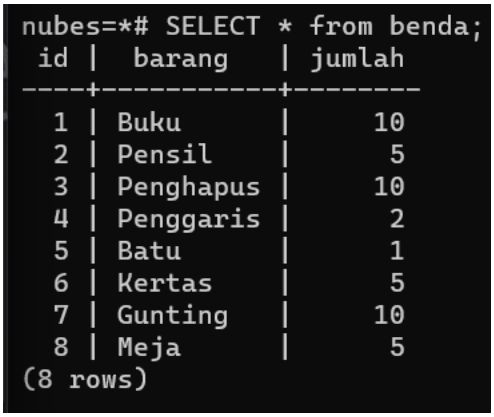
Dari hasil pembacaan yang ada, didapatkan bahwa data yang ada pada terminal T1 sudah mengalami perubahan yang terjadi akibat proses *inserting*, sedangkan data yang ada pada terminal T2 tidak mengalami perubahan sama sekali. Hal ini menunjukkan bahwa larangan fenomena *dirty read* tidak terjadi pada isolasi transaksi *read committed* karena terminal T2 tidak dapat membaca perubahan data pada terminal T1 sebelum proses *commit* dilakukan.

5. Setelah pembacaan, kemudian pada terminal T1 akan dilakukan *commit*.

T1


6. Setelah *commit* pada T1 telah dijalankan, terminal T2 sudah bisa melakukan pembacaan data yang sudah diubah pada terminal T1 sebelumnya. Hal ini memenuhi kondisi *dirty read* yaitu pembacaan perubahan data pada terminal lain baru bisa dilakukan setelah proses *commit*. Tetapi tidak memenuhi kondisi

nonrepeatable read dan *phantom read* karena terjadi perubahan data pada pembacaan dan eksekusi ulang data.

T2


```
nubes=*# SELECT * from benda;
 id |  barang   | jumlah
----+-----------+--------
  1 | Buku      |     10
  2 | Pensil    |      5
  3 | Penghapus |     10
  4 | Penggaris |      2
  5 | Batu      |      1
  6 | Kertas    |      5
  7 | Gunting   |     10
  8 | Meja      |      5
(8 rows)
```

7. Jika pada T2 sudah tidak ingin dilakukan perubahan pada data yang ada, bisa melakukan *commit* pada T2.

T2


```
nubes=*# COMMIT;
COMMIT
nubes=#
```

Berdasarkan proses yang telah dilakukan, urutan proses transaksi yang terjadi untuk pembuktian isolasi read committed yaitu sebagai berikut.

T1	**T2**	**Keterangan**
BEGIN;		Memulai transaksi T1
SHOW TRANSACTION ISOLATION LEVEL;		Melakukan pengecekan level isolasi transaksi T1
	BEGIN;	Memulai transaksi T2

	SHOW TRANSACTION ISOLATION LEVEL;	Melakukan pengecekan level isolasi transaksi T2
SELECT * FROM benda;		Melihat keseluruhan data yang ada pada tabel benda pada transaksi T1 (data yang ditampilkan yaitu data awal)
	SELECT * FROM benda;	Melihat keseluruhan data yang ada pada tabel benda pada transaksi T2 (data yang ditampilkan yaitu data awal)
INSERT INTO benda(id, barang, jumlah) VALUES(8,'Meja',5);		Melakukan penambahan data pada tabel benda untuk transaksi T1
SELECT * FROM benda;		Melihat keseluruhan data yang ada pada tabel benda pada transaksi T1 (data yang ditampilkan yaitu data yang sudah ditambah melalui *inserting*)
	SELECT * FROM benda;	Melihat keseluruhan data yang ada pada tabel benda pada transaksi T2 (data yang ditampilkan masih data awal) -> tidak terjadi *dirty read* (pembacaan perubahan data sebelum *commit*)
COMMIT;		Melakukan *commit* untuk T1
	SELECT * FROM benda;	Melihat keseluruhan data yang ada pada tabel benda pada transaksi T2 (data yang ditampilkan sudah menjadi data baru yang sudah ditambah melalui *inserting*) -> pembacaan data bisa dilakukan setelah *commit*
	COMMIT;	Melakukan *commit* untuk

		T2

d. Read Uncommitted

Isolasi read uncommitted merupakan isolasi yang paling tidak ketat jika dibandingkan dengan berbagai tingkat isolasi lainnya. Hal ini dikarenakan pada tingkat isolasi ini, hampir semua larangan fenomena diizinkan mulai dari *dirty read, nonrepeatable read, phantom read,* hingga *serialization anomaly*. Meskipun begitu, khusus untuk *dirty read*, pada DBMS PostgreSQL, larangan fenomena ini tidak mungkin terjadi (penanganan seperti *read commited*). Sedangkan pada DBMS lain, *dirty read* dalam *read uncommitted* dapat mungkin terjadi.

.

2. Implementasi Concurrency Control Protocol

Program diimplementasikan dalam bentuk *website* dan algoritma perhitungan menggunakan bahasa Python. Tautan *website*: https://concurrency-control.vercel.app/

Untuk *source code* dari algoritma implementasi Concurrency Control Protocol yang diterapkan dalam pengerjaan tugas kami terdapat pada link ini

a. Two-Phase Locking (2PL)

Two-Phase Locking adalah salah satu teknik *Concurrency Control Protocol* yang berfungsi untuk mengatasi permasalahan *concurrency* dalam *transaction*. Pada *Two-Phase Locking*, *Concurrency Control* ini memastikan bahwa *schedules* bersifat *conflict-serializable*. Suatu transaksi dapat dikatakan mengikuti protokol *Two-Phase Locking* apabila *Locking* dan *Unlocking* dapat dilakukan dalam dua tahap yaitu melalui fase pertama yaitu *Growing Phase* ketika transaksi akan mendapatkan *lock* tetapi tidak dapat melepaskan *lock* dan fase kedua yaitu *Shrinking Phase* ketika transaksi akan melepaskan lock dan tidak mungkin untuk mendapatkan lock. Tahapan pengujian dari protokol *Two-Phase Locking* yaitu sebagai berikut.

1. *Test case* 1

Dalam skenario uji *Two-Phase Locking* (2PL) ini, tiga transaksi, yaitu T1, T2, dan T3, terlibat dalam operasi *Shared Lock* (SL), *Read* (R), *Update Lock* (UPL), *Write* (W), dan *Unlock* (UL) terhadap data X.Transaksi T1 dimulai dengan memperoleh SL pada X, kemudian *read* nilai X. Transaksi T2 melakukan *shared lock* pada X dan juga membaca nilai X. Selanjutnya, T1 memasukkan *write* pada antrian (*Queue*) untuk X, tetapi T2 membatalkan (*Abort*) transaksinya. T1 kemudian melakukan *update lock* pada X, *write* nilai X, dan melakukan *upgrade lock* menjadi UL. Transaksi T3 membatalkan (*Abort*), dan T1 melakukan *upgrade lock* dan melakukan *commit*. Pada transaksi T2, setelah membatalkan transaksi, T2 kembali memperoleh *shared lock* pada X, *read* nilai X, melakukan *upgrade lock*, *write* nilai X, dan melakukan *upgrade lock* menjadi UL sebelum melakukan *commit*. Transaksi T3 hanya melakukan *shared lock* pada X, *read* nilai X, dan kemudian melakukan *commit*. Hasil akhir dari urutan transaksi ini adalah *commit* dari T1 dan T2, sedangkan T3 juga melakukan *commit*.

Concurrency Control

R1(X);R2(X);W1(X);W2(X);R3(X);C1;C2;C3 Submit

Choose Algorithms:

(•) 2PL ( ) OCC ( ) MVCC

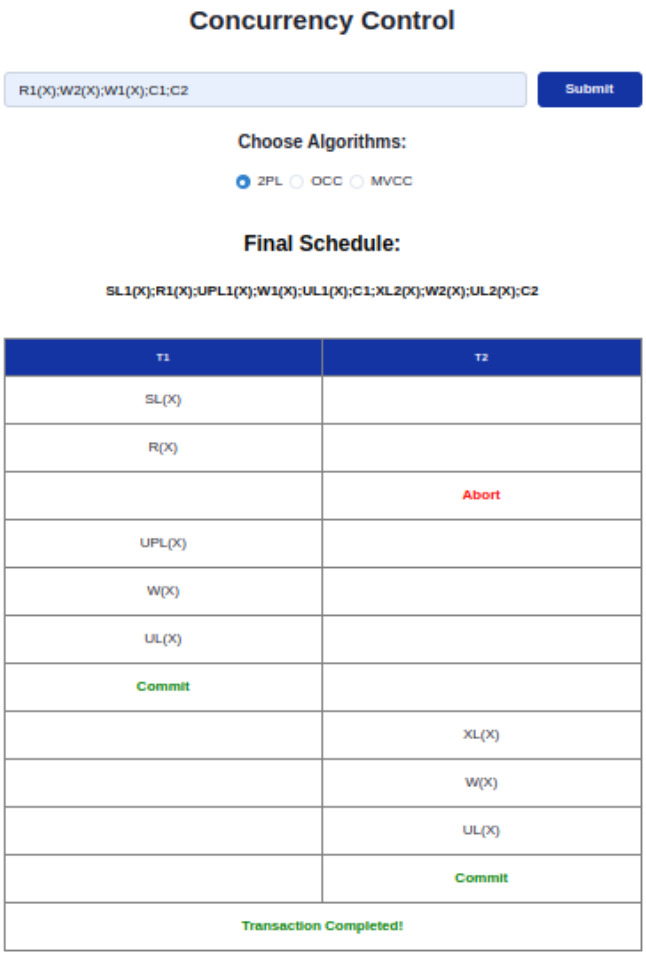
Final Schedule:

SL1(X);R1(X);UPL1(X);W1(X);UL1(X);C1;SL2(X);R2(X);UPL2(X);W2(X);UL2(X);C2;SL3(X);R3(X);C3

T1	T2	T3
SL(X)		
R(X)		
	SL(X)	
	R(X)	
Queue: W(X)		
	Abort	
UPL(X)		
W(X)		
		Abort
UL(X)		
Commit		
	SL(X)	
	R(X)	
	UPL(X)	
	W(X)	
	UL(X)	
	Commit	
		SL(X)
		R(X)
		Commit
Transaction Completed!		

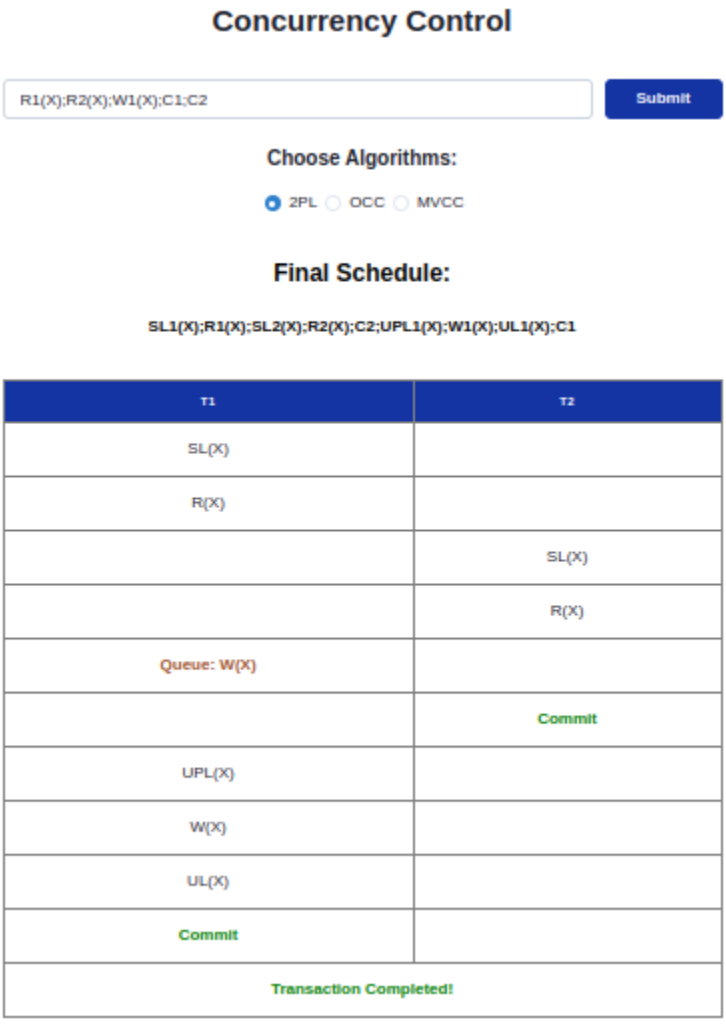
2. *Test case* 2

Dalam skenario uji *Two-Phase Locking* (2PL) ini, transaksi pertama (T1) dimulai dengan memperoleh *Shared Lock* (SL) terhadap data X, kemudian melakukan *read* pada nilai X. Namun, transaksi kedua (T2) dihentikan (*Abort*) sebelum melakukan operasi apa pun. Setelah itu, transaksi pertama (T1) melanjutkan dengan melakukan *Update Lock* (UPL) untuk X, menulis nilai baru X (W), dan melakukan *Unlock* (UL) setelah menyelesaikan transaksi (*Commit*). Pada sisi lain, transaksi kedua (T2) dimulai dengan mencoba memperoleh *Exclusive Lock* (XL) untuk X, namun dihentikan (*Abort*) sebelum menyelesaikan operasi apa pun. Setelah itu, transaksi kedua (T2) kembali dimulai dengan memperoleh XL untuk X, menulis nilai baru X (W), dan melakukan *unlock* setelah menyelesaikan transaksi (*Commit*). Dengan demikian, hasil kasus uji 2PL ini menunjukkan interaksi transaksi yang melibatkan pemilihan kunci berbagi dan eksklusif sesuai dengan protokol 2PL.



3. *Test case* 3

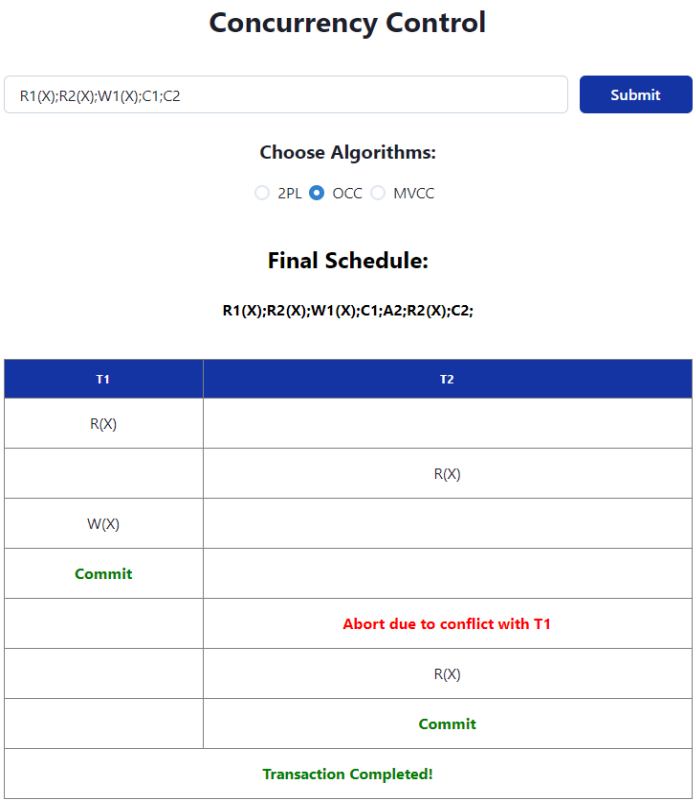
Dalam skenario ini, terdapat dua transaksi, T1 dan T2. Transaksi pertama (T1) dimulai dengan memperoleh *Shared Lock* (SL) untuk X dan melakukan *read*. Transaksi kedua (T2) kemudian memperoleh SL untuk X dan juga melakukan *read*. Transaksi pertama (T1) kemudian menempatkan *write* pada X ke dalam antrian (*Queue*) tanpa segera melaksanakannya. Pada saat yang bersamaan, T2 menyelesaikan transaksi dengan melakukan operasi *Commit*. Setelah itu, T1 melanjutkan dengan melakukan *Update Lock*(UPL) untuk X, menulis nilai baru X, dan melakukan *Unlock* (UL) setelah menyelesaikan transaksi (*Commit*). Hasil ini menunjukkan bahwa meskipun T1 dan T2 bersaing untuk mengakses X, transaksi T2 berhasil menyelesaikan operasinya sebelum T1 menyelesaikan operasi *write*, sehingga tidak terjadi *deadlock* atau konflik yang mengakibatkan pengulangan atau kegagalan transaksi.



b. Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control atau OCC merupakan salah satu teknik dalam *Concurrency Control Protocol* yang berfungsi untuk mengatasi permasalahan *concurrency* dalam *transaction*. Pada protokol ini, protokol mengasumsikan bahwa transaksi-transaksi dapat dijalankan secara konkuren dan independen tanpa adanya gangguan antara satu transaksi dengan transaksi lainnya. Mekanisme dari OCC adalah akan memberikan izin untuk melakukan perubahan tanpa pemeriksaan selama transaksi berlangsung. Setelah akhir transaksi tercapai melalui *commit*, akan dilakukan pengecekan validasi daan jika validasi aman, transaksi akan dieksekusi dan *database* akan mengalami pembaharuan. Sedangkan, jika terjadi konflik, transaksi dibatalkan dan melakukan proses *rollback*. Tahapan pengujian dari protokol *Optimistic Concurrency Control* yaitu sebagai berikut.

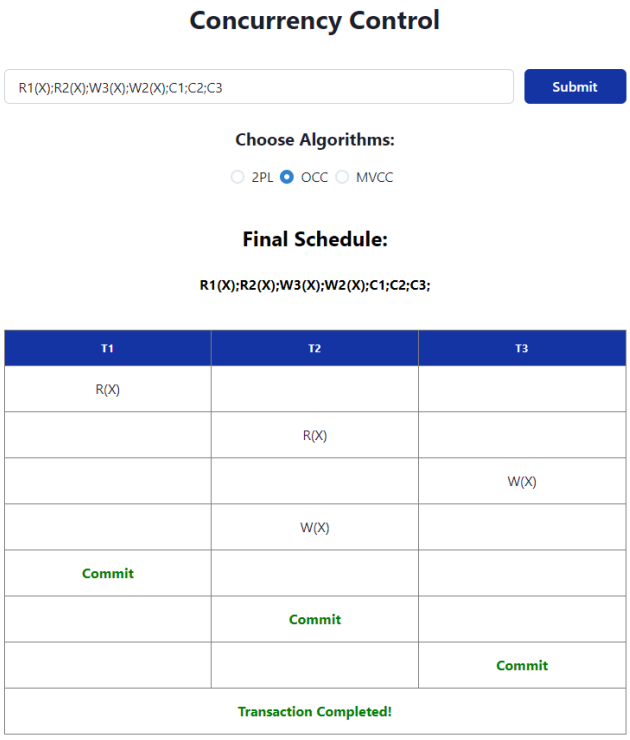
1. *Test case* 1



Dalam skenario ini terdapat 2 transaksi, yaitu T1 dan T2. Transaksi ini tidak dapat berjalan secara konkuren, karena terjadinya *Abort* pada T2. Hal ini disebabkan oleh operasi *Read* pada T2 awalnya sukses karena T1 belum ada operasi *Write*. Tetapi saat tahap validasi, terdapat operasi *Write* pada T1 yang terjadi setelah operasi Read pada T2, maka dari itu hasil Read pada T2 tidak sesuai lagi karena T1 telah di-*commit*, maka dari itu transaksi harus di-*abort* dan melakukan pembacaan ulang.

2. *Test case* 2

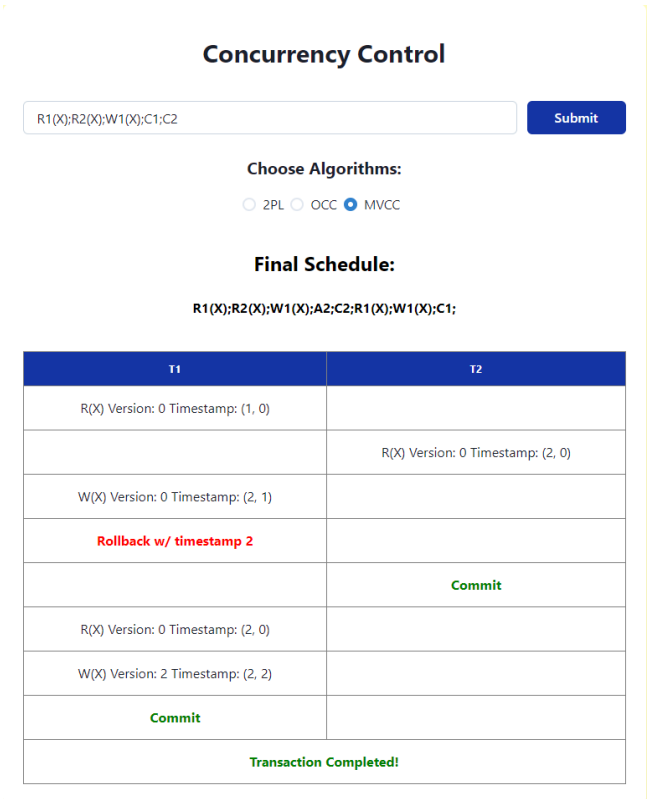
Dalam skenario ini terdapat 3 transaksi, T1, T2, dan T3. Transaksi ini berhasil berjalan secara konkuren, yaitu suatu hal yang tidak dapat dilakukan oleh *Two-Phase-Locking*. Transaksi ini mungkin terjadi karena tidak terjadinya *dirty reads*, atau semua hasil read tidak terganggu oleh operasi *write* yang berasal dari transaksi lainnya. Sehingga, transaksi ini dapat berjalan secara sukses.



c. Multiversion Timestamp Ordering Concurrency Control (MVCC)

Multiversion Timestamp Ordering Concurrency Control (MVCC) merupakan salah satu teknik dalam *Concurrency Control Protocol* yang berfungsi untuk mengatasi permasalahan concurrency dalam transaction. Alur yang ada pada protokol ini yaitu dengan membuat salinan duplikat dari record sehingga pembacaan dan update data dapat dilakukan secara aman (tidak saling memblokir) pada saat yang bersamaan. Dalam protokol ini, tiap transaksi diberikan timestamp sebagai penanda waktu mulai transaksi. MVCC memungkinkan pembacaan dan penulisan transaksi pada versi data yang sesuai dengan transaksi dan memastikan proses isolasi. Tiap versi data yang ada akan diberikan *read* dan *write timestamp* untuk penentuan versi data yang bisa dibaca oleh transaksi. MVCC memungkinkan tingkat isolasi yang tinggi dan efisien dalam mengelola konkurensi pada sistem database dengan menghindari pembacaan tertunda dan konflik penulisan. Tahapan pengujian dari protokol *Multiversion Timestamp Ordering Concurrency Control* (MVCC) yaitu sebagai berikut.

1. *Test case* 1

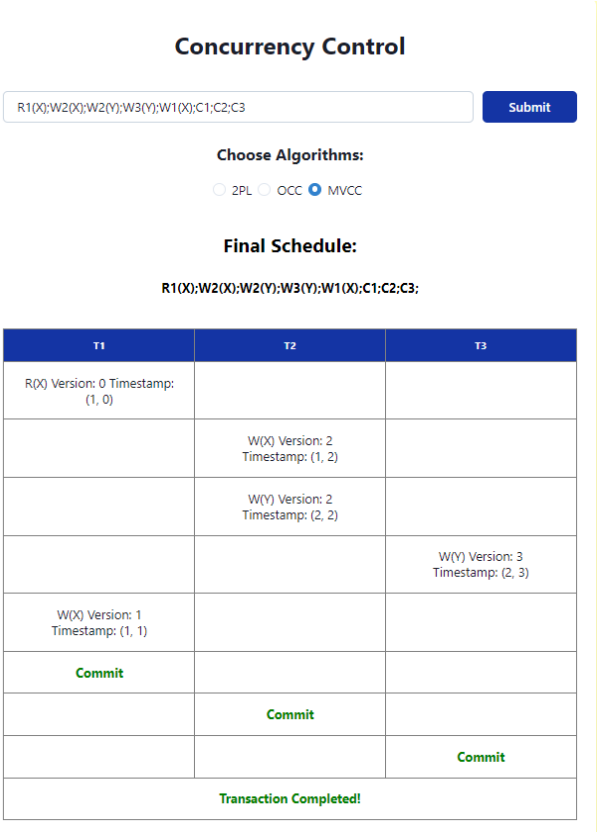


T1	T2
R(X) Version: 0 Timestamp: (1, 0)	
	R(X) Version: 0 Timestamp: (2, 0)
W(X) Version: 0 Timestamp: (2, 1)	
Rollback w/ timestamp 2	
	Commit
R(X) Version: 0 Timestamp: (2, 0)	
W(X) Version: 2 Timestamp: (2, 2)	
Commit	
Transaction Completed!	

Dalam skenario ini terdapat dua transaksi, T1 dan T2. Transaksi ini gagal berjalan secara konkuren, sehingga harus di-*abort* untuk dapat jalan secara serial. Hal ini dikarenakan saat T1 melakukan Write pada tabel X yang memiliki *Timestamp* (TST) = 0, operasi tersebut memiliki *Read Timestamp* (RST) = 2, dan *Write Timestamp* (WST) = 1. Terdapat kondisi dimana TST < WST, sehingga transaksi tersebut harus dilakukan rollback.

2. *Test case* 2

Dalam skenario uji ini, terdapat 3 buah transaksi yang berhasil berjalan secara konkuren. Hal ini disebabkan oleh kondisi-kondisi yang tidak menyebabkan *rollback* (TST < WST). Semua operasi berhasil melakukan update terhadap *Timestamp* dan tetap mempertahankan versionnya masing-masing.



3. Eksplorasi Recovery

a. *Write-Ahead Log*

Write-Ahead Log (WAL) adalah sebuah *rule* dimana setiap aktivitas semua perubahan yang dilakukan terhadap *record* dalam basis data harus tercatat dalam *transaction log*. Proses Write-Ahead Log melibatkan pencatatan setiap perubahan terhadap basis data pada *transaction log* yang disimpan pada *log file*, kemudian *log file* akan disimpan pada *stable storage*. Dengan adanya *transaction log*, semua transaksi bisa dilakukan *recovery* ketika terjadi kegagalan pada DBMS maupun *hardware*. Pada PostgreSQL, *Write-Ahead Log* dijadikan sebuah mekanisme *recovery* yang disimpan pada *directory* pg_wal.

b. *Continuous Archiving*

Continuous Archiving pada PostgreSQL merupakan mekanisme yang dapat diterapkan pada *Write-Ahead Log* (WAL). Dengan menerapkan *Continuous Archiving*, *file* WAL yang merekam perubahan pada data disalin ke penyimpanan yang lebih stabil. Hal ini memungkinkan untuk melakukan *Point-in-Time Recovery*, di mana kita dapat mengembalikan basis data ke kondisi tertentu pada waktu yang spesifik.

Untuk mengaktifkan *Continuous Archiving*, perlu dilakukan konfigurasi, seperti mengubah nilai wal_level menjadi 'archive' dan menentukan perintah shell yang akan digunakan dalam archive_command. Perintah ini akan dijalankan oleh server, untuk memindahkan file WAL yang telah selesai ke lokasi yang ditentukan oleh administrator. Pembuatan archive_command berfungsi untuk memeriksa keberadaan *file* arsip yang sudah ada di lokasi tersebut, sehingga menghindari *overwriting*.

c. *Point-in-Time Recovery*

Point-in-Time Recovery adalah proses *recovery* pada basis data yang memungkinkan pengguna untuk mengembalikan sistem ke suatu titik waktu tertentu. Dengan adanya Continuous Archiving yang memanfaatkan Write-Ahead Log (WAL), Point-in-Time Recovery menjadi mungkin dilakukan. Pengguna dapat menentukan

waktu tertentu dalam riwayat transaksi database dan memulihkan data ke kondisi tersebut.

d. Simulasi Kegagalan pada PostgreSQL

Implementasi proses *recovery* pada PostgreSQL diperlukan konfigurasi sebagai berikut.

a. Konfigurasi Continuous Archiving

1. Buatlah *directory* untuk menyimpan log arsip basis data.
2. Beri akses postgres untuk *write file*.

```
$ sudo chmod 700 /path/to/database_archive/
```

3. Tambahkan *user* postgres ke dalam *path archive*.

```
$ sudo chown postgres:postgres /path/to/database_archive/
```

4. Edit *file* config sebagai berikut.

```
archive_mode=on
archive_command = 'test ! -f /path/to/database_archive/%f && cp
%p /path/to/database_archive/%f'
wal_level = replica
```

5. *Restart* PostgreSQL.
6. Lakukan *test archiving* menggunakan *command* berikut. Jika berhasil melakukan konfigurasi, akan terdapat *log file* pada *directory* database_archive.

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

b. Konfigurasi Physical Backup dari Kluster Basis Data

1. Buatlah *directory backup* basis data.
2. Beri akses postgres untuk write file.

```
$ sudo chmod 700 /path/to/database_backup/
```

3. Tambahkan *user* postgres ke dalam *path backup*.

```
$ sudo chown postgres:postgres /path/to/database_backup/
```

4. Lakukan physical backup dengan command berikut.

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

Proses *backup* berhasil jika terdapat *file* dalam *directory* database_backup.

c. *Point-In Time Recovery* pada klaster basis data

1. Copy direktori pg_wal ke tempat lain.
2. Hapus direktori PostgreSQL.
3. Buat direktori baru untuk PostgreSQL
4. Copy file *physical backup* ke direktori baru untuk PostgreSQL yang baru dibuat.
5. Beri akses postgres untuk *write file* pada direktori baru.

```
$ sudo chmod 700 /path/to/
$ sudo chown -R postgres:postgres
/path/to/database_backup/
```

6. File WAL pada direktori pg_wal yang pada *physical backup* sudah lama, sehingga harus dihapus.
7. Copy direktori pg_wal yang sudah dipindahkan pada langkah 1 ke dalam direktori PostgreSQL yang baru.
8. Konfigurasi file config sebagai berikut.

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

9. Isi recovery target pada config. Terdapat berbagai jenis target, pilih salah satu. (recovery_target, recovery_target_lsn, recovery_target_name, recovery_target_time, or recovery_target_xid)
10. Buat file recovery signal.

11. Start PostgreSQL.

Setelah melakukan konfigurasi, kita dapat melakukan simulasi kegagalan. Skenario simulasi kegagalan yang dilakukan adalah apabila terjadi hal yang tak terduga seperti service DBMS yang mati.

1. Pada *database* 'postgres' terdapat tabel barang sebagai berikut. (data sebelum terjadi kegagalan)

```
postgres=# \dt
         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | barang | table | postgres
(1 row)
```

2. *Database administrator* secara rutin melakukan *continuous archiving*.

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

```
 pg_switch_wal
---------------
 0/2000078
(1 row)
```

3. *Database administrator* secara rutin melakukan *physical backup* klaster basis data.

```
:~$ sudo -u postgres pg_basebackup -D ~/pg_database_backup
```

4. Suatu saat, terjadi kegagalan karena *service* DBMS tidak dapat menyala. Data tidak dapat ditampilkan karena *service* DBMS yang mati.
5. Kemudian *database administrator* langsung melakukan *recovery* dengan langkah-langkah sebagai berikut.

1. Mematikan *service database*.

```
:~$ sudo service postgresql stop
```

2. Copy direktori pg_wal PostgreSQL ke direktori lain.

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

3. Hapus direktori PostgreSQL.

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

4. Buat direktori baru untuk PostgreSQL kemudian isi direktori dengan *backup database*.

```
:~$ sudo mkdir /var/lib/postgresql/14/main
:~$ sudo cp -a ~/pg_database_backup/. /var/lib/postgresql/14/main/
```

5. Beri izin akses postgres untuk *write file*.

```
~$ sudo chown postgres:postgres /var/lib/postgresql/14/main
~$ sudo chmod 700 /var/lib/postgresql/14/main
```

6. Hapus directory pg_wal yang sudah lama pada direktori baru PostgreSQL. Kemudian *copy* direktori pg_wal yang baru dipindahkan pada *step* 2 ke dalam direktori baru PostgreSQL.

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

7. Buat file recovery.signal sebagai *trigger* proses *recovery*.

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

8. Isi target *recovery* dengan waktu sebelum tabel barang terhapus pada *file* config.

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

```
recovery_target_time = '2022-12-01 13:55:30.000000'
```

9. Start *service database*.

```
~$ sudo service postgresql start
```

10. Lihat log recovery pada PostgreSQL (opsional).

```
~$  tail -n 100 /var/log/postgresql/postgresql-14-main.log
```

Basis data berhasil *recover*.

```
 LOG:  starting PostgreSQL 14.9 (Ubuntu 14.9-0ubuntu0.22.04.1) on x86_64-pc-linux-g
1ubuntu1~22.04) 11.4.0, 64-bit
 LOG:  listening on IPv4 address "127.0.0.1", port 5432
 LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
 LOG:  database system was interrupted; last known up at 2023-12-01 14:00:37 WIB
tabase_archive/00000002.history': No such file or directory
 LOG:  starting point-in-time recovery to 2022-12-01 13:55:30+07
 LOG:  restored log file "000000010000000000000010" from archive
 LOG:  redo starts at 0/10000028
 LOG:  consistent recovery state reached at 0/10000100
 LOG:  database system is ready to accept read-only connections
 LOG:  restored log file "000000010000000000000011" from archive
 LOG:  recovery stopping before commit of transaction 735, time 2023-12-01 14:02:14

 LOG:  pausing at the end of recovery
 HINT:  Execute pg_wal_replay_resume() to promote.
```

11. Cek isi basis data yang sudah *recover* (data setelah *recovery*).

```
postgres=# \dt
         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | barang | table | postgres
(1 row)
```

4. Pembagian Kerja

NIM	Nama	Bagian
13521008	Jason Rivalino	Eksplorasi Transaction Isolation, Frontend
13521010	Muhamad Salman Hakim Alfarisi	Eksplorasi Recovery
13521024	Ahmad Nadil	Two-Phase Locking, Optimistic Concurrency Control, Multiversion Timestamp Ordering, Frontend, Backend
13521026	Kartini Copa	Eksplorasi Recovery, Optimistic Concurrency Control

Referensi

[1] https://www.postgresql.org/docs/current/transaction-iso.html

[2] https://medium.com/@yosepnoventon/memahami-transaction-isolation-level-dan-read-phenomena-di-mysql-8d794cb1b2aa

[3] https://mkdev.me/posts/transaction-isolation-levels-with-postgresql-as-an-example

[4] https://www.postgresql.org/docs/current/wal-intro.html

[5] https://www.postgresql.org/docs/current/continuous-archiving.html