

Laporan Tugas Besar 2
IF3140 - Manajemen Basis Data

MEKANISME *CONCURRENCY CONTROL* DAN *RECOVERY*



Disusun oleh:
K01 - G06

Vincentius Lienardo	13518081
Nicholas Chen	13519029
Mohammad Dwinta Harits Cahyana	13519041
Nizamixavier Rafif Lutvie	13519085
Maximillian Lukman	13519153

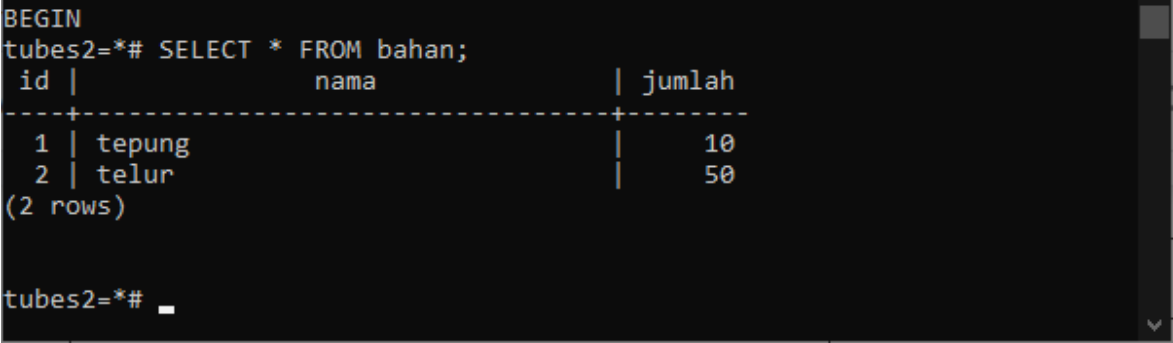
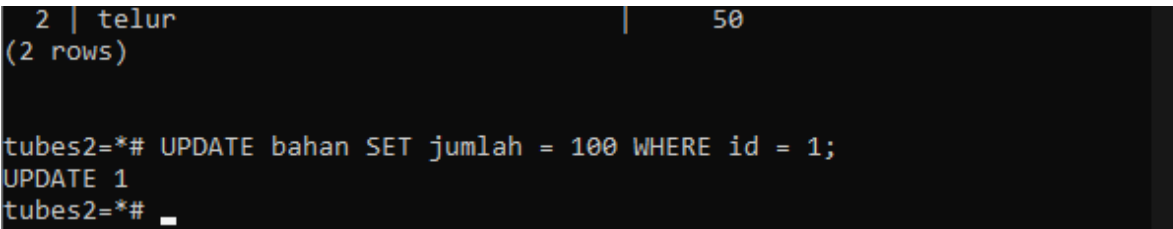
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2021

1. Eksplorasi *Concurrency Control*

A. *Serializability*

Serializability adalah derajat atau tingkat isolasi yang paling tinggi dimana isolasi ini mensimulasikan eksekusi transaksi secara satu per satu atau serial, bukan secara konkuren pada semua transaksi yang dilakukan. Cara kerjanya mirip dengan isolasi *repeatable read* dimana apabila menggunakan isolasi ini, aplikasi harus dapat mencoba kembali transaksi apabila terjadi kegagalan serialisasi (*serialization failure*). Perbedaan paling utama dengan isolasi *repeatable read* adalah bahwa isolasi ini memonitor kondisi yang dapat membuat eksekusi serangkaian transaksi konkuren menjadi tidak konsisten dan kondisi yang terdeteksi yang dapat menyebabkan anomali serialisasi akan memicu *serialization failure*.

Simulasi:

Transaksi	Keterangan
T1	Memulai transaksi T1 dan SELECT tabel 'bahan' sebelum transaksi 
T1	Lakukan perintah UPDATE pada tabel 'bahan' dengan <i>query</i> UPDATE bahan SET jumlah = 100 WHERE id = 1; 
T2	Memulai transaksi T2 dengan tingkat isolasi <i>serializable</i> dan SELECT tabel 'bahan' sebelum transaksi

```
Command Prompt - psql -U postgres
tubes2=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
tubes2=*# SELECT * FROM bahan;
 id |          nama          | jumlah
-----+-----+-----
  1 | tepung                 |    10
  2 | telur                  |    50
(2 rows)

tubes2=*#
```

T2	Lakukan perintah UPDATE pada tabel 'bahan' dengan <i>query</i> lain UPDATE bahan SET jumlah = 200 WHERE id = 1;
----	---

```
 id |          nama          | jumlah
-----+-----+-----
  2 | telur                  |    50
(2 rows)

tubes2=*# UPDATE bahan SET jumlah = 200 WHERE id = 1;
```

T1	Lakukan COMMIT pada transaksi T1
----	----------------------------------

```
Command Prompt - psql -U postgres
(2 rows)

tubes2=*# UPDATE bahan SET jumlah = 100 WHERE id = 1;
UPDATE 1
tubes2=*# COMMIT TRANSACTION;
COMMIT
tubes2=#
```

T2	Terjadi <i>error</i> pada T2 saat T1 di-commit
----	--

```
Select Command Prompt - psql -U postgres

tubes2=*# UPDATE bahan SET jumlah = 200 WHERE id = 1;
ERROR:  could not serialize access due to concurrent update
tubes2=!#
```

T2	T2 akan di- <i>rollback</i> pada saat di- <i>commit</i>
----	---

```

Command Prompt - psql -U postgres
tubes2=# UPDATE bahan SET jumlah = 200 WHERE id = 1;
ERROR: could not serialize access due to concurrent update
tubes2=# COMMIT TRANSACTION;
ROLLBACK
tubes2=#

```

Dari hasil simulasi dapat dilihat bahwa *commit* pada transaksi T2 gagal karena T1 telah di-*commit* sementara terjadi perubahan data pada T2 sehingga T2 akan *error* dan *rollback*.

B. Repeatable Read

Repeatable read adalah tingkat isolasi yang hanya melihat data yang dilakukan sebelum sebuah transaksi dijalankan dan tidak dapat melihat data yang belum di-*commit* maupun perubahan yang dilakukan selama eksekusi transaksi oleh transaksi yang bersamaan atau konkuren, transaksi juga tidak dapat memodifikasi data yang sedang dibaca oleh transaksi lainnya hingga transaksi selesai. Tingkat isolasi ini memberikan *exclusive lock* yang mencegah sebuah transaksi untuk melihat atau mengubah langsung perubahan data yang belum di-*commit*. Locks tersebut akan ditahan sampai transaksi selesai. Tingkat isolasi ini dapat mencegah *dirty read*, *uncommitted non-repeatable read*, dan juga *phantom read*.

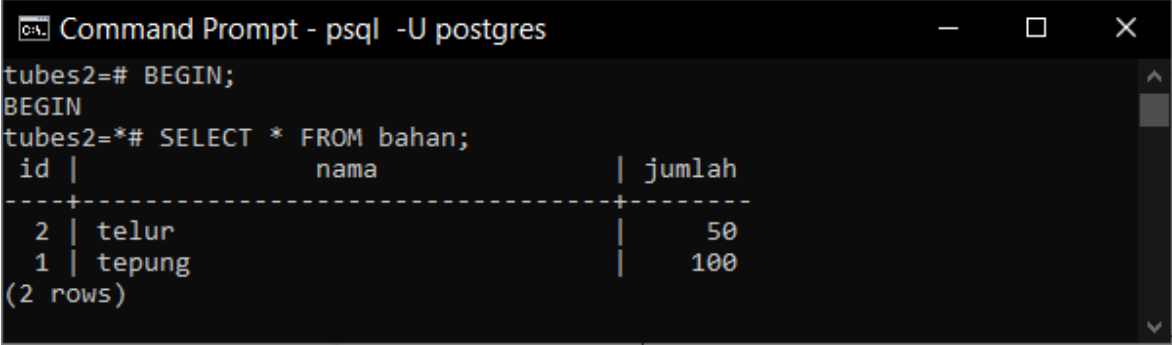

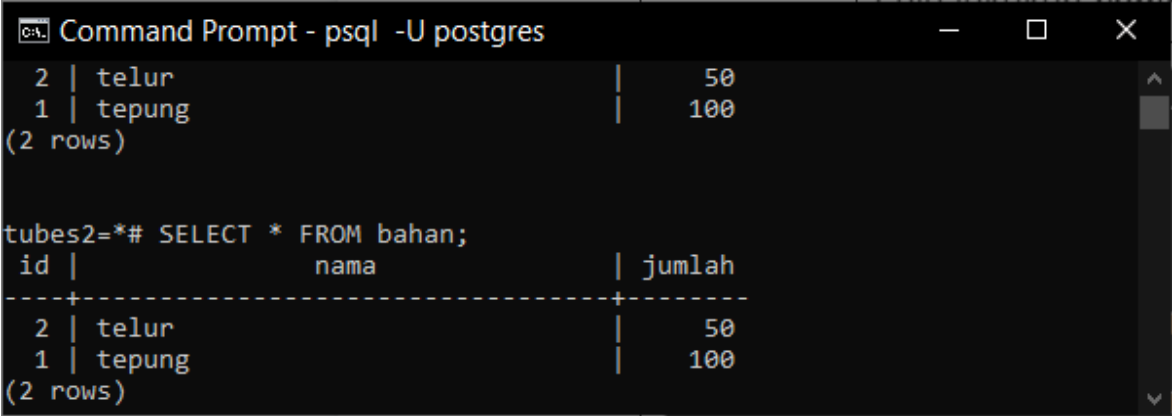
Simulasi:

Transaksi	Keterangan
T1	Memulai transaksi T1 dengan tingkat isolasi <i>repeatable read</i> dan SELECT tabel 'bahan' sebelum transaksi

```

Command Prompt - psql -U postgres
tubes2=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
tubes2=# SELECT * FROM bahan;
 id |          nama          | jumlah
----+-----
  2 | telur                  |     50
  1 | tepung                 |    100
(2 rows)
tubes2=#

```

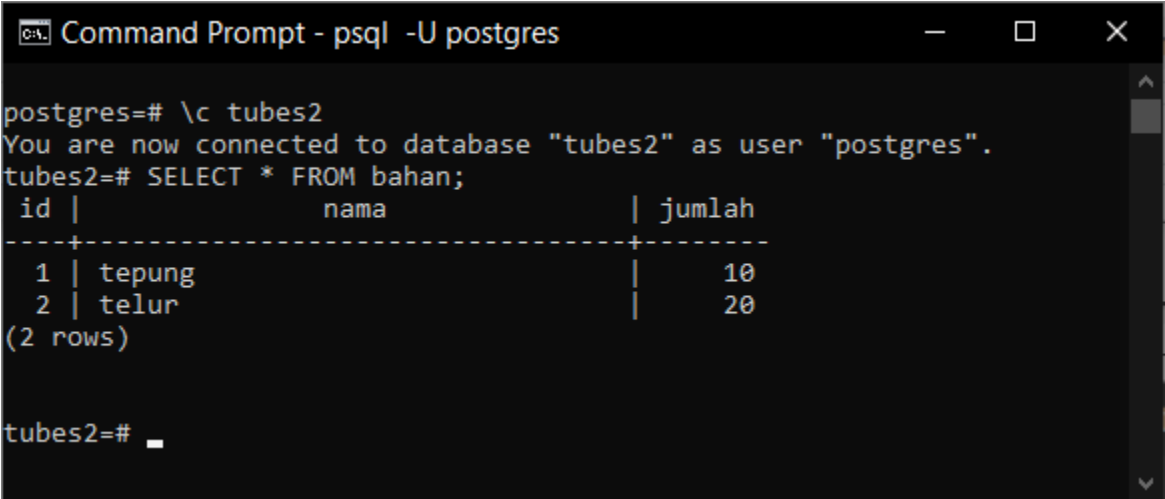
T2	Memulai transaksi T2 dan SELECT tabel 'bahan' sebelum transaksi
 <pre> Command Prompt - psql -U postgres tubes2=# BEGIN; BEGIN tubes2=# SELECT * FROM bahan; id nama jumlah ----+----- 2 telur 50 1 tepung 100 (2 rows) </pre>	
T2	Lakukan perintah UPDATE pada tabel 'bahan' dengan query UPDATE bahan SET jumlah = 200 WHERE nama = 'telur'; Lalu lakukan <i>commit</i> untuk transaksi T2
 <pre> Command Prompt - psql -U postgres 1 tepung 100 (2 rows) tubes2=# UPDATE bahan SET jumlah = 200 WHERE nama = 'telur'; UPDATE 1 tubes2=# COMMIT TRANSACTION; COMMIT tubes2=# </pre>	
T1	Lihat kembali data pada tabel 'bahan' dengan perintah SELECT pada T1
 <pre> Command Prompt - psql -U postgres 2 telur 50 1 tepung 100 (2 rows) tubes2=# SELECT * FROM bahan; id nama jumlah ----+----- 2 telur 50 1 tepung 100 (2 rows) </pre>	

Dari simulasi dapat dilihat bahwa data yang dihasilkan oleh perintah SELECT T1 tidak berubah atau tidak terlihat perubahannya walaupun T2 yang melakukan perubahan terhadap tabel 'bahan' telah di-*commit*, data yang terlihat pada T1 tetap sama saat sebelum melakukan *commit* pada T2.

C. Read Committed

Read committed adalah tingkat isolasi *default* pada PostgreSQL. Apabila sebuah transaksi yang menggunakan tingkat isolasi ini melakukan *query* SELECT, transaksi hanya dapat melihat data yang di-*commit* sebelum *query* dijalankan dan tidak akan dapat melihat data yang tidak di-*commit* atau perubahan yang terjadi selama eksekusi *query* oleh transaksi yang konkuren. Isolasi ini memungkinkan dua *query* SELECT mengembalikan data yang berbeda meskipun berada pada satu transaksi, hal ini terjadi apabila terdapat transaksi lain yang *commit* setelah *query* SELECT pertama dan sebelum *query* SELECT kedua. Pada *query* lain seperti UPDATE, DELETE, SELECT FOR UPDATE, dan SELECT FOR SHARE juga akan melihat data hasil *query* yang telah di-*commit* sebelum *query* dijalankan, bukan sebelum transaksi dijalankan. Jika data yang akan dibaca tersebut sedang diperbarui (*update*, *delete*, atau *locked*) oleh transaksi konkuren lainnya, maka *query* akan menunggu untuk transaksi tersebut hingga *commit* atau *roll-back*.

Tabel 'bahan' sebelum dijalankan transaksi:



```
Command Prompt - psql -U postgres

postgres=# \c tubes2
You are now connected to database "tubes2" as user "postgres".
tubes2=# SELECT * FROM bahan;
 id |          nama          | jumlah
----+-----
  1 | tepung                 |    10
  2 | telur                  |    20
(2 rows)

tubes2=#
```

Simulasi:

Transaksi 1 : UPDATE
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED; UPDATE bahan SET jumlah = 50 WHERE nama = 'telur'; COMMIT TRANSACTION;
Transaksi 2 : SELECT

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SELECT * FROM bahan;
```

Hasil Simulasi:

```
Command Prompt - psql -U postgres  
tubes2=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;  
BEGIN  
tubes2=# SELECT * FROM bahan;  
id | nama | jumlah  
---+---+---  
1 | tepung | 10  
2 | telur | 20  
(2 rows)  
  
tubes2=# UPDATE bahan SET jumlah = 50 WHERE nama = 'telur';  
UPDATE 1  
tubes2=# COMMIT TRANSACTION;  
COMMIT  
tubes2=#
```

```
Command Prompt - psql -U postgres  
tubes2=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;  
BEGIN  
tubes2=# SELECT * FROM bahan;  
id | nama | jumlah  
---+---+---  
1 | tepung | 10  
2 | telur | 20  
(2 rows)  
  
tubes2=# SELECT * FROM bahan;  
id | nama | jumlah  
---+---+---  
1 | tepung | 10  
2 | telur | 50  
(2 rows)  
  
tubes2=#
```

Dari hasil simulasi dapat dilihat bahwa pada 2 transaksi yang konkuren, data yang didapat oleh perintah SELECT pada transaksi 2 berbeda dengan data pada saat transaksi 1 belum di-*commit* karena transaksi 1 yang melakukan UPDATE pada tabel belum di-*commit* dan sesuai dengan tingkat isolasinya yaitu *read committed*, transaksi hanya dapat melihat data yang di-*commit* sebelum *query* dijalankan tetapi data yang telah diubah oleh *commit* sebelumnya dapat dilihat pada transaksi yang lain secara konkuren.

D. *Read Uncommitted*

Read uncommitted adalah tingkat isolasi yang paling rendah atau paling lemah dimana isolasi ini memungkinkan terjadinya *dirty reads*. Tingkat isolasi ini dapat membaca dan mengubah data yang telah dimodifikasi oleh transaksi yang belum di-*commit* karena tidak memakai *shared locks* dalam aksesnya sehingga dapat mempengaruhi transaksi yang sedang berjalan. Selain itu, tingkat isolasi ini juga tidak menggunakan *exclusive locks* untuk mencegah transaksi yang membaca baris yang sedang diubah oleh transaksi lain. Isolasi ini juga tidak mencegah transaksi lain saat mengakses data yang sedang dibaca kecuali transaksi sedang menambah atau menghapus tabel pada *database* sehingga dimungkinkan hasil dari pengaksesan atau pembacaan data tersebut berubah ketika terdapat transaksi-transaksi lain yang memodifikasi data dari tabel target yang ingin dicari.

2. Implementasi *Concurrency Control Protocol*

I. *Simple Locking*

I. *Screenshot* hasil Percobaan

Percobaan dengan kasus normal
R1(X); W2(Y); R1(Y); W1(Y); C2; C1
<pre>Enter transaction 1: R1(X) Enter transaction 2: W2(Y) Enter transaction 3: R1(Y) Enter transaction 4: W1(Y) Enter transaction 5: C2 Enter transaction 6: C1 Here are the process: XL1(X) R1(X) UL1(X) XL2(Y) W2(Y) UL2(Y) XL1(Y) R1(Y) W1(Y) UL1(Y) C2 C1 Database Content: X: 0 Y: 1</pre>

Percobaan dengan kasus *blindwrite*.

R1(X); W2(X); W2(Y); R3(X); W3(Y); W1(Y); C1; C2; C3

```
Enter transaction 1: R1(X)
Enter transaction 2: W2(X)
Enter transaction 3: W2(Y)
Enter transaction 4: R3(X)
Enter transaction 5: W3(Y)
Enter transaction 6: W1(Y)
Enter transaction 7: C1
Enter transaction 8: C2
Enter transaction 9: C3
```

Here are the process:

```
XL1(X)
R1(X)
UL1(X)
XL2(X)
W2(X)
UL2(X)
XL2(Y)
W2(Y)
UL2(Y)
XL3(X)
R3(X)
UL3(X)
XL3(Y)
W3(Y)
UL3(Y)
XL1(Y)
W1(Y)
UL1(Y)
C1
C2
C3
```

Database Content:

```
X: 2
Y: 1
```

Percobaan dengan kasus deadlock antara T1 dan T2

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

```
Enter transaction 1: R1(X)
Enter transaction 2: R2(X)
Enter transaction 3: W2(X)
Enter transaction 4: W1(X)
Enter transaction 5: C2
Enter transaction 6: C1
```

Here are the process:

```
XL1(X)
R1(X)
A2
UL2(X)
XL1(X)
W1(X)
UL1(X)
C1
```

Database Content:

X: 1

Percobaan dengan kasus transaksi dengan dirty read gagal rollback akibat transaksi yang lebih muda telah commit terlebih dahulu

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

```
Enter the number of transaction you want to input: 8
```

Input 8 transaction one by one with following format.

R - Read

W - Write

e.x: R1(X), W3(Z), C1, C3

```
Enter transaction 1: R1(X)
Enter transaction 2: W2(Y)
Enter transaction 3: R3(Y)
Enter transaction 4: C3
Enter transaction 5: R2(X)
Enter transaction 6: W1(X)
Enter transaction 7: C1
Enter transaction 8: C2
```

Here are the process:

```
XL1(X)
R1(X)
XL2(Y)
W2(Y)
UL2(Y)
XL3(Y)
R3(Y)
UL3(Y)
C3
A2
A3
```

Recoverability error! cannot rollback transaction T3 because already committed.

II. Analisis dari hasil algoritma yang diterapkan

Algoritma protokol Simple Locking adalah algoritma sederhana yang dapat memungkinkan beberapa transaksi untuk berjalan secara konkuren di satu waktu yang sama. Namun algoritma ini memiliki banyak kekurangan diantaranya seperti tidak dapat memastikan transaksi berjalan secara serial (not serializable), dapat terjadi deadlock (kasus ini pada program ditangani dengan cara menerapkan protokol wait-die sebagai deadlock prevention), dan dapat terjadi dirty read antar transaksi yang menyebabkan simple locking tidak recoverable.

Pada hasil percobaan diatas ditunjukkan bahwa protokol wait-die berhasil mencegah Algoritma Simple Locking masuk ke dalam deadlock. Namun dapat dilihat pula saat terjadi dirty read dan ternyata transaksi yang melakukan write perlu rollback sedangkan transaksi yang membaca telah commit, pada Algoritma Simple Locking akan terjadi error karena tidak recoverable.

Untuk membuat performa Algoritma Simple Locking ini meningkat, dapat diterapkan Algoritma Two-Phase Locking yang dapat memastikan transaksi bersifat serializable, dan recoverable.

II. ***Serial Optimistic Concurrency Control (OCC)***

I. Screenshot hasil Percobaan

Berikut bentuk *schedule* dari program OCC, *schedule* harus dituliskan secara *hardcode* pada file main.cpp

KASUS 1:

```
Transaction one(1);
Transaction two(2);
Transaction three(3);

Action act1(one, "start", '-');
Action act2(one, "read", 'x');
Action act3(two, "start", '-');
Action act4(two, "validate", '-');
Action act5(two, "write", 'x');
Action act6(two, "write", 'y');
Action act7(three, "start", '-');
Action act8(three, "validate", '-');
Action act9(three, "write", 'y');
Action act10(one, "validate", '-');
Action act11(one, "write", 'y');
Action act12(one, "finish", '-');
Action act13(two, "finish", '-');
Action act14(three, "finish", '-');

vector<Action> actions;
```

Representasi dari *schedule* tersebut adalah sebagai berikut:

T1(one)	T2(two)	T3(three)
<Start>		
R(X)		
	<Start>	
	<Validate>	
	W(X)	
	W(Y)	
		<Start>
		<Validate>
		W(Y)
<Validate>		

Write(Y)		
<Finish>		
	<Finish>	
		<Finish>

Pada *schedule* di atas seluruh proses *write* didorong ke belakang(sesuai kebutuhan OCC) dan sebelum dilakukan *write*, transaksi memasuki proses validasi. Selain itu proses *commit* disamakan dengan *finish* pada tabel.

Berikut keluaran program dengan *schedule* di atas:

```
C:\Users\mdhar\Documents\Kuliah\Semester 5\Manajemen Basis Dat
T1:
Start time: 1
Validate time: 9
Finish time: 12
Read set: x
Write set: y

T2:
Start time: 3
Validate time: 3
Finish time: 13
Read set:
Write set: x y

T3:
Start time: 7
Validate time: 7
Finish time: 14
Read set:
Write set: y

Transaksi T2 berhasil dijalankan
Transaksi T3 berhasil dijalankan
Transaksi T1 gagal divalidasi dengan T2 dan harus dirollback.
Transaksi T1 berhasil divalidasi dengan T3.
----ROLLBACK----
Transaksi T1 berhasil dijalankan
```

KASUS 2:

```
Transaction one(1);
Transaction two(2);

Action act1(one, "start", '-');
Action act2(one, "read", 'b');
Action act3(two, "start", '-');
Action act4(two, "read", 'b');
Action act5(two, "read", 'a');
Action act6(one, "read", 'a');
Action act7(one, "validate", '-');
Action act8(one, "finish", 'b');
Action act9(two, "validate", 'c');
Action act10(two, "write", 'b');
Action act11(two, "write", 'a');
Action act12(two, "finish", '-');

vector<Action> actions;
```

Representasi dari *schedule* tersebut adalah sebagai berikut:

T1(one)	T2(two)
<Start>	
R(B)	
	<Start>
	R(B)
	R(A)
R(A)	
<Validate>	
<Finish>	
	<Validate>
	W(B)
	W(A)
	<Finish>

Berikut keluaran program dengan *schedule* di atas:

```
T1:
Start time: 1
Validate time: 6
Finish time: 8
Read set: b a
Write set:

T2:
Start time: 3
Validate time: 8
Finish time: 12
Read set: b a
Write set: b a

Transaksi T1 berhasil dijalankan
Transaksi T2 berhasil divalidasi dengan T1.
Transaksi T2 berhasil dijalankan
```

II. Analisis dari hasil algoritma yang diterapkan

Pada kasus 1, transaksi yang pertama kali divalidasi adalah T2, lalu T3, dan T1. Hal ini sesuai dengan urutan validasi pada *schedule*. Karena T2 merupakan transaksi yang pertama berjalan maka T2 tidak perlu divalidasi sehingga dapat langsung berhasil dieksekusi.

Selanjutnya T3 harus melakukan pengecekan terhadap T2 terlebih dahulu, dapat diamati dari tangkapan layar bahwa waktu validasi T3 berada di antara waktu mulai dan berakhirnya T2. Oleh karena itu T3 divalidasi dengan T2. Pada tangkapan layar terlihat bahwa himpunan *read* dari T3 tidak beririsan dengan himpunan *write* T2, maka T3 berhasil dieksekusi

Terakhir T1 melakukan pengecekan terhadap T2, lalu T3. Terlihat bahwa waktu validasi T1 berada diantara T2, maka dilakukan validasi yang sama dengan T3. Lain halnya dengan T3, T1 gagal validasi dengan T2 disebabkan adanya himpunan irisan *read* T1 dan *write* T2(yakni x). Sementara T1 berhasil divalidasi dengan T3 (karena T3 hanya menuliskan y).

Karena T1 gagal dieksekusi pada percobaan sebelumnya, maka T1 dirollback dan diberi *timestamp* baru setelah semua transaksi selesai. Setelah dilakukan rollback T1 berhasil dieksekusi.

Pada kasus 2, urutan validasi adalah T1, lalu T2. T1 sebagai transaksi yang pertama berjalan tidak perlu melewati validasi dan berhasil dieksekusi. Sementara itu T2 melakukan validasi pada masa hidup T1, maka T2 harus divalidasi terhadap T1. Dapat diamati pada tangkapan layar bahwa himpunan *read* T2 dan *write* T1 tidak beririsan. Karena hal tersebut T2 berhasil dieksekusi tanpa mengalami *rollback* terlebih dahulu.

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

I. *Screenshot* hasil Percobaan

Berikut adalah screenshot hasil percobaan yang telah dilakukan.

Kasus 1:

```
PS C:\git\06_K01_Tubes2MBD\MultiversionConcurrencyControl> py mvcc.py
Masukkan jumlah data item: 4
Masukkan nama data item 1: A
Masukkan nama data item 2: B
Masukkan nama data item 3: C
Masukkan nama data item 4: D
Masukkan jumlah transaksi: 3
Asumsikan timestamp transaksi sesuai dengan urutan
masukan nama transaksi mulai dari input pertama sebagai timestamp terkecil
Masukkan nama transaksi 1: 1
Masukkan nama transaksi 2: 2
Masukkan nama transaksi 3: 3
Masukkan isi jadwal dalam satu baris (dipisahkan menggunakan semicolon), contoh: R1(X);R2(Y);W1(X)
R1(A);R2(A);R3(B);R1(B);W3(C);W2(C);R1(C);C1;R2(D);
W3(B);C3;W2(D);C2
R1(A)
Read isi item A versi 0
R2(A)
Read isi item A versi 0
R3(B)
Read isi item B versi 0
R1(B)
Read isi item B versi 0
W3(C)
Dibuat item C versi 3
W2(C)
Dibuat item C versi 2
R1(C)
Read isi item C versi 0
C1()
Commit Transaksi 1
R2(D)
Read isi item D versi 0
W3(B)
Dibuat item B versi 3
C3()
Commit Transaksi 3
W2(D)
Dibuat item D versi 2
C2()
Commit Transaksi 2
Schedule berhasil dijalankan sehingga serializable
```

Kasus 2:

```
PS C:\git\06_K01_Tubes2MBD\MultiversionConcurrencyControl> py mvcc.py
Masukkan jumlah data item: 2
Masukkan nama data item 1: X
Masukkan nama data item 2: Y
Masukkan jumlah transaksi: 3
Asumsikan timestamp transaksi sesuai dengan urutan masukan nama transaksi mulai dari input pertama sebagai timestamp terkecil
Masukkan nama transaksi 1: 1
Masukkan nama transaksi 2: 2
Masukkan nama transaksi 3: 3
Masukkan isi jadwal dalam satu baris (dipisahkan menggunakan semicolon), contoh: R1(X);R2(Y);W1(X)
R1(X);W2(X);W2(Y);W3(Y);W1(Y);R1(X);W1(Y);R1(X);W1(Y);C1;C2;C3
R1(X)
Read isi item X versi 0
W2(X)
Dibuat item X versi 2
W2(Y)
Dibuat item Y versi 2
W3(Y)
Dibuat item Y versi 3
W1(Y)
Dibuat item Y versi 1
R1(X)
Read isi item X versi 0
W1(Y)
Overwrite isi item Y versi 1
R1(X)
Read isi item X versi 0
W1(Y)
Overwrite isi item Y versi 1
C1()
Commit Transaksi 1
C2()
Commit Transaksi 2
C3()
Commit Transaksi 3
Schedule berhasil dijalankan sehingga serializable
```

Kasus 3:

```
PS C:\git\06_K01_Tubes2MBD\MultiversionConcurrencyCon
trol> py mvcc.py
Masukkan jumlah data item: 1
Masukkan nama data item 1: q
Masukkan jumlah transaksi: 3
Asumsikan timestamp transaksi sesuai dengan urutan ma
sukan nama transaksi mulai dari input pertama sebagai
timestamp terkecil
Masukkan nama transaksi 1: 1
Masukkan nama transaksi 2: 2
Masukkan nama transaksi 3: 3
Masukkan isi jadwal dalam satu baris (dipisahkan meng
gunakan semicolon), contoh: R1(X);R2(Y);W1(X)
R1(q);W1(q);R2(q);W2(q);)R1(q);W1(q);W2(q);R3(q)W3(q)
;C1;C2;C3
R1(q)
Read isi item q versi 0
W1(q)
Dibuat item q versi 1
R2(q)
Read isi item q versi 1
W2(q)
Dibuat item q versi 2
R1(q)
Read isi item q versi 1
W1(q)
Rollback transaksi 1
Terdapat transaksi yang di-abort
```

II. Analisis dari hasil algoritma yang diterapkan

Kunci dari protokol MVCC adalah:

- Aksi *write* yang berhasil menghasilkan sebuah versi baru dari sebuah *data item*
- Digunakan *timestamp* sebagai label sebuah versi
- Saat dilakukan aksi *read(Q)*, digunakan versi dari Q yang sesuai berdasarkan *timestamp* dari transaksi yang memanggil aksi *read* tersebut, dan mengembalikan nilai dari versi tersebut

Dari hasil analisis percobaan, dapat disimpulkan bahwa aksi *read* selalu berhasil. Selain itu, sebuah aksi *write* oleh T_i gagal jika terdapat transaksi lain T_j yang sesuai urutan serialisasi urutan dari nilai *timestamp* dan harusnya membaca hasil dari *write* T_i , ternyata sudah membaca versi yang dibuat oleh transaksi yang lebih lama daripada T_i .

Dapat disimpulkan juga bahwa protokol tersebut menjamin *serializability* dari sebuah schedule.

3. Eksplorasi Recovery

Dasar Teori WAL, Continuous Archiving, dan PITR

- Write-Ahead Log

Write-Ahead Logging (WAL) merupakan teknik yang menyatakan bahwa setiap perubahan pada data, baik itu perubahan pada tabel maupun *index*, hanya dapat dilakukan jika log yang berisi perubahan yang akan dilakukan telah di-*flush* ke *permanent storage*. Dengan teknik ini, integritas data, *atomicity (undo rule)*, dan *durability (redo rule)* dapat dijamin. Tidak diperlukan untuk melakukan *flush* pada *data pages* ke *disk* setiap sebuah transaksi *commit* karena apabila terjadi *crash* pada sistem, dapat dilakukan *recovery* pada basis data menggunakan log.

WAL dapat mengurangi kebutuhan untuk melakukan *write* pada *disk* karena hanya log yang perlu di-*flush* untuk memastikan bahwa sebuah transaksi telah *commit*. Log ditulis secara berurutan sehingga *cost* untuk melakukan sinkronisasi pada log jauh lebih sedikit daripada melakukan *flush* pada *data pages*. WAL juga mendukung *on-line backup* dan *point-in-time recovery* dengan melakukan *archive* pada data WAL yang memungkinkan untuk menjalankan *revert* pada setiap waktu yang tersedia pada WAL.

- Continuous Archiving

Continuous Archiving merupakan teknik di mana file WAL di-*copy* ke *secondary storage*. Sebelumnya, pada WAL, log mencatat setiap perubahan yang terjadi pada basis data. Apabila terjadi *crash* pada sistem, basis data dapat di-*restore* dengan mengulangi entri-entri log yang dibuat pada *checkpoint* terakhir. Dengan *Continuous Archiving*, keberadaan log ini dapat digunakan sebagai strategi untuk melakukan *backup* pada basis data dengan menggabungkan *file-system backup* dan file WAL.

Recovery dapat dilakukan dengan melakukan *restore* pada *file-system backup* dan kemudian mengulangi entri-entri log pada file WAL sehingga dapat mengembalikan *state* basis data. *Continuous Archiving* menjamin proses *archive* file WAL dapat dilakukan secara terus-menerus. Penggunaan ini sangat membantu basis data yang berukuran besar karena tidak diperlukan untuk melakukan *backup* secara penuh.

- Point-in-Time Recovery

Point-in-Time Recovery (PITR) merupakan teknik di mana dapat dilakukan penghentian *replay* kapan pun dan tetap memiliki *snapshot* basis data yang konsisten pada saat itu. Hal ini membuat sangat mungkin untuk dilakukan proses *restore* basis data ke *state* mana pun sejak *base backup* dilakukan. Proses *Point-in-Time Recovery* terdiri dari *backup* dan *restore*. *Backup* merupakan proses menyimpan *state* basis data, sedangkan *restore* merupakan proses pemulihan terhadap *backup* yang telah dilakukan pada basis data.

Dalam PostgreSQL, langkah-langkah untuk melakukan proses *backup* adalah sebagai berikut.

1. Memodifikasi file `postgresql.conf` untuk mendukung *archive log*.
2. Membuat *base backup* dan simpan ke *remote storage*.
3. Membuat *WAL backup* dan simpan ke *remote storage*.

Sedangkan langkah-langkah untuk melakukan proses *recovery* adalah sebagai berikut.

1. Ekstrak file dari *base backup*.
2. Salin file dari folder `pg_xlog`.
3. Membuat file `recovery.conf`.
4. Memulai proses *recovery*.

Simulasi Kegagalan dan Recovery

Akan disimulasikan kegagalan pada PostgreSQL dan proses *recovery* yang dilakukan. Berikut merupakan perintah kode untuk melakukan *backup* pada basis data lalu melakukan *recovery* dengan teknik *Point-in-Time Recovery (PITR)* yang berasal dari referensi tutorial <https://www.scalingpostgres.com/tutorials/postgresql-backup-point-in-time-recovery/>.

```
# create directory for archive logs
sudo -H -u postgres mkdir /var/lib/postgresql/pg_log_archive

# enable archive logging
sudo nano /etc/postgresql/10/main/postgresql.conf

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'

# restart cluster
sudo systemctl restart postgresql@10-main
```

```

# create database with some data
sudo su - postgres
psql -c "create database test;"
psql test -c "
create table posts (
  id integer,
  title character varying(100),
  content text,
  published_at timestamp without time zone,
  type character varying(100)
);

insert into posts (id, title, content, published_at, type) values
(100, 'Intro to SQL', 'Epic SQL Content', '2018-01-01', 'SQL'),
(101, 'Intro to PostgreSQL', 'PostgreSQL is awesome!', now(), 'PostgreSQL');
"

# archive the logs
psql -c "select pg_switch_wal();" # pg_switch_xlog(); for versions < 10

# backup database
pg_basebackup -Ft -D /var/lib/postgresql/db_file_backup

# stop DB and destroy data
sudo systemctl stop postgresql@10-main
rm /var/lib/postgresql/10/main/* -r
ls /var/lib/postgresql/10/main/

# restore
tar xvf /var/lib/postgresql/db_file_backup/base.tar -C /var/lib/postgresql/10/main/
tar xvf /var/lib/postgresql/db_file_backup/pg_wal.tar -C /var/lib/postgresql/10/main/pg_wal/

# add recovery.conf
nano /var/lib/postgresql/10/main/recovery.conf

  restore_command = 'cp /var/lib/postgresql/pg_log_archive/%f %p'

# start DB
sudo systemctl start postgresql@10-main

# verify restore was successful
psql test -c "select * from posts;"

##### Do PITR to a Specific Time #####

# backup database and gzip
pg_basebackup -Ft -X none -D - | gzip > /var/lib/postgresql/db_file_backup.tar.gz

# wait
psql test -c "insert into posts (id, title, content, type) values

```

```

(102, 'Intro to SQL Where Clause', 'Easy as pie!', 'SQL'),
(103, 'Intro to SQL Order Clause', 'What comes first?', 'SQL');"

# archive the logs
psql -c "select pg_switch_wal();" # pg_switch_xlog(); for versions < 10

# stop DB and destroy data
sudo systemctl stop postgresql@10-main
rm /var/lib/postgresql/10/main/* -r
ls /var/lib/postgresql/10/main/

# restore
tar xvfz /var/lib/postgresql/db_file_backup.tar.gz -C /var/lib/postgresql/10/main/

# add recovery.conf
nano /var/lib/postgresql/10/main/recovery.conf

    restore_command = 'cp /var/lib/postgresql/pg_log_archive/%f %p'
    recovery_target_time = '2018-02-22 15:20:00 EST'

# start DB
sudo systemctl start postgresql@10-main

# verify restore was successful
psql test -c "select * from posts;"
tail -n 100 /var/log/postgresql/postgresql-10-main.log

# complete and enable database restore
psql -c "select pg_wal_replay_resume();"

```

Langkah-langkah *PITR* di atas akan disimulasikan sebagai bahan pembelajaran dan eksplorasi yang direpresentasikan oleh beberapa *screenshot* di bawah ini.

Berikut merupakan data awal pada tabel **posts**.

```

test=# SELECT * FROM posts;

```

id	title	content	published_at	type
100	Intro to SQL	Epic SQL Content	2018-01-01 00:00:00	SQL
101	Intro to PostgreSQL	PostgreSQL is awesome!	2021-11-20 20:56:24.076818	PostgreSQL

```

(2 rows)

```

Kemudian, akan dilakukan penambahan data pada **posts**.

```

postgres@vctl1e18-desktop:~$ psql test -c "insert into posts (id, title, content, type) values
> (102, 'Intro to SQL Where Clause', 'Easy as pie!', 'SQL'),
> (103, 'Intro to SQL Order Clause', 'What comes first?', 'SQL');"
INSERT 0 2

```


Data yang sudah ditambahkan dapat diverifikasi dengan melakukan *query* pada **posts**.

```
postgres@vctlie18-desktop:~$ psql test -c "select * from posts;"
 id | title | content | published_at | type
-----+-----+-----+-----+-----
 100 | Intro to SQL | Epic SQL Content | 2018-01-01 00:00:00 | SQL
 101 | Intro to PostgreSQL | PostgreSQL is awesome! | 2021-11-20 20:56:24.076818 | PostgreSQL
 102 | Intro to SQL Where Clause | Easy as pie! |  | SQL
 103 | Intro to SQL Order Clause | What comes first? |  | SQL
(4 rows)
```

Dilakukan *backup* pada basis data terlebih dahulu.

```
postgres@vctlie18-desktop:~$ tar xvf /var/lib/postgresql/db_file_backup/base.tar -C /var/lib/postgresql/10/main/
backup_label
tablespace_map
pg_replslot/
pg_twophase/
pg_multixact/
pg_multixact/members/
pg_multixact/members/0000
pg_multixact/offsets/
pg_multixact/offsets/0000
base/
base/13083/
base/13083/3467
base/13083/12934_vm
base/13083/3606
base/13083/112
base/13083/2607_fsm
base/13083/1259
base/13083/2328_vm
base/13083/3600
base/13083/2666
base/13083/2685
base/13083/3598_vm
base/13083/2692
base/13083/2606
base/13083/2669
base/13083/3599
base/13083/12941
base/13083/2704
base/13083/2620_vm
base/13083/3602_fsm
base/13083/2224_vm
base/13083/2703
base/13083/2618_fsm
base/13083/2674
base/13083/2606_vm
base/13083/12929_vm
```

Kemudian, *service* basis data dihentikan dan basis data dihapus.

```
postgres@vctlie18-desktop:~$ sudo systemctl stop postgresql@10-main
[sudo] password for postgres:
postgres@vctlie18-desktop:~$ rm /var/lib/postgresql/10/main/* -r
postgres@vctlie18-desktop:~$ ls /var/lib/postgresql/10/main/
```

Lalu, *service* basis data dinyalakan kembali dan proses *recovery* dijalankan melalui file *recovery.conf*. Proses *recovery* berhasil dan data di dalam **posts** terjaga.

```
postgres@vctlie18-desktop:~$ sudo systemctl start postgresql@10-main
postgres@vctlie18-desktop:~$ psql test -c "select * from posts;"
 id | title | content | published_at | type
-----+-----+-----+-----+-----
 100 | Intro to SQL | Epic SQL Content | 2018-01-01 00:00:00 | SQL
 101 | Intro to PostgreSQL | PostgreSQL is awesome! | 2021-11-20 20:56:24.076818 | PostgreSQL
 102 | Intro to SQL Where Clause | Easy as pie! |  | SQL
 103 | Intro to SQL Order Clause | What comes first? |  | SQL
(4 rows)
```

Berikut merupakan log proses *recovery*, dapat dilihat bahwa dilakukan proses *redo* yang dimulai dari log 0/3000028 sampai 0/40001E8.

```
postgres@vctlie18-desktop:~$ tail -n 28 /var/log/postgresql/postgresql-10-main.log
2021-11-20 21:08:25.871 WIB [1004] LOG: redo starts at 0/4000060
2021-11-20 21:08:25.871 WIB [1004] LOG: invalid record length at 0/4000140: wanted 24, got 0
2021-11-20 21:08:25.871 WIB [1004] LOG: redo done at 0/4000108
2021-11-20 21:08:25.881 WIB [947] LOG: database system is ready to accept connections
2021-11-20 21:08:26.403 WIB [1126] [unknown]@[unknown] LOG: incomplete startup packet
2021-11-20 21:11:33.693 WIB [947] LOG: received fast shutdown request
2021-11-20 21:11:33.694 WIB [947] LOG: aborting any active transactions
2021-11-20 21:11:33.696 WIB [947] LOG: worker process: logical replication launcher (PID 1012) exited with exit code 1
2021-11-20 21:11:33.696 WIB [1006] LOG: shutting down
2021-11-20 21:11:33.743 WIB [947] LOG: database system is shut down
2021-11-20 21:12:08.576 WIB [8838] LOG: listening on IPv4 address "127.0.0.1", port 5432
2021-11-20 21:12:08.576 WIB [8838] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2021-11-20 21:12:08.587 WIB [8839] LOG: database system was interrupted; last known up at 2021-11-20 20:58:21 WIB
2021-11-20 21:12:08.635 WIB [8839] LOG: starting archive recovery
2021-11-20 21:12:08.653 WIB [8839] LOG: restored log file "000000010000000000000003" from archive
2021-11-20 21:12:08.672 WIB [8839] LOG: redo starts at 0/3000028
2021-11-20 21:12:08.672 WIB [8839] LOG: consistent recovery state reached at 0/30000F8
2021-11-20 21:12:08.674 WIB [8838] LOG: database system is ready to accept read only connections
2021-11-20 21:12:08.690 WIB [8839] LOG: restored log file "000000010000000000000004" from archive
cp: cannot stat '/var/lib/postgresql/pg_log_archive/000000010000000000000005': No such file or directory
2021-11-20 21:12:08.706 WIB [8839] LOG: redo done at 0/40001E8
2021-11-20 21:12:08.724 WIB [8839] LOG: restored log file "000000010000000000000004" from archive
cp: cannot stat '/var/lib/postgresql/pg_log_archive/00000002.history': No such file or directory
2021-11-20 21:12:08.745 WIB [8839] LOG: selected new timeline ID: 2
2021-11-20 21:12:08.776 WIB [8839] LOG: archive recovery complete
cp: cannot stat '/var/lib/postgresql/pg_log_archive/00000001.history': No such file or directory
2021-11-20 21:12:08.886 WIB [8838] LOG: database system is ready to accept connections
```

4. Kesimpulan dan Saran

A. Kesimpulan

Dari hasil eksplorasi *concurrency control* yang telah dilakukan, dapat diketahui bahwa tingkat isolasi yang paling aman dalam menjalankan sebuah transaksi adalah *serializability* karena memang tingkat isolasi ini mengatur setiap eksekusi konkuren dari serangkaian transaksi dengan menjalankannya satu per satu dalam urutan tertentu dan akan error apabila terdapat 2 transaksi yang mengubah sebuah tabel secara bersamaan. Selain itu, pada tingkat isolasi *repeatable read*, dari hasil simulasi yang diujicobakan, diketahui bahwa tingkat isolasi ini meskipun sebuah transaksi telah melakukan perubahan pada data dan melakukan *commit*, transaksi yang berjalan bersamaan tidak dapat melihat perubahannya sehingga tidak memungkinkan adanya fenomena *non-repeatable read* tetapi tetap dapat terjadi *serialization anomaly*. Sementara, pada tingkat isolasi *read committed*, perubahan yang dilakukan sebuah transaksi dapat dilihat perubahannya pada transaksi yang lain sehingga memungkinkan adanya *non-repeatable read* pada transaksinya.

Dari hasil implementasi *concurrency control* yang telah dilakukan, Algoritma Simple Locking Protocol memiliki banyak kekurangan diantaranya seperti tidak dapat memastikan transaksi berjalan secara serial (not serializable), dapat terjadi deadlock (kasus ini pada program ditangani dengan cara menerapkan protokol wait-die sebagai deadlock prevention), dan dapat terjadi dirty read antar transaksi yang menyebabkan simple locking tidak recoverable. Untuk membuat performa Algoritma Simple Locking ini meningkat, dapat diterapkan Algoritma Two-Phase Locking yang dapat memastikan transaksi bersifat serializable, dan recoverable. Algoritma *Multiversion Timestamp Ordering Protocol* dapat memastikan transaksi dapat berjalan secara serial (serializable). Selain itu, protokol juga menjamin terhindarnya dari *deadlock*. Akan tetapi, masih dapat terjadi *cascading rollback*. Protokol akan selalu berhasil dan tidak pernah menunggu dalam melakukan aksi *read*. Protokol akan membuat sebuah versi baru jika melakukan aksi *write* sehingga memiliki kemungkinan yang lebih kecil untuk terjadinya *rollback*.

Dari hasil eksplorasi *recovery*, *Continuous Archiving* dan *Point-in-Time Recovery* sama-sama menggunakan *WAL* dalam implementasinya. *Continuous Archiving* menggunakan *base backup* di mana file *WAL* ini disalin ke *secondary storage* yang prosesnya dilakukan secara terus-menerus, sedangkan *Point-in-Time Recovery*

memungkinkan terjadinya penghentian proses *replay* kapan pun dan tetap memiliki basis data yang konsisten.

Dari hasil simulasi kegagalan dan *recovery* yang dilakukan dengan menggunakan *PITR*, dapat dilihat bahwa *base backup* dan file *WAL* memiliki faktor penting dalam proses *recovery*. *PITR* telah terbukti dapat memulihkan kondisi basis data ke *state* tertentu. Namun, kekurangannya adalah basis data tidak *available* pada saat proses *recovery* sedang berjalan sehingga dapat membutuhkan waktu yang relatif lama apabila basis data yang ingin di-*recover* berukuran cukup besar.

B. Saran

Untuk pemilihan tingkat isolasi pada saat melakukan *concurrency control*, sebaiknya diperhatikan kebutuhan dan juga relasi-relasi dari basis data yang dimiliki seperti misalnya apabila ingin menggunakan isolasi *serializability*, transaksi-transaksi yang dijalankan akan memakan waktu yang sedikit lebih lama karena transaksi akan dijalankan satu per satu sehingga faktor waktu tersebut harus diperhatikan.

Saran yang didapat dari pengerjaan tugas besar ini adalah pemilihan protokol konkurensi, seperti *Simple Locking*, *OCC*, dan *MVCC* dapat dilakukan bergantung pada penggunaan dan kondisi tertentu, begitu pun pemilihan teknik *recovery*, seperti *Continuous Archiving* dan *PITR* karena setiap protokol/teknik yang digunakan memiliki kelebihan dan kekurangannya masing-masing. Selain itu, beberapa protokol seperti *Simple Locking* protocol hanya dapat digunakan sebagai bahan pembelajaran dan tidak diimplementasikan ke kasus *real* karena kekurangan yang telah dijelaskan sebelumnya.

5. Pembagian Kerja

13518081 Vincentius Lienardo

- Bagian C: Eksplorasi *Recovery*

13519029 Nicholas Chen

- Bagian B.a: *Simple Locking (exclusive locks only)*

13519041 Mohammad Dwinta Harits Cahyana

- Bagian B.b: *Serial Optimistic Concurrency Control (OCC)*

13519085 Nizamixavier Rafif Lutvie

- Bagian B.c: *Multiversion Timestamp Ordering Concurrency Control (MVCC)*

13519153 Maximillian Lukman

- Bagian A: Eksplorasi *Concurrency Control*

Referensi

<https://www.postgresql.org/docs/7.2/xact-serializable.html>

<https://www.postgresql.org/docs/8.3/tutorial-transactions.html>

<https://www.postgresql.org/docs/9.1/wal-intro.html>

<https://www.postgresql.org/docs/9.1/continuous-archiving.html>

<https://www.postgresql.org/docs/9.1/sql-start-transaction.html>

<https://www.postgresql.org/docs/9.5/transaction-iso.html>

<https://www.scalingpostgres.com/tutorials/postgresql-backup-point-in-time-recovery/>

www.javatpoint.com/dbms-lock-based-protocol#:~:text=Simplistic%20lock%20protocol,item%20after%20completing%20the%20transaction