

## **TUGAS BESAR 2**

**IF3140 Manajemen Basis Data**

### **MEKANISME CONCURRENCY CONTROL DAN RECOVERY**

**KELAS 02**

**Dosen : Latifa Dwiyanti, S.T., M.T.**



**Disusun oleh kelompok 11**

Muhammad Helmi Hibatullah	13520014
Hana Fathiyah	13520047
Jason Kanggara	13520080
Muhammad Gilang Ramadhan	13520137

**Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung  
2022**

# BAB I

## Eksplorasi Concurrency Control

### 1.1. Serializability

Menurut definisinya, *Serializable* adalah tingkat isolasi yang memberikan derajat isolasi yang paling tinggi. Tingkatan ini mengemulasikan eksekusi transaksi yang serial untuk semua schedulennya. Semua transaksi pada tingkat *serializability* hanya akan melihat data yang telah di-*commit* oleh transaksi lainnya sebelum transaksi tersebut dimulai. Khusus untuk operasi yang mengubah data pada sebuah tuple, jika tuple tersebut telah diubah oleh transaksi lainnya yang berjalan secara konkuren dan transaksi konkuren tersebut telah *commit*, maka operasi tersebut akan di-*rollback*. Kemudian tingkat isolasi ini juga melihat apakah transaksi-transaksi yang berjalan secara konkuren konsisten dengan setidaknya sebuah eksekusi serial transaksi-transaksi tersebut. Jika kebutuhan tersebut tidak terpenuhi, maka telah terjadi sebuah kesalahan terhadap *serialization*. Akibatnya, terjadi *error serialization failure* pada transaksi *schedule* yang dieksekusi. Gambar di bawah ini merupakan contoh simulasi transaksi pada level *serializability*

```
tubes2_kel11_k02=# begin;
BEGIN
tubes2_kel11_k02=# SELECT * FROM tubes_2;
      nim |      name      | jamtidur
-----+-----+-----
 13520014 | Muhammad Helmi Hibatullah |      8
 13520047 | Hana Fathiyah      |      9
 13520080 | Jason Kanggara     |      6
 13520137 | Muhammad Gilang Ramadhan |      7
(4 rows)

tubes2_kel11_k02=# UPDATE tubes_2 SET jamtidur=5 WHERE
nim=13520047;
UPDATE 1
tubes2_kel11_k02=# SELECT * FROM tubes_2;
      nim |      name      | jamtidur
-----+-----+-----
 13520014 | Muhammad Helmi Hibatullah |      8
 13520080 | Jason Kanggara     |      6
 13520137 | Muhammad Gilang Ramadhan |      7
 13520047 | Hana Fathiyah      |      5
(4 rows)

tubes2_kel11_k02=# commit;
COMMIT

Type "help" for help.
tubes2_kel11_k02=# begin transaction isolation level serializable;
BEGIN
tubes2_kel11_k02=# SELECT * FROM tubes_2;
      nim |      name      | jamtidur
-----+-----+-----
 13520014 | Muhammad Helmi Hibatullah |      8
 13520047 | Hana Fathiyah      |      9
 13520080 | Jason Kanggara     |      6
 13520137 | Muhammad Gilang Ramadhan |      7
(4 rows)

tubes2_kel11_k02=# UPDATE tubes_2 SET jamtidur=5 WHERE nim=13520047;
ERROR:  could not serialize access due to concurrent update
tubes2_kel11_k02=!#
```

Gambar 1.1. Contoh transaksi pada level *serializability*

Urutan transaksi di atas jika dituliskan dalam tabel adalah sebagai berikut.

Tabel 1.1. Contoh transaksi pada level *serializability*

T1	T2
SELECT * FROM tubes_2;	
	SELECT * FROM tubes_2;
UPDATE tubes_2 SET jamtidur=5 WHERE	

nim=13520047;	
	UPDATE tubes_2 SET jamtidur=5 WHERE nim=13520047;
SELECT * FROM tubes_2;	
COMMIT;	
	ABORT;

Dalam hal ini, *update* yang dilakukan pada transaksi T2 tidak dapat berjalan karena masih terdapat transaksi yang belum di-*commit* pada transaksi T1.

## 1.2. Repeatable Read

Menurut definisinya, Repeatable read merupakan tingkatan isolasi yang menjamin pembacaan suatu transaksi akan membaca data yang sama ketika membaca suatu baris dua kali. Hal ini dilakukan dengan implementasi sistem *locking* dimana transaksi akan memiliki *read lock* pada setiap baris yang dibaca dan *write lock* pada semua baris yang di *insert*, *update*, dan *delete*. Level isolasi yang menjamin pembacaan dua operasi *read* pada baris yang sama di suatu transaksi akan dapat melihat data yang sama. Namun, level isolasi ini juga lebih *restrictive* dimana hal ini juga akan mengurangi tingkat konkurensi dari transaksi yang terjadi. Jika dilihat dari tingkat konkurensinya, *repeatable read* memiliki tingkat konkurensi yang lebih rendah daripada *read committed* akan tetapi masih lebih tinggi dari pada *serializability*. Gambar di bawah ini merupakan contoh simulasi transaksi pada level *repeatable read*.

```

tubes2_kel11_k02=# begin;
BEGIN
tubes2_kel11_k02=# SELECT * FROM tubes_2;
 nim | name | jamtidur
-----+-----+-----
13520014 | Muhammad Helmi Hibatullah | 8
13520080 | Jason Kanggara | 6
13520137 | Muhammad Gilang Ramadhan | 7
13520047 | Hana Fathiyah | 5
(4 rows)

tubes2_kel11_k02=# UPDATE tubes_2 SET jamtidur=5 WHERE
 nim=13520014;
UPDATE 1
tubes2_kel11_k02=# COMMIT;
COMMIT
tubes2_kel11_k02=#

tubes2_kel11_k02=# begin transaction isolation level repeatable read;
BEGIN
tubes2_kel11_k02=# SELECT * FROM tubes_2;
 nim | name | jamtidur
-----+-----+-----
13520014 | Muhammad Helmi Hibatullah | 8
13520080 | Jason Kanggara | 6
13520137 | Muhammad Gilang Ramadhan | 7
13520047 | Hana Fathiyah | 5
(4 rows)

tubes2_kel11_k02=#

```

Gambar 1.2. Contoh transaksi pada level *repeatable read*

Urutan transaksi di atas jika dituliskan dalam tabel adalah sebagai berikut.

Tabel 1.2. Contoh transaksi pada level *repeatable read*

T1	T2
----	----

SELECT * FROM tubes_2;	
	SELECT * FROM tubes_2;
UPDATE tubes_2 SET jamtidur=5 WHERE nim=13520014;	
COMMIT;	
	SELECT * FROM tubes_2;

Dalam hal ini, jika dilakukan *SELECT* pada transaksi T2, masih menghasilkan data yang sama kedua kalinya padahal terdapat transaksi T1 yang melakukan perubahan data dan melakukan *commit* di dalamnya.

### 1.3. Read Committed

Menurut definisinya, *Read committed* adalah level isolasi yang memastikan bahwa data yang dibaca telah di commit pada saat data tersebut dibaca, sehingga hal tersebut dapat mencegah pelanggaran isolasi *dirty read*. Pada *read committed*, ketika suatu transaksi berjalan pada level isolasi ini, operasi *SELECT* hanya melihat (membaca) data yang di commit sebelum querynya dimulai dan tidak pernah melihat data yang tidak di *commit* atau perubahan yang di-commit selama eksekusi *query* dari transaksi konkuren. Kelemahan dari *Read committed* yaitu hanya dapat memastikan bahwa data tersebut telah di commit saja, akibatnya pelanggaran seperti *non repeatable read* dan *phantom phenomena* dapat tetap terjadi karena level isolasi ini tidak akan memastikan bahwa data tidak akan berubah, data dapat diubah secara bebas setelah dibaca. Berikut pada gambar di bawah ini disajikan contoh simulasi transaksi pada level *read committed*.

```

tubes2_kel11_k02=# begin;
BEGIN
tubes2_kel11_k02=# SELECT * FROM tubes_2;
 nim | name | jamtidur
-----+-----+-----
13520080 | Jason Kanggara | 6
13520137 | Muhammad Gilang Ramadhan | 7
13520047 | Hana Fathiyah | 5
13520014 | Muhammad Helmi Hibatullah | 5
(4 rows)

tubes2_kel11_k02=# UPDATE tubes_2 SET jamtidur=4 WHERE
nim=13520137;
UPDATE 1
tubes2_kel11_k02=# SELECT * FROM tubes_2;
 nim | name | jamtidur
-----+-----+-----
13520080 | Jason Kanggara | 6
13520047 | Hana Fathiyah | 5
13520014 | Muhammad Helmi Hibatullah | 5
13520137 | Muhammad Gilang Ramadhan | 4
(4 rows)

tubes2_kel11_k02=# COMMIT;
COMMIT
tubes2_kel11_k02=#

tubes2_kel11_k02=# begin transaction isolation level read committed;
BEGIN
tubes2_kel11_k02=# SELECT * FROM tubes_2;
 nim | name | jamtidur
-----+-----+-----
13520080 | Jason Kanggara | 6
13520137 | Muhammad Gilang Ramadhan | 7
13520047 | Hana Fathiyah | 5
13520014 | Muhammad Helmi Hibatullah | 5
(4 rows)

tubes2_kel11_k02=# SELECT * FROM tubes_2;
 nim | name | jamtidur
-----+-----+-----
13520080 | Jason Kanggara | 6
13520047 | Hana Fathiyah | 5
13520014 | Muhammad Helmi Hibatullah | 5
13520137 | Muhammad Gilang Ramadhan | 4
(4 rows)

tubes2_kel11_k02=#

```

Gambar 1.3. Contoh transaksi pada level *read committed*

Urutan transaksi di atas jika dituliskan dalam tabel adalah sebagai berikut.

Tabel 1.3. Contoh transaksi pada level *read committed*

T1	T2
SELECT * FROM tubes_2;	
UPDATE tubes_2 SET jamtidur=4 WHERE nim=13520014;	
SELECT * FROM tubes_2;	
	SELECT * FROM tubes_2;
COMMIT;	
	SELECT * FROM tubes_2;

Dalam hal ini, transaksi T2 tidak dapat membaca hasil perubahan data yang dilakukan oleh transaksi T1 sebelum transaksi T1 melakukan *COMMIT*. Data perubahan yang dilakukan oleh transaksi T1 dapat dibaca oleh transaksi T2 ketika perubahan tersebut telah di-*commit* oleh transaksi T1.

#### 1.4. Read Uncommitted

Menurut definisinya, *Read uncommitted* adalah tingkat isolasi yang memberikan derajat isolasi yang paling rendah dan merupakan tingkat isolasi yang paling tidak membatasi (the least restrictive isolation levels). Transaksi yang berjalan pada tingkat *read uncommitted* tidak mengeluarkan shared lock untuk mencegah transaksi lain memodifikasi data yang sedang dibaca oleh transaksi saat ini. Selain itu, transaksi pada *read uncommitted* tidak akan terblokir oleh exclusive lock sehingga pada tingkatan ini suatu transaksi dapat membaca suatu perubahan item data oleh transaksi lain yang belum di commit. Maka dari itu, dimungkinkan adanya dirty read.

Selain itu juga, pada *read uncommitted* ini eksekusi suatu transaksi tidak terisolasi dengan transaksi lainnya. Artinya, transaksi yang berjalan pada tingkat *read uncommitted* dalam menyelesaikan task atau operasi yang terdapat didalamnya bergantung (dependent) dengan transaksi lain karena perubahan item data yang dibuat oleh transaksi lain terlihat meskipun transaksi tersebut belum menyelesaikan seluruh task-nya (belum commit).

Dikarenakan eksekusi transaksi pada tingkatan ini tidak terisolasi, maka pada tingkatan *read uncommitted* juga dimungkinkan terjadinya *non-repeatable reads* dan *phantom reads*. *Non-repeatable reads* adalah suatu kondisi dimana suatu transaksi membaca data yang sama dua kali, namun memperoleh hasil yang berbeda. Sedangkan *phantom reads* adalah suatu kondisi ketika suatu transaksi sedang melakukan pembacaan data, terjadi penambahan *record* baru atau penghapusan suatu *record* oleh transaksi lain pada data (relasi) yang sedang dibaca. Hal ini mengakibatkan, transaksi melakukan pembacaan *record* yang sudah hilang atau tidak membaca keseluruhan *record* pada data.

Adapun pada tabel di bawah ini disajikan contoh *schedule* eksekusi transaksi pada level *read uncommitted* dengan nilai awal data X dan Y pada basis data adalah 2 dan 3. Misalkan kedua transaksi berjalan secara konkuren, yaitu T1 dan T2.

T1	T2	Keterangan
R1(X)		Nilai yang terbaca : X = 2
R1(Y)		Nilai yang terbaca : Y = 3
	R2(Y)	Nilai yang terbaca : Y = 3
	$Y = Y + 2 = 5$	
	W2(Y)	Nilai Y berubah menjadi 5
R1(Y)		Nilai yang terbaca : Y = 5 Maka terjadi : <ul style="list-style-type: none"> <li>• <b>Dirty read</b> karena membaca hasil modifikasi T2 padahal T2 belum <i>commit</i></li> <li>• <b>Non-repeatable reads</b> karena hasil <i>read</i> data B saat ini berbeda dengan hasil <i>read</i> sebelumnya</li> </ul>
$X = X - 1 = 1$		
W1(X)		Nilai X berubah menjadi 1
C1		
	C2	

## BAB II

### Implementasi Concurrency Control Protocol

#### 2.1. Simple Locking (exclusive locks only)

*Simple locking* merupakan versi praktis dari *two phase locking* yang ketika transaksi masuk ke sistem, setiap transaksi harus meminta exclusive lock untuk read maupun write transaction. Jika *request lock* ditolak karena ada transaksi yang lain yang sedang melakukan *locking* terhadap data tersebut maka terdapat dua kemungkinan yaitu pertama, jika transaksi hanya *request single read* atau *write* maka transaksi tersebut *wait for lock* hingga *lock* diperoleh kemudian transaksi dapat berjalan. Kedua, jika transaksi tidak hanya *single read / write* maka dilakukan *release* terhadap semua *lock* sebelum *request lock* ini ditolak, *abort* transaksi, dan ulangi transaksi. Setelah itu, semua *lock* dapat dibebaskan setelah *commit/abort*.

#### 2.2.1 Screenshot Hasil Percobaan

Contoh masukkan

R1(X),R2(Y),R1(Y)

Hasil keluaran program

L1(X) : Grant Lock-X(X) to T1  
R1(X) : T1 reads X  
L2(Y) : Grant Lock-X(Y) to T2  
R2(Y) : T2 reads Y  
C2 : Commit T2  
UL2(Y): Unlock Lock-X(2) from T2  
L1(Y) : Grant Lock-X(Y) to T1  
R1(Y) : T1 reads Y  
C1 : Commit T1  
UL1(X): Unlock Lock-X(1) from T1  
UL1(Y): Unlock Lock-X(1) from T1

Program dapat dijalankan dengan menjalankan *command* tersebut, lalu pengguna dapat memilih apakah masukkan berupa file .txt atau dengan mengetik transaksinya langsung pada terminal.

python simpleLocking.py

### 2.1.2. Analisis dan Hasil dari Algoritma yang Diterapkan

Berikut merupakan rincian program yang diterapkan untuk implementasi simulasi Simple Locking Concurrency Control Protocol dengan sifat Exclusive Locks Only.

#### 1. File simpleLocking.py

File diimplementasikan dengan metode object oriented berupa class SimpleLocking yang berisi method untuk menjalankan proses concurrency control dengan protocol Simple Locking (exclusive locks only). Method/fungsi yang terdapat pada class SimpleLocking adalah sebagai berikut

- def concurrency\_control(self)  
Merupakan fungsi utama yang menampung seluruh *helper function* untuk menjalankan algoritma Simple Locking Exclusive Locks Only. Pada method ini, akan dijalankan proses untuk melakukan locking dari transaksi, memeriksa juga apakah akan dilakukan rollback atau tidak, commit transaksi, transaksi dalam proses waiting, unlock transaksi, dan juga menjalankan fungsi fungsi lainnya yang dapat membantu proses concurrency control
- def can\_lock(self, transaction)  
Merupakan fungsi untuk menentukan apakah transaksi dapat di lock atau tidak dengan memeriksa lock dictionary yang sudah diinisialisasi pada `__init__` class SimpleLocking
- def lock(self, transaction)  
Merupakan fungsi untuk melakukan lock transaksi jika transaksi belum di-lock
- def unlock(self, transaction)  
Merupakan fungsi untuk melakukan unlock transaksi
- def check\_rollback(self, transaction)  
Merupakan fungsi untuk memeriksa apakah terdapat transaksi yang harus di-rollback. Jika iya, maka transaksi yang sudah di proses akan dimasukkan ke dalam list of rollback yang sudah diinisialisasi
- def check\_remaining(self, transaction)  
Merupakan fungsi untuk memeriksa apakah masih terdapat transaksi yang belum diproses saat sedang rollback. Jika iya, maka transaksi tersebut pada nomor transaksi yang sama akan dimasukkan ke list of rollback.
- def check\_queue(self, transaction)  
Merupakan fungsi untuk memeriksa apakah terdapat transaksi yang sedang dalam tahap waiting. Jika terdapat transaksi yang waiting, maka saat sudah siap dijalankan, transaksi tersebut akan dijalankan proses locking
- def is\_commit(self, transaction)  
Merupakan fungsi untuk menentukan apakah transaksi pada nomor transaksi tersebut merupakan transaksi terakhir pada list of transaction. Jika iya, maka setelah setelah dijalankan transaksi tersebut, akan dilakukan commit dan unlock



lock yang ada

- `def execute_rollback(self, transaction)`

Merupakan fungsi untuk menjalankan rollback, mulai dari abort, memasukkan transaksi ke list of rollback, unlock lock yang ada pada nomor transaksi yang di rollback, dan dilanjutkan untuk mengulangi proses locking

- `def still_exist(self, transaction)`

Merupakan fungsi untuk menentukan apakah nomor transaksi masih ada atau tidak.

## 2.2. Serial Optimistic Concurrency Control

*Serial Optimistic Concurrency Control* (OCC) adalah salah satu versi dari *Optimistic Concurrency Control* (OCC) dimana proses validasi transaksi dilakukan secara serial. Adapun *Concurrency Control* jenis ini sendiri adalah salah satu protokol *Concurrency Control* yang tidak menggunakan *locking* dan memungkinkan transaksi berlanjut hingga sebelum tahap *commit*, dimana sebelum commit akan dilakukan pengecekan untuk melihat apakah operasi yang mereka lakukan mengalami konflik atau tidak terhadap objek. Pada *Optimistic Concurrency Control* (OCC) proses penulisan data pada database ditunda hingga akhir transaksi dan selalu melakukan pelacakan item data yang dibaca/ditulis oleh transaksi untuk kepentingan validasi.

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

- Begin:** Tahap dilakukannya pencatatan timestamp yang menandakan dimulainya suatu transaksi.
- Read and execution (modify) phase:** Tahap dilakukannya pembacaan data pada *database* dan penulisan data hanya pada *temporary local variables* saja.
- Validation phase :** Tahap dimana transaksi telah menjalankan semua operasi yang terdapat didalamnya dan melakukan "*validation test*" untuk menentukan apakah *local variables* dapat dituliskan pada database tanpa melanggar serializability. Pada "*validation test*" akan diperiksa apakah terdapat transaksi lain, yang berjalan konkuren dengan transaksi tersebut, yang telah melakukan modifikasi data pada *database* untuk *local variables* tersebut.

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

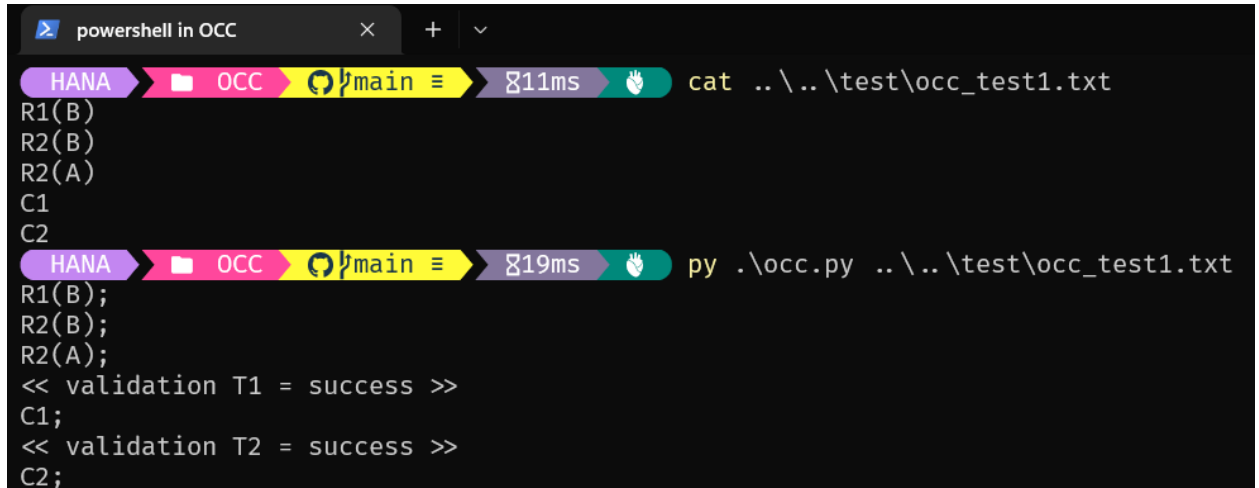
- StartTS(Ti) : Waktu ketika Transaksi Ti memulai Read and execution (modify) phase.
- ValidationTS(Ti) : Waktu ketika Transaksi Ti memulai proses validasi (memasuki validation phase).
- FinishTS(Ti) : Waktu ketika Transaksi Ti selesai melakukan penulisan data pada basis data (waktu write phase berakhir).

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.

- finishTS(Ti) < startTS(Tj)

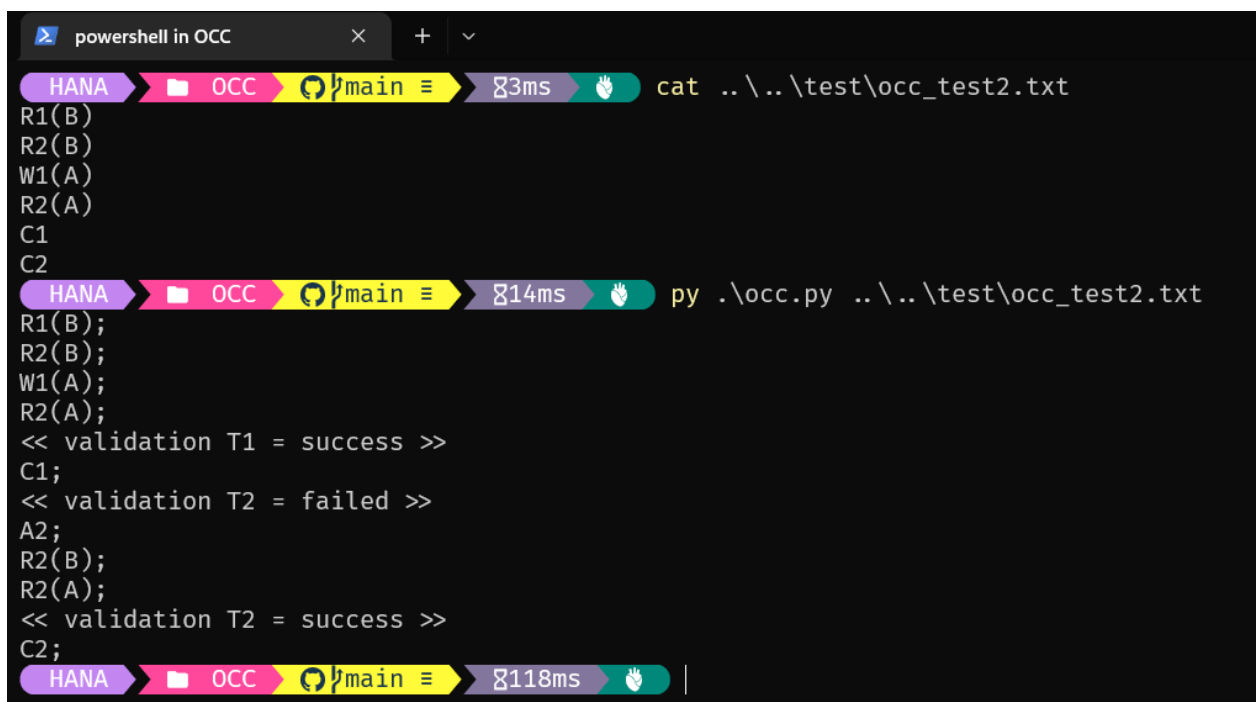
- b.  $\text{startTS}(T_j) < \text{finishTS}(T_i) < \text{validationTS}(T_j)$  dan *Write Set* pada  $T_i$  tidak beririsan dengan *Read Set* dan *Write Set* dari  $T_j$

### 2.2.1. Screenshot Hasil Percobaan



```
powershell in OCC
HANA OCC main 811ms cat ../../test/occ_test1.txt
R1(B)
R2(B)
R2(A)
C1
C2
HANA OCC main 819ms py .\occ.py ../../test/occ_test1.txt
R1(B);
R2(B);
R2(A);
<< validation T1 = success >>
C1;
<< validation T2 = success >>
C2;
```

Gambar 2.2.1. Percobaan 1, Hasil Validasi Kedua Transaksi Sukses



```
powershell in OCC
HANA OCC main 83ms cat ../../test/occ_test2.txt
R1(B)
R2(B)
W1(A)
R2(A)
C1
C2
HANA OCC main 814ms py .\occ.py ../../test/occ_test2.txt
R1(B);
R2(B);
W1(A);
R2(A);
<< validation T1 = success >>
C1;
<< validation T2 = failed >>
A2;
R2(B);
R2(A);
<< validation T2 = success >>
C2;
HANA OCC main 8118ms
```

Gambar 2.2.2. Percobaan 2, Hasil Validasi Transaksi T1 sukses dan T2 gagal

```
powershell in OCC
HANANA OCC main 82ms cat ../test/occ_test3.txt
R1(B)
R2(B)
W1(A)
W2(A)
C2
C1
HANANA OCC main 823ms py .\occ.py ../test/occ_test3.txt
R1(B);
R2(B);
W1(A);
W2(A);
<< validation T2 = success >>
C2;
<< validation T1 = success >>
C1;
HANANA OCC main 8127ms
```

Gambar 2.2.3. Percobaan 3, Hasil Validasi untuk *Test Case 3*

Program dijalankan dengan mengetikkan terlebih dahulu urutan transaksi pada suatu *file*. Kemudian, kita dapat memindahkan direktori ke dalam direktori *src*. Setelah itu, kita dapat menjalankan program dengan perintah sebagai berikut.

```
py .\occ.py <path file>
```

atau

```
python .\occ.py <path file>
```

### 2.2.2. Analisis dari Hasil Algoritma yang Diterapkan

Berikut merupakan rincian program yang diterapkan dalam hal implementasi serial optimistic concurrency control.

#### 1. File *transaction\_occ.py*

File ini berisi kelas dan fungsi yang diperlukan di dalam program, yakni sebagai berikut.

- class *Action*(*NamedTuple*)

*Attribute:*

*operation*, menandakan proses yang dilakukan untuk setiap tahap, apakah *write*, *read*, *commit*, atau *abort*.

*time*, menandakan waktu terjadinya proses atau aksi tersebut

*value*, menandakan nilai yang disimpan pada setiap aksi

*Method:*

*print*, untuk mencetak setiap aksi

- def *build\_action*(*action\_str*: str, *time*: int) → *Action*:

merupakan fungsi yang menerjemahkan setiap aksi dalam waktu tertentu ke dalam kelas aksi. Keluaran dari fungsi ini adalah kelas Action

- class Transaction

*Attribute:*

number, menyimpan nomor transaksi

startTS, menyimpan waktu start

finishTS, menyimpan waktu finish

actions, menyimpan list aksi yang dilakukan

state, menghitung aksi

read\_sets, himpunan aksi read dalam transaksi nomor tertentu

write\_sets, himpunan aksi write dalam transaksi nomor tertentu

is\_end, penanda apakah transaksi tersebut sudah selesai atau belum

*Method:*

add\_action, menambahkan aksi ke dalam actions, write\_sets, dan read\_sets

get\_current\_action, menambahkan action terkini

increment\_state, penambahan state

## 2. File occ.py

File ini berisi algoritma yang diterapkan dalam OCC. Program diawali dengan pembacaan file transaksi yang dalam hal ini digunakan library sys untuk inputnya. Selanjutnya, dilakukan penulisan nomor transaksi berapa saja yang ada di dalam file tersebut. Penulisan ini diawali dengan pengecekan apakah suatu nomor transaksi tersebut sudah terdapat di dalam *key dictTrans* atau belum. Jika, belum, buat kelas baru dengan nomor transaksi yang sedang dicek dan waktu yang sedang berlangsung. Jika sudah, tambahkan aksi yang baru di dalam kelas pada *dictTrans* tersebut. Lakukan sampai seluruh transaksi terbaca oleh file.

Selanjutnya, dilakukan traversal untuk setiap transaksi yang ditandai dengan *key* di dalam *dictTrans*. Selama masih ada transaksi yang belum selesai, traversal tersebut akan terus dijalankan. Untuk setiap transaksi, jika transaksi tersebut sudah selesai, transaksi tersebut tidak akan diperiksa. Jika suatu transaksi belum selesai, akan diperiksa waktu aksi yang sedang berlangsung, jika waktu aksi  $\leq$  waktu sistem, transaksi akan dijalankan. Apabila aksi pada suatu transaksi tersebut adalah *commit*, dilakukan pengecekan FinishTS dan startTS pada transaksi lainnya.

Dalam hal pengecekan FinishTS dan startTS, kami menggunakan prinsip de Morgan, yakni sebagai berikut.

```
if (not (dictTrans[i].finishTS < dictTrans[j].startTS) and
    not((dictTrans[j].startTS < dictTrans[i].finishTS < clock) and
        (len(dictTrans[i].write_sets.intersection(dictTrans[j].read_sets)) == 0))):
    is_executed = False
```

Prinsip ini berlawanan dengan prinsip kesuksesan suatu transaksi pada OCC, yakni

apabila finishTS transaksi lain < startTS transaksi tersebut atau startTS transaksi tersebut < finishTS transaksi lain < validationTS transaksi tersebut yang diistilahkan dengan clock. Jika berlawanan dengan prinsip keberhasilan suatu transaksi, maka transaksi tersebut tidak dieksekusi dan dilakukan perintah abort untuk menghentikan transaksi. Apabila seluruh transaksi berhasil, akan dicetak validation success, apabila gagal akan dicetak validation abort. Apabila terdapat transaksi yang gagal/abort, startTS pada transaksi tersebut akan diubah menjadi waktu terakhir transaksi lain yang berlangsung dan transaksi tersebut akan dimulai dari awal. Proses ini dijalankan terus menerus sampai seluruh transaksi selesai..

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

Pada Multiversion Timestamp Ordering, sebuah daftar objek maupun nilai tentatif objek tersebut disimpan pada setiap objek. Dimana daftar tersebut mewakili informasi dari nilai objek tersebut. Misalkan instruksi adalah read, maka akan dikembalikan nilai terbaru dari Qk dengan Qk adalah versi dari Q yang write timestamp terbesar yang kurang dari atau sama dengan timestamp dari transaksi yang memprosesnya. Apabila read timestamp dari Qk lebih kecil daripada timestamp transaksi yang memprosesnya, maka read timestamp tersebut akan diubah menjadi nilai dari timestamp yang memprosesnya.

Sementara itu, pada instruksi write, apabila timestamp dari transaksi yang sedang memproses kurang dari read timestamp Qk (Dengan Qk adalah versi dari Q yang write timestamp terbesar yang kurang dari atau sama dengan timestamp dari transaksi yang memprosesnya), maka transaksi akan dirollback. Jika tidak, akan diperiksa lagi apakah timestamp dari transaksi pemroses sama dengan write timestamp Qk, bila sama, maka instruksi transaksi pemroses akan melakukan overwrite konten Qk. Bila tidak, maka akan dibuat versi terbaru Qi dari Q dengan write dan read timestamp Qi adalah timestamp transaksi pemroses.

Adapun untuk source code dari MVCC disajikan sebagai berikut.

main.py
<pre> from cc import *  cc = CC("../test/mvcc_test1.txt", 40) cc.execute() </pre>

File main.py berisi kode program yang digunakan untuk menjalankan file test case yang ada pada folder test di directory IF3140\_K02\_G11 yang merupakan root dari semua file kode program tugas besar 2 IF3140 Manajemen Basis Data.

mvcc.py
<pre> from storage import * </pre>

```

class MVCC :
    def __init__(self):
        """
        Initiate MVCC class
        """
        return

    def read(self, key, nilai, storage, timestamp):
        """
        Method to implementing MVCC read rule's
        """

        index = storage.getHighestwriteTS(timestamp, key)
        nilai = storage.data[index][1]
        if(storage.data[index][2] <= timestamp): # R-TS(Qk) <= TS(Ti)
            storage.data[index][2] = timestamp
        return True

    def write(self, key, nilai, storage, timestamp):
        """
        Method to implementing MVCC write rule's
        """

        index = storage.getHighestwriteTS(timestamp, key)
        if(storage.data[index][2] <= timestamp):
            if(storage.data[index][3] != timestamp):
                # Create a new version Qi of Q
                latestVer = storage.getLatestVersion(key) + 1
                storage.addElement(key, nilai, timestamp, timestamp, latestVer)
            else: # TS(Ti) = W-TS(Qk)
                # Overwrite konten Qk
                storage.data[index][1] = nilai
        return True

```

```
else: # R-TS(Qk) > TS(Ti)
    # Rollback Ti
    return False
```

Adapun untuk kelas ini tidak memiliki atribut, tetapi tetap memiliki method-method yang dapat disajikan sebagai berikut.

- `__init__`  
Deskripsi : Untuk melakukan inisialisasi instans kepada kelas MVCC
- `read`  
Method `read` berisi instruksi yang melakukan `read` pada MVCC. Kemudian, akan dikembalikan nilai terbaru dari `Qk` dengan `Qk` yang merupakan versi dari `Q` yang `write timestamp` terbesar yang kurang dari atau sama dengan *timestamp* dari transaksi yang memprosesnya melalui pengembalian `idx` dari elemen `Qk`. Lalu diperiksa apakah `read timestamp` dari `Qk` lebih kecil daripada `timestamp` transaksi yang memprosesnya, bila ya maka *read timestamp* tersebut akan diubah menjadi nilai dari `timestamp` yang memprosesnya. Setelah itu, pada akhirnya `read` akan selalu mengembalikan `true`, sebagaimana instruksi `read` yang tidak akan berujung pada `abort`.
- `write`  
Method `write` berisi instruksi yang melakukan `write` pada MVCC. Apabila `timestamp` dari transaksi yang sedang memproses kurang dari `read timestamp` `Qk` (Dengan `Qk` adalah versi dari `Q` yang `write timestamp` terbesar yang kurang dari atau sama dengan `timestamp` dari transaksi yang memprosesnya melalui pengembalian `idx` dari elemen `Qk`), maka transaksi akan di-rollback (Return false). Jika tidak, akan diperiksa lagi apakah `timestamp` dari transaksi pemroses (`timestamp`) sama dengan `write timestamp` `Qk` (`storage.data[idx][2]`). Bila sama, maka instruksi transaksi pemroses akan melakukan `overwrite` konten `Qk` (`storage.data[idx][1] = nilai`). Bila tidak, maka akan dibuat versi terbaru `Qi` dari `Q` dengan `write` dan `read timestamp` `Qi` akan diassign dengan *timestamp* transaksi pemroses.

cc.py

```
from storage import Storage
from mvcc import MVCC
```

```
class CC:
```

```
    """ Transaction tabel with timestamp """
```

```
        trTbl = {}
```

```
    """ List contain transaction id which has been aborted """ trIDAborted = [] """ Queue to store
        transaction instruction"""
```

```

        inQueue = []
''' Queue contain transaction id which has been aborted '''
        inQueueAborted = []
''' Object storage '''
        ObjectStorage = None
''' MVCC processor '''
        mvcc_processor = None

def __init__(self, file : str, countItem : int):
    self.mvcc_processor = MVCC()
    self.inQueue = open(file, "r").read().split(" ")
    self.ObjectStorage = Storage(countItem)

def execute(self):
    tsCount = 0
    while (len(self.inQueue) != 0 or len(self.inQueueAborted) != 0):
        # Initiate transaction which has been aborted
        if (self.inQueue == []):
            self.inQueue = self.inQueueAborted.copy()
            self.trTbl.clear()
            self.inQueueAborted.clear()
            self.trIDAborted.clear()

        # Processing intruction
        currIn = self.inQueue.pop()

        # Variabel to store key (object)
        key = None

        # Variabel to store information of key (object)
        value = None

        # Instruction type

```



```
method = ""

# Transaction ID
trID = None

# Transaction Timestamp
trTS = None

# Checking instruction type
if(currln.find("w") != -1): # Write
    method = "w"
    currln_split = currln.split(method)
    trID = int(currln_split[0])
    kvPair = currln_split[1].split("|")

    key = int(kvPair[0])
    value = int(kvPair[1])

elif(currln.find("r") != -1): # Read
    method = "r"
    currln_split = currln.split(method)
    trID = int(currln_split[0])
    key = int(currln_split[1])

else:
    print("Error detected at "+str(currln)+". Please use r or w!")
    break

if (len(currln_split) < 2):
    print("Error detected at "+str(currln)+". Please use correct format on file test.txt
(Example: 1r0 or 2r1,100).")
    break
```

```

if(trID not in self.trTbl.keys()): # If transaction ID not in transaction table
    trTS = tsCount # Assign trTS value with tsCount
    tsCount += 1
    self.trTbl[trID] = trTS

else:
    # Otherwise, Assign trTS value with transaction table value from trID
    trTS = self.trTbl[trID]

if(trID not in self.trIDAborted): # If transaction hasn't aborted
    # Execute read
    if (method == "r"):
        result = self.mvcc_processor.read(key, value, self.ObjectStorage, trTS)

    # Execute write
    if (method == "w"):
        result = self.mvcc_processor.write(key, value, self.ObjectStorage, trTS)

    # If result status is false, transaction will be aborting
    if (not result):
        self.trIDAborted.append(trID)

    # Add instruction to Aborted queue
    if (trID in self.trIDAborted):
        self.inQueueAborted.append(currIn)

else: # Otherwise
    # Move to inQueueAborted, if it has been aborted
    self.inQueueAborted.append(currIn)

print("-----")
print("Current Instruction: ", currIn)
print("Instruction Queue: ", self.inQueue)

```

```
print("Transaction Table: ", self.trTbl)
print("Aborted Transactions ID: ", self.trIDAborted)
print("Aborted instructions Queue: ", self.inQueueAborted)
print("Storage: \n")
self.ObjectStorage.showStorage()
```

File cc.py berisi suatu kelas yang berfungsi untuk melakukan concurrency control. Adapun kelas ini memiliki atribut-atributnya yaitu sebagai berikut.

- trTbl  
Deskripsi : Dictionary untuk menyimpan id transaksi dengan timestampnya.
- trIDAborted  
Deskripsi : Menyimpan id transaksi yang telah di-abort.
- inQueue  
Deskripsi : Menyimpan instruksi-instruksi transaksi.
- inQueueAborted  
Deskripsi : Menyimpan instruksi transaksi yang telah di-rollback.
- ObjectStorage  
Deskripsi : *Storage* dari *concurrency control*.
- mvcc\_processor  
Deskripsi : Processor pada MVCC

Untuk method dalam kelas cc yaitu sebagai berikut.

- \_\_init\_\_  
Deskripsi : Untuk melakukan inisialisasi concurrency control, seperti prosessor, variabel yang menyimpan instruksi, dan inisialisasi jumlah storage, serta memproses file transaksi supaya dapat disimpan instruksi-instruksi di dalamnya pada suatu variabel.
- execute  
Deskripsi : Untuk menjalankan transaksi, pada awalnya kita akan menyiapkan variabel tsCount untuk menghitung timestamp-timestamp dari instruksi-instruksi transaksi, kemudian selama masih ada transaksi yang masih dapat dijalankan ataupun adanya transaksi terabort yang masih dapat dijalankan, akan dilakukan perulangan untuk memproses instruksi-instruksi transaksi.

Adapun untuk prosesnya adalah sebagai berikut.

1. Pertama-pertama akan dilakukan inisialisasi transaksi yang pernah di abort apabila inQueue kosong dengan cara mengcopy isi dari inQueueAborted ke inQueue, lalu mereset inQueueAborted, trTbl, dan trIDAborted. Selanjutnya kita akan menghapus instruksi yang sedang diproses melalui .pop dan menyimpannya ke currIn, lalu menginisialisasi variabel-variabel seperti method,

trID, trTS, key, dan value. Setelah itu, akan diperiksa jenis instruksinya, apabila valid, program akan lanjut, dan sebaliknya.

2. Selanjutnya, akan diperiksa apakah transaction ID (trID) ada pada transaction table, jika ya, maka akan diassign nilai trTS dengan nilai transaction table dari trID. Bila belum ada, maka akan diassign nilai trTS dengan tsCount, lalu nilai tsCount di-*increment*, dan nilai transaction table untuk trID akan diisi dengan trTS.
3. Kemudian, akan dicek apabila transaksi telah diabort atau tidak, bila ya, maka akan ditambahkan ke inQueueAborted, bila belum, maka transaksi akan diproses sesuai dengan instruksinya (read/write) oleh prosessor MVCC. Pada pemrosesan instruksi transaksi, akan dikirim status mengenai instruksi transaksi tersebut, bila gagal, maka transaksi akan diabort dan akan ditambahkan ke inQueueAborted yang menampung transaksi gagal. Lalu, akan dicetak ke layar instruksi yang sedang diproses, antrian dari instruksi, tabel transaksi, ID transaksi yang dibatalkan, dan antrian transaksi yang dibatalkan.

storage.py

```
class Storage:
    data = []

    def __init__(self, length):
        """
        Initiate storage with 5 element of data
        such as key, val, writeTS, readTS, dan version
        """
        for i in range(length):
            self.data.append([i,0,0,0,0])

    def getLatestVersion(self,key):
        """
        Method to return new version of a element
        """
        version = 0
        for i in self.data:
            current_Key = i[0]
            current_Ver = i[4]
```

```

        if(current_Key == key):
            if(current_Ver > version):
                version = current_Ver
        return version

def getHighestwriteTS(self,timestamp,key):
    """
        Method to find idx that denote the version of Q
        whose write timestamp is the largest write timestamp
        less than or equal to TS(Ti)
    """
    index = 0
    wts = self.data[0][3]
    for i in self.data:
        current_Key = i[0]
        current_WTS = i[3]
        if(key==current_Key):
            if(current_WTS>wts):
                if(current_WTS<=timestamp): # Qi terdeteksi
                    index = i
                    wts = current_WTS
    return index

def addElement(self,key,val,rTS,wTS,version):
    """
        Method to add new element to data
    """
    self.data.append([key,val,rTS,wTS,version])

def showStorage(self):
    """
        Method to show storage information
    """

```

```
""  
for i in self.data:  
    print("-----")  
    print("Read R-TS: ",i[2])  
    print("Read W-TS: ",i[3])  
    print("Version: ",i[4])  
    print("Key: ",i[0])  
    print("Value: ",i[1])  
    print()
```

File `storage.py` berisikan kelas `storage` yang terdiri dari atribut `data` yang memiliki 5 buah elemen yaitu `key`, `val`, `readTS`, `writeTS`, dan `version` yang masing-masing direpresentasikan dengan index array 0 sampai dengan 4. Sementara itu, untuk method pada kelas tersebut dapat disajikan sebagai berikut.

- `__init__`

Deskripsi : Dilakukan inisialisasi pada atribut `data` pada `storage`, dengan suatu array yang panjangnya 5, yang mana isinya terdiri dari `i` dimana `i` sebagai index yang diincrement dari 0 sampai dengan `length` yang kemudian diassign sebagai `key`-nya, kemudian untuk `val`, `readTS`, `writeTS`, dan `version` diassign dengan nilai 0.

- `getLatestVersion`

Deskripsi : Dikembalikan versi terbaru dari suatu elemen. Pada method ini, akan ditelusuri semua objek dari array, hanya saja karena ditelusuri semuanya, dapat dibuat sebuah `key` agar dapat diketahui bahwa objek tersebut adalah objek yang dicari atau tidak.

- `getHighestwriteTS`

Deskripsi : Dikembalikan index dari `Qk`, dengan `Qk` adalah versi dari `Q` yang `write timestamp` terbesar yang kurang dari atau sama dengan `timestamp` dari transaksi yang memprosesnya. Seperti biasa, karena akan ditelusuri semua objek pada `data`, maka diperlukan `key` untuk mengecek apakah objek tersebut objek yang kita cari. Pada setiap objek, akan diperiksa apakah `keynya (current_Key)` sama dengan `key` yang diminta, dan apakah `write timestampnya` lebih besar daripada `write timestamp` yang sekarang dianggap maksimal, dan apakah `write timestampnya` lebih kecil atau kurang dari sama dengan `timestamp` dari transaksi yang sedang memprosesnya.

## BAB III

### Eksplorasi Recovery

#### 3.1. Write-Ahead Log

PostgreSQL memiliki mekanisme *recovery* menggunakan *transaction log* yang dinamakan *write-ahead log*. Semua aktivitas yang mengubah suatu data akan dicatat di dalam *log* tersebut. Dalam mekanisme ini, semua perubahan akan dicatat terlebih dahulu di dalam *log* sebelum perubahan benar-benar dilakukan. Dengan ini, jika terjadi kegagalan apapun pada proses transaksi, maka Postgres dapat melakukan *recovery* dengan melihat kembali *transaction log* sesaat sebelum sistem mengalami kegagalan.

#### 3.2. Continuous Archiving

Hasil berkas *log* dari mekanisme *write-ahead log* dapat dilakukan mekanisme tambahan dengan mengarsipkan *log* secara terus menerus ke tempat penyimpanan yang lebih stabil. Hal ini disebut juga *continuous archiving*. Dalam mekanisme ini, berkas *log* akan disalin dan disimpan ke suatu tempat yang *persistent*. Hasil salinan ini dapat digunakan untuk melakukan mekanisme selanjutnya, yaitu *point-in-time recovery*.

#### 3.3. Point-in-Time Recovery

Seperti namanya, *point-in-time recovery* akan mengembalikan basis data ke keadaan sebelumnya di waktu tertentu di masa lalu. Pada PostgreSQL, aktivitas ini akan membuat sebuah *timeline* baru dari *database* tersebut. Sebuah *timeline* menunjukkan satu buah kemungkinan jalannya *timeline*, ditandai dengan berkas *log* di *timeline* tersebut. Fitur *timeline* ini digunakan apabila setelah melakukan proses *recovery* pengguna hendak melakukan *rollback* proses *recovery* tersebut dan kembali ke versi basis data yang belum melakukan *recovery*.

#### 3.4. Simulasi Recovery

Metode *recovery* yang digunakan adalah *Point-in-time recovery* (PITR). Sebelum melakukan *recovery*, perlu dilakukan konfigurasi untuk dapat melakukan *continuous archiving* pada berkas `postgresql.conf` dan membuat direktori penyimpanan arsip. Berikut adalah langkah-langkahnya.

1. Buat direktori baru untuk menyimpan arsip berkas *log*. Misal nama direktorinya adalah `dbarchive`.
2. Sunting berkas konfigurasi postgres `postgresql.conf` sebagai berikut.

`wal_level = replica`

```
archive_mode = on
archive_command = 'test ! -f /path/to/dbarchive/%f && cp %p /path/to/dbarchive/%f'
restore_command = 'cp /path/to/dbarchive/%f%p'
```

Jangan lupa untuk mengganti /path/to/dbarchive ke direktori dbarchive.

3. Buat direktori baru untuk menyimpan *backup*. Misal nama direktorinya adalah dbbackup. Kemudian lakukan *backup* dengan perintah berikut.

```
pg_basebackup -U postgres --progress -D path/to/dbbackup
```

4. Berikut adalah database yang ada pada sistem sebelum dilakukan simulasi.

```
postgres=# \dt
              List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | account        | table | postgres
 public | borrower       | table | postgres
 public | branch         | table | postgres
 public | customer       | table | postgres
 public | depositor      | table | postgres
 public | loan           | table | postgres
 public | sekolah_top1000_utbk | table | postgres
(7 rows)
```

Berikut adalah data pada tabel branch.

```
postgres=# SELECT * FROM branch;
 branch_name | branch_city | assets
-----+-----+-----
 Downtown   | Brooklyn   | 900000
 Redwood    | Palo Alto  | 2100000
 Perryridge | Horseneck  | 1700000
 Mianus     | Horseneck  | 400200
 Round Hill | Horseneck  | 8000000
 Pownal     | Bennington | 400000
 North Town | Rye        | 3700000
 Central    | Rye        | 400280
 Brighton   | Brooklyn   | 7000300
 West Coast | Vice City  | 100300
(10 rows)
```

5. Misalnya akan ditambahkan dua data baru ke tabel branch. Namun sebelum menambahkan data, akan diganti *WAL log* yang akan digunakan sehingga kita dapat melakukan *recovery* ke keadaan dimana data belum ditambahkan (0/5000160).



```
postgres=# SELECT pg_switch_wal();
pg_switch_wal
-----
0/5000160
(1 row)
```

```
postgres=# SELECT * FROM branch;
 branch_name | branch_city | assets
-----+-----+-----
Downtown    | Brooklyn   | 900000
Redwood     | Palo Alto  | 2100000
Perryridge  | Horseneck  | 1700000
Mianus      | Horseneck  | 400200
Round Hill  | Horseneck  | 8000000
Pownal      | Bennington | 400000
North Town  | Rye        | 3700000
Central     | Rye        | 400280
Brighton    | Brooklyn   | 7000300
West Coast  | Vice City  | 100300
Contoh1     | Bandung    | 250
Contoh2     | Bandung    | 10
(12 rows)
```

```
postgres=# SELECT pg_switch_wal();
pg_switch_wal
-----
0/6000740
(1 row)
```

6. Ubah `recover_target_lsn` pada `postgresql.conf` menjadi `'0/5000160'` dan matikan servis PostgreSQL.
7. Akan dilakukan simulasi kegagalan sistem dengan menghapus semua data pada direktori data PostgreSQL. Sebelum menghapus, salin folder `pg_wal` ke tempat yang aman terlebih dahulu.
8. Karena folder data pada direktori PostgreSQL sekarang sudah kosong, maka servis PostgreSQL tidak akan dapat berjalan. Untuk mengembalikannya seperti semula, salin semua data dari direktori `dbbackup` ke folder data PostgreSQL.
9. Salin folder `pg_wal` yang sebelumnya sudah disimpan dan timpa folder `pg_wal` di folder data PostgreSQL.

10. Buat berkas *signal* untuk memberi tahu postgres untuk memulai dalam mode *recovery*.  
Buat berkas kosong *recovery.signal* pada folder data PostgreSQL.
11. Mulai kembali servis PostgreSQL dan basis data akan kembali ke keadaan semula di mana tabel *branch* hanya memiliki sepuluh data.

```
postgres=# SELECT * FROM branch;
 branch_name | branch_city | assets
-----+-----+-----
Downtown    | Brooklyn   | 900000
Redwood     | Palo Alto  | 2100000
Perryridge  | Horseneck  | 1700000
Mianus      | Horseneck  | 400200
Round Hill  | Horseneck  | 8000000
Pownal      | Bennington | 400000
North Town  | Rye        | 3700000
Central     | Rye        | 400280
Brighton    | Brooklyn   | 7000300
West Coast  | Vice City  | 100300
(10 rows)
```

## **BAB IV**

### **Kesimpulan dan Saran**

#### **4.1. Kesimpulan**

Ketika beberapa transaksi dijalankan secara bersamaan atau konkuren di dalam database, terdapat kemungkinan bahwa konsistensi data tidak lagi dipertahankan. Sistem memiliki kepentingan untuk mengontrol interaksi antara transaksi yang bersamaan. Kontrol ini dicapai melalui salah satu dari beberapa mekanisme yang dikenal sebagai *concurrency control*.

*Simple locking* merupakan himpunan *rules* yang menyatakan kapan transaksi dapat melakukan *lock* dan *unlock* terhadap setiap item data dalam database.

Di dalam proses OCC, dilakukan proses validasi, yakni metode *concurrency control* sesuai dengan kasus yang ditangani dan sebagian besar transaksi yang dilakukan adalah *read-only*, sehingga tingkat konflik di antara transaksi ini rendah. Waktu tetap yang unik dikaitkan dengan setiap transaksi dalam sistem. Urutan *serializability* ditentukan oleh *time* transaksi. Transaksi dalam skema ini tidak pernah tertunda. Namun, transaksi tersebut harus lulus tes validasi untuk dapat diselesaikan. Jika tidak lulus uji validasi, sistem mengembalikannya ke keadaan awal.

MVCC didasarkan pada pembuatan versi baru item data untuk setiap transaksi yang melakukan *write* pada item tersebut. Saat operasi *read* dijalankan, sistem memilih salah satu versi untuk dibaca. Skema ini memastikan bahwa versi yang akan dibaca selanjutnya dipilih dengan memastikan *serializability* berdasarkan *time*. Dalam MVCC, operasi *read* selalu berhasil.

DBMS dirancang sedemikian rupa agar dapat mengantisipasi adanya kegagalan sistem dengan mencoba untuk kembali ke keadaan semula sebelum terjadinya kegagalan. Dalam simulasi yang sudah dijelaskan, walaupun folder yang sangat penting pada PostgreSQL seperti folder data itu terhapus, PostgreSQL masih dapat melakukan *recovery* ke titik dimana kegagalan sistem belum terjadi.

#### **4.2. Saran**

Menurut kelompok kami, tidak banyak referensi terkait contoh kasus *concurrency control* dengan proses OCC. Bahkan, pada buku referensi dan *slide* yang digunakan dalam perkuliahan pun, hanya terdapat satu contoh saja. Sebaiknya, diberikan referensi proses transaksi yang dilakukan dalam kasus *concurrency control* baik dengan proses OCC, maupun proses lainnya.

Selain itu, sebaiknya terdapat *expected output* atau *test case* di dalam spesifikasi dalam hal pembuatan simulasi proses *concurrency control* supaya simulasi yang dilakukan dapat dipastikan kebenarannya.

Untuk melakukan simulasi proses *recovery* pada basis data, dipastikan harus mengikuti langkah-langkahnya dengan benar dan teliti. Jika proses tidak diikuti dengan baik dan teliti, kesalahan akan sangat mungkin terjadi, seperti servis PostgreSQL yang tidak dapat dimulai kembali setelah mencoba melakukan *backup*.

## LAMPIRAN

### Link Repository Github

[https://github.com/hanafathiyah/IF3140\\_K02\\_G11](https://github.com/hanafathiyah/IF3140_K02_G11)

### Pembagian Tugas

NIM	Nama	Tugas
13520014	Muhammad Helmi Hibatullah	Laporan dan Pengujian Keseluruhan Bab III, Kesimpulan dan Saran
13520047	Hana Fathiyah	Pengujian Bab I, Source code OCC + screenshot dan analisis, Kesimpulan dan Saran
13520080	Jason Kanggara	Source code Simple Locking + screenshot dan analisis
13520137	Muhammad Gilang Ramadhan	Teori Bab I dan Bab II, Source code MVCC + screenshot dan analisis

## REFERENSI

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition

- Chapter 18: Concurrency Control