IF3140 MANAJEMEN BASIS DATA

MEKANISME CONCURRENCY CONTROL DAN RECOVERY

K01 Kelompok 03

Anggota :

Arleen Chrysantha Gunardi	13521059
Yobel Dean Christopher	13521067
Saddam Annais Shaquille	13521121
Reza Pahlevi Ubaidillah	13521165

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

Daftar Tabel

Daftar Gambar

1. Eksplorasi Transaction Isolation

Transaction Isolation adalah suatu konsep basis data untuk mengatur tingkat isolasi antar transaksi-transaksi yang sedang berjalan bersamaan. Hal ini digunakan untuk menjaga *integrity* dan *consistency* dalam basis data. Jika banyak transaksi berjalan dalam satu waktu, kemungkinan terjadinya *conflict* menjadi meningkat. Oleh karena itu, basis data menggunakan mekanisme isolasi untuk mengontrol bagaumana transaksi-transaksi tersebut dapat saling berinteraksi sehingga *integrity* dan *consistency* dapat terjaga.

SQL menyediakan 4 level isolasi, yaitu *serializable*, *repeatable read*, *read committed*, dan *read uncommitted*. Level isolasi yang paling ketat adalah *serializable* yang memastikan bahwa hasil dari setiap transaksi *concurrent* akan memiliki hasil yang sama seperti menjalankan ketika tidak *concurrent*. Tiga level isolasi lainnya didefinisikan berdasarkan fenomena-fenomena yang tidak mungkin terjadi untuk tiap level tersebut. Fenomena-fenomena yang tidak diperbolehkan untuk tiap level isolasi adalah berikut:

1. *Dirty read*
   Sebuah transaksi membaca data yang ditulis oleh transaksi yang belum melakukan *commit*.
2. *Non repeatable read*
   Sebuah transaksi membaca ulang data yang sebelumnya sudah dibaca dan menemukan bahwa data tersebut telah diubah oleh transaksi lain.
3. *Phantom read*
   Sebuah transaksi menjalankan ulang suatu query pencarian dan menemukan bahwa hasil yang memenuhi query pencarian tersebut mengembalikan baris data yang berbeda karena telah diubah oleh transaksi lain yang baru melakukan *commit*.
4. *Serialization anomaly*
   Hasil dari banyak transaksi yang *concurrent* tidak konsisten dengan semua urutan yang mungkin ketika dijalankan satu per satu.

Berikut merupakan mapping antara level tiap isolasi yang ada dan juga fenomena-fenomena apa saja yang mungkin terjadi dan tidak mungkin terjadi:

Tabel 1.1. Isolasi - Fenomena di tiap Level Isolasi

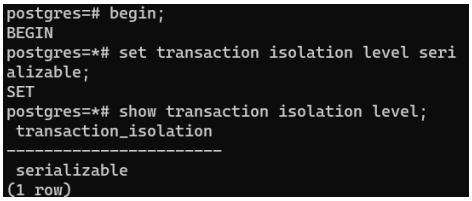
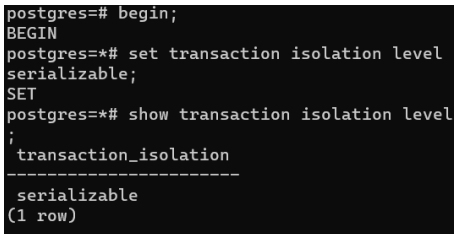
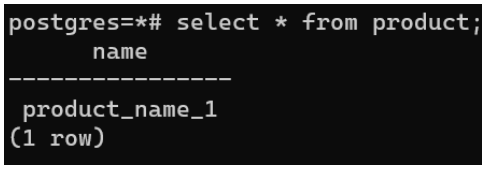
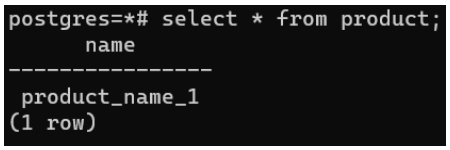
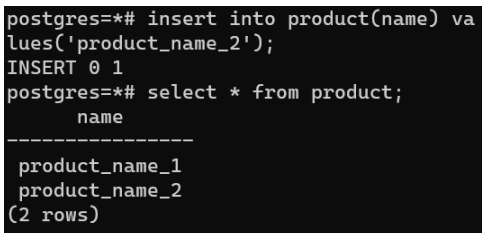
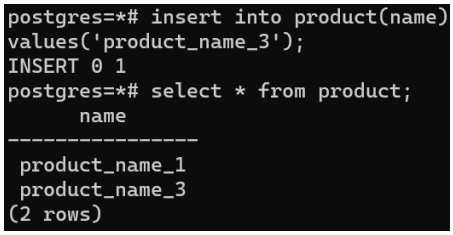
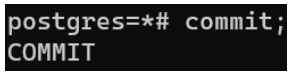
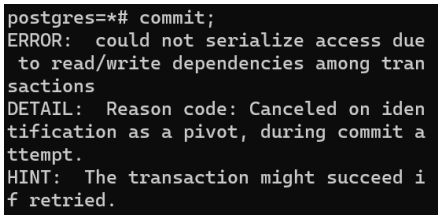
	Dirty Read	*Non repeatable read*	*Phantom read*	*Serialization anomaly*
Serializable	Tidak mungkin	Tidak mungkin	Tidak mungkin	Tidak mungkin
Repeatable read	Tidak mungkin	Tidak mungkin	Mungkin	Mungkin
Read committed	Tidak mungkin	Mungkin	Mungkin	Mungkin
Read uncommitted	Mungkin	Mungkin	Mungkin	Mungkin

a. Serializable

Pada level isolasi Serializable, basis data memastikan bahwa transaksi berjalan seperti mereka dieksekusi secara serial atau satu per satu. Oleh karena itu, level isolasi ini memastikan tidak mungkin terjadinya *dirty read*, *non repeatable read*, *phantom read*, dan *serialization anomaly*. Cara kerja dari level isolasi ini mirip dengan level isolasi *repeatable read*. Hal yang menjadi perbedaan antara level isolasi ini dengan level isolasi *repeatable read* adalah level ini memonitor apakah terjadi *serialization anomaly* atau tidak.

Karena suatu transaksi hanya dapat membaca data jika data tersebut telah dilakukan *commit* oleh transaksi lain, level isolasi ini dapat mencegah *dirty read*. Suatu transaksi tidak akan berjalan jika terdapat transaksi lain yang ingin mengubah suatu data sehingga dapat mencegah non repeatable read. Suatu transaksi *insert* atau *delete* tidak akan berjalan ketika terdapat transaksi lain yang menggunakan data tersebut sehingga *phantom read* dapat dicegah. Serialization anomaly juga tidak akan dicegah karena level isolasi ini memiliki memonitor apakah akan terjadi *serialization anomaly* atau tidak.

Simulasi

T1	T2	Keterangan
		T1 memulai transaksi dan set menjadi level isolasi serializable
		T2 memulai transaksi dan set menjadi level isolasi serializable
		T1 membaca relasi product
		T2 membaca relasi product
		T1 menambah data di relasi product kemudian membaca relasi lagi
		T2 menambah data di relasi product kemudian membaca relasi lagi
		T1 melakukan commit
		T2 melakukan commit Namun gagal

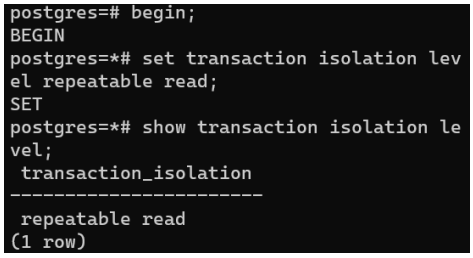
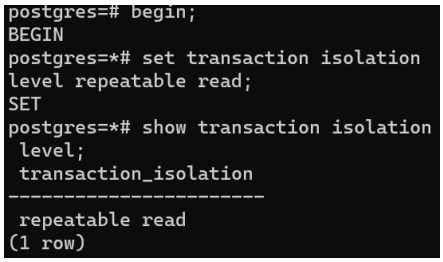
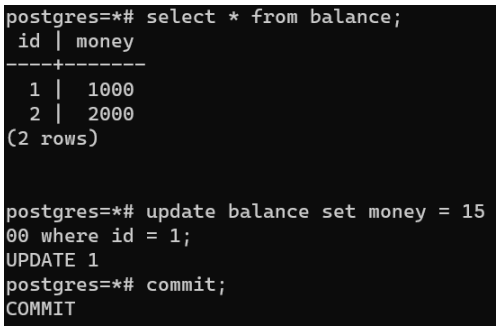
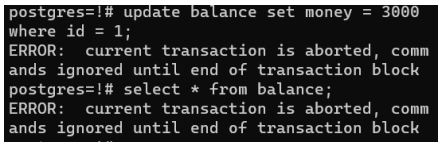
Pada simulasi tersebut terlihat bahwa level isolasi serializable tidak memperbolehkan insert data baru ke relasi tersebut (di *repeatable read* diperbolehkan). Hal ini dapat berguna untuk menghindari fenomena-fenomena *dirty read*, *non repeatable read*, *phantom read*, dan *serialization anomaly*.

b. Repeatable Read

Pada level isolasi Repeatable Read, ketika sebuah transaksi ingin membaca suatu data, transaksi tersebut akan membuat *snapshot* dari data tersebut dan menggunakan *snapshot* ini sepanjang transaksi berlangsung. *Snapshot* ini menjaga supaya tiap row baris data tidak berubah selama transaksi berlangsung. Akan tetapi, perubahan jumlah baris data tersebut tetap dapat dilakukan oleh transaksi lain dan terbaca oleh transaksi yang telah berjalan. *Snapshot* tersebut menjaga kekonsistenan data selama transaksi berlangsung sehingga level isolasi ini akan menghindari *non repeatable read* akan tetapi tidak menghindari *phantom read*.

Jika terdapat suatu transaksi lain yang melakukan perubahan pada data yang sama, transaksi ini tidak akan melihat perubahan tersebut hingga selesai dan melakukan *commit*. Jika suatu transaksi lain sedang menggunakan data sama seperti yang sedang digunakan juga oleh transaksi saat ini, transaksi saat ini akan menunggu sampai transaksi tersebut selesai. Jika transaksi lain melakukan *rollback* atau *commit* tetapi tidak ada data yang berubah (pada data yang digunakan oleh transaksi saat ini), transaksi saat ini dapat melanjutkan transaksinya. Akan tetapi, jika *commit* dari transaksi lain merubah data yang digunakan juga oleh transaksi saat ini, transaksi saat ini akan melakukan *roll back.*

Simulasi

T1	T2	Keterangan
		T1 memulai transaksi dan set menjadi level isolasi repeatable read
		T2 memulai transaksi dan set menjadi level isolasi repeatable read
		T1 membaca relasi balance dan melakukan update dengan mengubah salah satu data
		T2 melakukan update dengan mengubah data yang sama seperti T1

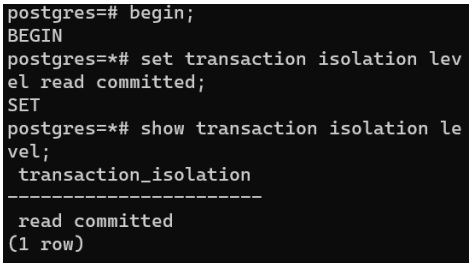
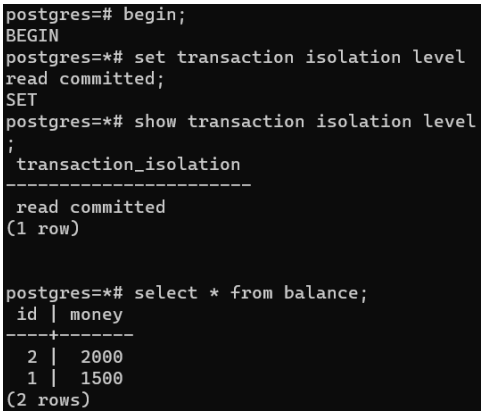
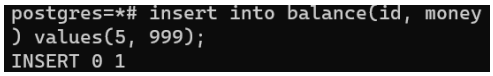
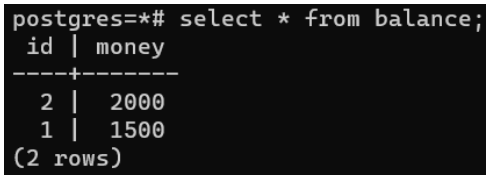
Pada simulasi tersebut terlihat bahwa level isolasi repeatable read tidak memperbolehkan merubah data yang sama (di *read committed* diperbolehkan). Hal ini dapat berguna untuk menghindari fenomena-fenomena *dirty read* dan *non repeatable read*.

c. Read Committed

Tingkat isolasi ini menjamin bahwa setiap *read* yang dilakukan oleh transaksi hanya akan menggunakan data yang telah di-*commit* oleh transaksi lain yang telah selesai sebelumnya. Hal ini mengakibatkan transaksi tidak akan melihat perubahan yang dilakukan oleh transaksi lain yang belum dikonfirmasi sehingga mencegah *dirty read*.

Akan tetapi, pada level isolasi *read committed*, data yang telah dibaca oleh transaksi saat ini tetap dapat mengalami perubahan oleh transaksi lain yang melakukan *commit*. Oleh karena itu, level isolasi ini masih ada kemungkinan untuk *non repeatable read* dan *phantom read.*

Simulasi

T1	T2	Keterangan
		T1 memulai transaksi dan set menjadi level isolasi read comitted
		T1 memulai transaksi dan set menjadi level isolasi read comitted dan membaca relasi balance
		T1 memasukkan data baru ke dalam relasi balance;
		Pembacaan relasi balance tidak ada yang berubah

```
postgres=*# commit;
COMMIT
```

T1 melakukan commit

```
postgres=*# select * from balance;
 id | money
----+-------
  2 |  2000
  1 |  1500
  5 |   999
(3 rows)
```

Pembacaan di T2 berubah dan sekarang berisi data dari T1

Pada simulasi tersebut terlihat bahwa level isolasi read committed tidak memperbolehkan *dirty read*. Data yang belum di-*commit* oleh T1 tidak terlihat di T2. Akan tetapi, setelah di-*commit,* data tersebut dapat dilihat oleh T2.

d. Read Uncommitted

Tingkat isolasi ini memungkinkan transaksi untuk membaca perubahan yang belum di-commit (*dirty read*) yang dilakukan oleh transaksi lain. Pada tingkat isolasi ini, tidak ada mekanisme yang mencegah transaksi melihat perubahan data yang belum dikonfirmasi, yang berarti transaksi dapat menggunakan data yang tidak konsisten.

2. Implementasi Concurrency Control Protocol

a. Two-Phase Locking (2PL)

Two-Phase Locking merupakan protokol *concurrency control* yang memastikan *conflict-serializable schedules*. Algoritma ini bertujuan untuk menjamin konsistensi dan isolasi dari setiap transaksi yang berjalan pada suatu DBMS. 2PL terdiri dari dua fase, yaitu *growing phase* dan *shrinking phase*.

Pada *Two-Phase Locking*, terdapat dua jenis *lock*, yaitu *shared lock* dan *exclusive lock*. DBMS membuat *shared lock* ketika membaca suatu *item* data pada suatu transaksi. *Shared lock* dapat dipegang oleh lebih dari satu transaksi. Berbeda dengan *shared lock*, *exclusive lock* dibuat ketika DBMS menuliskan *item* data dengan nilai baru. *Exclusive lock* untuk suatu *item* hanya dapat dibuat ketika tidak ada transaksi lain yang memegang *lock* terhadap *item* tersebut. *Shared lock* bisa di-*upgrade* secara otomatis menjadi *exclusive lock* pada DBMS dengan *automatic acquisition of locks* ketika suatu transaksi memiliki *shared lock* untuk suatu *item* kemudian dilakukan *write* pada transaksi dan *item* yang sama (dengan catatan *item* tidak dipegang oleh *lock* pada transaksi lain). Jika suatu transaksi sedang memegang *lock* untuk suatu *item*, maka instruksi terhadap *item* tersebut pada transaksi lain harus menunggu terlebih dahulu hingga *lock* dilepaskan. *Lock* dilepaskan ketika suatu transaksi melakukan *commit*.

Selain itu, *Two-Phase Locking* juga memiliki mekanisme untuk mencegah terjadinya *deadlock*. Pencegahan *deadlock* pada simulasi ini dilakukan dengan *wound-wait*. Pada skema *wound-wait*, transaksi yang lebih tua (*older*) memaksa *rollback* (*wound*) transaksi yang lebih muda (*younger*). Transaksi yang lebih muda menunggu transaksi yang lebih tua. Dengan skema ini, *rollback* yang perlu dilakukan lebih sedikit jika dibandingkan dengan skema pencegahan *deadlock* lainnya, yaitu skema *wait-die*.

Di tugas besar ini, simulasi *Two-Phase Locking* (2PL) diimplementasikan dengan pemrograman sederhana dalam Python. Adapun repositori program dapat diakses pada tautan berikut ini.

i. Struktur Data Program Simulasi

Berikut merupakan struktur data dasar yang digunakan dalam program.

Tabel 2.1 Struktur Data Program Simulasi 2PL

```python
from enum import Enum

class InstructionType(Enum):
    READ = 'read'
    WRITE = 'write'
    COMMIT = 'commit'
    ABORT = 'abort'

class TransactionValidationBasedPhase(Enum):
    READ_AND_EXECUTION = 1
    VALIDATION = 2
    WRITE = 3
```

```python
from enums import InstructionType

class Instruction:
    def __init__(self, input_sequence):
        # input_sequence: Rx(A) / Wx(A) / Cx
        if (input_sequence[0]=='R'):
            self.type = 'read'
            self.transaction_id = input_sequence[1]
            self.item = input_sequence[3]
        elif (input_sequence[0]=='W'):
            self.type = 'write'
            self.transaction_id = input_sequence[1]
            self.item = input_sequence[3]
        elif (input_sequence[0]=='C'):
            self.type = 'commit'
            self.transaction_id = input_sequence[1]
            self.item = ''

    def __str__(self):
        result = ''
        if (self.type=='read'):
            result += (f"R{self.transaction_id}({self.item})")
        elif (self.type=='write'):
            result += (f"W{self.transaction_id}({self.item})")
        elif (self.type=='commit'):
```

```
            result += (f"C{self.transaction_id}")
        return result
```

```
from typing import List
from instruction import Instruction

class Transaction:
    def __init__(self, id: str, instructions:List[Instruction]) -> None:
        self.id:str = id
        self.instructions: List[Instruction] = instructions
```

```
from queue import Queue
from transaction import Transaction
from transaction_occ import TransactionOCC
from instruction import Instruction
from typing import List

class Schedule:
    def __init__ (self, input_sequence:str, type:str='base'):
        self.instructions = Queue() # list (queue) of instructions
        self.rollbacked_list = Queue() # list of transactions rollbacked
        self.logging = Queue() # list (queue) of final schedule
        # input_sequence: Rx(A);Wx(A);Cx;Rx(B);Cx
        input_list = input_sequence.split(";")
        temp_transactions = {}
        for input in input_list:
            input_instruction = Instruction(input)
            self.instructions.put(input_instruction)
            if(input_instruction.transaction_id not in
temp_transactions.keys()):

temp_transactions.update({input_instruction.transaction_id: []})

temp_transactions[input_instruction.transaction_id].append(input_instruc
tion)

        self.transactions = {}
        for tid, tins in temp_transactions.items():
            if(type == 'base'):
                transaction = Transaction(tid, tins)
```

```
            elif(type == 'occ'):
                transaction = TransactionOCC(tid, tins)
            self.transactions.update({tid:transaction})

    def add_transaction(self, transaction: Transaction) -> None:
        self.transactions.append(transaction)

    def add_instruction(self, instruction):
        self.instructions.put(instruction)

    def rollback(self, transaction_id):
        self.rollbacked_list.put(transaction_id)

    def execute_instruction(self):
        if(self.instructions.empty() and not
self.rollbacked_list.empty()):
            transaction =
self.transactions.get(self.rollbacked_list.get())
            for e in transaction.instructions:
                self.instructions.put(e)

        instruction = self.instructions.get()
        if (instruction.transaction_id in self.rollbacked_list.queue):
            return self.execute_instruction()

        print(instruction)
        self.logging.put(instruction)
        return instruction

    def get_logged_instructions(self, transaction_id):
        logged = []
        for instruction in list(self.logging.queue):
            if (instruction.transaction_id == transaction_id):
                logged.append(instruction)
        return logged

    def get_instructions(self, transaction_id):
        instructions = []
        for ins in list(self.instructions.queue):
```

```
            if (ins.transaction_id == transaction_id):
                instructions.append(ins)
        return instructions

    def remove_instructions(self, transaction_id):
        for _ in range(len(self.instructions.queue)):
            ins = self.instructions.get()
            if (ins.transaction_id != transaction_id):
                self.instructions.put(ins)

    def restore_instruction(self, instruction):
        # put instruction to the first element of self.instructions
        self.add_instruction(instruction)
        for _ in range(self.instructions.qsize()-1):
            curr = self.instructions.get()
            self.add_instruction(curr)

if __name__ == '__main__':
    schedule = Schedule('R1(A);R2(A)', 'occ')
    schedule.rollback('1')
    x = schedule.execute_instruction()
    y = schedule.execute_instruction()
```

ii. Algoritma *Two-Phase Locking*

Berikut merupakan implementasi dari algoritma *Two-Phase Locking* pada program simulasi.

Tabel 2.2 Algoritma *Two-Phase Locking*

```
from collections import deque
import re
from schedule import *

class TwoPhaseLock:
    def __init__(self, input_sequence):
        self.schedule = Schedule(input_sequence)
        self.locks = {} # {'instruction': instruction, 'locktype':
```

```
x_lock/s_lock}
        self.instruction_queue = deque([])
        self.waiting_list = {} # {'transaction_id': [list of
transactions waiting for the transaction]}
        for ins in list(self.schedule.instructions.queue):
            if (ins.transaction_id not in self.waiting_list):
                self.waiting_list[ins.transaction_id] = []
        self.executed = []

    def add_schedule_instruction(self, instruction):
        # apend instruction to schedule
        self.schedule.add_instruction(instruction)

    def execute_instruction(self):
        # execute current instruction
        return self.schedule.execute_instruction()

    def check_lock_avail(self, lock_type, item, transaction_id):
        # check if item can acquire x_lock on transaction_id
        # return true/false and transaction it waits for (-1 if true)
        for ins,lock in self.locks.items():
            if (lock_type == 'x_lock'):
                if (ins.item==item and
ins.transaction_id!=transaction_id):
                    # if there's another transaction that holds a lock
on the item, then x_lock is not available
                    return False, ins.transaction_id
            else: # s_lock
                if (lock=='x_lock' and ins.item==item and
ins.transaction_id!=transaction_id):
                    # if there's another transaction that holds an
x_lock on the item, then s_lock is not available
                    return False, ins.transaction_id
        return True, -1

    def acquire_lock(self, instruction):
        if (instruction.type=='read'):
            is_s_lock_avail = self.check_lock_avail('s_lock',
instruction.item, instruction.transaction_id)
```

```
            if (not is_s_lock_avail[0]):
                # lock not avail, add to queue and waiting list
                self.instruction_queue.append(instruction)
                if (instruction.transaction_id not in
self.waiting_list[is_s_lock_avail[1]]):

self.waiting_list[is_s_lock_avail[1]].append(instruction.transaction_id)
                return False
            # else, acquire lock
            self.locks[instruction] = 's_lock'

        elif (instruction.type=='write'):
            is_x_lock_avail = self.check_lock_avail('x_lock',
instruction.item, instruction.transaction_id)
            if (not is_x_lock_avail[0]):
                # lock not avail, add to queue and waiting list
                self.instruction_queue.append(instruction)
                if (instruction.transaction_id not in
self.waiting_list[is_x_lock_avail[1]]):

self.waiting_list[is_x_lock_avail[1]].append(instruction.transaction_id)
                return False
            # else, acquire lock
            self.locks[instruction] = 'x_lock'
            # automatic upgrade lock from s_lock to x_lock
            for ins,lock in self.locks.items():
                if (lock=='s_lock' and
ins.transaction_id==instruction.transaction_id and
ins.item==instruction.item):
                    self.release_lock(ins)
                    break
        return True

    def release_lock(self, instruction):
        # release lock
        self.locks.pop(instruction)

    def commit(self, instruction):
        # commit, if not available then add to queue
```

```
        if (not self.check_commit_avail(instruction.transaction_id)):
            self.add_to_queue(instruction)
            return False
        return True

    def check_commit_avail(self, transaction_id):
        # check if commit can be executed
        # commit can not be executed if there are instructions within
the same transaction waiting in the queue
        for ins in self.instruction_queue:
            if (ins.transaction_id == transaction_id):
                return False
        return True

    def add_to_queue(self, instruction):
        self.instruction_queue.append(instruction)

    def search_queue(self, transaction_id):
        # search if there's any instructions within transaction_id
waiting in the queue
        for ins in self.instruction_queue:
            if (ins.transaction_id == transaction_id):
                return True
        return False

    def remove_from_queue(self, transaction_id):
        # remove all instructions within transaction_id from the queue
        for _ in range(len(self.instruction_queue)):
            ins = self.instruction_queue.popleft()
            if (ins.transaction_id != transaction_id):
                self.add_to_queue(ins)

    def execute_queue(self):
        # execute instructions waiting in the queue
        instruction = self.instruction_queue.popleft()
        if (instruction.type=='read' or instruction.type=='write'):
            is_lock_acquired = self.acquire_lock(instruction)
            if (is_lock_acquired):
                print(f"Current instruction: {instruction}")
```

```
                print(f"Action\t\t:acquire {self.locks[instruction]} for
{instruction.item} on T{instruction.transaction_id}")
                self.executed.append(instruction)
            return is_lock_acquired, instruction

        elif (instruction.type=='commit'):
            is_commit = self.commit(instruction)
            if (is_commit):
                print(f"Current instruction: {instruction}")
                commit_instructions =
self.get_executed_instructions(instruction.transaction_id)
                cnt = 0
                print("Action\t\t:", end='')
                for ins in commit_instructions:
                    if (ins.type!='commit'):
                        if (ins in self.locks):
                            print('\t\t' if cnt>0 else '', end='')
                            print(f"release {self.locks[ins]} for
{ins.item} on T{ins.transaction_id}")
                            self.release_lock(ins)
                            cnt+=1
                self.executed.append(instruction)
            return is_commit, instruction

    def abort(self, transaction_id):
        # abort mechanism
        instructions =
self.schedule.get_logged_instructions(transaction_id)
        for ins in instructions:
            if (ins.type!='commit'):
                # release locks
                cnt = 0
                if (ins in self.locks):
                    print(f"\trelease {self.locks[ins]} for {ins.item}
on T{ins.transaction_id}")
                    self.release_lock(ins)
                    cnt+=1
            # remove instructions from executed instructions list
            if (ins in self.executed):
```

```
                self.executed.remove(ins)
        # remove waiting instructions from the queue
        self.remove_from_queue(transaction_id)
        # adding all instructions within the transactions to the queue
        transaction =
self.schedule.get_logged_instructions(transaction_id) # executed
instructions
        [self.add_to_queue(ins) for ins in transaction]
        upcoming_ins = self.schedule.get_instructions(transaction_id) #
upcoming instructions
        [self.add_to_queue(ins) for ins in upcoming_ins]
        self.schedule.remove_instructions(transaction_id)
        # clear the waiting list
        self.waiting_list[transaction_id] = []

    def get_executed_instructions(self, transaction_id):
        # returns all executed instructions within transaction_id
        result = []
        for ins in self.executed:
            if (ins.transaction_id == transaction_id):
                result.append(ins)
        return result

    def print_queue(self):
        print('Queue\t\t: [', end='')
        cnt = 0
        for ins in self.instruction_queue:
            print(ins, end='')
            print(', ' if cnt<len(self.instruction_queue)-1 else '',
end='')
            cnt+=1
        print(']')

    def print_locks(self):
        print('Locks\t\t: [', end='')
        cnt = 0
        for ins,lock in self.locks.items():
            print(f"{lock}{ins.transaction_id}({ins.item})", end='')
            print(', ' if cnt<len(self.locks)-1 else '', end='')
```

```
            cnt+=1
        print(']')

    def print_waiting_list(self):
        print('Waiting list\t: [', end='')
        cnt = 0
        for waited,transaction_list in self.waiting_list.items():
            if (len(transaction_list)>0):
                print(f"T{waited} is waited by ", end='')
                print(', '.join([f"T{i}" for i in transaction_list]),
end=';')
        print(']')

    def print_executed(self):
        print('FINAL SCHEDULE\t: ', end='')
        cnt = 0
        for ins in self.executed:
            print(ins, end=';')
            cnt+=1
        print()

    def run(self):
        while (not self.schedule.instructions.empty() or
len(self.instruction_queue)>0):
            if (not self.schedule.instructions.empty()):
                current_instruction = self.execute_instruction()
                print(f"Current instruction: {current_instruction}")

                # if there's any instructions within the same
transaction waiting in queue, then the current instruction also added to
the queue (waiting)
                if
(self.search_queue(current_instruction.transaction_id)):
                    self.add_to_queue(current_instruction)
                    print("Action\t\t:Waiting. Instruction added to
queue")
                    self.print_queue()
                    self.print_locks()
                    self.print_waiting_list()
```

```
                    self.print_executed()

print("=================================================================
=")
                else:
                    if (current_instruction.type=='read' or
current_instruction.type=='write'):
                        is_lock_acquired =
self.acquire_lock(current_instruction)
                        if (not is_lock_acquired):
                            print(f"Action\t\t:{'s_lock' if
current_instruction.type=='read' else 'x_lock'} not granted. Instruction
added to queue")
                            if
(len(self.waiting_list[current_instruction.transaction_id])>0):
                                # if there's any transaction waiting for
this transaction
                                for t_id in
self.waiting_list[current_instruction.transaction_id]:
                                    print(f"Action\t\t:Abort T{t_id}")
                                    self.abort(t_id)
                                    current_wound =
self.instruction_queue.popleft()

self.schedule.restore_instruction(current_wound)

                        else:
                            print(f"Action\t\t:acquire
{self.locks[current_instruction]} for {current_instruction.item} on
T{current_instruction.transaction_id}")
                            self.executed.append(current_instruction)
                        self.print_queue()
                        self.print_locks()
                        self.print_waiting_list()
                        self.print_executed()

print("=================================================================
=")
```

```
                    elif (current_instruction.type=='commit'):
                        is_commit = self.commit(current_instruction)
                        if (is_commit):
                            commit_instructions =
self.schedule.get_logged_instructions(current_instruction.transaction_id
)
                            cnt = 0
                            for ins in commit_instructions:
                                if (ins.type!='commit'):
                                    # release locks
                                    if (ins in self.locks):
                                        print(f"Action\t\t:release
{self.locks[ins]} for {ins.item} on T{ins.transaction_id}")
                                        self.release_lock(ins)
                                cnt+=1
                            # clear the waiting list

self.waiting_list[current_instruction.transaction_id] = []

                            self.executed.append(current_instruction)
                            self.print_queue()
                            self.print_locks()
                            self.print_waiting_list()
                            self.print_executed()

print("================================================================
=")
                            # if commited, then execute all queue
                            for _ in range
(len(self.instruction_queue)):
                                exec_queue = self.execute_queue()
                                # print(f"Current instruction:
{exec_queue[1]}")
                                self.print_queue()
                                self.print_locks()
                                self.print_waiting_list()
                                self.print_executed()

print("================================================================
```

```
=")
                        else:
                            print("Action\t\t:Not executed. Instruction
added to queue")

        self.print_executed()

if __name__ == '__main__':
    print('Two Phase Locking')

print("======================================================================
=")

    # receive input
    is_valid = False
    while (not is_valid):
        print("Input your schedule:")
        sequence = input()

print("======================================================================
=")
        is_valid =
re.search("^(?:[RW]\d+\([A-Z]\)|C\d+)(?:;(?:[RW]\d+\([A-Z]\)|C\d+))*$",
sequence)

    locking = TwoPhaseLock(sequence)

    locking.run()
```

iii. Percobaan 1

Percobaan pertama dilakukan dengan melakukan *input schedule* sebagai berikut:

R1(A);W2(A);W2(B);W3(B);W1(A);C1;C2;C3

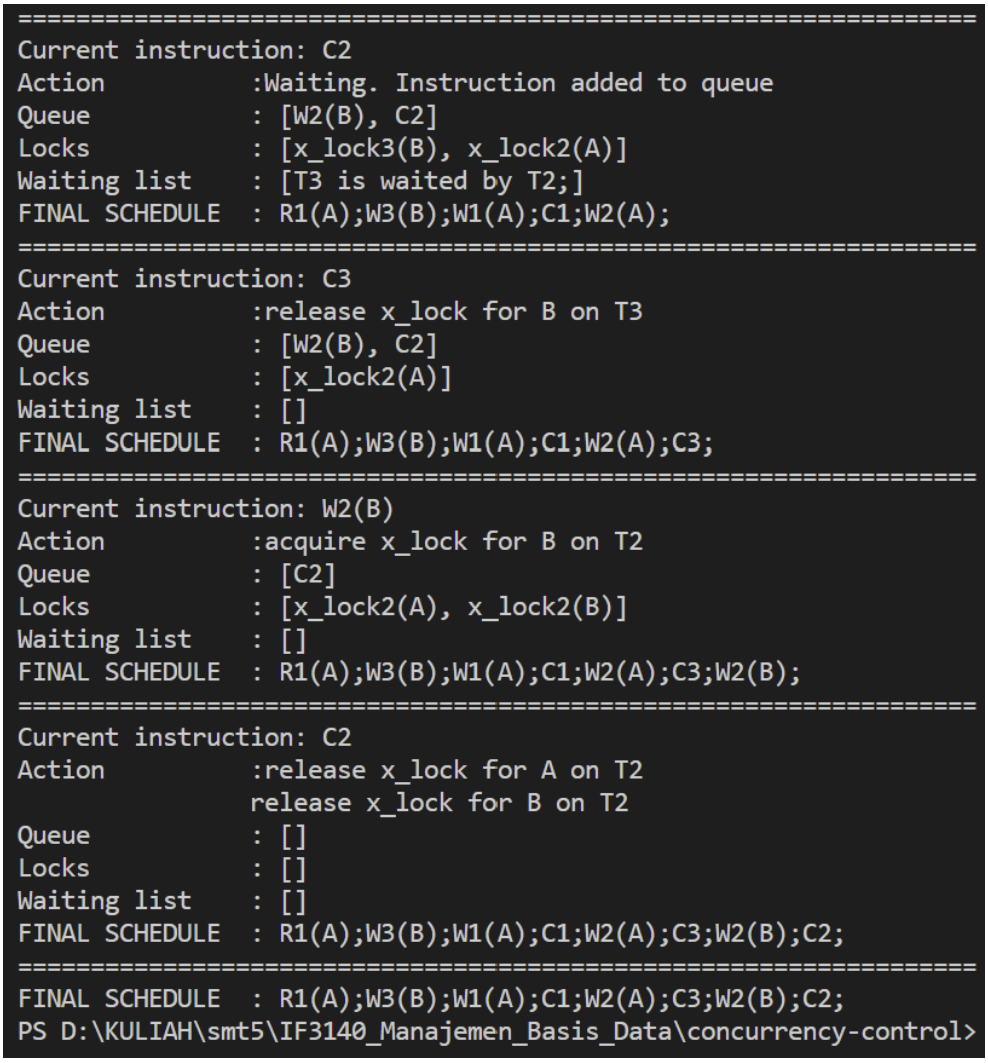
Berikut merupakan hasil dari percobaan 1 ketika dijalankan pada program simulasi *Two-Phase Locking*.

```
PS D:\KULIAH\smt5\IF3140_Manajemen_Basis_Data\concurrency-control> python src/two_phase_locking.py
Two Phase Locking
======================================================================
Input your schedule:
R1(A);W2(A);W2(B);W3(B);W1(A);C1;C2;C3
======================================================================
```

Gambar 2.1 Percobaan 1 *Two-Phase Locking*: *Input*

```
============================================================
Current instruction: R1(A)
Action          :acquire s_lock for A on T1
Queue           : []
Locks           : [s_lock1(A)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);
============================================================
Current instruction: W2(A)
Action          :x_lock not granted. Instruction added to queue
Queue           : [W2(A)]
Locks           : [s_lock1(A)]
Waiting list    : [T1 is waited by T2;]
FINAL SCHEDULE  : R1(A);
============================================================
Current instruction: W2(B)
Action          :Waiting. Instruction added to queue
Queue           : [W2(A), W2(B)]
Locks           : [s_lock1(A)]
Waiting list    : [T1 is waited by T2;]
FINAL SCHEDULE  : R1(A);
============================================================
Current instruction: W3(B)
Action          :acquire x_lock for B on T3
Queue           : [W2(A), W2(B)]
Locks           : [s_lock1(A), x_lock3(B)]
Waiting list    : [T1 is waited by T2;]
FINAL SCHEDULE  : R1(A);W3(B);
============================================================
Current instruction: W1(A)
Action          :acquire x_lock for A on T1
Queue           : [W2(A), W2(B)]
Locks           : [x_lock3(B), x_lock1(A)]
Waiting list    : [T1 is waited by T2;]
FINAL SCHEDULE  : R1(A);W3(B);W1(A);
============================================================
Current instruction: C1
Action          :release x_lock for A on T1
Queue           : [W2(A), W2(B)]
Locks           : [x_lock3(B)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);W3(B);W1(A);C1;
============================================================
Current instruction: W2(A)
Action          :acquire x_lock for A on T2
Queue           : [W2(B)]
Locks           : [x_lock3(B), x_lock2(A)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);W3(B);W1(A);C1;W2(A);
============================================================
```

Gambar 2.2 Percobaan 1 *Two-Phase Locking*: Hasil 1



Gambar 2.3 Percobaan 1 *Two-Phase Locking*: Hasil 2

Program akan mengeksekusi instruksi secara berurutan. Pertama-tama, R1(A) dieksekusi dan *shared lock* untuk A pada T1 diperoleh. Selanjutnya, W2(A) dieksekusi. Namun, karena terdapat T1 yang sedang memegang *lock* terhadap A, maka instruksi W2(A) tidak bisa dieksekusi dan *exclusive lock* untuk A pada T2 tidak bisa diperoleh, sehingga W2(A) menunggu *lock* A pada T1 dilepaskan (masuk ke *queue*). Dengan kata lain, T2 kini menunggu T1. Selanjutnya, W2(B) dieksekusi. Namun, karena terdapat instruksi pada T2 yang sedang menunggu T1, maka instruksi-instruksi berikutnya pada T2 tidak dapat dieksekusi dan harus turut menunggu (masuk ke *queue*).

Berikutnya, W3(B) dieksekusi dan *exclusive lock* diperoleh untuk B pada T3. Selanjutnya adalah W1(A). W1(A) dieksekusi dan *shared lock* untuk A pada T1 di-*upgrade* menjadi *exclusive lock*. Transaksi 1 melakukan *commit*, sehingga *lock* A pada T1 dilepas.

Karena *lock* untuk A pada T1 sudah dilepas, maka kini instruksi pada *queue* yang sedang menunggu T1 tadi bisa dieksekusi. W2(A) dieksekusi dan *exclusive lock* untuk A pada T2 diperoleh. Selanjutnya, C2 tidak dapat dieksekusi karena masih ada W2(B) yang menunggu pada queue. Program membaca instruksi berikutnya, yaitu C3, sehingga *lock* untuk B pada T3 dilepaskan. Selanjutnya, W2(B) dieksekusi dan *exclusive lock* untuk B pada T2 diperoleh. Terakhir, C2 dieksekusi, sehingga *lock* untuk A dan B pada T2 dilepaskan.

Jadi, *final schedule* untuk kasus ini adalah sebagai berikut.

R1(A);W3(B);W1(A);C1;W2(A);C3;W2(B);C2;

iv. Percobaan 2

Percobaan kedua dilakukan dengan melakukan *input schedule* sebagai berikut:

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

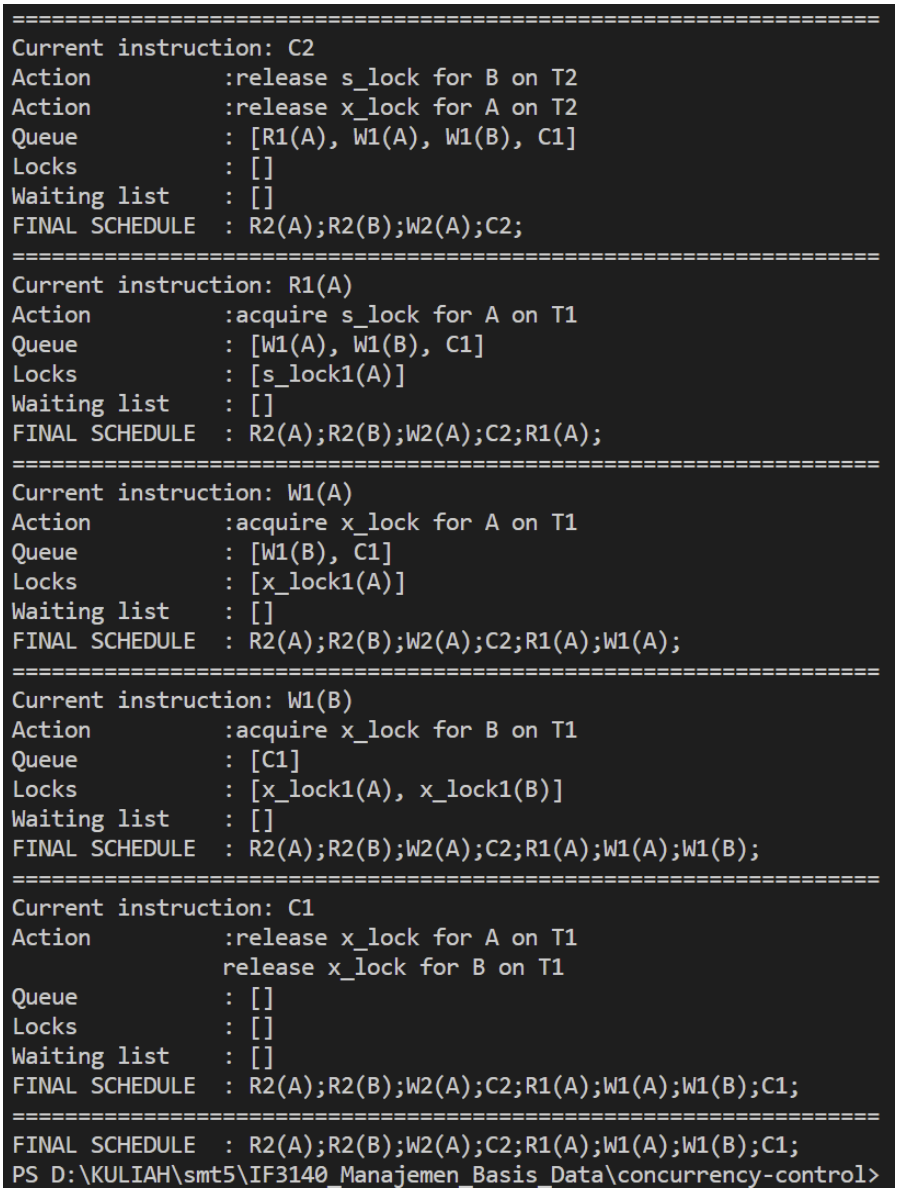
Berikut merupakan hasil dari percobaan 2 ketika dijalankan pada program simulasi *Two-Phase Locking*.

```
PS D:\KULIAH\smt5\IF3140_Manajemen_Basis_Data\concurrency-control> python src/two_phase_locking.py
Two Phase Locking
======================================================================
Input your schedule:
R1(A);R2(A);W1(A);R2(B);W2(A);W1(B);C1;C2
======================================================================
```

Gambar 2.4 Percobaan 2 *Two-Phase Locking*: *Input*

```
==================================================================
Current instruction: R1(A)
Action          :acquire s_lock for A on T1
Queue           : []
Locks           : [s_lock1(A)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);
==================================================================
Current instruction: R2(A)
Action          :acquire s_lock for A on T2
Queue           : []
Locks           : [s_lock1(A), s_lock2(A)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);R2(A);
==================================================================
Current instruction: W1(A)
Action          :x_lock not granted. Instruction added to queue
Queue           : [W1(A)]
Locks           : [s_lock1(A), s_lock2(A)]
Waiting list    : [T2 is waited by T1;]
FINAL SCHEDULE  : R1(A);R2(A);
==================================================================
Current instruction: R2(B)
Action          :acquire s_lock for B on T2
Queue           : [W1(A)]
Locks           : [s_lock1(A), s_lock2(A), s_lock2(B)]
Waiting list    : [T2 is waited by T1;]
FINAL SCHEDULE  : R1(A);R2(A);R2(B);
==================================================================
Current instruction: W2(A)
Action          :x_lock not granted. Instruction added to queue
Action          :Abort T1
        release s_lock for A on T1
Queue           : [R1(A), W1(A), W1(B), C1]
Locks           : [s_lock2(A), s_lock2(B)]
Waiting list    : [T2 is waited by T1;]
FINAL SCHEDULE  : R2(A);R2(B);
==================================================================
Current instruction: W2(A)
Action          :acquire x_lock for A on T2
Queue           : [R1(A), W1(A), W1(B), C1]
Locks           : [s_lock2(B), x_lock2(A)]
Waiting list    : [T2 is waited by T1;]
FINAL SCHEDULE  : R2(A);R2(B);W2(A);
==================================================================
```

Gambar 2.5 Percobaan 2 *Two-Phase Locking*: Hasil 1



Gambar 2.6 Percobaan 2 *Two-Phase Locking*: Hasil 2

Program akan mengeksekusi instruksi secara berurutan. Pertama-tama, R1(A) dieksekusi dan *shared lock* untuk A pada T1 diperoleh. R2(A) juga dieksekusi dan *shared lock* untuk A pada T2 diperoleh. Kemudian, ketika W1(A) hendak dieksekusi, *exclusive lock* gagal diperoleh karena terdapat *lock* untuk A pada transaksi lain (T2), sehingga W1(A) harus menunggu hingga *lock* A pada T2 dilepaskan (masuk ke *queue*). Dengan kata lain, T1 menunggu T2.

Selanjutnya, R2(B) dieksekusi dan *shared lock* untuk B pada T2 diperoleh. Kemudian, program akan mengeksekusi W2(A), namun gagal karena terdapat *lock* A pada T1, sehingga T2 harus menunggu T1. Ini merupakan kondisi *deadlock*, di mana dua transaksi saling menunggu. Oleh karena itu, mekanisme *wound-wait* dijalankan. Transaksi yang lebih tua, yaitu T2 memaksa (*wound*) T1 untuk melakukan *rollback*. T1 di-*abort* dan di-*rollback*, sehingga seluruh instruksi pada T1 dihapuskan dari *schedule* dan masuk ke *queue*. *Lock* A pada T1 dilepaskan.

W2(A) tetap dieksekusi dan berhasil dieksekusi karena kini tidak ada *lock* untuk A di transaksi lain. *Exclusive lock* untuk A pada T2 berhasil diperoleh. Kemudian, program membaca instruksi selanjutnya, yaitu C2. Dengan *commit*, *lock* untuk B dan A pada T2 dilepaskan.

Karena T2 sudah selesai dieksekusi seluruhnya, maka instruksi-instruksi T1 pada *queue* dapat dieksekusi. R1(A) dieksekusi kembali dan *shared lock* untuk A pada T1 diperoleh. W1(A) juga dieksekusi dengan meng-*upgrade shared lock* menjadi *exclusive lock* untuk A pada T1. Selanjutnya, W1(B) dieksekusi dan *exclusive lock* untuk B pada T1 diperoleh. Terakhir, *commit* pada T1 dilakukan dan *lock* untuk A dan B dilepaskan.

Dengan demikian, *final schedule* untuk percobaan ini adalah sebagai berikut.

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

v. Percobaan 3

Percobaan ketiga dilakukan dengan melakukan *input schedule* sebagai berikut:

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

Berikut merupakan hasil dari percobaan 3 ketika dijalankan pada program simulasi *Two-Phase Locking*.

```
PS D:\KULIAH\smt5\IF3140_Manajemen_Basis_Data\concurrency-control> python src/two_phase_locking.py
Two Phase Locking
======================================================================
Input your schedule:
R1(A);R1(B);W1(A);W3(B);R2(B);W1(C);R2(A);C1;C2;C3
======================================================================
```

Gambar 2.7 Percobaan 3 *Two-Phase Locking*: *Input*

```
==================================================================
Current instruction: R1(A)
Action          :acquire s_lock for A on T1
Queue           : []
Locks           : [s_lock1(A)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);
==================================================================
Current instruction: R1(B)
Action          :acquire s_lock for B on T1
Queue           : []
Locks           : [s_lock1(A), s_lock1(B)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);R1(B);
==================================================================
Current instruction: W1(A)
Action          :acquire x_lock for A on T1
Queue           : []
Locks           : [s_lock1(B), x_lock1(A)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);R1(B);W1(A);
==================================================================
Current instruction: W3(B)
Action          :x_lock not granted. Instruction added to queue
Queue           : [W3(B)]
Locks           : [s_lock1(B), x_lock1(A)]
Waiting list    : [T1 is waited by T3;]
FINAL SCHEDULE  : R1(A);R1(B);W1(A);
==================================================================
Current instruction: R2(B)
Action          :acquire s_lock for B on T2
Queue           : [W3(B)]
Locks           : [s_lock1(B), x_lock1(A), s_lock2(B)]
Waiting list    : [T1 is waited by T3;]
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);
==================================================================
Current instruction: W1(C)
Action          :acquire x_lock for C on T1
Queue           : [W3(B)]
Locks           : [s_lock1(B), x_lock1(A), s_lock2(B), x_lock1(C)]
Waiting list    : [T1 is waited by T3;]
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);
==================================================================
```

Gambar 2.8 Percobaan 3 *Two-Phase Locking*: Hasil 1

```
================================================================
Current instruction: R2(A)
Action          :s_lock not granted. Instruction added to queue
Queue           : [W3(B), R2(A)]
Locks           : [s_lock1(B), x_lock1(A), s_lock2(B), x_lock1(C)]
Waiting list    : [T1 is waited by T3, T2;]
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);
================================================================
Current instruction: C1
Action          :release s_lock for B on T1
Action          :release x_lock for A on T1
Action          :release x_lock for C on T1
Queue           : [W3(B), R2(A)]
Locks           : [s_lock2(B)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;
================================================================
Queue           : [R2(A), W3(B)]
Locks           : [s_lock2(B)]
Waiting list    : [T2 is waited by T3;]
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;
================================================================
Current instruction: R2(A)
Action          :acquire s_lock for A on T2
Queue           : [W3(B)]
Locks           : [s_lock2(B), s_lock2(A)]
Waiting list    : [T2 is waited by T3;]
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;R2(A);
================================================================
Current instruction: C2
Action          :release s_lock for B on T2
Action          :release s_lock for A on T2
Queue           : [W3(B)]
Locks           : []
Waiting list    : []
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;R2(A);C2;
================================================================
Current instruction: W3(B)
Action          :acquire x_lock for B on T3
Queue           : []
Locks           : [x_lock3(B)]
Waiting list    : []
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;R2(A);C2;W3(B);
================================================================
```

```
================================================================
Current instruction: C3
Action          :release x_lock for B on T3
Queue           : []
Locks           : []
Waiting list    : []
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;R2(A);C2;W3(B);C3;
================================================================
FINAL SCHEDULE  : R1(A);R1(B);W1(A);R2(B);W1(C);C1;R2(A);C2;W3(B);C3;
```

Gambar 2.9 Percobaan 3 *Two-Phase Locking*: Hasil 2

Program akan mengeksekusi instruksi secara berurutan. Pertama-tama, R1(A) dieksekusi dan *shared lock* untuk A pada T1 diperoleh. R1(B) juga dieksekusi dan *shared lock* untuk B pada T1 diperoleh. Selanjutnya, W1(A) dieksekusi dan *shared lock* A pada T1 di-*upgrade* menjadi *exclusive lock*.

Kemudian, W3(B) gagal dieksekusi karena terdapat *lock* untuk B pada transaksi lainnya (T1), sehingga W3(B) harus menunggu hingga *lock* B pada T1 dilepas (masuk ke *queue*). Dengan kata lain, T3 menunggu T1. Selanjutnya, R2(B) dieksekusi dan *shared lock* untuk B pada T2 diperoleh. W1(C) dieksekusi dan *exclusive lock* untuk C pada T1 diperoleh.

R2(A) gagal dieksekusi karena T1 memiliki *exclusive lock* untuk A, sehingga transaksi lain tidak dapat memperoleh *lock* untuk A. R2(A) harus menunggu hingga *lock* A pada T1 dilepas (masuk ke *queue*). Dengan kata lain, T2 menunggu T1. Jadi, T1 ditunggu oleh dua transaksi, yaitu T3 dan T2.

Commit pada T1 dilakukan, sehingga *lock* untuk B, A, dan C dilepaskan. Karena *lock* pada T1 sudah dilepaskan, maka instruksi-instruksi yang menunggu dapat dijalankan. R2(A) dijalankan dan *shared lock* untuk A pada T2 diperoleh. Kemudian, *commit* pada T2 dilakukan dan *lock* A pada T2 dilepaskan. W3(B) juga dieksekusi dan *exclusive lock* untuk B pada T3 diperoleh. Terakhir, *commit* pada T3 dieksekusi dan *lock* B pada T3 dilepaskan.

Dengan demikian, *final schedule* untuk percobaan ini adalah sebagai berikut.

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

b. Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control merupakan protokol *concurrency control* yang menjamin konsistensi dan isolasi dari setiap transaksi yang berjalan pada suatu DBMS. Sesuai dengan namanya, algoritma ini memiliki asumsi yang “optimis” bahwa semuanya terjadi dengan lancar dan minim konflik. Algoritma OCC menunda identifikasi dan resolusi konflik hingga mencapai akhir transaksi atau ketika perlu di-*resolve*.

OCC menggunakan protokol yang dilandaskan atas validasi (*validation-based protocol*). Algoritma ini terdiri dari tiga fase, yaitu *read and execution phase*, *validation phase*, dan *write phase*. Jadi, pengecekan hanya dilakukan di akhir transaksi dan transaksi dieksekusi secara

optimis. Jika dengan pengecekan validasi di akhir transaksi tidak ada pelanggaran, maka transaksi dieksekusi.

Pada OCC, setiap transaksi memiliki *timestamp*, yaitu *StartTS* ketika transaksi pertama dieksekusi, *ValidationTS* ketika memasuki fase validasi, dan *FinishTS* ketika transaksi selesai *write phase*. Terdapat aturan pada OCC jika TS(Ti) < TS(Tj), maka validasi dapat dikatakan sukses dan Tj dapat di-*commit* jika (*finishTS(Ti)* < *startTS(Tj))* atau (*startTS(Tj)* < *finishTS(Ti)* < *validationTS(Tj)* dan himpunan *item* data yang di-*write* Ti tidak beririsan dengan himpunan *item* data yang di-*read* oleh Tj*)*. Jika salah satu syarat tidak terpenuhi, maka validasi gagal dan Tj di-*abort*.

Di tugas besar ini, simulasi *Optimistic Concurrency Control* (OCC) diimplementasikan dengan pemrograman sederhana dalam Python. Adapun repositori program dapat diakses pada tautan berikut ini.

i. Struktur Data Program Simulasi

Struktur data program simulasi OCC sama dengan struktur data pada simulasi 2PL yang tercantum pada Tabel 2.1 dengan tambahan sebagai berikut.

Tabel 2.3 Struktur Data Program Simulasi OCC

```
from transaction import Transaction
from typing import List
from instruction import Instruction
from enums import TransactionValidationBasedPhase, InstructionType
# from time import perf_counter_ns

class TransactionOCC(Transaction):
    def __init__(self, id: str, instructions: List[Instruction]) ->
None:
        super().__init__(id, instructions)
        self.start_ts:int = None
        self.validation_ts:int = None
        self.finish_ts:int = None
        self.data_items_written = set()
        self.data_items_read = set()
        for instruction in self.instructions:
            if instruction.type == 'write':
```

```
                self.data_items_written.add(instruction.item)
            elif instruction.type == 'read':
                self.data_items_read.add(instruction.item)

    def start(self, current_timestamp) -> None:
        self.start_ts = current_timestamp

    def validate(self, transactions, current_timestamp) -> bool:
        self.validation_ts = current_timestamp
        res = True
        for ti in transactions.values():
            if ti.id == self.id:
                continue
            if ti.validation_ts and ti.validation_ts <
self.validation_ts:
                # TS(Ti) < TS(Tj)
                if ti.finish_ts and ti.finish_ts < self.start_ts:
                    # print('gala')
                    pass
                    # return False
                elif ti.finish_ts and self.start_ts < ti.finish_ts <
self.validation_ts and
len(ti.data_items_written.intersection(self.data_items_read))==0:
                    # print('galb')
                    pass
                    # return False
                else:
                    return False
        return True

    def commit(self, current_timestamp) -> None:
        self.finish_ts = current_timestamp

if __name__ == '__main__':
    ia = Instruction('R1(A)')
    ib = Instruction('R2(A)')
    tc = TransactionOCC('1', [ia, ib])
    print(tc.data_items_read)
```

ii. Algoritma *Optimistic Concurrency Control*

Berikut merupakan implementasi dari algoritma *Two-Phase Locking* pada program simulasi.

Tabel 2.4 Algoritma *Optimistic Concurrency Control*

```
from schedule import Schedule
from collections import deque
from queue import Queue
from instruction import Instruction
from transaction_occ import TransactionOCC

class OCC:
    def __init__(self, input_sequence: str) -> None:
        self.schedule: Schedule = Schedule(input_sequence, type='occ')


    def _transform_schedule(self) -> None:
        instructions = list(self.schedule.instructions.queue)
        writes = {}
        for ins in instructions:
            if ins.type == 'write':
                if(ins.transaction_id not in writes.keys()):
                    writes.update({ins.transaction_id: []})
                writes[ins.transaction_id].append(ins)

        for tid,arr in writes.items():
            for el in arr:
                instructions.remove(el)
        for tid, arr in writes.items():
            commit_idx = len(instructions)-1
            for i in range(len(instructions)):
                ins = instructions[i]
                if ins.type == 'commit' and ins.transaction_id==tid:
                    commit_idx = i
                    break
            instructions[:commit_idx] += arr

        self.schedule.instructions = Queue()
```

```
        self.schedule.instructions.queue = deque(instructions)

        temp_transactions = {}
        for ins in instructions:
            if(ins.transaction_id not in temp_transactions.keys()):
                temp_transactions.update({ins.transaction_id:[]})
            temp_transactions.get(ins.transaction_id).append(ins)

        for tid, tins in temp_transactions.items():
            self.schedule.transactions.update({tid:TransactionOCC(tid,
tins)})



    def get_transaction(self, transaction_id):
        for tid, t in self.schedule.transactions.items():
            if tid == transaction_id:
                tj = t
        return tj


    def run(self) -> None:
        processed = set()
        while not (self.schedule.instructions.empty() and
self.schedule.rollbacked_list.empty()):
            instruction = self.schedule.execute_instruction()
            tj = self.get_transaction(instruction.transaction_id)

            if tj.id not in processed:
                tj.start(self.schedule.current_timestamp-1)
            processed.add(tj.id)

            if instruction.type == 'write':
                valid = tj.validate(self.schedule.transactions,
self.schedule.current_timestamp-1)
                if valid:
                    while not (instruction.type == 'commit' and
instruction.transaction_id == tj.id):
                        instruction =
```

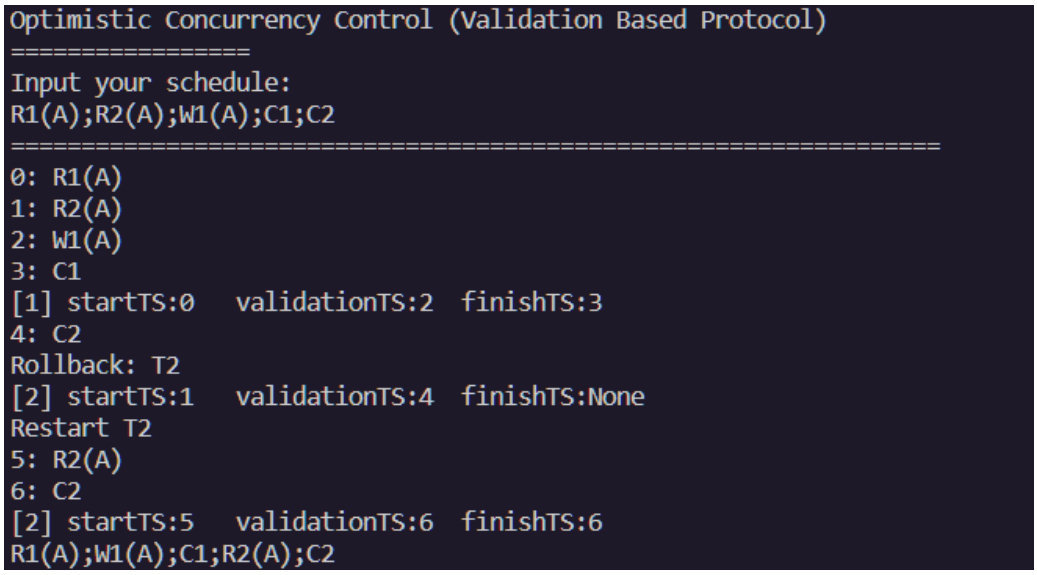
```
self.schedule.execute_instruction()
                    tj.commit(self.schedule.current_timestamp-1)
                    print(f'[{tj.id}]
startTS:{tj.start_ts}\tvalidationTS:{tj.validation_ts}\tfinishTS:{tj.fin
ish_ts}')
                else:
                    print(f'rollback: {tj.id}')
                    self.schedule.rollback(tj.id)
                    processed.remove(tj.id)
            elif instruction.type == 'commit':
                valid = tj.validate(self.schedule.transactions,
self.schedule.current_timestamp-1)
                print(f'[{tj.id}]
startTS:{tj.start_ts}\tvalidationTS:{tj.validation_ts}\tfinishTS:{tj.fin
ish_ts}')
                if valid:
                    tj.commit(self.schedule.current_timestamp-1)
                else:
                    print(f'rollback: {tj.id}')
                    self.schedule.rollback(tj.id)
                    processed.remove(tj.id)
```

iii. Percobaan 1

Percobaan pertama dilakukan dengan melakukan *input schedule* sebagai berikut:

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

Berikut merupakan hasil dari percobaan 1 ketika dijalankan pada program simulasi *Optimistic Concurrency Control*.



Gambar 2.10 Percobaan 1 *Optimistic Concurrency Control*

Program akan mengeksekusi R1 dengan data item A, dilanjutkan dengan R2 dengan data item A juga. Selanjutnya masuk ke fase validasi transaksi 1, di mana hasilnya ternyata valid dan W1 dieksekusi, kemudian transaksi 1 dilakukan commit. Berikutnya, transaksi 2 masuk ke fase validasi dan berusaha melakukan commit, tetapi tidak berhasil karena transaksi 1 commit pada fase read transaksi 2, sehingga transaksi 2 rollback. Akhirnya transaksi 2 bisa berjalan secara tidak konkuren dan berhasil commit.

Dengan demikian, *final schedule* untuk percobaan ini adalah sebagai berikut.

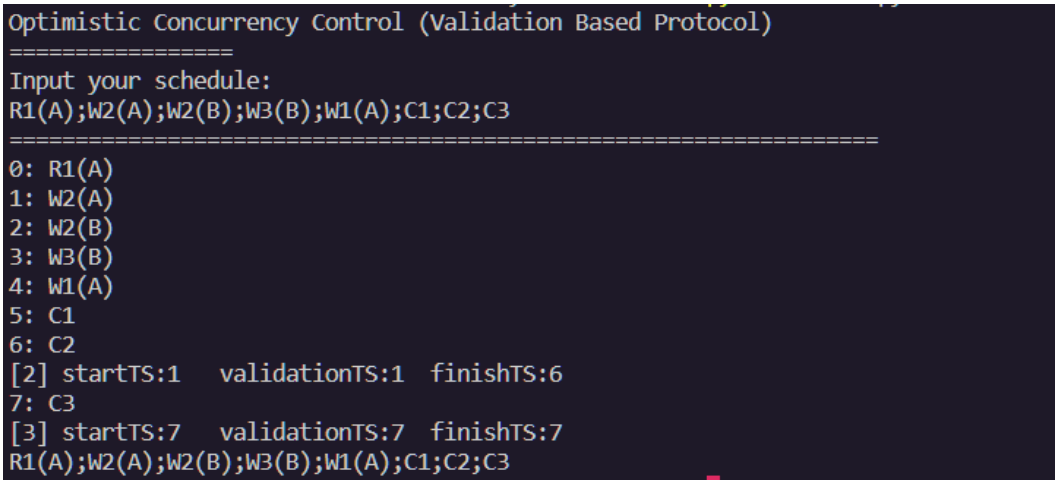
R1(A);W1(A);C1;R2(A);C2

iv. Percobaan 2

Percobaan pertama dilakukan dengan melakukan *input schedule* sebagai berikut:

R1(A);W2(A);W2(B);W3(B);W1(A);C1;C2;C3

Berikut merupakan hasil dari percobaan 2 ketika dijalankan pada program simulasi *Optimistic Concurrency Control*.



Gambar 2.11 Percobaan 2 *Optimistic Concurrency Control*

Pada schedule ini, transaksi 2 dan 3 melakukan *blind write*. Transaksi dimulai dengan membaca A pada transaksi 1. Kemudian, transaksi 2 bisa langsung saja mennulis data A dan B karena merupakan *blind write*. Hal yang sama juga untuk transaksi 3. Selanjutnya, transaksi 1 memasuki fase validasi dan validasinya diterima, sehingga bisa langsung dilakukan commit. Hal ini dilanjutkan oleh transaksi 2 dan 3 yang juga berhasil melakukan commit.

Dengan demikian, *final schedule* untuk percobaan ini adalah sebagai berikut.

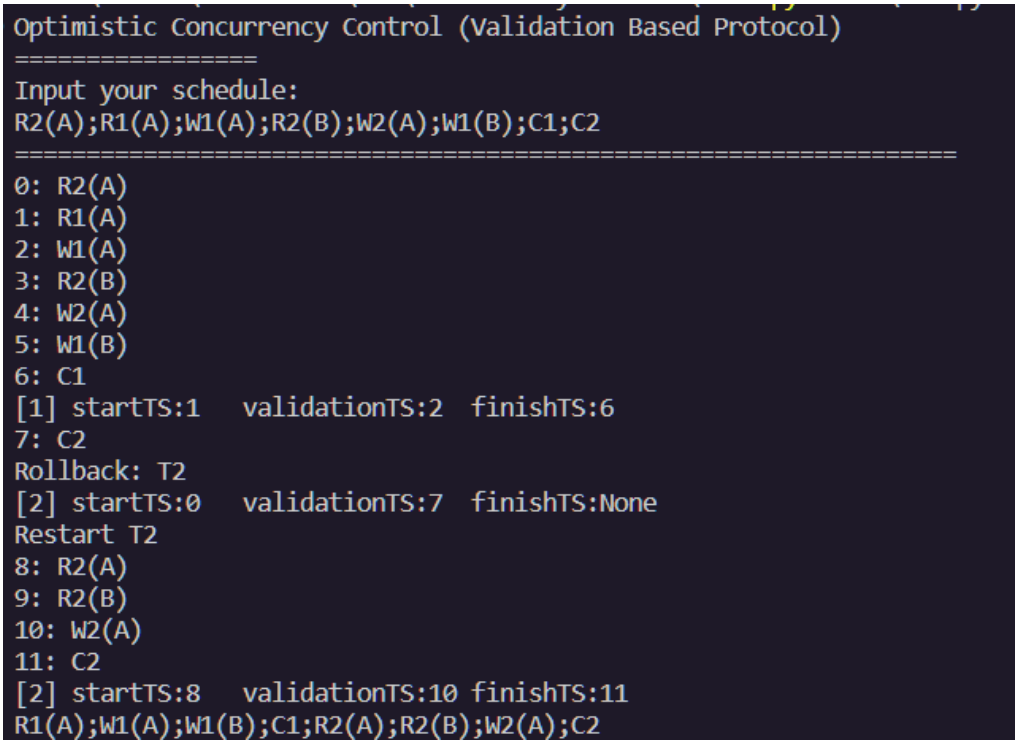
R1(A);W2(A);W2(B);W3(B);W1(A);C1;C2;C3

v. Percobaan 3

Percobaan pertama dilakukan dengan melakukan *input schedule* sebagai berikut:

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

Berikut merupakan hasil dari percobaan 3 ketika dijalankan pada program simulasi *Optimistic Concurrency Control*.



Gambar 2.11 Percobaan 3 *Optimistic Concurrency Control*

Pada *schedule* ini, dimulai dengan *read* A pada transaksi 2 dan 1. Kemudian, transaksi 1 melakukan *write* A. Transaksi 2 juga melakukan *read* B dan *write* A, lalu transaksi 1 melakukan *write* B. Ketika transaksi 1 di-*commit*, maka dilakukan validasi / pengecekan terhadap *start timestamp* (1), *validation timestamp* (2, ketika melakukan *write* A), dan *finish timestamp* (6). *Commit* transaksi 1 diterima. Sedangkan transaksi 2, ketika dilakukan validasi, tidak memenuhi syarat, sehingga transaksi 2 di-*rollback* dan dimulai kembali. Selanjutnya transaksi 2 dapat menjalankan instruksinya secara tidak konkuren sehingga berhasil finish pada timestamp 11.

Dengan demikian, *final schedule* untuk percobaan ini adalah sebagai berikut.

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

3. Eksplorasi Recovery

a. *Write-Ahead Log*

Write-Ahead Logging (WAL) adalah sebuah metode dalam manajemen basis data yang digunakan untuk memastikan integritas transaksi dan pemulihan data setelah kegagalan sistem. Konsep utama di balik WAL adalah bahwa sebelum melakukan perubahan pada basis data utama, perubahan tersebut terlebih dahulu dicatat dalam sebuah log yang disebut sebagai "*write-ahead log*". Proses ini melibatkan penulisan entri log sebelum data sebenarnya diubah dalam basis data, sehingga jika terjadi kegagalan sistem atau kerusakan, sistem dapat menggunakan log tersebut untuk memulihkan basis data ke keadaan konsisten terakhir.

Secara rinci, langkah-langkah utama dalam *Write-Ahead Logging* adalah sebagai berikut. Pertama, saat transaksi dimulai, sistem mencatat entri log untuk setiap perubahan yang akan dilakukan. Entri ini mencakup informasi seperti operasi yang dilakukan, nilai sebelum perubahan, dan nilai setelah perubahan. Kemudian, perubahan sebenarnya diterapkan pada basis data. Setelah perubahan berhasil, entri log dicatat sebagai "*commit record*" menandakan bahwa transaksi telah selesai dengan sukses.

b. *Continuous Archiving*

Continuous Archiving adalah suatu konsep dalam manajemen basis data yang melibatkan proses berkelanjutan untuk menyimpan salinan cadangan (*backup*) basis data secara terus-menerus dan sistematis. Tujuan utama dari *Continuous Archiving* adalah untuk memastikan keberlanjutan operasional dan pemulihan yang efisien dalam situasi kegagalan atau kehilangan data. Proses ini seringkali diimplementasikan bersama dengan metode *Write-Ahead Logging* (WAL) untuk meningkatkan tingkat ketahanan dan pemulihan basis data.

Pada dasarnya, Continuous Archiving melibatkan penyimpanan berkelanjutan dari salinan cadangan transaksional, yang mencakup log transaksi dan *snapshot* basis data, ke suatu lokasi penyimpanan eksternal. Log transaksi terus-menerus direkam selama operasi basis data berlangsung, sementara *snapshot* basis data mencerminkan keadaan basis data pada titik waktu tertentu. Proses ini dapat dilakukan secara periodik atau bahkan secara real-time tergantung pada kebutuhan dan tingkat kekritisan data. Dengan menyimpan

perubahan basis data secara terus-menerus, sistem dapat mengembalikan basis data ke keadaan konsisten terakhir sebelum terjadinya kegagalan tanpa kehilangan banyak data.

c. *Point-in-Time Recovery*

Point-in-Time Recovery (PITR) adalah suatu metode pemulihan dalam manajemen basis data yang memungkinkan pengembalian basis data ke suatu titik waktu tertentu di masa lalu. Konsep utama di balik PITR adalah memberikan kemampuan untuk mengembalikan basis data ke keadaan yang konsisten pada suatu titik waktu sebelum terjadinya kegagalan atau kesalahan, sehingga meminimalkan kerugian data. Untuk melaksanakan PITR, suatu basis data biasanya harus mendukung *Continuous Archiving* dan memiliki log transaksi yang mencatat semua perubahan yang terjadi pada basis data. Log transaksi ini menjadi kunci dalam proses PITR. Saat terjadi kegagalan atau kesalahan, administrator basis data dapat menggunakan log transaksi untuk mengidentifikasi titik waktu yang akan digunakan sebagai dasar untuk pemulihan.

Pada Postgresql, *Point-in-Time Recovery* dapat dilakukan dengan menerapkan kedua metode sebelumnya, yaitu *Continuous Archiving* dan *Write Ahead Log*. Langkah-langkah umum dalam PITR mencakup analisis log transaksi untuk menentukan titik waktu yang diinginkan, menghentikan operasi basis data, mengembalikan salinan cadangan basis data (*backup*) yang sesuai dengan titik waktu tersebut, dan menerapkan log transaksi untuk merestorasi perubahan yang terjadi setelah titik waktu tersebut. Hasilnya adalah basis data yang kembali ke keadaan yang konsisten dan sesuai dengan titik waktu yang dipilih.

d. Simulasi Kegagalan pada PostgreSQL

Sebelum menjalankan simulasi, langkah awal yang perlu diambil adalah menyiapkan dua direktori, yaitu 'basebackup' dan 'wal_archive'. Setelah itu, melakukan penyesuaian pada konfigurasi PostgreSQL dengan membuka file konfigurasi yang terletak pada path '`/etc/postgresql/16/data/postgresql.conf`'. Beberapa parameter yang perlu diubah dalam konfigurasi ini adalah sebagai berikut:

- `wal_Level =replica or archive`
  Menentukan tingkat *Write-Ahead Logging* (WAL) yang diaktifkan. Perekaman WAL aktif untuk tujuan replika atau arsip.
- `archive_mode=on`
  Memulai proses *Continuous Archiving* dan menyimpan *file-file* WAL ke dalam direktori arsip yang telah ditentukan.
- `archive_command = 'cp -i %p C:/Users/Yobel/Documents/wal_archive/%'`
  Menyalin *file* WAL yang aktif ke dalam direktori arsip yang telah ditentukan.
- `archive_timeout = 60`
  Menentukan waktu tunggu sebelum menyimpan *file* WAL ke dalam direktori arsip.

DATA SEBELUM KEGAGALAN TERJADI

Pada tahap ini, fokusnya adalah menambahkan data awal ke dalam database sebelum simulasi kegagalan dimulai. Proses ini dimulai dengan penambahan 5 data pertama ke dalam database PostgreSQL. Data ini mungkin mencakup informasi yang relevan dengan operasional atau kebutuhan bisnis sistem tersebut. Langkah ini bertujuan untuk menciptakan keadaan awal atau *baseline* data sebelum kegagalan atau pemulihan dilakukan.

```
myDB=# INSERT INTO tableI SELECT generate_series(1,5) AS id, md5(random()::text) AS descr;
INSERT 0 5
myDB=# SELECT * FROM tableI;
 id |               name
----+----------------------------------
  1 | a1b53e4eff935424a44a701895511ce6
  2 | 881ff9ce847d4351004e3b9a74e5c9cb
  3 | 96cf2ffaa9db28bacd7f89e236ee943d
  4 | d7a0729333e8476d46f3ea0506e18f97
  5 | 5b70af14b0ca087f515be2c7a3494961
(5 rows)


myDB=# SELECT now();
              now
-------------------------------
 2023-12-01 17:08:16.698868+07
(1 row)
```

```
PS C:\Users\Yobel\Documents\wal_archive> ls


    Directory: C:\Users\Yobel\Documents\wal_archive


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         12/1/2023   5:08 PM       16777216 000000010000000000000001
```

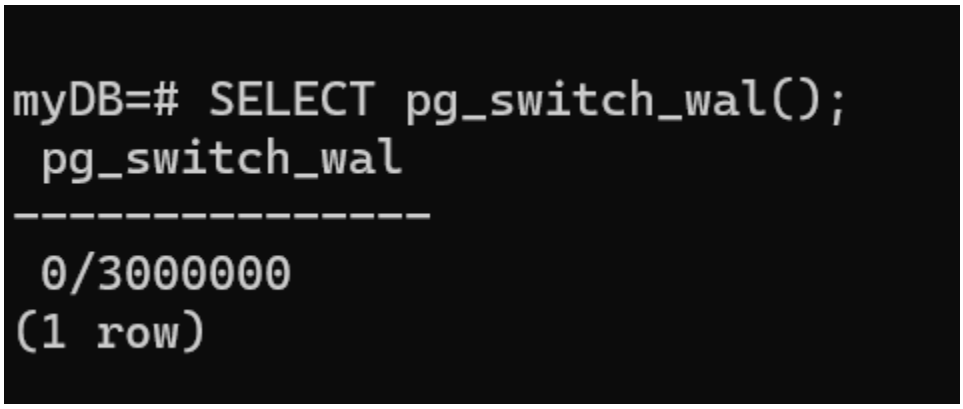
Gambar 3.1 Data Sebelum Kegagalan Terjadi

Setelah data awal dimasukkan ke dalam database, langkah selanjutnya dalam simulasi kegagalan PostgreSQL melibatkan penggunaan perintah `Switch_wal`. Langkah ini bertujuan untuk memastikan bahwa sistem berada dalam keadaan yang diinginkan, yakni dalam *state basebackup* atau *checkpoint*.

Log transaksi PostgreSQL mencatat setiap perubahan data yang terjadi dalam sistem. `Switch_wal` digunakan untuk menandai titik referensi penting dalam log transaksi ini. Saat `Switch_wal` dijalankan, PostgreSQL menciptakan titik yang jelas dan dapat diidentifikasi dalam log transaksi, menunjukkan bahwa operasi-operasi yang tercatat setelah titik ini merupakan perubahan data yang signifikan.

Setelah `Switch_wal` dijalankan, PostgreSQL berada dalam keadaan *basebackup* atau *checkpoint*. Keadaan ini menunjukkan bahwa sistem telah mencapai suatu titik tertentu dalam log transaksi yang dianggap aman dan konsisten. *Basebackup* adalah salinan lengkap dari

seluruh *database* pada suatu titik waktu tertentu, sedangkan *checkpoint* adalah titik spesifik dalam log transaksi di mana data dianggap aman dan terkonsolidasi.



Gambar 3.2 Data pg_switch_wal

Sebelum melanjutkan ke operasi selanjutnya, suatu langkah terstandarisasi diterapkan. Langkah ini mencakup dua proses utama, yaitu penghapusan data dari *backup* sebelumnya dan menggantinya dengan *basebackup* baru. Penghapusan data *backup* sebelumnya dapat dilakukan dengan menggunakan perintah `rm -rf` yang akan menghapus semua data dari direktori tersebut secara rekursif. Setelah itu, perintah `pg_basebackup` digunakan untuk membuat *basebackup* baru dengan mengisi ulang direktori tersebut dengan salinan data saat ini.

Penambahan 5 set data ke-2 ke dalam *database*.

```
myDB=# INSERT INTO tableI SELECT generate_series(6,10) AS id, md5(random()::text) AS descr;
INSERT 0 5
myDB=# SELECT * FROM tableI;
 id |               name
----+----------------------------------
  1 | a1b53e4eff935424a44a701895511ce6
  2 | 881ff9ce847d4351004e3b9a74e5c9cb
  3 | 96cf2ffaa9db28bacd7f89e236ee943d
  4 | d7a0729333e8476d46f3ea0506e18f97
  5 | 5b70af14b0ca087f515be2c7a3494961
  6 | c7045d8def4367c58390b63ce2f28419
  7 | d9e4fdc4f683fce90bb0d6416c6cfba3
  8 | 53faaa63018de7a4715614554118e096
  9 | 67597ba2e9fd1cb28ac93d0bd700edf1
 10 | 98bf0b4f5abb7e869c363488adee6ac9
(10 rows)


myDB=# SELECT COUNT(1) from tableI;
 count
-------
    10
(1 row)


myDB=# SELECT now();
              now
-------------------------------
 2023-12-01 17:21:53.378463+07
(1 row)
```

```
    Directory: C:\Users\Yobel\Documents\wal_archive


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         12/1/2023   5:10 PM       16777216 000000010000000000000001
-a----         12/1/2023   5:11 PM       16777216 000000010000000000000002
-a----         12/1/2023   5:19 PM       16777216 000000010000000000000003
-a----         12/1/2023   5:19 PM            338 000000010000000000000003.00000028.backup
-a----         12/1/2023   5:21 PM       16777216 000000010000000000000004
```

Gambar 3.3 Penambahan Set Data Kedua

Penambahan 5 set data ke-3 ke dalam *database*.

```
myDB=# INSERT INTO tableI SELECT generate_series(11,15) AS id, md5(random()::text) AS descr;
INSERT 0 5
myDB=# SELECT * FROM tableI;
 id |               name
----+----------------------------------
  1 | a1b53e4eff935424a44a701895511ce6
  2 | 881ff9ce847d4351004e3b9a74e5c9cb
  3 | 96cf2ffaa9db28bacd7f89e236ee943d
  4 | d7a0729333e8476d46f3ea0506e18f97
  5 | 5b70af14b0ca087f515be2c7a3494961
  6 | c7045d8def4367c58390b63ce2f28419
  7 | d9e4fdc4f683fce90bb0d6416c6cfba3
  8 | 53faaa63018de7a4715614554118e096
  9 | 67597ba2e9fd1cb28ac93d0bd700edf1
 10 | 98bf0b4f5abb7e869c363488adee6ac9
 11 | 014959ce54839efd1051cbb55163cea7
 12 | 436169040ecd01b3120b4e01b6c13554
 13 | a3bfcd1ef9bbdf5ebc645948de24b60b
 14 | bdf3591da858b91749fa642782cbf64d
 15 | 938445d2321ab700d4c307711691c257
(15 rows)


myDB=# SELECT COUNT(1) from tableI;
 count
-------
    15
(1 row)


myDB=# SELECT now();
              now
-------------------------------
 2023-12-01 17:25:44.82288+07
(1 row)
```

Gambar 3.4 Penambahan Set Data Ketiga

Setiap operasi penambahan data akan dicatat dalam log transaksi. Dengan penambahan data secara bertahap, kita dapat memeriksa dan menganalisis log transaksi untuk melihat perubahan data yang terjadi selama simulasi kegagalan.

SAAT KEGAGALAN TERJADI

Untuk mensimulasikan kegagalan pada PostgreSQL, matikan layanan (*service*) PostgreSQL.

```
myDB=# SELECT * FROM tableI;
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
!?>
```

Gambar 3.5 Mematikan Layanan PostgreSQL

DATA SETELAH DILAKUKAN RECOVERY

```
postgres=# \c myDB;
You are now connected to database "myDB" as user "postgres".
myDB=# SELECT * FROM tableI;
 id |              name
----+----------------------------------
  1 | a1b53e4eff935424a44a701895511ce6
  2 | 881ff9ce847d4351004e3b9a74e5c9cb
  3 | 96cf2ffaa9db28bacd7f89e236ee943d
  4 | d7a0729333e8476d46f3ea0506e18f97
  5 | 5b70af14b0ca087f515be2c7a3494961
  6 | c7045d8def4367c58390b63ce2f28419
  7 | d9e4fdc4f683fce90bb0d6416c6cfba3
  8 | 53faaa63018de7a4715614554118e096
  9 | 67597ba2e9fd1cb28ac93d0bd700edf1
 10 | 98bf0b4f5abb7e869c363488adee6ac9
(10 rows)
```

Gambar 3.6 Data Setelah Dilakukan *Recovery*

Langkah selanjutnya adalah menyesuaikan konfigurasi PostgreSQL dengan menambahkan `restore_command` dan `recovery_target_time`. `Restore_command` diatur untuk menentukan bagaimana pemulihan dari arsip log dilakukan, sementara `recovery_target_time` menetapkan target waktu pemulihan yaitu `2023-12-01 17:25:44`.

Proses pemulihan dimulai dari *basebackup* atau *checkpoint* terakhir sebelum kegagalan terjadi. Log secara *roll-forward* dibaca dan eksekusi terhadap database dilakukan sesuai dengan log yang dihasilkan. Proses ini berlanjut hingga waktu pada log mencapai target waktu yang ditentukan.

4. Pembagian Kerja

Tabel 4.1 Pembagian Kerja

NIM	Nama	Bagian
13521059	Arleen Chrysantha Gunardi	Implementasi *Concurrency Control Protocol*: *Two Phase Locking* (2PL)
13521067	Yobel Dean Christopher	Eksplorasi *Recovery*
13521121	Saddam Annais S	Eksplorasi *Transaction Isolation*
13521165	Reza Pahlevi Ubaidillah	Implementasi *Concurrency Control Protocol*: *Optimistic Concurrency Control* (OCC)

Referensi

1. PostgreSQL: Documentation: 16: 13.2. Transaction Isolation
2. PostgreSQL: Documentation: 16: 30.3. Write-Ahead Logging (WAL)
3. PostgreSQL: Documentation: 16: 26.3. Continuous Archiving and Point-in-Time Recovery (PITR)
4. PPT Kuliah IF3140 Manajemen Basis Data Concurrency Control Bagian 1
5. PPT Kuliah IF3140 Manajemen Basis Data Concurrency Control Bagian 2