

TUGAS 2
IF3140 MANAJEMEN BASIS DATA
MEKANISME *CONCURRENCY CONTROL* DAN *RECOVERY*



Disusun oleh
Kelompok 02 Kelas 02

Diky Restu Maulana	13520017
Felicia Sutandijo	13520050
Rayhan Kinan Muhannad	13520065
Muhammad Naufal Satriandana	13520168

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2022

Daftar Isi

Daftar Isi	2
A. Eksplorasi Concurrency Control	3
Penjelasan Serializable, Repeatable Read, Read Committed, dan Read Uncommitted	3
Simulasi Serializable, Repeatable Read, dan Read Committed	5
B. Implementasi Concurrency Control Protocol	18
1. Simple locking	22
2. OCC	30
3. MVCC	32
C. Eksplorasi Recovery	33
Penjelasan Write-Ahead Log, Continuous Archiving, dan Point-In-Time Recovery	33
Tahapan Konfigurasi pada PostgreSQL	34
Simulasi Kegagalan pada PostgreSQL dan Proses Recovery	39
Kesimpulan dan Saran	42
Pembagian Kerja Kelompok	43
Referensi	43

A. Eksplorasi Concurrency Control

Penjelasan *Serializable*, *Repeatable Read*, *Read Committed*, dan *Read Uncommitted*

Standar SQL menyediakan empat level isolasi transaksi, yaitu *serializable*, *repeatable read*, *read committed*, dan *read uncommitted*, terurut dari yang paling ketat. Level-level tersebut didefinisikan dengan cara mendefinisikan fenomena-fenomena yang tidak boleh terjadi. Fenomena-fenomena yang tidak diperbolehkan pada level-level tertentu tersebut antara lain:

1. *Dirty read*

Sebuah transaksi membaca data yang telah ditulis oleh transaksi konkuren lain yang belum di-*commit*.

2. *Nonrepeatable read*

Sebuah transaksi membaca ulang data yang telah dibaca sebelumnya dan menemukan bahwa data telah diubah oleh transaksi lain yang sudah di-*commit*.

3. *Phantom read*

Sebuah transaksi mengeksekusi kembali sebuah *query* yang mengembalikan baris-baris yang memenuhi kondisi pencarian, namun ketika dieksekusi kembali baris-baris yang memenuhi kondisi pencarian tersebut telah berubah karena transaksi lain yang sudah di-*commit*.

4. *Serialization anomaly*

Hasil dari *commit* sebuah grup transaksi tidak konsisten dengan seluruh urutan yang mungkin dari transaksi-transaksi tersebut yang dijalankan satu per satu.

Berikut fenomena apa saja yang diperbolehkan untuk setiap level isolasi.

Tabel 1.1. Level Isolasi Transaksi

Level isolasi	<i>Dirty read</i>	<i>Nonrepeatable read</i>	<i>Phantom read</i>	<i>Serialization anomaly</i>
<i>Serializable</i>	Tidak diperbolehkan	Tidak diperbolehkan	Tidak diperbolehkan	Tidak diperbolehkan
<i>Repeatable read</i>	Tidak diperbolehkan	Tidak diperbolehkan	Diperbolehkan, namun tidak di PG	Diperbolehkan
<i>Read committed</i>	Tidak diperbolehkan	Diperbolehkan	Diperbolehkan	Diperbolehkan
<i>Read Uncommitted</i>	Diperbolehkan, namun tidak di PG	Diperbolehkan	Diperbolehkan	Diperbolehkan

1. Serializable

Level isolasi *transaksi serializable* merupakan level isolasi transaksi yang paling ketat. Pada level ini, transaksi-transaksi seolah-olah dijalankan secara serial (berurutan) dan tidak konkuren (bersamaan). Aplikasi-aplikasi yang menggunakan level isolasi ini harus siap untuk mengulangi transaksi-transaksi yang gagal karena kegagalan pelaksanaan serial. Yang membedakan level ini dengan level isolasi transaksi *repeatable read* adalah level *serializable* memonitor apakah terjadi *serialization anomaly* atau tidak, yang bila ditemukan akan menyebabkan *serialization failure*. Proses monitor ini dapat memakan *overhead* yang lebih banyak.

Pada level isolasi ini, tidak mungkin terjadi keempat fenomena yang telah disebutkan, yaitu *fenomena dirty read*, *nonrepeatable read*, *phantom read*, dan *serialization anomaly*. Fenomena *dirty read* tidak mungkin terjadi karena transaksi hanya membaca data yang sudah di-*commit*. Fenomena *nonrepeatable read* tidak mungkin terjadi karena tidak ada transaksi lain yang akan mengubah data ketika transaksi pertama sedang berjalan, sehingga bila data dibaca kembali akan tetap sama. Fenomena *phantom read* juga tidak mungkin terjadi karena tidak ada transaksi lain yang mengubah data saat transaksi pertama berjalan, sehingga *query* yang sama bila dijalankan akan menghasilkan hasil yang sama pula. Terakhir, *serialization anomaly* tidak diperbolehkan terjadi dan terus dipantau oleh sistem agar tidak terjadi.

2. Repeatable Read

Level isolasi transaksi *repeatable read* hanya membaca data yang telah di-*commit* oleh transaksi-transaksi sebelumnya, dan tidak membaca data yang belum di-*commit* ataupun perubahan yang terjadi ketika transaksi sedang dilaksanakan oleh transaksi lain yang konkuren (namun sebuah *query* tetap melihat perubahan yang dilakukan oleh transaksinya sendiri, meskipun belum di-*commit*). Aplikasi-aplikasi yang menggunakan level isolasi ini juga harus siap untuk mengulangi transaksi-transaksi yang gagal karena kegagalan pelaksanaan serial.

Level isolasi transaksi ini tidak memungkinkan terjadinya fenomena *dirty read* dan *nonrepeatable read*. Fenomena *phantom read* masih dimungkinkan, namun tidak diperbolehkan pada PostgreSQL. Fenomena *dirty read* tidak mungkin terjadi karena transaksi hanya membaca data yang telah di-*commit*. Fenomena *repeatable read* juga tidak mungkin terjadi karena transaksi hanya membaca data yang telah di-*commit*, sehingga

pembacaan ulang pasti menghasilkan hasil yang sama karena tidak dipengaruhi oleh transaksi-transaksi konkuren lainnya.

3. Read Committed

Level isolasi transaksi *read committed* merupakan level isolasi transaksi *default* pada PostgreSQL. Ketika sebuah transaksi menggunakan level isolasi ini, maka sebuah *query* SELECT hanya akan melihat data yang di-*commit* sebelum *query* dieksekusi. Namun, *query* SELECT setelahnya dapat menghasilkan keluaran yang berbeda karena bisa saja telah terjadi perubahan lain oleh transaksi yang konkuren yang di-*commit* setelah *query* pertama namun sebelum *query* kedua.

Level isolasi transaksi ini tidak memungkinkan terjadinya *dirty read*, namun ketiga fenomena lainnya masih mungkin terjadi. Fenomena *dirty read* tidak mungkin terjadi karena *query* hanya membaca data yang sudah di-*commit* saat pelaksanaan *query*.

4. Read Uncommitted

Level isolasi transaksi *read uncommitted* memperbolehkan *dirty read*, *nonrepeatable read*, *phantom read*, dan *serialization anomaly*. Namun, pada PostgreSQL, *read uncommitted* tidak dapat digunakan dan akan ditangani seperti *read committed*.

Simulasi Serializable, Repeatable Read, dan Read Committed

Untuk memahami perbedaan-perbedaan dari level isolasi transaksi yang berbeda, akan dilakukan simulasi dari setiap derajat isolasi pada PostgreSQL, kecuali level isolasi transaksi *read uncommitted*. Hal ini karena tidak terdapat level isolasi transaksi *read uncommitted* pada PostgreSQL. Simulasi akan dilakukan pada dua terminal, satu terminal untuk menjalankan *query*, dan yang lainnya untuk mengecek isolasi.

1. Serializable

Transaksi pada kedua terminal diawali dengan perintah `BEGIN` dan diatur level isolasinya dengan perintah `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`.

```
postgres=# BEGIN;  
BEGIN  
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
SET
```

Gambar 1.1. Awal Transaksi

Kemudian, pada terminal pertama dilakukan INSERT pada tabel yang sudah ada. Hasil INSERT ditampilkan menggunakan perintah SELECT. Terlihat bahwa INSERT telah berhasil, namun transaksi belum di-*commit*.

```
postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
postgres=# INSERT INTO tubes2_13520050 VALUES (123);
INSERT 0 1
postgres=# SELECT * FROM tubes2_13520050 WHERE col1=123;
 col1
-----
   123
(1 row)

postgres=# |
```

Gambar 1.2. Terminal 1: INSERT data 123

Pada terminal kedua, dicoba dilakukan INSERT dengan nilai yang sama. Karena col1 merupakan *primary* key, maka *query* tersebut harus menunggu transaksi pada terminal pertama.

```
postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
postgres=# INSERT INTO tubes2_13520050 VALUES (123);
|
```

Gambar 1.3. Terminal 2: INSERT data 123

Pada terminal pertama, transaksi di-*commit*.

```
postgres=# COMMIT;
COMMIT
```

Gambar 1.4. Terminal 1: COMMIT

Setelah dilakukan *commit* pada terminal pertama, *query* di terminal kedua gagal karena col1 merupakan *primary* key dan nilai yang hendak dimasukkan ke col1 merupakan nilai yang sama, yaitu 123. Transaksi mengalami *rollback* ketika dicoba untuk di-*commit*.

```

SET
postgres=# INSERT INTO tubes2_13520050 VALUES (123);
ERROR:  duplicate key value violates unique constraint "tubes2_13520050_pkey"
DETAIL:  Key (col1)=(123) already exists.
postgres=# COMMIT;
ROLLBACK
postgres=# |

```

Gambar 1.5. Terminal 1: INSERT gagal

Pada simulasi ini, terlihat bahwa level isolasi transaksi *serializable* tidak memperbolehkan keempat fenomena *dirty read*, *nonrepeatable read*, *phantom read*, dan *serialization anomaly*.

Kedua transaksi tersebut dapat digambarkan pada tabel berikut, dengan transaksi pada terminal pertama diberi nama T1 dan transaksi pada terminal kedua diberi nama T2.

Tabel 1.2. Simulasi Serializable

T1	T2	Keterangan
	BEGIN;	Memulai transaksi T2.
	SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	Memilih level isolasi <i>serializable</i> di T2.
BEGIN;		Memulai transaksi T1.
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;		Memilih level isolasi <i>serializable</i> di T1.
INSERT INTO tubes2_13520050 VALUES (123);		Menambahkan data 123 pada tabel tubes2_13520050.
SELECT * FROM tubes2_13520050 WHERE col1=123;		Membaca data 123.
	INSERT INTO	Menambahkan data

	tubes2_13520050 VALUES (123);	dengan nilai yang sama, yaitu 123. Gagal melakukan perubahan, muncul <i>error</i> .
COMMIT;		T1 melakukan <i>commit</i> .
	ROLLBACK;	T2 melakukan <i>rollback</i> karena terjadi <i>error</i> .

2. Repeatable Read

Seperti pada simulasi sebelumnya, transaksi dimulai dengan perintah BEGIN. Kemudian, level isolasi transaksi ditentukan dengan perintah SET TRANSACTION ISOLATION LEVEL REPEATABLE READ.

```
postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
```

Gambar 1.6. Awal Transaksi

Pada terminal pertama, dilakukan perintah SELECT untuk memeriksa tabel tubes2_13520050 pada saat ini. Hasil SELECT menunjukkan 2 buah baris yang berisi 1 dan 123.

```
postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
    1
   123
(2 rows)

postgres=# |
```

Gambar 1.7. Terminal 1: SELECT

Kemudian, pada terminal kedua dilakukan perintah INSERT untuk menambahkan data 12345 ke col1 pada tabel tubes2_13520050. Perintah SELECT dijalankan untuk mengecek apakah INSERT telah berhasil. Perhatikan bahwa transaksi ini belum di-*commit*.

```
postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
postgres=# INSERT INTO tubes2_13520050 VALUES (12345);
INSERT 0 1
postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
      1
     123
    12345
(3 rows)

postgres=# |
```

Gambar 1.8. Terminal 2: INSERT data 12345

Kemudian, pada terminal pertama dilakukan pembacaan ulang pada tabel yang telah di-*insert* oleh transaksi pada terminal kedua. Hasil pembacaan ulang menggunakan perintah SELECT masih menghasilkan hasil yang sama, yaitu 1 dan 123. Baris data 12345 tidak ikut terbaca.

```

postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
    1
   123
(2 rows)

postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
    1
   123
(2 rows)

postgres=# |

```

Gambar 1.9. Terminal 1: SELECT

Setelah itu, transaksi pada terminal kedua di-*commit*.

```

postgres=# COMMIT;
COMMIT

```

Gambar 1.10. Terminal 2: COMMIT

Kembali dilakukan pembacaan ulang pada terminal pertama untuk mengecek apakah terjadi fenomena *nonrepeatable read*. Hasil tetap sama, dan data 12345 tidak ikut terbaca.

```

postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
    1
   123
(2 rows)

```

Gambar 1.11. Terminal 1: SELECT

Ketika transaksi pada terminal pertama hendak melakukan INSERT ke dalam tabel dengan nilai yang sama (12345), *query* INSERT gagal karena col1 merupakan *primary key*

dan tidak boleh terdapat duplikat pada nilainya. Transaksi ini kemudian ketika dicoba untuk di-*commit* akan melakukan *rollback*.

```
postgres=# INSERT INTO tubes2_13520050 VALUES (12345);
ERROR:  duplicate key value violates unique constraint "tubes2_13520050_pkey"
DETAIL:  Key (col1)=(12345) already exists.
postgres=# COMMIT;
ROLLBACK
postgres=# |
```

Gambar 1.12. Terminal 1: INSERT data 12345

Dari simulasi ini, dapat terlihat bahwa level isolasi transaksi *repeatable read* tidak memperbolehkan *dirty read*, *nonrepeatable read*, dan *phantom read*, namun memperbolehkan *serialization anomaly*.

Kedua transaksi tersebut dapat digambarkan pada tabel berikut, dengan transaksi pada terminal pertama diberi nama T1 dan transaksi pada terminal kedua diberi nama T2.

Tabel 1.3. Simulasi Repeatable Read

T1	T2	Keterangan
	BEGIN;	Memulai transaksi T2.
	SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;	Memilih level isolasi <i>repeatable read</i> di T2.
BEGIN;		Memulai transaksi T1.
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;		Memilih level isolasi <i>repeatable read</i> di T1.
SELECT * FROM tubes2_13520050;		Membaca data yang ada pada tabel tubes2_13520050. Hasil <i>query</i> merupakan 2 baris data pada col1, yaitu 1 dan 123.

	INSERT INTO tubes2_13520050 VALUES (12345);	Menambahkan data baru, yaitu 12345 ke tabel tubes2_13520050.
	SELECT * FROM tubes2_13520050;	Memastikan bahwa penambahan data berhasil.
SELECT * FROM tubes2_13520050;		Melakukan pembacaan data pada tabel tubes2_13520050 dan hasil tetap menunjukkan 2 baris data, yaitu 1 dan 123.
	COMMIT;	T2 melakukan <i>commit</i> .
SELECT * FROM tubes2_13520050;		Melakukan pembacaan data pada tabel tubes2_13520050 dan hasil tetap menunjukkan 2 baris data, yaitu 1 dan 123.
INSERT INTO tubes2_13520050 VALUES (12345);		Gagal melakukan perubahan, muncul <i>error</i> .
ROLLBACK;		Melakukan <i>rollback</i> T1 karena mengalami <i>error</i> .

3. Read Committed

Transaksi pada kedua terminal diawali dengan perintah BEGIN dan diatur level isolasinya dengan perintah SET TRANSACTION ISOLATION LEVEL READ COMMITTED.

```
postgres=# BEGIN;  
BEGIN  
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SET  
postgres=# |
```

Gambar 1.13. Awal Transaksi

Pada terminal pertama, dilakukan perintah SELECT untuk membaca data pada tabel saat ini. Dari hasil perintah, terlihat bahwa pada pembacaan *query* saat ini, terdapat 3 buah baris data yang berisi 1, 123, dan 12345.

```
postgres=# BEGIN;  
BEGIN  
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SET  
postgres=# SELECT * FROM tubes2_13520050;  
col1  
-----  
      1  
     123  
    12345  
(3 rows)  
  
postgres=# |
```

Gambar 1.14. Terminal 1: SELECT

Kemudian, dimasukkan sebuah data baru pada transaksi di terminal kedua menggunakan perintah INSERT. Data yang dimasukkan adalah 1234567. Setelah itu, dilakukan perintah SELECT untuk memeriksa apakah data telah berhasil dimasukkan. Hasil dari perintah SELECT menunjukkan bahwa ada 4 buah baris data pada col1 tabel tubes2_13520050, yaitu 1, 123, 12345, dan 1234567. Artinya, INSERT pada tabel telah berhasil. Perhatikan bahwa transaksi belum di-*commit*.

```

postgres=# BEGIN;
BEGIN
postgres=# SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET
postgres=# INSERT INTO tubes2_13520050 VALUES (1234567);
INSERT 0 1
postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
      1
     123
    12345
   1234567
(4 rows)

postgres=# |

```

Gambar 1.15. Terminal 2: INSERT data 1234567

Untuk membuktikan tidak terjadinya *dirty read*, dilakukan pembacaan ulang dengan perintah yang sama, yaitu `SELECT * FROM tubes2_13520050` untuk melihat data yang ada pada tabel `tubes2_13520050` pada terminal pertama. Karena transaksi pada terminal kedua belum di-*commit*, maka data baru yang ditambahkan, yaitu 1234567, tidak ikut terbaca. Hasil *read* masih sama dengan *read* yang pertama, dengan tiga baris yang berisi 1, 123, dan 12345. Hal ini menunjukkan tidak terjadinya *dirty read*.

```

postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
      1
     123
    12345
(3 rows)

```

Gambar 1.16. Terminal 1: SELECT

Kemudian, dilakukan *commit* pada transaksi di terminal kedua.

```

postgres=# COMMIT;
COMMIT

```

Gambar 1.17. Terminal 2: COMMIT

Setelah transaksi pada terminal kedua di-*commit*, dilakukan pembacaan ulang pada terminal pertama untuk memeriksa apakah terjadi fenomena *nonrepeatable read*. Hasil dari *query* SELECT menunjukkan empat buah baris data pada col1 tabel tubes2_13520050 yang berisi 1, 123, 12345, dan 1234567. Data terbaru yang baru saja dimasukkan dan di-*commit* oleh transaksi pada terminal kedua ikut terbaca. Artinya, terjadi fenomena *nonrepeatable read* karena hasil *read* berbeda dari hasil *read* yang pertama. Hal ini sekaligus berarti bahwa *phantom read* juga dapat terjadi.

```
postgres=# SELECT * FROM tubes2_13520050;
 col1
-----
      1
     123
    12345
   1234567
(4 rows)
```

Gambar 1.18. Terminal 1: SELECT

Ketika transaksi pada terminal pertama hendak melakukan INSERT ke dalam tabel dengan nilai yang sama (1234567), *query* INSERT gagal karena col1 merupakan *primary key* dan tidak boleh terdapat duplikat pada nilainya. Transaksi ini kemudian ketika dicoba untuk di-*commit* akan melakukan *rollback*.

```
postgres=# INSERT INTO tubes2_13520050 VALUES (1234567);
ERROR:  duplicate key value violates unique constraint "tubes2_13520050_pkey"
DETAIL:  Key (col1)=(1234567) already exists.
postgres=# COMMIT;
ROLLBACK
```

Gambar 1.19. Terminal 1: INSERT data 1234567

Dari simulasi ini, dapat dilihat bahwa level isolasi transaksi *read committed* tidak memperbolehkan *dirty read*, namun memperbolehkan *nonrepeatable read*, *phantom read*, dan *serialization anomaly*.

Pada tabel, transaksi satu yang terjadi di terminal pertama diberi nama T1 dan transaksi kedua yang terjadi di terminal pertama.

Tabel 1.4. Simulasi Read Committed

T1	T2	Keterangan
----	----	------------

	BEGIN;	Memulai transaksi T2.
	SET TRANSACTION ISOLATION LEVEL READ COMMITTED;	Memilih level isolasi <i>read committed</i> pada T2.
BEGIN;		Memulai transaksi T1.
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;		Memilih level isolasi <i>read committed</i> di T1.
SELECT * FROM tubes2_13520050;		Melihat data yang ada pada tabel tubes2_13520050. Hasil dari <i>query</i> ini adalah 3 buah baris data pada col1, yaitu 1, 123, dan 12345.
	INSERT INTO tubes2_13520050 VALUES (1234567);	Menambahkan data 1234567 pada tabel tubes2_13520050.
	SELECT * FROM tubes2_13520050;	Memastikan bahwa penambahan data berhasil.
SELECT * FROM tubes2_13520050;		Melihat data yang ada pada tabel tubes2_13520050. Hasil dari <i>query</i> ini tetap 3 buah baris data pada col1, yaitu 1, 123, dan 12345.
	COMMIT;	Melakukan <i>commit</i> pada T2.
SELECT * FROM		Melihat data yang ada

tubes2_13520050;		pada tabel tubes2_13520050. Hasil dari <i>query</i> ini berubah menjadi 4 buah baris data pada col1, yaitu 1, 123, dan 12345, dan 1234567.
INSERT INTO tubes2_13520050 VALUES (12345);		Mencoba memasukkan data baru dengan nilai yang sama.
ROLLBACK;		T1 <i>rollback</i> .

B. Implementasi Concurrency Control Protocol

Berikut merupakan beberapa framework yang penulis *develop* dan gunakan untuk mengimplementasikan algoritma Concurrency Control Protocol. Untuk kode selengkapnya dapat diakses pada pranala berikut [ini](#).

process/Data.py

```
class Data:
    def __init__(self) -> None:
        self.value: int = 0

    def getValue(self) -> int:
        return self.value

    def setValue(self, value: int) -> None:
        self.value = value
```

process/FileAccess.py

```
import math

class FileAccess:
    def __init__(self, filename: str) -> None:
        self.filename: str = filename

    def read(self) -> int:
        file = open(self.filename, "rb")
        content = file.read()
        file.close()

        return int.from_bytes(content, byteorder="big", signed=False)

    def write(self, value: int) -> None:
        file = open(self.filename, "wb")

        length = math.ceil(value / (1 << 8))
        content = value.to_bytes(length, byteorder="big",
signed=False)
        file.write(content)
```

```
file.close()
```

process/Query.py

```
from typing import Callable, List

from process.FileAccess import FileAccess
from process.Data import Data

class Query:
    def __init__(self, *args: str) -> None:
        self.fileNames: List[str] = []
        self.fileAccesses: List[FileAccess] = []

        for filename in args:
            self.fileNames.append(filename)
            self.fileAccesses.append(FileAccess(filename))

    def getFileNames(self) -> List[str]:
        return self.fileNames

    def execute(self) -> None:
        raise NotImplementedError()

class ReadQuery(Query):
    def __init__(self, *args: str) -> None:
        super().__init__(*args)

    def execute(self, *args: Data) -> None:
        for i in range(len(args)):
            data = args[i]
            fileAccess = self.fileAccesses[i]
            data.setValue(fileAccess.read())

class WriteQuery(Query):
    def __init__(self, *args: str) -> None:
```

```

        super().__init__(*args)

    def execute(self, *args: Data) -> None:
        for i in range(len(args)):
            data = args[i]
            fileAccess = self.fileAccesses[i]
            fileAccess.write(data.getValue())

class FunctionQuery(Query):
    def __init__(self, *args: str, **kwargs: Callable[..., int]) -> None:
        super().__init__(*args)
        self.function = kwargs.get("function", lambda *args: args[0])

    def execute(self, *args: Data) -> None:
        value: List[int] = []
        for data in args:
            value.append(data.getValue())
        args[0].setValue(self.function(*value))

class DisplayQuery(Query):
    def __init__(self, *args: str, **kwargs: Callable[..., int]) -> None:
        super().__init__(*args)
        self.function = kwargs.get("function", lambda *args: args)

    def execute(self, *args: Data) -> None:
        value: List[int] = []
        for data in args:
            value.append(data.getValue())

        result = self.function(*value)
        if type(result) is tuple:
            print(" ".join(map(str, result)))
        if type(result) is int:
            print(result)

```

process/Transaction.py

```
from typing import Dict, List

from process.Query import Query
from process.Data import Data

class Transaction:
    def __init__(self, startTimestamp: int, listOfQuery: List[Query])
    -> None:
        self.startTimestamp = startTimestamp
        self.listOfQuery = listOfQuery
        self.queryIndex = 0
        self.dictData: Dict[str, Data] = dict()

    def getStartTimestamp(self):
        return self.startTimestamp

    def getLength(self) -> int:
        return len(self.listOfQuery)

    def getCurrentQuery(self) -> Query:
        return self.listOfQuery[self.queryIndex]

    def isFinished(self) -> bool:
        return self.queryIndex == self.getLength()

    def nextQuery(self) -> None:
        self.queryIndex += 1

    def rollback(self, newTimestamp) -> None:
        self.startTimestamp = newTimestamp
        self.queryIndex = 0
        self.dictData: Dict[str, Data] = dict()

    def commit(self) -> None:
        for i in range(self.getLength()):
            currentQuery = self.listOfQuery[i]
```

```

currentFileNames = currentQuery.getFileNames()
currentData: List[Data] = []

for filename in currentFileNames:
    currentData.append(self.dictData.setdefault(filename,
Data()))

currentQuery.execute(*currentData)

```

1. Simple Locking

Simple locking merupakan salah satu concurrency control protocol yang menggunakan “lock” sebagai pengendali konkurensi. Sebenarnya, terdapat dua jenis lock yang umum, yakni simple lock dan exclusive lock. Shared lock akan diberikan kepada transaksi yang ingin membaca suatu data, sedangkan exclusive lock akan diberikan kepada transaksi yang ingin menulis ke suatu data. Akan tetapi, dalam implementasi ini, hanya exclusive lock yang akan digunakan.

Cara kerja simple locking adalah sebagai berikut. Transaksi-transaksi yang berjalan secara konkuren akan diberikan lock sesuai dengan urutan kedatangannya. Transaksi yang ingin membaca atau menulis ke dalam suatu data akan melakukan permintaan lock untuk data tersebut. Jika lock tidak sedang dipegang oleh transaksi lain, maka sistem akan memberikannya, tetapi jika sedang dipegang, maka transaksi tersebut harus menunggu hingga transaksi yang sedang memegang locknya melepaskannya kembali, dalam kasus ini hingga transaksi tersebut melakukan commit.

Screenshot:

a) Hasil percobaan

```

Select input type:
1. From a file
2. From terminal
Enter your choice: 2
Enter the input: R1(X); W2(X); W2(Y); W3(Y); W1(X); C1; C2; C3;
Verbose? (y/n): y

```

Gambar 2.1.1 pemilihan jenis input terminal dan verbose

Dalam percobaan ini, pengguna akan diminta sebuah jenis input. Pengguna memasukkan angka 2 karena user ingin memasukkan input dari terminal. Format input adalah operasi yang dipisah dengan titik koma, seperti contoh di atas. Kemudian pengguna akan diberi pilihan apakah verbose atau tidak. Jika memilih verbose, semua proses yang dilakukan akan diperlihatkan state program saat itu. Dalam kasus ini, pengguna memilih 'y', artinya verbose.

```
User input      : ['R1(X)', 'W2(X)', 'W2(Y)', 'W3(Y)', 'W1(X)', 'C1', 'C2', 'C3']
Schedule       : ['R1(X)', 'W2(X)', 'W2(Y)', 'W3(Y)', 'W1(X)', 'C1', 'C2', 'C3']
Lock Table     : {}
Waiting Queue  : []
Final Schedule : []

Operation      : R1(X)
Schedule       : ['W2(X)', 'W2(Y)', 'W3(Y)', 'W1(X)', 'C1', 'C2', 'C3']
Lock Table     : {'X': '1'}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)']

Operation      : W2(X)
Schedule       : ['W3(Y)', 'W1(X)', 'C1', 'C3']
Lock Table     : {'X': '1'}
Waiting Queue  : ['W2(X)', 'W2(Y)', 'C2']
Final Schedule : ['L1(X)', 'R1(X)']

Operation      : W3(Y)
Schedule       : ['W1(X)', 'C1', 'C3']
Lock Table     : {'X': '1', 'Y': '3'}
Waiting Queue  : ['W2(X)', 'W2(Y)', 'C2']
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'W3(Y)']

Operation      : W1(X)
Schedule       : ['C1', 'C3']
Lock Table     : {'X': '1', 'Y': '3'}
Waiting Queue  : ['W2(X)', 'W2(Y)', 'C2']
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'W3(Y)', 'W1(X)']

Operation      : C1
Schedule       : ['W2(X)', 'W2(Y)', 'C2', 'C3']
Lock Table     : {'X': None, 'Y': '3'}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'W3(Y)', 'W1(X)', 'C1', 'U1(X)']

Operation      : W2(X)
Schedule       : ['W2(Y)', 'C2', 'C3']
Lock Table     : {'X': '2', 'Y': '3'}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'W3(Y)', 'W1(X)', 'C1', 'U1(X)', 'L2(X)', 'W2(X)']
```

```

Operation      : W2(Y)
Schedule       : ['C3']
Lock Table     : {'X': '2', 'Y': '3'}
Waiting Queue  : ['W2(Y)', 'C2']
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'WB(Y)', 'W1(X)', 'C1', 'U1(X)', 'L2(X)', 'W2(X)']

Operation      : C3
Schedule       : ['W2(Y)', 'C2']
Lock Table     : {'X': '2', 'Y': None}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'WB(Y)', 'W1(X)', 'C1', 'U1(X)', 'L2(X)', 'W2(X)', 'C3', 'U3(Y)']

Operation      : W2(Y)
Schedule       : ['C2']
Lock Table     : {'X': '2', 'Y': '2'}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'WB(Y)', 'W1(X)', 'C1', 'U1(X)', 'L2(X)', 'W2(X)', 'C3', 'U3(Y)', 'L2(Y)', 'W2(Y)']

Operation      : C2
Schedule       : []
Lock Table     : {'X': None, 'Y': None}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'WB(Y)', 'W1(X)', 'C1', 'U1(X)', 'L2(X)', 'W2(X)', 'C3', 'U3(Y)', 'L2(Y)', 'W2(Y)', 'C2', 'U2(X)', 'U2(Y)']

Final Schedule : ['L1(X)', 'R1(X)', 'L3(Y)', 'WB(Y)', 'W1(X)', 'C1', 'U1(X)', 'L2(X)', 'W2(X)', 'C3', 'U3(Y)', 'L2(Y)', 'W2(Y)', 'C2', 'U2(X)', 'U2(Y)']

```

Gambar 2.1.2 dan 2.1.3 hasil output percobaan pertama

Karena user memilih verbose, maka untuk setiap langkah yang diambil program, akan ditampilkan state program pada saat itu. Dari gambar, dapat dilihat semua komponen kosong kecuali schedule. Komponen yang dimaksud disini adalah schedule, lock table, waiting queue, dan final schedule. Komponen-komponen ini akan digunakan program untuk menghasilkan output.

Dapat dilihat bahwa setiap langkah, program akan mengeksekusi schedule secara berurutan. Jika suatu operasi dalam schedule tidak memungkinkan untuk dilakukan, seperti karena lock yang dibutuhkan sedang digunakan transaksi lain, maka operasi tersebut dan semua operasi setelahnya dari transaksi yang sama akan dimasukkan ke waiting queue. Dalam kasus ini, contohnya dapat dilihat pada langkah operation W2(X). Operasi tersebut dan seluruh operasi dari transaksi 2 akan ditaruh di waiting queue karena lock untuk X dipegang oleh transaksi 1. Ketika sebuah transaksi melakukan commit, maka semua operasi dalam waiting queue akan ditaruh ke depan schedule, kemudian dicoba untuk dieksekusi kembali. Hal ini karena lock akan dilepaskan pada saat commit. Hal ini terus diulang hingga semua schedule habis. Jika schedule habis namun masih ada operasi di waiting queue, maka operasi tersebut akan dimasukkan kembali ke schedule dan dijalankan kembali. Hasil schedule yang baru akan ditempatkan di array final schedule dan akan ditampilkan ke layar.


```

Select input type:
1. From a file
2. From terminal
Enter your choice: 1
Enter the file name: D:\ProjectKuliah\MBD\Tubes 2\concurrency-control-protocol\tes.txt
Verbose? (y/n): y
User input      : ['R1(X)', 'R2(Y)', 'R1(Y)', 'R2(X)', 'C1', 'C2']
Schedule       : ['R1(X)', 'R2(Y)', 'R1(Y)', 'R2(X)', 'C1', 'C2']
Lock Table     : {}
Waiting Queue  : []
Final Schedule : []

Operation      : R1(X)
Schedule       : ['R2(Y)', 'R1(Y)', 'R2(X)', 'C1', 'C2']
Lock Table     : {'X': '1'}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)']

Operation      : R2(Y)
Schedule       : ['R1(Y)', 'R2(X)', 'C1', 'C2']
Lock Table     : {'X': '1', 'Y': '2'}
Waiting Queue  : []
Final Schedule : ['L1(X)', 'R1(X)', 'L2(Y)', 'R2(Y)']

Operation      : R1(Y)
Schedule       : ['R2(X)', 'C2']
Lock Table     : {'X': '1', 'Y': '2'}
Waiting Queue  : ['R1(Y)', 'C1']
Final Schedule : ['L1(X)', 'R1(X)', 'L2(Y)', 'R2(Y)']

Operation      : R2(X)
Schedule       : []
Lock Table     : {'X': '1', 'Y': '2'}
Waiting Queue  : ['R1(Y)', 'C1', 'R2(X)', 'C2']
Final Schedule : ['L1(X)', 'R1(X)', 'L2(Y)', 'R2(Y)']

Schedule      : []
Lock Table     : {'X': '1', 'Y': '2'}
Waiting Queue  : ['R1(Y)', 'C1', 'R2(X)', 'C2']
Final Schedule : ['L1(X)', 'R1(X)', 'L2(Y)', 'R2(Y)']

Deadlock Detected

```

Gambar 2.1.4 percobaan 2

Pada percobaan 2, pengguna memilih input dari file. Hasil input tersebut akan ditampilkan di awal sebagai user input. Akan tetapi, dalam kasus ini, terlihat bahwa terjadi deadlock. Transaksi 2 sedang menunggu lock X yang dipegang oleh transaksi 1, namun di saat yang sama, transaksi 1 sedang menunggu lock Y yang sedang dipegang transaksi 2. Hal ini menyebabkan deadlock. Dalam implementasinya, waiting queue yang masih tersisa akan diproses kembali. Namun dalam kasus ini, walaupun setelah diproses kembali, akan tetap tersisa waiting queue yang sama dengan waiting queue sebelum diproses kembali. Program mendeteksi waiting queue yang sama dan menyatakan bahwa schedule yang diberikan akan deadlock.

b) Analisis algoritma

```
class SimpleLockingControl():
    def __init__(self, schedule):
        self.schedule = schedule # example ['R1(X)', 'W2(X)', 'W2(Y)',
        'W3(Y)', 'W1(X)', 'C1', 'C2', 'C3']
        self.lockTable = {} # key = data item, value = transaction that
        has the lock for that data item, example: {'X': '1', 'Y': '2'}
        self.waitingQueue = []
        self.finalSchedule = []
        self.previousWaitingQueue = []

    def lock(self, transactionId, dataItem):
        if dataItem in self.lockTable and self.lockTable[dataItem] !=
        transactionId and self.lockTable[dataItem] != None:
            return False
        else:
            self.lockTable[dataItem] = transactionId
            return True

    def unlock(self, transactionId, dataItem):
        if self.lockTable[dataItem] == transactionId:
            self.lockTable[dataItem] = None
            return True
        else:
            return False

    def hasLock(self, transactionId, dataItem):
        if dataItem in self.lockTable:
            return self.lockTable[dataItem] == transactionId
        else:
            return False
```

```

def getTransactionId(self, operation):
    return operation[1:operation.find("(")] if operation[0] != 'C'
else operation[1:]

def getDataItem(self, operation):
    return operation[operation.find("(") + 1:operation.find(")")] if
operation[0] != 'C' else None

def printState(self):
    print (f"{'Schedule':20} : {self.schedule}")
    print (f"{'Lock Table':20} : {self.lockTable}")
    print (f"{'Waiting Queue':20} : {self.waitingQueue}")
    print (f"{'Final Schedule':20} : {self.finalSchedule}")
    print ()

def run(self, verbose=False):
    if verbose:
        self.printState()

    while len(self.schedule) > 0:
        operation = self.schedule.pop(0)
        if verbose:
            print (f"{'Operation':20} : {operation}")
        if operation[0] == 'R' or operation[0] == 'W':
            transactionId = self.getTransactionId(operation)
            dataItem = self.getDataItem(operation)
            if self.hasLock(transactionId, dataItem):
                self.finalSchedule.append(operation)
            else:
                if self.lock(transactionId, dataItem):
                    self.finalSchedule.append('L' + operation[1:])
                self.finalSchedule.append(operation)

```

```

        else:
            # add to waiting queue every operation for the
transaction
            temp = []
            temp.append(operation)
            for op in self.schedule:
                if self.getTransactionId(op) ==
transactionId:
                    temp.append(op)
            for op in temp[1:]:
                self.schedule.remove(op)
            self.waitingQueue += temp

    if operation[0] == 'C':
        transactionId = self.getTransactionId(operation)
        # remove all locks for the transaction
        self.finalSchedule.append(operation)
        for dataItem in self.lockTable:
            if self.unlock(transactionId, dataItem):
                self.finalSchedule.append('U' + transactionId +
('(' + dataItem + ')')
                # remove all operations for the transaction from the
waiting queue and add them in front of the schedule
                while len(self.waitingQueue) > 0:
                    self.schedule.insert(0, self.waitingQueue.pop())
        if verbose:
            self.printState()
        while len(self.waitingQueue) > 0:
            # if the waiting queue is the same as the previous one, then
there is a deadlock
            if self.waitingQueue == self.previousWaitingQueue:
                raise Exception("Deadlock Detected")

```

```
self.previousWaitingQueue = self.waitingQueue
self.run(verbose)

return self.finalSchedule
```

Gambar 2.1.5 Algoritma simple locking

Proses utama algoritma simple locking terjadi di metode run(). Metode ini akan menerima argumen boolean yang mengindikasikan apakah program berjalan secara verbose atau tidak. Dapat dilihat bahwa secara garis besar, program akan berjalan terus hingga semua operasi dalam schedule berhasil dilakukan. Dalam prosesnya, terdapat dua kasus utama, yakni jika operasi bertipe read atau write, dan jika operasi bertipe commit. Jika operasi bertipe read atau write, maka akan diperiksa jika transaksi dari operasi tersebut sudah memiliki lock untuk data tersebut. Jika sudah, langsung tambahkan ke final schedule, namun jika belum, maka transaksi akan meminta lock. Jika locknya berhasil didapatkan, maka akan ditambahkan operasi lock dan operasi tadi ke dalam final schedule. Namun jika tidak berhasil diberikan, maka operasi tersebut beserta semua operasi lainnya dari transaksi yang sama di dalam schedule akan dimasukkan ke waiting queue. Jika operasi bertipe commit, maka semua lock yang dipegang oleh transaksi tersebut akan dilepaskan dan operasi commit dan unlock akan dimasukkan ke dalam final schedule. Jika ada operasi di waiting list, maka operasi tersebut akan dikembalikan ke depan schedule untuk dilaksanakan kembali. Setelah semua operasi di schedule sudah dilaksanakan, maka waiting queue akan diperiksa. Jika ada, maka pindahkan ke schedule dan jalankan kembali. Akan tetapi, jika program mendeteksi waiting list yang sama dari sebelumnya, maka terjadi deadlock.

2. OCC

OCC atau *Optimistic Concurrency Control* merupakan salah satu protokol yang digunakan oleh DBMS untuk mengatasi permasalahan *concurrency* pada *transaction*. Protokol ini dikategorikan sebagai “optimistik” dikarenakan protokol mengasumsikan seluruh transaksi berjalan secara *concurrent* ketika transaksi sedang berjalan. Dalam protokol ini, tidak dilakukan pengecekan saat transaksi dijalankan. Pembaruan dalam transaksi tidak diterapkan langsung ke *database* hingga akhir transaksi tercapai. Semua pembaruan diterapkan ke salinan lokal *item* data yang disimpan untuk transaksi. Di akhir eksekusi transaksi, terdapat fase validasi yang memeriksa apakah terdapat pembaruan transaksi yang melanggar *serializability*. Jika tidak ada pelanggaran *serializability* maka transaksi dieksekusi serta *database* diperbarui atau transaksi dilakukan *rollback* dan kemudian diulang kembali.

Screenshot:

a) Hasil percobaan

main.py

```
from process.Query import ReadQuery, WriteQuery, FunctionQuery,
DisplayQuery
from manager.SerialOptimistic import SerialOptimisticTransaction,
SerialOptimisticControl

if __name__ == "__main__":
    T25 = SerialOptimisticTransaction(25, [
        ReadQuery("binary/B"),
        DisplayQuery("binary/B", function=lambda B: B),
        WriteQuery("binary/B"),
        ReadQuery("binary/A"),
        DisplayQuery("binary/A", function=lambda A: A),
        WriteQuery("binary/A")
    ])

    T26 = SerialOptimisticTransaction(26, [
        ReadQuery("binary/B"),
        DisplayQuery("binary/B", function=lambda B: B),
        FunctionQuery("binary/B", function=lambda B: B + 50),
        DisplayQuery("binary/B", function=lambda B: B),
```

```

        WriteQuery("binary/B"),
        ReadQuery("binary/A"),
        DisplayQuery("binary/A", function=lambda A: A),
        FunctionQuery("binary/A", function=lambda A: A + 50),
        DisplayQuery("binary/A", function=lambda A: A),
        WriteQuery("binary/A")
    ])

    concurrencyManager = SerialOptimisticControl(
        [T25, T26],
        [25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 26, 25,
25, 25]
    )

    concurrencyManager.run()

```

Terminal

```

[BEGIN TRANSACTION 25]
[READ: binary/B]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[WRITE: binary/B]
[BEGIN TRANSACTION 26]
[READ: binary/B]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[FUNCTION: FunctionQuery("binary/B" function=lambda B: B + 50)]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[WRITE: binary/B]
[READ: binary/A]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[FUNCTION: FunctionQuery("binary/A" function=lambda A: A + 50)]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[WRITE: binary/A]
[ROLLBACK TRANSACTION 26]
[READ: binary/A]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[WRITE: binary/A]
[COMMIT TRANSACTION 25]
RESULT:
1200
1200
[BEGIN TRANSACTION 41]
[READ: binary/B]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]

```

```

[FUNCTION: FunctionQuery("binary/B" function=lambda B: B + 50)]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[WRITE: binary/B]
[READ: binary/A]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[FUNCTION: FunctionQuery("binary/A" function=lambda A: A + 50)]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[WRITE: binary/A]
[COMMIT TRANSACTION 41]
RESULT:
1200
1250
1200
1250

```

b) Analisis Algoritma

manager/SerialOptimistic.py

```

from __future__ import annotations
from typing import List, Set
import inspect
import sys

from manager.ConcurrencyControl import ConcurrencyControl
from process.Transaction import Transaction
from process.Query import Query, ReadQuery, WriteQuery,
DisplayQuery, FunctionQuery

class SerialOptimisticTransaction(Transaction):
    def __init__(self, startTimestamp: int, listOfQuery:
List[Query]) -> None:
        super().__init__(startTimestamp, listOfQuery)

        # Menambahkan end timestamp
        self.endTimestamp: int = sys.maxsize

        # Mencatat data items written dan data items read
        self.dataItemWritten: Set[str] = set()
        self.dataItemRead: Set[str] = set()

```



```

    def validationTest(self, counterTimestamp: int, other:
SerialOptimisticTransaction) -> bool:
        if self.startTimestamp <= other.startTimestamp:
            return True
        if self.startTimestamp >= other.endTimestamp:
            return True
        if self.startTimestamp < other.endTimestamp and
counterTimestamp >= other.endTimestamp and
self.dataItemRead.intersection(other.dataItemWritten) == set():
            return True
        return False

    def nextQuery(self) -> None:
        currentQuery = self.getCurrentQuery()
        if type(currentQuery) is WriteQuery:
            for filename in currentQuery.getFileNames():
                self.dataItemWritten.add(filename)
                print(f"[WRITE: {filename}]")

        if type(currentQuery) is ReadQuery:
            for filename in currentQuery.getFileNames():
                self.dataItemRead.add(filename)
                print(f"[READ: {filename}]")

        if type(currentQuery) is DisplayQuery:
            print(
                f"[DISPLAY:
{inspect.getsource(currentQuery.function).replace(' ',
'').strip()}]"
            )

        if type(currentQuery) is FunctionQuery:
            print(
                f"[FUNCTION:
{inspect.getsource(currentQuery.function).replace(' ',
'').strip()}]"
            )

        super().nextQuery()

```

```

def rollback(self, newTimestamp: int) -> None:
    self.dataItemWritten = set()
    self.dataItemRead = set()

    super().rollback(newTimestamp)

class SerialOptimisticControl(ConcurrencyControl):
    def __init__(self, listOfTransaction:
List[SerialOptimisticTransaction], schedule: List[int]) -> None:
        super().__init__(listOfTransaction, schedule)

    def run(self):
        tempSchedule: List[int] = [timestamp for timestamp in
self.schedule]
        activeTimestamp: List[int] = []
        counter = 0

        while tempSchedule:
            currentTimestamp = tempSchedule.pop(0)

            if currentTimestamp not in activeTimestamp:
                activeTimestamp.append(currentTimestamp)
                print(f"[BEGIN TRANSACTION {currentTimestamp}]")

                transaction: SerialOptimisticTransaction =
self.getTransaction(
                    currentTimestamp
                )

                transaction.nextQuery()

            if transaction.isFinished():
                valid = all(transaction.validationTest(
                    currentTimestamp +
                    counter, self.getTransaction(timestamp)
                ) for timestamp in activeTimestamp)

```

```

        if valid:
            print(
                f"[COMMIT TRANSACTION
{transaction.getStartTimestamp()}]"
            )
            print("RESULT:")
            transaction.commit()
            transaction.endTimestamp = currentTimestamp +
counter

        else:
            tempSchedule = list(filter(
                lambda X: X != currentTimestamp,
tempSchedule
            ))
            activeTimestamp = list(filter(
                lambda X: X != currentTimestamp,
activeTimestamp
            ))
            newTimestamp = currentTimestamp + \
                counter + len(tempSchedule)

            print(f"[ROLLBACK TRANSACTION
{currentTimestamp}]")
            transaction.rollback(newTimestamp)

            tempSchedule.extend(
                newTimestamp for _ in
range(transaction.getLength())
            )

            counter += 1

```

Pendekatan optimis pada algoritma OCC di atas tersebut didasarkan pada asumsi bahwa sebagian besar operasi tidak akan *conflict* antara satu dengan lainnya. Pendekatan optimis tersebut tidak memerlukan adanya mekanisme *lock* atau *timestamp*. Sebaliknya, transaksi dieksekusi tanpa dilakukan pengecekan *serializability* hingga

fase validasi. Dengan menggunakan pendekatan optimis, setiap transaksi bergerak melalui tiga fase, yaitu fase baca, fase validasi, dan fase penulisan.

(i) Selama fase baca, transaksi membaca *database*, kemudian mengeksekusi transaksi tersebut yang diperlukan, dan menyimpan hasil transaksi tersebut di dalam RAM atau *memory*. Semua operasi pemutakhiran transaksi dicatat dalam RAM atau *memory* tersebut tidak dapat diakses oleh transaksi lainnya.

(ii) Selama fase validasi, transaksi akan dicek untuk memastikan bahwa perubahan yang dilakukan tidak akan mempengaruhi integritas dan konsistensi *database* dengan memperhitungkan *write set* serta *read set* dari setiap transaksi yang terlibat dalam *schedule*. Jika transaksi valid, maka transaksi masuk ke fase penulisan. Jika transaksi, transaksi akan dilakukan *rollback* dan perubahan akan dihapus

(iii) Selama fase penulisan, perubahan diterapkan secara permanen ke dalam *database*.

3. MVCC

MVCC atau *Multi-Version Concurrency Control* merupakan salah satu protokol yang digunakan oleh DBMS untuk mengatasi permasalahan *concurrency* pada *transaction*. Skema *multi-version* memiliki arti bahwa *database* akan menyimpan versi lama dari seluruh data yang tercatat pada *database* tersebut yang dapat diakses oleh transaksi yang sedang berjalan untuk menambahkan *degree of concurrency* yang terjadi antara transaksi yang terlibat. Setiap operasi *write* yang berhasil ditambahkan pada *database* akan membentuk versi baru dari data tersebut. Selain itu, setiap versi dari data yang tercatat di dalam *database* akan diberi *write timestamp* dan *read timestamp* untuk mencatat versi mana yang boleh dibaca oleh transaksi. Ketika suatu operasi *read* membaca suatu data, maka versi yang disediakan oleh *concurrency controller* haruslah sesuai dengan *timestamp* dari mulainya transaksi tersebut agar konsistensi dan integritas *database* terjaga.

Screenshot:

a) Hasil percobaan

main.py

```
from process.Query import ReadQuery, WriteQuery, FunctionQuery,
DisplayQuery
```

```

from manager.MultiversionTimestampOrdering import
MultiversionTransaction, MultiversionControl

if __name__ == "__main__":
    T25 = MultiversionTransaction(25, [
        ReadQuery("binary/B"),
        DisplayQuery("binary/B", function=lambda B: B),
        WriteQuery("binary/B"),
        ReadQuery("binary/A"),
        DisplayQuery("binary/A", function=lambda A: A),
        WriteQuery("binary/A")
    ])

    T26 = MultiversionTransaction(26, [
        ReadQuery("binary/B"),
        DisplayQuery("binary/B", function=lambda B: B),
        FunctionQuery("binary/B", function=lambda B: B + 50),
        DisplayQuery("binary/B", function=lambda B: B),
        WriteQuery("binary/B"),
        ReadQuery("binary/A"),
        DisplayQuery("binary/A", function=lambda A: A),
        FunctionQuery("binary/A", function=lambda A: A + 50),
        DisplayQuery("binary/A", function=lambda A: A),
        WriteQuery("binary/A")
    ])

    concurrencyManager = MultiversionControl(
        [T25, T26],
        [25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 25,
25, 25]
    )

    concurrencyManager.run()

```

Terminal

```

[BEGIN TRANSACTION 25]
[READ: binary/B]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[WRITE: binary/B]
[BEGIN TRANSACTION 26]

```

```

[READ: binary/B]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[FUNCTION: FunctionQuery("binary/B" function=lambda B: B + 50)]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[WRITE: binary/B]
[READ: binary/A]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[FUNCTION: FunctionQuery("binary/A" function=lambda A: A + 50)]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[WRITE: binary/A]
[COMMIT TRANSACTION 26]
RESULT:
1250
1300
1250
1300
[READ: binary/A]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[ROLLBACK TRANSACTION 25]
[BEGIN TRANSACTION 27]
[READ: binary/B]
[DISPLAY: DisplayQuery("binary/B" function=lambda B: B)]
[WRITE: binary/B]
[READ: binary/A]
[DISPLAY: DisplayQuery("binary/A" function=lambda A: A)]
[WRITE: binary/A]
[COMMIT TRANSACTION 27]
RESULT:
1300
1300

```

b) Analisis Algoritma

manager/MultiversionTimestampOrdering.py

```

from typing import Callable, Dict, List
import inspect

from manager.ConcurrencyControl import ConcurrencyControl
from process.Transaction import Transaction
from process.Query import Query, ReadQuery, WriteQuery,
DisplayQuery, FunctionQuery
from process.FileAccess import FileAccess
from process.Data import Data

```

```

class Multiversion(Data):
    def __init__(self) -> None:
        self.value: int = 0
        self.writeTimestamp: int = 0
        self.readTimestamp: int = 0

    def getValue(self) -> int:
        return self.value

    def setValue(self, value: int) -> None:
        self.value = value

    def getWriteTimestamp(self) -> None:
        return self.writeTimestamp

    def setWriteTimestamp(self, writeTimestamp: int):
        self.writeTimestamp = writeTimestamp

    def getReadTimestamp(self) -> int:
        return self.readTimestamp

    def setReadTimestamp(self, readTimestamp: int) -> None:
        self.readTimestamp = readTimestamp

class MultiversionAccess(FileAccess):
    listOfVersion: Dict[str, List[Multiversion]] = dict()

    def __init__(self, filename: str) -> None:
        self.filename: str = filename
        self.initialValue = super().read()

    def refresh(self):
        self.initialValue = super().read()

    def read(self, timestamp: int) -> int:
        if
len(MultiversionAccess.listOfVersion.setdefault(self.filename,

```

```

[[])) == 0:

    multiversion = Multiversion()
    multiversion.setValue(self.initialValue)
    multiversion.setWriteTimestamp(0)
    multiversion.setReadTimestamp(timestamp)

MultiversionAccess.listOfVersion[self.filename].append(
    multiversion
)

    return self.initialValue

else:
    index = 0
    for i in
range(len(MultiversionAccess.listOfVersion[self.filename])):
        if
MultiversionAccess.listOfVersion[self.filename][i].writeTimestamp
<= timestamp:
            index = i
        else:
            break

    if
MultiversionAccess.listOfVersion[self.filename][index].readTimesta
mp < timestamp:

MultiversionAccess.listOfVersion[self.filename][index].readTimesta
mp = timestamp

    return
MultiversionAccess.listOfVersion[self.filename][index].getValue()

    def write(self, value: int, timestamp: int) -> None:
        if
len(MultiversionAccess.listOfVersion.setdefault(self.filename,
[[])) == 0:

            multiversion = Multiversion()
            multiversion.setValue(value)

```



```

        multiversion.setWriteTimestamp(timestamp)
        multiversion.setReadTimestamp(0)

    MultiversionAccess.listOfVersion[self.filename].append(
        multiversion
    )

    else:
        index = 0
        for i in
range(len(MultiversionAccess.listOfVersion[self.filename])):
            if
MultiversionAccess.listOfVersion[self.filename][i].writeTimestamp
<= timestamp:
                index = i
            else:
                break

        if timestamp <
MultiversionAccess.listOfVersion[self.filename][index].readTimesta
mp:
            raise Exception("ROLLBACK")

        elif timestamp ==
MultiversionAccess.listOfVersion[self.filename][index].writeTimest
amp:
MultiversionAccess.listOfVersion[self.filename][index].setValue(
    value
)

        else:
            multiversion = Multiversion()
            multiversion.setValue(value)
            multiversion.setWriteTimestamp(timestamp)
            multiversion.setReadTimestamp(0)

    MultiversionAccess.listOfVersion[self.filename].append(
        multiversion

```

```

    )

    def commit(self, timestamp: int):
        if
len(MultiversionAccess.listOfVersion.setdefault(self.filename,
[])) == 0:
            # Do nothing
            pass
        else:
            index = 0
            for i in
range(len(MultiversionAccess.listOfVersion[self.filename])):
                if
MultiversionAccess.listOfVersion[self.filename][i].writeTimestamp
<= timestamp:
                    index = i
                else:
                    break

            multiversion =
MultiversionAccess.listOfVersion[self.filename][index]
            value = multiversion.getValue()
            super().write(value)

class MultiversionQuery(Query):
    def execute(self, timestamp: int, *args: Data) -> None:
        raise NotImplementedError()

    def commit(self, timestamp: int, *args: Data):
        raise NotImplementedError()

class MultiversionReadQuery(MultiversionQuery, ReadQuery):
    def __init__(self, *args: str) -> None:
        super().__init__(*args)

    self.multiversionAccesses: List[MultiversionAccess] = []

```

```

        for filename in args:

self.multiversionAccesses.append(MultiversionAccess(filename))

    def execute(self, timestamp: int, *args: Data) -> None:
        for i in range(len(args)):
            data = args[i]
            multiversionAccess = self.multiversionAccesses[i]
            data.setValue(multiversionAccess.read(timestamp))

    def commit(self, timestamp: int, *args: Data):
        for i in range(len(args)):
            data = args[i]
            multiversionAccess = self.multiversionAccesses[i]
            data.setValue(multiversionAccess.read(timestamp))

class MultiversionWriteQuery(MultiversionQuery, WriteQuery):
    def __init__(self, *args: str) -> None:
        super().__init__(*args)

        self.multiversionAccesses: List[MultiversionAccess] = []

        for filename in args:

self.multiversionAccesses.append(MultiversionAccess(filename))

    def execute(self, timestamp: int, *args: Data) -> None:
        for i in range(len(args)):
            data = args[i]
            multiversionAccess = self.multiversionAccesses[i]
            multiversionAccess.write(data.getValue(), timestamp)

    def commit(self, timestamp: int, *args: Data):
        for i in range(len(args)):
            data = args[i]
            multiversionAccess = self.multiversionAccesses[i]
            multiversionAccess.write(data.getValue(), timestamp)
            multiversionAccess.commit(timestamp)

```

```

class MultiversionFunctionQuery(MultiversionQuery, FunctionQuery):
    def __init__(self, *args: str, **kwargs: Callable[..., int])
-> None:
        super().__init__(*args)
        self.function = kwargs.get("function", lambda *args:
args[0])

    def execute(self, timestamp: int, *args: Data) -> None:
        value: List[int] = []
        for data in args:
            value.append(data.getValue())
        args[0].setValue(self.function(*value))

    def commit(self, timestamp: int, *args: Data) -> None:
        value: List[int] = []
        for data in args:
            value.append(data.getValue())
        args[0].setValue(self.function(*value))

class MultiversionDisplayQuery(MultiversionQuery, DisplayQuery):
    def __init__(self, *args: str, **kwargs: Callable[..., int])
-> None:
        super().__init__(*args)
        self.function = kwargs.get("function", lambda *args: args)

    def execute(self, timestamp: int, *args: Data) -> None:
        pass

    def commit(self, timestamp: int, *args: Data) -> None:
        value: List[int] = []
        for data in args:
            value.append(data.getValue())

        result = self.function(*value)
        if type(result) is tuple:
            print(" ".join(map(str, result)))

```

```

        if type(result) is int:
            print(result)

class MultiversionTransaction(Transaction):
    def __init__(self, startTimestamp: int, listOfQuery:
List[Query]) -> None:
        newListOfQuery: List[Query] = []

        for query in listOfQuery:
            if type(query) is ReadQuery:
                newListOfQuery.append(
                    MultiversionReadQuery(*query.getFileNames())
                )
            elif type(query) is WriteQuery:
                newListOfQuery.append(
                    MultiversionWriteQuery(*query.getFileNames())
                )
            elif type(query) is FunctionQuery:
                newListOfQuery.append(
                    MultiversionFunctionQuery(
                        *query.getFileNames(),
function=query.function)
                )
            elif type(query) is DisplayQuery:
                newListOfQuery.append(
                    MultiversionDisplayQuery(
                        *query.getFileNames(),
function=query.function)
                )
            else:
                raise Exception("TYPE INVALID")

        super().__init__(startTimestamp, newListOfQuery)

    def nextQuery(self) -> None:
        currentQuery: MultiversionQuery = self.getCurrentQuery()
        currentFileNames = currentQuery.getFileNames()
        currentData: List[Data] = []

```

```

        for filename in currentFileNames:
            currentData.append(self.dictData.setdefault(filename,
Data()))

        currentQuery.execute(self.getStartTimestamp(),
*currentData)

        if type(currentQuery) is MultiversionWriteQuery:
            for filename in currentQuery.getFileNames():
                print(f"[WRITE: {filename}]")

        if type(currentQuery) is MultiversionReadQuery:
            for filename in currentQuery.getFileNames():
                print(f"[READ: {filename}]")

        if type(currentQuery) is MultiversionDisplayQuery:
            print(
                f"[DISPLAY:
{inspect.getsource(currentQuery.function).replace(',', '
').strip()}]"
            )

        if type(currentQuery) is MultiversionFunctionQuery:
            print(
                f"[FUNCTION:
{inspect.getsource(currentQuery.function).replace(',', '
').strip()}]"
            )

        super().nextQuery()

def rollback(self, newTimestamp) -> None:
    newListOfQuery: List[MultiversionQuery] = []

    for query in self.listOfQuery:
        if type(query) is MultiversionReadQuery:
            newListOfQuery.append(
                MultiversionReadQuery(*query.getFileNames())

```

```

        )
        elif type(query) is MultiversionWriteQuery:
            newListOfQuery.append(
                MultiversionWriteQuery(*query.getFileNames())
            )
        elif type(query) is MultiversionFunctionQuery:
            newListOfQuery.append(
                MultiversionFunctionQuery(
                    *query.getFileNames(),
function=query.function)
            )
        elif type(query) is MultiversionDisplayQuery:
            newListOfQuery.append(
                MultiversionDisplayQuery(
                    *query.getFileNames(),
function=query.function)
            )
        else:
            raise Exception("TYPE INVALID")

        self.listOfQuery = newListOfQuery

        super().rollback(newTimestamp)

    def commit(self) -> None:
        MultiversionAccess.listOfVersion.clear()

        for i in range(self.getLength()):
            currentQuery: MultiversionQuery = self.listOfQuery[i]
            currentFileNames = currentQuery.getFileNames()
            currentData: List[Data] = []

            for filename in currentFileNames:
currentData.append(self.dictData.setdefault(filename, Data()))

            currentQuery.commit(self.getStartTimestamp(),
*currentData)

```

```

class MultiversionControl(ConcurrencyControl):
    def __init__(self, listOfTransaction:
List[MultiversionTransaction], schedule: List[int]) -> None:
        super().__init__(listOfTransaction, schedule)

    def run(self):
        tempSchedule: List[int] = [timestamp for timestamp in
self.schedule]

        activeTimestamp: List[int] = []
        maxTimestamp = 0

        while tempSchedule:
            currentTimestamp = tempSchedule.pop(0)

            if currentTimestamp not in activeTimestamp:
                activeTimestamp.append(currentTimestamp)
                maxTimestamp = max(maxTimestamp, currentTimestamp)

            print(f"[BEGIN TRANSACTION {currentTimestamp}]")

            transaction: MultiversionTransaction =
self.getTransaction(
                currentTimestamp
            )

            try:
                transaction.nextQuery()

                if transaction.isFinished():
                    print(
                        f"[COMMIT TRANSACTION
{transaction.getStartTimestamp()}]"
                    )
                    print("RESULT:")
                    transaction.commit()

            except:
                tempSchedule = list(filter(

```



```

        lambda X: X != currentTimestamp, tempSchedule
    ))
    activeTimestamp = list(filter(
        lambda X: X != currentTimestamp,
activeTimestamp
    ))

    newTimestamp = maxTimestamp + 1

    print(f"[ROLLBACK TRANSACTION
{currentTimestamp}]")
    transaction.rollback(newTimestamp)

    tempSchedule.extend(
        newTimestamp for _ in
range(transaction.getLength())
    )

```

Pendekatan *multi-version* pada algoritma MVCC di atas tersebut mengharuskan *database* untuk menyimpan beberapa versi dari setiap data yang dibaca dan diubah oleh transaksi. Selain itu, setiap versi dari data tersebut juga terdapat *write timestamp* yang menyatakan *timestamp* saat terakhir kali data tersebut diubah oleh suatu transaksi serta *read timestamp* yang menyatakan *timestamp* saat terakhir kali data tersebut dibaca oleh suatu transaksi. Pada operasi *read*, *database* akan menyediakan data dengan versi terbaru namun masih lebih tua dibandingkan *timestamp* transaksi dan transaksi akan mengubah *read timestamp* dari data tersebut. Pada operasi *write*, terlebih dahulu *database* akan melakukan pengecekan mengenai apakah versi terbaru dari data tersebut telah dibaca oleh transaksi dengan *timestamp* yang lebih baru dibandingkan operasi *write* tersebut. Jika hal tersebut terbukti benar, maka transaksi yang mengandung operasi *write* tersebut akan dilakukan *rollback* dan diulang dengan *timestamp* yang berbeda. Jika hal tersebut tidak terbukti benar, maka transaksi akan menuliskan data pada versi baru yang disediakan oleh *database*.

C. Eksplorasi Recovery

Penjelasan *Write-Ahead Log*, *Continuous Archiving*, dan *Point-In-Time Recovery*

1. *Write-Ahead Log*

Write-Ahead Log (WAL) adalah salah satu mekanisme *recovery* yang tersedia dalam basis data PostgreSQL menggunakan *Transaction Log* atau *Log*. *Transaction Log* ini mencatat semua aktivitas perubahan nilai suatu *record* dalam basis data. *Write-Ahead Log* dilakukan dengan cara setiap perubahan yang akan dilakukan terhadap basis data harus tercatat terlebih dahulu di dalam *log file*, dan *log file* tersebut harus dapat tersimpan dengan aman di dalam *disk* atau *stable storage*.

Dengan menggunakan *Write-Ahead Log*, jika terjadi *crash* baik pada sistem operasi, PostgreSQL, ataupun *hardware*, maka basis data dapat dilakukan *recovery* dengan melihat *log* transaksi yang sedang berjalan ketika sistem mengalami *crash*. Pada PostgreSQL, file *Write-Ahead Log* disimpan di *pg_wal/ subdirectory* pada *directory* kluster data.

2. *Continuous Archiving*

Continuous Archiving adalah mekanisme tambahan yang dapat dilakukan pada *Write-Ahead Log* (WAL) File. Dengan menggunakan *Continuous Archiving*, maka file WAL ini akan di-copy ke tempat penyimpanan yang lebih stabil (*persistent*). Hasil *copy*-an ini dapat digunakan untuk melakukan *Point-in-Time Recovery*.

Untuk melakukan WAL *Archiving*, terlebih dahulu harus dilakukan beberapa konfigurasi, seperti mengubah *wal_level* menjadi *archive*, dan menspesifikasikan shell command yang digunakan di dalam konfigurasi parameter *archive_command*.

3. *Point-in-Time Recovery*

Point-in-Time Recovery (PITR) adalah suatu cara yang dapat dilakukan oleh *database administrator* untuk melakukan *restore* atau *recover* sekumpulan data atau database pada waktu tertentu di masa lalu. PITR dapat digunakan ketika seseorang dengan tidak sengaja menghapus *table* atau *records* di dalam basis data atau sesuatu hal salah terjadi dan menyebabkan basis data menjadi rusak. Untuk melakukan PITR pada PostgreSQL, terlebih dahulu harus dilakukan *backing up* dari basis data yang sedang berjalan dan melakukan *archiving* file WAL.

Tahapan Konfigurasi pada PostgreSQL

Note : Proses ini dilakukan pada sistem operasi berbasis Linux dan PostgreSQL 14

a. Melakukan konfigurasi Continuous Archiving pada kluster basis data

Membuat direktori baru untuk menyimpan *archive* dari file WAL

```
$ mkdir database_archive
```

Berikan akses izin untuk menulis file kepada *default user* PostgreSQL.

```
$ sudo chown postgres:postgres database_archive
```

Lakukan konfigurasi pada file *postgresql.conf* untuk memperbolehkan *archiving*. Membuka file *postgresql.conf* dapat dilakukan dengan perintah berikut.

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

Setelah membuka file konfigurasi, cari variabel bernama *archive_mode*. Lakukan *uncomment* dengan menghilangkan simbol *#* pada awal baris. Ubah nilai *archive_mode* tersebut menjadi *on* seperti dibawah ini.

```
. . .  
archive_mode = on  
. . .
```

Lakukan pula konfigurasi untuk mengatur perintah yang harus dilakukan klaster basis data untuk melakukan *archiving*. Cari variabel bernama *archive_command*. Lakukan *uncomment* dengan menghilangkan simbol *#* pada awal baris. Ubah nilai *archive_command* tersebut menjadi *on* seperti dibawah ini.

```
. . .  
archive_command = 'test ! -f /path/to/database_archive/%f && cp %p  
/path/to/database_archive/%f'  
. . .
```

Perintah di atas melakukan operasi untuk mengecek apakah file WAL sudah ada di dalam *archive*. Jika belum, maka akan melakukan *copy* file WAL ke dalam direktori *archive*. Ganti nilai */path/to/database_archive* dengan *path* menuju direktori *database_archive* yang tadi sudah dibuat.

Terakhir, lakukan konfigurasi pada variabel *wal_level*. Variabel tersebut digunakan untuk menyatakan seberapa banyak informasi yang harus dituliskan ke dalam *log*. Untuk *continuous archiving*, maka nilai variabel *wal_level* minimal harus *replica*.

```
. . .  
wal_level = replica  
. . .
```

Sekarang simpan perubahan konfigurasi dan keluar dari file konfigurasi tersebut. Untuk mengimplementasikan perubahan pada file konfigurasi, maka diperlukan *restart* klaster basis data dengan perintah dibawah ini.

```
$ sudo systemctl restart postgresql@14-main
```

Setelah sukses melakukan *restart*, maka klaster basis data akan melakukan *archiving* setiap file WAL apabila sudah penuh. Secara *default*, setiap file WAL memiliki kapasitas WAL file sebesar 16MB.

Jika ingin melakukan *archiving* segera, bisa dilakukan suatu perintah agar memaksa klaster basis data untuk berubah dan meng-*archive* file WAL saat ini. Perintah yang dimaksud adalah sebagai berikut.

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

Setelah klaster basis data sukses melakukan *archiving* dengan meng-*copy* file WAL ke dalam *archive*, kita bisa melakukan *backup* secara fisik setiap file data dari klaster basis data.

b. Melakukan *Physical Backup* dari kluster basis data PostgreSQL

Sangat penting untuk melakukan *backup* secara rutin untuk mencegah apabila ada hal-hal yang tak diinginkan terjadi. Untuk melakukan *Point-in-Time Recovery*, PostgreSQL mengharuskan kita untuk melakukan *physical backup* klaster basis data. *Physical backup* tersebut dilakukan dengan cara meng-copy semua file basis data dari direktori PostgreSQL.

Pada tahap sebelumnya, kita sudah membuat direktori *database_archive* untuk menyimpan seluruh *archive* file WAL. Pada tahap ini kita perlu membuat direktori baru bernama *database_backup* untuk menyimpan *physical backup* dari basis data.

Buat direktori baru bernama *database_backup*.

```
$ mkdir database_backup
```

Berikan izin untuk melakukan *write* pada direktori *database_backup* kepada *user* postgres.

```
$ sudo chown postgres:postgres database_backup
```

Lakukan *physical backup* pada basis data PostgreSQL dengan perintah sebagai berikut.

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

Ubah nilai */path/to/* dengan *path* menuju direktori *database_backup* tersimpan. Dengan melakukan *physical backup* dari klaster basis data, sekarang kita dapat melakukan *Point-in-Time Recovery*.

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

Setelah melakukan *physical backup* dan melakukan *archiving* file WAL, sekarang kita dapat melakukan *Point-In Time Recovery* (PITR) jika kita ingin melakukan *rollback* basis data.

Apabila PostgreSQL masih berjalan, maka kita perlu mematikannya terlebih dahulu. Proses ini dapat dilakukan dengan perintah sebagai berikut.

```
$ sudo systemctl stop postgresql@14-main
```

Setelah mematikan PostgreSQL, kita perlu menghapus semua file pada direktori data PostgreSQL. Akan tetapi, kita perlu memindahkan direktori *pg_wal* ke tempat lain karena mungkin masih mengandung *unarchived* file WAL yang penting untuk proses *recovery*.

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

Sekarang kita dapat menghapus semua file pada direktori data PostgreSQL dan membuat direktori baru dengan isi kosong.

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

Lalu, *copy* semua file *physical backup* yang sudah kita buat pada tahap sebelumnya ke dalam direktori PostgreSQL yang masih kosong tadi.

```
$ sudo cp -a /path/to/database_backup/. /var/lib/postgresql/14/main/
```

Pastikan direktori PostgreSQL tadi sudah diberikan akses izin untuk *user postgres*.

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

File WAL pada direktori *pg_wal* yang di-*copy* dari *physical backup* sudah usang dan tidak berguna. Kita perlu mengganti file tersebut dengan file WAL yang sudah kita *copy* terlebih dahulu sebelum menghapus direktori data PostgreSQL.

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

Setelah melakukan *restore* direktori data PostgreSQL, kita perlu melakukan konfigurasi pada pengaturan *recovery* untuk memastikan *server* basis data me-*recover* file WAL yang di-*archive* dengan benar. Konfigurasi *recovery settings* dapat ditemukan pada file *postgresql.conf*. Buka file konfigurasi dengan perintah berikut.

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

Kemudian cari variabel bernama *restore_command* dan hilangkan karakter *#* untuk melakukan *uncomment*. Variabel *restore_command* mirip seperti variabel *archive_command* yang ada pada tahap sebelumnya. Jika *archive_command* melakukan copy ke archive, maka *restore_command* melakukan restore ke dalam basis data.

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

Jangan lupa mengganti */path/to/database_archive/* dengan *path* ke direktori *archive*. Selanjutnya, kita harus menspesifikasikan kumpulan variabel *recovery_target* untuk menentukan “patokan” bagi klaster basis data saat melakukan *recovery*. Selengkapnya ada di dalam dokumentasi berikut [PostgreSQL Recovery Target](#). Jika tidak dispesifikasikan, maka klaster basis data akan membaca seluruh *log* pada file WAL yang ada di dalam *archive*. Setelah mengatur variabel *restore_command* dan *recovery_target*, simpan perubahan dan keluar dari file konfigurasi.

Sebelum melakukan *restart* klaster basis data, kita perlu memberitahukan kepada PostgreSQL untuk memulai pada mode *recovery*. Untuk melakukannya, kita perlu membuat file kosong pada direktori data PostgreSQL bernama *recovery.signal*.

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

Sekarang kita dapat melakukan *restart* klaster basis data dengan perintah berikut.

```
$ sudo systemctl start postgresql@14-main
```

Jika basis data sukses dijalankan, basis data akan masuk ke dalam mode *recovery*. Ketika sudah mencapai target dari proses *recovery*, file *recovery.signal* akan secara otomatis terhapus. Untuk melihat proses *recovery* yang terjadi, kita bisa melihat *log* dengan perintah sebagai berikut.

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

Simulasi Kegagalan pada PostgreSQL dan Proses Recovery

Basis data yang digunakan adalah [nation.sql](#)

Data tabel awal:

```
dikyrest@LAPTOP-BVV7FVGB:~$ psql -U postgres
psql (14.6 (Ubuntu 14.6-1.pgdg20.04+1))
Type "help" for help.

postgres=# \c nation
You are now connected to database "nation" as user "postgres".

nation=# \dt
          List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | _continents    | table | postgres
public | _countries     | table | postgres
public | _country_languages | table | postgres
public | _country_stats | table | postgres
public | _languages     | table | postgres
public | _regions      | table | postgres
(6 rows)

nation=#
```

Gambar 3.1 Daftar Tabel pada Database *nation*

Database Administrator melakukan *physical backup* klaster basis data dengan perintah berikut.

```
dikyrest@LAPTOP-BVV7FVGB:~$ mkdir ~/database_backup
dikyrest@LAPTOP-BVV7FVGB:~$ sudo chown postgres:postgres ~/database_backup
dikyrest@LAPTOP-BVV7FVGB:~$ sudo -u postgres pg_basebackup -D ~/database_backup
dikyrest@LAPTOP-BVV7FVGB:~$
```

Gambar 3.2 Backup Database

Kemudian, seorang *user* yang tidak bertanggung jawab menghapus tabel *_countries*.

```
nation=# drop table _countries;
DROP TABLE
nation=#
```

Gambar 3.3 Penghapusan Tabel *_countries*

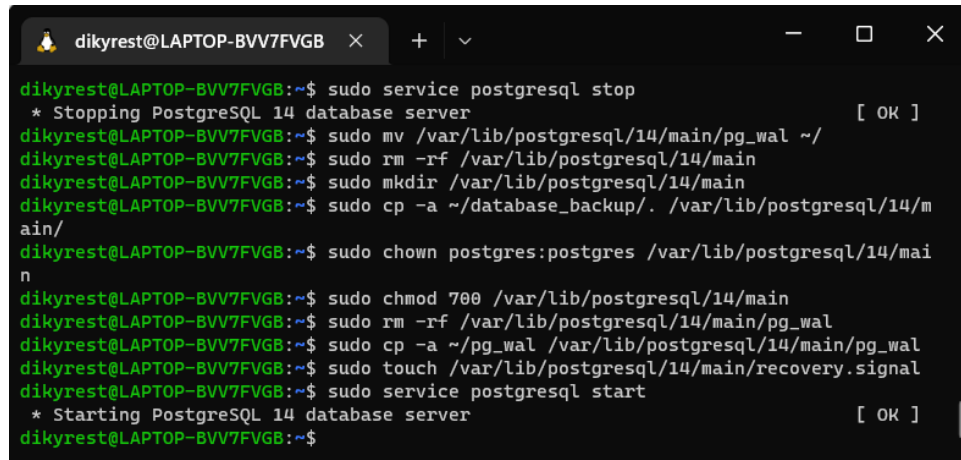
```
dikyrest@LAPTOP-BVV7FVGB:~$ psql -U postgres
psql (14.6 (Ubuntu 14.6-1.pgdg20.04+1))
Type "help" for help.

postgres=# \dt
          List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | _continents    | table | postgres
public | _country_languages | table | postgres
public | _country_stats | table | postgres
public | _languages     | table | postgres
public | _regions      | table | postgres
(5 rows)

(END)
```

Gambar 3.4 Daftar Tabel setelah Penghapusan

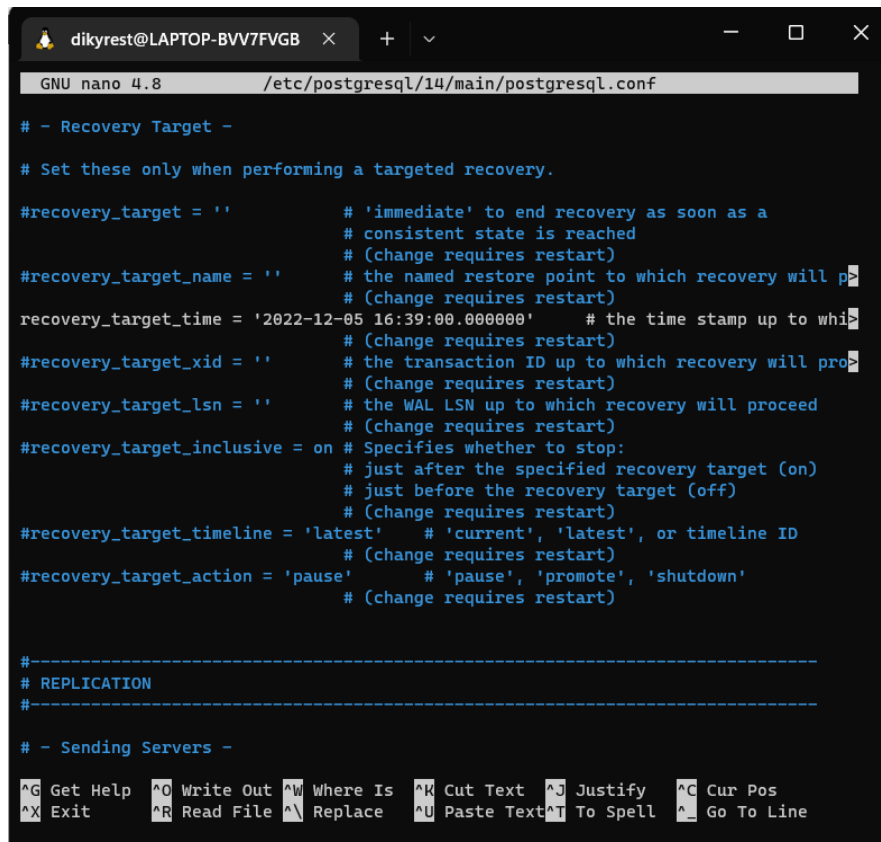
Database Administrator kemudian melakukan proses *restore* dari file yang sudah di-backup sebagai berikut.

A terminal window titled 'dikyrest@LAPTOP-BVV7FVGB' showing a series of commands to stop the PostgreSQL service, move the old WAL files, remove the old main directory, create a new main directory, copy backup files, set permissions, and start the service again. The output shows the service stopping and starting successfully.

```
dikyrest@LAPTOP-BVV7FVGB:~$ sudo service postgresql stop
* Stopping PostgreSQL 14 database server [ OK ]
dikyrest@LAPTOP-BVV7FVGB:~$ sudo mv /var/lib/postgresql/14/main/pg_wal ~/
dikyrest@LAPTOP-BVV7FVGB:~$ sudo rm -rf /var/lib/postgresql/14/main
dikyrest@LAPTOP-BVV7FVGB:~$ sudo mkdir /var/lib/postgresql/14/main
dikyrest@LAPTOP-BVV7FVGB:~$ sudo cp -a ~/database_backup/. /var/lib/postgresql/14/main/
dikyrest@LAPTOP-BVV7FVGB:~$ sudo chown postgres:postgres /var/lib/postgresql/14/main
dikyrest@LAPTOP-BVV7FVGB:~$ sudo chmod 700 /var/lib/postgresql/14/main
dikyrest@LAPTOP-BVV7FVGB:~$ sudo rm -rf /var/lib/postgresql/14/main/pg_wal
dikyrest@LAPTOP-BVV7FVGB:~$ sudo cp -a ~/pg_wal /var/lib/postgresql/14/main/pg_wal
dikyrest@LAPTOP-BVV7FVGB:~$ sudo touch /var/lib/postgresql/14/main/recovery.signal
dikyrest@LAPTOP-BVV7FVGB:~$ sudo service postgresql start
* Starting PostgreSQL 14 database server [ OK ]
dikyrest@LAPTOP-BVV7FVGB:~$
```

Gambar 3.5 *Restore Database*

Isi dari variabel *recovery_target* adalah sebagai berikut. Pemilihan *recovery_target_time* yang demikian dikarenakan pada waktu sekian basis data belum mengalami kerusakan akibat dihapusnya tabel *_countries* oleh user yang tidak bertanggung jawab.

A terminal window showing the configuration of the PostgreSQL recovery target in the postgresql.conf file. The file is opened with nano 4.8. The configuration includes recovery_target_time set to '2022-12-05 16:39:00.000000'.

```
GNU nano 4.8 /etc/postgresql/14/main/postgresql.conf

# - Recovery Target -

# Set these only when performing a targeted recovery.

#recovery_target = ''           # 'immediate' to end recovery as soon as a
                                # consistent state is reached
                                # (change requires restart)
#recovery_target_name = ''      # the named restore point to which recovery will p
                                # (change requires restart)
recovery_target_time = '2022-12-05 16:39:00.000000' # the time stamp up to whi
                                # (change requires restart)
#recovery_target_xid = ''       # the transaction ID up to which recovery will pro
                                # (change requires restart)
#recovery_target_lsn = ''       # the WAL LSN up to which recovery will proceed
                                # (change requires restart)
#recovery_target_inclusive = on # Specifies whether to stop:
                                # just after the specified recovery target (on)
                                # just before the recovery target (off)
                                # (change requires restart)
#recovery_target_timeline = 'latest' # 'current', 'latest', or timeline ID
                                # (change requires restart)
#recovery_target_action = 'pause' # 'pause', 'promote', 'shutdown'
                                # (change requires restart)

#-----
# REPLICATION
#-----

# - Sending Servers -

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Paste Text ^T To Spell  ^_ Go To Line
```

Gambar 3.6 *Konfigurasi Recovery Target*

Kemudian, basis data berhasil *ter-restore* dengan sukses.

```
dikyrest@LAPTOP-BVV7FVGB x + v
2022-12-05 16:47:25.319 WIB [7179] LOG: starting PostgreSQL 14.6 (Ubuntu 14.6-1.pgdg20.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0, 64-bit
2022-12-05 16:47:25.321 WIB [7179] LOG: listening on IPv4 address "127.0.0.1", port 5432
2022-12-05 16:47:25.324 WIB [7179] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2022-12-05 16:47:25.332 WIB [7180] LOG: database system was interrupted; last known up at 2022-12-05 16:40:00 WIB
cp: cannot stat '~/database_archive/00000002.history': No such file or directory
2022-12-05 16:47:25.703 WIB [7180] LOG: starting point-in-time recovery to 2022-12-05 16:39:00+07
cp: cannot stat '~/database_archive/000000010000000000000000000004': No such file or directory
2022-12-05 16:47:25.710 WIB [7180] LOG: redo starts at 0/4000028
2022-12-05 16:47:25.713 WIB [7180] LOG: consistent recovery state reached at 0/4000138
2022-12-05 16:47:25.713 WIB [7179] LOG: database system is ready to accept read-only connections
cp: cannot stat '~/database_archive/000000010000000000000000000005': No such file or directory
2022-12-05 16:47:25.716 WIB [7180] LOG: recovery stopping before commit of transaction 757, time 2022-12-05 16:44:46.211286+07
2022-12-05 16:47:25.716 WIB [7180] LOG: pausing at the end of recovery
2022-12-05 16:47:25.716 WIB [7180] HINT: Execute pg_wal_replay_resume() to promote
dikyrest@LAPTOP-BVV7FVGB:~$
```

Gambar 3.7 Isi *Transaction Log*

```
dikyrest@LAPTOP-BVV7FVGB x + v
dikyrest@LAPTOP-BVV7FVGB:~$ psql -U postgres
psql (14.6 (Ubuntu 14.6-1.pgdg20.04+1))
Type "help" for help.

postgres=# \c nation
You are now connected to database "nation" as user "postgres".
nation=# \dt
          List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | _continents    | table | postgres
 public | _countries     | table | postgres
 public | _country_languages | table | postgres
 public | _country_stats | table | postgres
 public | _languages     | table | postgres
 public | _regions       | table | postgres
(6 rows)
```

Gambar 3.8 Daftar Tabel setelah *Restore*

Pada *log* dapat terlihat bahwa basis data mulai melakukan *redo* pada 0/4000028 dan berhasil mencapai *consistent recovery state* pada 0/4000138.

Kesimpulan dan Saran

Terdapat empat tingkatan level isolasi transaksi, yaitu, dari yang paling ketat, *serializable*, *repeatable read*, *read committed*, dan *read uncommitted*. Keempat level isolasi transaksi tersebut didefinisikan dengan fenomena apa saja yang diperbolehkan terjadi. Fenomena-fenomena tersebut yaitu fenomena *dirty read*, *nonrepeatable read*, *phantom read*, dan *serialization anomaly*.

Pada PostgreSQL, level isolasi transaksi yang dapat digunakan hanya tiga, yaitu *serializable*, *repeatable read*, dan *read committed*, dengan *default* level isolasi transaksi *read committed*. Level isolasi *read uncommitted* diimplementasikan sama seperti *read committed*. Level isolasi transaksi dapat diatur untuk setiap transaksi menggunakan *query SET TRANSACTION ISOLATION LEVEL <level isolasi transaksi>*.

Terdapat tiga algoritma yang diimplementasikan, yaitu algoritma *simple locking*, *optimistic concurrency control*, dan *multi-version concurrency control*. Setiap algoritma tersebut memiliki kelebihan dan kekurangannya masing - masing. Pertama - tama, algoritma *simple locking* memiliki kelebihan utama yaitu cukup mudah diimplementasikan oleh *database*. Namun, algoritma *simple locking* umumnya tidak menghasilkan *schedule* dengan *degree of concurrency* yang tinggi. Kemudian, algoritma *optimistic concurrency control* memiliki kelebihan utama yaitu setiap transaksi dapat dieksekusi tanpa harus memperhitungkan setiap operasi yang dijalankan oleh *scheduler*. Tetapi, algoritma *optimistic concurrency control* membutuhkan *memory overhead size* yang cukup besar dikarenakan adanya pencatatan *read set* dan *write set* dari setiap transaksi yang konkuren. Terakhir, algoritma *multi-version concurrency control* memiliki kelebihan utama yaitu memungkinkan untuk mencapai *degree of concurrency* yang tinggi dikarenakan menggunakan versi data yang cukup bebas. Kekurangan dari algoritma *multi-version concurrency control* adalah membutuhkan *disk size* yang besar dikarenakan adanya pencatatan setiap versi dari data yang tersimpan.

Recovery adalah proses pemulihan sistem basis data saat terjadi suatu kegagalan sistem agar kembali ke keadaan yang konsisten. Kegagalan tersebut dapat diakibatkan oleh sistem yang mengalami *crash* karena kesalahan perangkat keras atau perangkat lunak, kegagalan media seperti *head crash*, atau kesalahan perangkat lunak dalam aplikasi seperti kesalahan logis pada program dalam pengaksesan basis data.

Pada sistem basis data PostgreSQL, terdapat beberapa macam mekanisme untuk melakukan proses *recovery*, antara lain *Write-Ahead Log*, *Continuous Archiving*, dan *Point-in-Time Recovery*. Ketiga mekanisme tersebut memiliki caranya masing-masing dan ada keterhubungan. Sebagai contoh, mekanisme *Continuous Archiving* memanfaatkan *Write-Ahead Log* dalam prosesnya. Begitupun dengan *Point-in-Time Recovery* yang dapat memanfaatkan *Continuous Archiving* dalam melakukan proses *recovery* hingga mencapai suatu titik waktu tertentu.

Untuk pengerjaan tugas besar ini, sebaiknya diberikan basis data yang dapat digunakan untuk melakukan simulasi dan implementasi *concurrency control* dan *recovery* agar eksplorasi dapat dilakukan dengan lebih mudah dan terstandar. Selain itu, *source code* yang telah dibuat sebelumnya dapat lebih ditekankan untuk mengikuti *design pattern* OOP.

Pembagian Kerja Kelompok

NIM	Nama	Kerja
13520017	Diky Restu Maulana	Eksplorasi <i>recovery</i>
13520050	Felicia Sutandijo	Eksplorasi <i>concurrency control</i>
13520065	Rayhan Kinan Muhannad	Implementasi <i>concurrency control</i> : <i>Framework Python, OCC, MVCC</i>
13520168	Muhammad Naufal Satriandana	Implementasi <i>concurrency control</i> : <i>simple locking</i>

Referensi

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

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

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

<https://www.digitalocean.com/community/tutorials/how-to-set-up-continuous-archiving-and-perform-point-in-time-recovery-with-postgresql-12-on-ubuntu-20-04>