IF3140 MANAJEMEN BASIS DATA

MEKANISME CONCURRENCY CONTROL DAN RECOVERY

K02 Kelompok 12

Anggota :

Kevin John Wesley Hutabarat	13521042
Naufal Syifa Firdaus	13521050
Ahmad Ghulam Ilham	13521118
I Putu Bakta Hari Sudewa	13521150

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

1. Eksplorasi Transaction Isolation

Isolasi transaksi adalah konsep pada sistem basis data yang adalah suatu transaksi tidak mempengaruhi dan dipengaruhi oleh transaksi lain yang sedang berjalan pada waktu yang bersamaan. Isolasi ditujukan untuk memastikan hasil dari transaksi tidak merusak atau mengubah hasil dari transaksi lain. Konsep ini diterapkan agar data dalam sistem konsisten dan terjaga integritasnya.

Pada eksplorasi ini akan dilakukan sejumlah simulasi untuk menunjukan kekuatan dan kelemahan bagi setiap tingkat isolasi. Berikut merupakan entitas pada basis data yang digunakan untuk keperluan simulasi pada tiap tingkatannya.

	person
PK	id (integer) name (varchar 255) job (varchar 255)

a. Read Uncommitted

- **Deskripsi**

  Berikut adalah tingkat isolasi transaksi paling rendah. Transaksi tidak dibatasi dalam proses *read* untuk membaca data apapun termasuk data yang sedang dimodifikasi oleh transaksi lain dan belum *commit*. *Commit* diperlukan untuk membuat perubahan yang dilakukan oleh transaksi menjadi permanen. Jika sebuah transaksi A membaca data yang diubah oleh transaksi B, ketika transaksi B yang mengubah data tersebut dibatalkan maka data tidak jadi berubah tapi transaksi A sudah terlanjur membaca data yang diubah sehingga data menjadi tidak valid. Fenomena tersebut disebut *dirty read*.

- **Dampak**

  Tingkat Isolasi *read uncommitted* memiliki performa yang paling tinggi karena semua transaksi dapat berjalan secara paralel dan bersamaan dan mengakses data apapun pada basis data tanpa restriksi seperti *lock*. Namun, hal ini

berdampak buruk pada integritas dan validitas transaksi. Salah satunya seperti fenomena *dirty read* yang dideskripsikan sebelumnya. Selain itu, *non-repeatable read* juga dapat terjadi. Fenomena ini adalah ketika transaksi A membaca sebuah data, kemudian data tersebut diubah dan di-*commit* oleh transaksi B sebelum transaksi A selesai. Lalu ketika transaksi A yang belum selesai kembali lagi membaca data yang sama, data tersebut berbeda dari data awal yang pertama dibaca karena telah diubah oleh transaksi B. Fenomena *phantom read* juga sangat mungkin terjadi pada tingkat isolasi transaksi yang paling rendah ini. *Phantom read* adalah kejadian dimana transaksi A membaca barisan baris sesuai querynya dan transaksi B melakukan *insert* atau *update* pada tabel sehingga ketika transaksi A mengeksekusi query pembacaan kembali data yang ditampilkan bertambah atau berubah pada tabel.

- **Contoh Kasus**

b. Read Committed

- **Deskripsi**

  Pada tingkat isolasi *read committed,* data yang sedang diubah oleh transaksi A tidak bisa dibaca oleh transaksi lainnya sebelum transaksi A *commit* perubahan tersebut. Hal ini mencegah terjadinya *dirty read* sehingga data yang dibaca oleh semua transaksi adalah data yang dijamin sudah *commit* dan permanen.

- **Dampak**

  Performa dari tingkat isolasi ini menurun daripada tingkat isolasi *read uncommitted* karena ada mekanisme penguncian pada data yang belum di-*commit* oleh sebuah transaksi sehingga transaksi lain yang ingin membaca data tersebut harus menunggu. Walaupun begitu, *non-repeatable read* dan *phantom read* masih dapat terjadi pada tingkat ini. Hal itu dikarenakan pada saat transaksi memproses sebuah data, sebelum transaksi tersebut selesai, transaksi lain dapat mengubah data tersebut dan *commit* sebelum transaksi awal selesai.

- **Simulasi**

  Pada simulasi dibawah ini transaksi pertama belum menyelesaikan prosesnya sedangkan transaksi kedua mengubah data yang sama secara bersamaan. *Non-repeatable read* terjadi pada simulasi berikut, transaksi satu membaca data dua kali dan pada tiap pembacaannya data yang didapatkan berbeda.

Transaksi 1

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
sample=*# SELECT * FROM person WHERE id = 1;
 id | name |     job
----+------+--------------
  1 | adul | pengangguran
(1 row)


sample=*# SELECT * FROM person WHERE id = 1;
 id | name |   job
----+------+---------
  1 | adul | pelawak
(1 row)


sample=*# COMMIT;
COMMIT
sample=#
```

Transaksi 2

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
sample=*# UPDATE person SET job = 'pelawak'  WHERE id = 1;
UPDATE 1
sample=*# COMMIT;
COMMIT
sample=#
```

Simulasi berikutnya transaksi 2 mengubah data id satu dan transaksi 1 membaca data yang sama ketika transaksi 2 belum *commit*. Namun karena tingkat isolasinya adalah *read committed*, transaksi satu membaca data dengan kondisi yang sebelum diubah oleh transaksi 2 karena belum *commit*.

Transaksi 1

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
sample=*# SELECT * FROM person WHERE id = 1;
 id | name |   job
----+------+---------
  1 | adul | pelawak
(1 row)


sample=*#
```

Transaksi 2

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN
sample=*# UPDATE person SET job = 'politisi'  WHERE id = 1;
UPDATE 1
sample=*#
```

c. Repeatable Read

- **Deskripsi**

  Tingkat isolasi *repeatable read* tidak memperbolehkan transaksi lain menulis, memperbarui, atau membaca data yang sedang diakses sebuah transaksi yang sedang berlangsung. Hal ini mengatasi permasalah *dirty write* dan *non-repeatable read* dalam pengimplementasiannya. Ketika sebuah transaksi mengakses data, sistem basis data akan menerapkan *lock* pada data tersebut sehingga transaksi lain tidak bisa membaca, menulis, dan memperbarui data tersebut.

- **Dampak**

Mekanisme *lock* pada data yang sedang dalam transaksi merestriksi konkurensi dari transaksi lainnya yang menggunakan data yang sama, oleh karena itu, performa dari tingkat isolasi ini lebih buruk dari tingkat sebelumnya yaitu *read committed*. Walaupun begitu, integritas dari hasil transaksi lebih terjaga karena masalah *dirty write* dan *non-repeatable read* tidak akan terjadi pada tingkat isolasi ini. Fenomena *phantom read* masih dapat terjadi sehingga tingkat isolasi ini belum sepenuhnya sempurna untuk menjaga kestabilan data hasil transaksi.

Namun, perlu dicatat bahwa PostgreSQL pada implementasinya tidak memperbolehkan *phantom* read terjadi pada tingkatan isolasi ini walaupun pada sistem basis data lain dapat terjadi. Pada *repeatable read* juga masih dapat terjadi fenomena *serialization anomaly* lain ketika transaksi A membaca dan mengubah suatu data dan transaksi B membaca dan mengubah suatu data yang sama dan keduanya *commit* bersamaan. Hal ini dapat menyebabkan inkonsistensi hasil *commit* transaksi keduanya yang akan disimulasikan pada demonstrasi dibawah.

- **Simulasi**

  Simulasi berikut menunjukan penanganan fenomena *non-repeatable read* pada transaksi. Transaksi 1 mengakses data id satu, lalu transaksi 2 mengubah data yang sama dan *commit*. Pada pembacaan transaksi 1 yang kedua kalinya, data yang didapatkan adalah data sebelum transaksi 2 *commit*. Hal ini mengatasi pembacaan data yang berbeda pada transaksi 1 yang belum selesai.

Transaksi 1

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
sample=*# SELECT * FROM person WHERE id = 1;
 id | name |   job
----+------+----------
  1 | adul | politisi
(1 row)


sample=*# SELECT * FROM person WHERE id = 1;
 id | name |   job
----+------+----------
  1 | adul | politisi
(1 row)


sample=*#
```

Transaksi 2

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
sample=*# UPDATE person SET job = 'pilot' WHERE id = 1;
UPDATE 1
sample=*# COMMIT;
COMMIT
```

Simulasi dibawah menunjukan penanganan *phantom read* pada *repeatable read* di PostgreSQL. Kedua read yang dilakukan transaksi 1 mendapatkan hasil yang sama walaupun pada transaksi 2 data ada yang bertambah sehingga mencegah *phantom read*. Hal ini unik pada sistem basis data PostgreSQL dan mungkin tidak terjadi pada sistem basis data lainnya.

Transaksi 1

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
sample=*# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
(3 rows)


sample=*# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
(3 rows)
```

Transaksi 2

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
sample=*# INSERT INTO person VALUES (4, 'giring','musisi');
INSERT 0 1
sample=*# COMMIT;
COMMIT
sample=# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
  4 | giring | musisi
(4 rows)
```

Simulasi dibawah menunjukan fenomena anomali lainnya ketika kedua transaksi membaca dan mengubah data secara bersamaan. Pada *repeatable read*, hal ini masih dapat terjadi yang dapat menyebabkan inkonsistensi pada data hasil transaksi.

Transaksi 1

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
sample=*# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
  4 | giring | politisi
(4 rows)


sample=*# INSERT INTO person VALUES (5, 'raisa', 'singer');
INSERT 0 1
sample=*# commit;
COMMIT
```

Transaksi 2

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
sample=*# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
  4 | giring | politisi
(4 rows)


sample=*# INSERT INTO person VALUES (5, 'abdur', 'komedian');
INSERT 0 1
sample=*# commit;
COMMIT
```

d. Serializable

- **Deskripsi**

Sebuah transaksi yang serializable akan diperlakukan seolah-olah transaksi tersebut adalah satu-satunya transaksi yang sedang berjalan. Dalam kata lain, tidak ada transaksi yang mengakses data yang sama berjalan bersamaan. Semua transaksi dieksekusi seolah secara berurut setelah transaksi sebelumnya selesai.Ketika transaksi A melakukan sebuah proses pencarian dengan query, semua data yang memenuhi query tersebut akan dikunci aksesnya untuk transaksi lain yang berjalan dan mencegah penyisipan dan penghapusan baris dalam query tersebut. Hal ini menyebabkan fenomena *phantom read* tidak akan terjadi pada tingkat isolasi ini. Anomali lainnya yang disebutkan pada bagian *repeatable read* diatas tidak dapat terjadi pada tingkat *serializable* karena transaksi dianggap tidak dapat diselesaikan secara serial atau berurut.

- **Dampak**

  Dengan mekanisme *locking* dan pengurutan transaksi yang kompleks dan seolah berurut, tingkat isolasi *serializable* memiliki performa paling buruk diantara tingkat isolasi lainnya. Meskipun begitu tingkat ini menawarkan keamanan integritas data dan transaksi yang paling tinggi karena tingkat ini mengatasi permasalahan *dirty read*, *non-repeatable read*, *phantom read,* dan anomali lainnya.

- **Simulasi**

  Pada simulasi dibawah, kedua transaksi melakukan pembacaan dan penambahan data pada tabel. Namun hal ini tidak diperbolehkan oleh tingkat isolasi *serializable* karena transaksi dianggap tidak dapat diselesaikan secara serial. Hal ini menyebabkan integrasi data terjaga dengan baik. Perubahan pada transaksi 1 di rollback dan dibatalkan karena transaksi 2 menambah data terlebih dahulu sesuai urutan prosesnya.

Transaksi 1

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
sample=*# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
  4 | giring | politisi
  5 | abdur  | komedian
  5 | raisa  | singer
(6 rows)


sample=*# INSERT INTO person VALUES (6, 'sby','pelukis');
INSERT 0 1
sample=*# commit;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
sample=#
```

Transaksi 2

```
sample=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
sample=*# SELECT * FROM person;
 id |  name  |   job
----+--------+----------
  2 | jokowi | presiden
  3 | lewis  | pembalap
  1 | adul   | pilot
  4 | giring | politisi
  5 | abdur  | komedian
  5 | raisa  | singer
(6 rows)


sample=*# INSERT INTO person VALUES (6, 'laura','aktor');
INSERT 0 1
sample=*# commit;
COMMIT
sample=#
```

…

2. Implementasi Concurrency Control Protocol

a. Two-Phase Locking (2PL)

Implementasi dilakukan dengan menggunakan bahasa pemrograman Python. Untuk Two-Phase Locking, terdapat dua fase dalam protokol ini, yaitu *growing phase* dan *shrinking phase.* Transaksi hanya dapat menerima *lock* pada fase *growing* dan hanya dapat melepas lock pada fase *shrinking*. Dalam implementasi ini, digunakan *rigorous two-phase locking*, yaitu transaksi hanya dapat melepas *lock* pada saat transaksi melakukan *commit*, baik itu *shared lock* ataupun *exclusive lock*.

Untuk mengatasi *deadlock,* dalam implementasi ini digunakan *deadlock handling* berupa skema *wait-die*. Dalam skema ini, jika ada transaksi yang lebih tua ingin me-*lock* item data yang masih di-*lock* oleh transaksi yang lebih muda, maka transaksi yang lebih tua akan menunggu sampai *lock* pada item data dilepas. Jika ada transaksi yang lebih muda ingin me-*lock* item data yang masih di-*lock* oleh transaksi yang lebih tua, transaksi yang lebih muda ini akan *rollback.*

Berikut adalah *source code* pada program utama untuk implementasi protokol *two-phase locking*

```
class TPLocking:
    def __init__(self, data):
        self.lockTable = {}
        self.queue = deque()
        self.schedule = []
        self.transactions = []
        self.status = 'running'
        self.data = data
        self.description = []

    def initiate_lock(self,data_item):
        self.lockTable[data_item] = {
            'state': 'unlocked',
            'transactions': set(),
        }

    def isQueue(self,transaction: Transaction) -> bool:
        found = False

        for x in self.queue:
            for data_item in self.lockTable:
```

```
                transaction_info = x[1]
                if(transaction_info != None):
                    if (transaction_info.transaction_id == transaction.transaction_id):
                        found = True
        return found

    def shared_lock(self, transaction: Transaction, data_item) -> bool:
        #New data_item
        if data_item not in self.lockTable.keys():
            self.initiate_lock(data_item)

        data = self.lockTable[data_item]

        if (data['state'] == 'unlocked' or data['state'] == 'shared'):
            transaction.state = 'running'
            data['state'] = 'shared'
            data['transactions'].add(transaction.transaction_id)

            transaction.assign_lock(data_item,'S')
            transaction.addOperationList('S',data_item)
            self.schedule.append(['S',transaction.transaction_id,data_item])
            self.description.append(f"Shared lock granted to T{transaction.transaction_id} on data item
{data_item}")
            self.state = 'running'
            return True
        else:
            #Deadlock handling with wait-die scheme
            if (transaction.state=='aborted' or self.state == 'rollback'):
                self.queue.append(['S',transaction,data_item])
                return False
            if (transaction.state == 'waiting' or self.state == 'queue'):
                return False
            if self.waitdie(transaction, self.getHoldingTransaction(data_item)):
                transaction.state = 'waiting'
                self.queue.append(['S',transaction,data_item])
            else:
                self.abort(transaction)
                self.shared_lock(transaction, data_item)
            self.state = 'running'
            return True

    def exclusive_lock(self,transaction: Transaction, data_item: str) -> bool:
        #New data_item
        if data_item not in self.lockTable.keys():
            self.initiate_lock(data_item)

        data = self.lockTable[data_item]

        #No transaction locking data  item
        if (data['state'] == 'unlocked'):
```

```
            transaction.state = 'running'
            data['state'] = 'exclusive'
            data['transactions'].add(transaction.transaction_id)

            transaction.assign_lock(data_item,'X')
            transaction.addOperationList('X',data_item)
            self.schedule.append(['X',transaction.transaction_id,data_item])
            self.description.append(f"Exclusive lock granted to T{transaction.transaction_id} on data
item {data_item}")
            self.state = 'running'
            return True

        #Upgrade Lock
        elif (data['state'] == 'shared' and len(data['transactions']) == 1 and
next(iter(data['transactions'])) == transaction.transaction_id):
            print(f"[!] lock {data_item} upgraded to exclusive lock")
            transaction.state = 'running'
            data['state'] = 'exclusive'

            transaction.delete_lock(data_item)
            transaction.assign_lock(data_item,'X')
            transaction.addOperationList('X', data_item)
            self.schedule.append(['X',transaction.transaction_id,data_item])
            self.description.append(f"Exclusive lock granted to T{transaction.transaction_id} on data
item {data_item}")
            self.state = 'running'
            return True
        #Other
        else:
            if ( transaction.state == 'aborted' or self.state == 'rollback'):
                self.queue.append(['X',transaction,data_item])
                return False
            if (transaction.state == 'waiting' or self.state=='queue'):
                return False
            #Deadlock handling with wait-die mechanism
            if self.waitdie(transaction, self.getHoldingTransaction(data_item)):
                self.description[len(self.description)-1] += f"T{transaction.transaction_id} should be
waiting for T{self.getHoldingTransaction(data_item)}"
                transaction.state = 'waiting'
                self.queue.append(['X',transaction,data_item])

            else:
                self.description[len(self.description)-1] += f"T{transaction.transaction_id} cannot lock
{data_item}, {data_item} hold by T{self.lockTable[data_item]['transactions'].__str__()[2]}"
                self.abort(transaction)
                self.exclusive_lock(transaction, data_item)
            self.state = 'running'
            return False

    def commit(self, transaction: Transaction) -> None:
```

```
        if self.isQueue(transaction):
            #Cannot commit since operation is not completed
            self.queue.append(['C',transaction])
        else:
            print(f"Transaction {transaction.transaction_id} commited")

            #Commit transaction
            transaction.state = 'commited'
            self.schedule.append(['C',transaction.transaction_id])
            self.description.append(f"T{transaction.transaction_id} commited")

            #Unlock all data item locked by transaction
            for x in transaction.lock:
                self.unlock(transaction, x[0])
            self.checkQueue()

    def abort(self, transaction: Transaction) -> None:
        #transaction aborted
        print(f"Transaction {transaction.transaction_id} aborted")
        transaction.state = 'aborted'
        self.state = 'rollback'

        self.schedule.append(['A',transaction.transaction_id])
        self.description.append(f"T{transaction.transaction_id} aborted, rollback")
        #Unlock data item hold by transaction
        i = 0
        while i < (len(transaction.lock)):
            self.unlock(transaction,transaction.lock[i][0])
            i += 1

        #Jalankan ulang transaksi
        self.rerun_transaction(transaction)

    def rerun_transaction(self,transaction: Transaction):
        for op in transaction.OperationList:
            self.queue.append([op[0],transaction,op[1]])

    def unlock(self, transaction: Transaction,data_item) :
        if data_item in self.lockTable.keys():
            data = self.lockTable[data_item]

            #remove lock from lockTable
            data['transactions'].discard(transaction.transaction_id)

            #remove lock from transaction
            transaction.delete_lock(data_item)

            #add to schedule
            self.schedule.append(['UL',transaction.transaction_id,data_item])
            self.description.append(f"{data_item} unlocked")
```

```
            #Run other transaction in queue if no transaction hold data item
            if not(data['transactions']):
                data['state']  = 'unlocked'

    def waitdie(self, younger_transaction: Transaction, older_transaction_id):
        return self.lookForTrx(older_transaction_id) is not None and younger_transaction.timestamp <
self.lookForTrx(older_transaction_id).timestamp

    def getHoldingTransaction(self,data_item) -> str:
        return next(iter(self.lockTable[data_item]['transactions']), None)

    def checkQueue(self):
        self.state = 'queue'
        if (len(self.queue) != 0):
            stop =False
            while not(stop) and len(self.queue) != 0:
                curr_ops = self.queue.popleft()
                type = curr_ops[0]
                transaction = curr_ops[1]
                transaction.state = 'queue'
                if (len(curr_ops) == 2):
                    if type == 'C':
                        self.commit(transaction)
                elif len(curr_ops) == 3:
                    data_item = curr_ops[2]
                    status = self.lock(transaction,data_item,type)

                    if not(status):
                        if curr_ops not in (self.queue):
                            self.queue.append(curr_ops)
                        stop = True
                    else:
                        pass

    def lookForTrx(self, trx_id):
        for trx in self.transactions:
            if trx.transaction_id == trx_id:
                return trx
        return None

    def lock(self, transaction, data_item, operation) -> bool:
        if (operation == 'R' or operation == 'S'):
            return self.shared_lock(transaction,data_item)
        elif (operation == 'W' or operation == 'X'):
            return self.exclusive_lock(transaction,data_item)

    def printSchedule(self):
        for sch in self.schedule:
            if(len(sch) == 2):
```

```
                print(f"{sch[0]}{sch[1]}")
            elif (len(sch) == 3):
                if (sch[0] == 'S' or sch[0] == 'X'):
                    print(f"Lock-{sch[0]}{sch[1]}({sch[2]})")
                else:
                    print(f"{sch[0]}{sch[1]}({sch[2]})")

    def run(self):
        curr_timestamp = 1
        for op in self.data:
            print(op)
            if (len(op) == 2):
                trx = self.lookForTrx(op[1])
                if (trx == None):
                    raise Exception("Unknown transaction request to be commited")
                if (op[0] == 'C'):
                    self.commit(trx)
                else:
                    print("Operation invalid!")

            elif(len(op) == 5):
                #Initialize
                ops = op[0]
                trx = self.lookForTrx(op[1])
                if trx==None:
                    trx = Transaction(op[1], curr_timestamp)
                    curr_timestamp += 1
                    self.transactions.append(trx)
                data = op[3]

                #Classified which action will be taken
                self.schedule.append([ops,op[1],op[3]])
                self.description.append("")
                status = self.lock(trx,data,ops)
        self.checkQueue()
        self.checkQueue()

    def scheduleToString(self,sch) -> str:
        result = ""
        if (sch[0] == 'S' or sch[0] == 'X'):
            result += f"lock-{sch[0]}{sch[1]}({sch[2]})"
        elif (sch[0] == 'UL'):
            result += f"UL{sch[1]}({sch[2]})"
        elif (sch[0] == 'R' or sch[0] == 'W'):
            result += f"{sch[0]}{sch[1]}({sch[2]})"
        else:
            result += f"{sch[0]}{sch[1]}"
        return result

    def __repr__(self):
```

```
        size = 14
        sizeDesc = 20
        schedule = ""
        for i in range (len(self.transactions)):
            schedule += " "*6 + "T" + str(self.transactions[i].transaction_id) + " "*6 + "|"
        schedule += " Description"
        schedule += "\n" + "-"*((len(self.transactions)*size)+sizeDesc) + "\n"

        for i in range (len(self.schedule)):
            for j in range (len(self.transactions)):
                if (self.schedule[i][1] == self.transactions[j].transaction_id):
                    word = self.scheduleToString(self.schedule[i])
                    if (len(word) % 2 == 0):
                        blank = (size-len(word))//2
                        schedule += " "*blank + word + " "*blank + "|"
                    elif (len(word) % 2 == 1):
                        blank = (size-len(word))//2
                        schedule += " "*blank + word + " "*blank + " |"
                else:
                    schedule += " "*size + "|"
            schedule += f" {self.description[i]}"
            schedule += "\n"
        return str(schedule)
```

Setelah program diimplementasi, dilakukan percobaan dengan *sequence* sebagai berikut: R1(X); R2(Y); R3(Z); W1(X); W3(Y); W2(Y); W2(A); W3(A); C1; C2; C3. Program menghasilkan *schedule* sebagai berikut

T1	T2	T3	Description
R1(X)			
lock-S1(X)			Shared lock granted to T1 on data item X
	R2(Y)		
	lock-S2(Y)		Shared lock granted to T2 on data item Y
		R3(Z)	
		lock-S3(Z)	Shared lock granted to T3 on data item Z
W1(X)			
lock-X1(X)			Exclusive lock granted to T1 on data item X
		W3(Y)	
		A3	T3 aborted, rollback
		UL3(Z)	Z unlocked
	W2(Y)		
	lock-X2(Y)		Exclusive lock granted to T2 on data item Y
	W2(A)		
	lock-X2(A)		Exclusive lock granted to T2 on data item A
		W3(A)	
C1			T1 commited
UL1(X)			X unlocked
		lock-S3(Z)	Shared lock granted to T3 on data item Z
		A3	T3 aborted, rollback
		UL3(Z)	Z unlocked
	C2		T2 commited
	UL2(Y)		Y unlocked
	UL2(A)		A unlocked
		lock-X3(A)	Exclusive lock granted to T3 on data item A
		lock-S3(Z)	Shared lock granted to T3 on data item Z
		lock-S3(Z)	Shared lock granted to T3 on data item Z
		lock-X3(Y)	Exclusive lock granted to T3 on data item Y
		C3	T3 commited
		UL3(A)	A unlocked
		UL3(Z)	Z unlocked
		UL3(Z)	Z unlocked
		UL3(Y)	Y unlocked

Dari *schedule* tersebut, terlihat bahwa T1 melakukan *shared lock* saat akan membaca X. Kemudian saat T1 ingin menulis X, *lock* akan otomatis di-*upgrade* menjadi *exclusive lock* karena X hanya di-*lock* oleh T1. Namun dapat dilihat pada baris ke sembilan, saat T3 ingin menulis ke data item Y, *lock* tidak dapat diberikan karena Y sedang di-*shared lock* oleh T2. Dengan skema *wait-die,* T3 yang lebih muda daripada T2 akan melakukan *rollback* dan *abort*. Kemudian dapat dilihat bahwa T2 juga ingin menulis ke item data A. Namun, *lock* tidak dapat diberikan karena A sedang di-*lock* oleh T3. Karena T2 lebih tua daripada T3, T2 akan menunggu sampai T3 membuka lock terhadap item data A akibat skema *wait-die*.

Jika diperhatikan, ada potensi untuk terjadinya *deadlock* pada *sequence* ini karena T3 ingin mendapatkan *exclusive lock* terhadap item data Y yang sedang di-*lock* oleh T2, sementara T2 juga ingin mendapatkan *exclusive lock* terhadap item data A yang sedang di-*lock* oleh T3. Jika tidak diterapkan *deadlock handling*, sequence ini akan mengalami *deadlock* dan tidak dapat selesai. Dapat dilihat juga T3 menjalankan ulang operasinya dari awal saat mengalami *rollback.* Transaksi melakukan *release lock* pada item data yang di-*lock* hanya pada saat transaksi melakukan *commit.*

Hasil *schedule* akhir yang diperoleh adalah R1(X); Lock-S1(X); R2(Y); Lock-S2(Y) ;R3(Z); Lock-S3(Z); W1(X); Lock-X1(X); W3(Y); A3; UL3(Z); W2(Y); Lock-X2(Y); W2(A); Lock-X2(A); W3(A); C1; UL1(X); Lock-S3(Z); A3; UL3(Z); C2; UL2(Y); UL2(A); Lock-X3(A); Lock-S3(Z); Lock-S3(Z); Lock-X3(Y); C3; UL3(A); UL3(Z); UL3(Z); UL3(Y)

b. Optimistic Concurrency Control (OCC)

OCC merupakan salah satu *concurrency control* bertipe optimistik yang artinya mengizinkan adanya dua atau lebih transaksi membaca atau melakukan manipulasi data yang sama serta berjalan secara konkuren tanpa adanya *locking* atau *blocking* satu sama lain selama tidak terjadi konflik. Manipulasi data dilakukan pada suatu "lingkungan" yang terpisah antar transaksi sehingga tidak perlu dilakukan *locking* agar beberapa transaksi dapat berjalan konkuren. OCC *validation based* dibagi menjadi tiga fase, yaitu fase *read and execution*, fase *validation*, dan fase *write*. *Key point* dari OCC ini adalah semua *read* dan *execution* yang dilakukan terjadi pada *local memory* transaksi. *Read* dan *execution* yang dilakukan nantinya akan divalidasi sebelum dilakukan *commit*. Jika terjadi konflik, maka transaksi yang *commit* lebih akhir akan di-*rollback*.

Pada bagian ini implementasi OCC menggunakan bahasa pemrograman python. Jika terjadi konflik dan harus dilakukan *rollback*, maka operasi-operasi yang di-*rollback* akan dieksekusi kembali tepat setelahnya, sehingga *rollback* tepat terjadi hanya sekali untuk tiap transaksi. Berikut merupakan kode implementasi dari OCC,

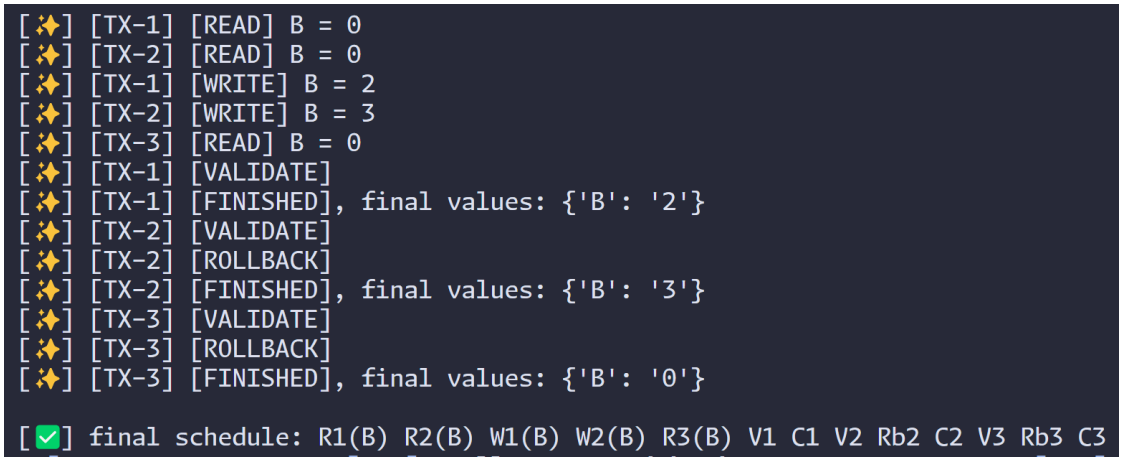
```
class OptimisticCC:
    def __init__(self, operations: List[List[str]]) -> None:
        self.operations = operations
        self.tx_timestamp: Dict[str, Dict[str, float]] = {}
        self.tx_written: Dict[str, set[str]] = {}
        self.tx_read: Dict[str, set[str]] = {}
        self.tx_resource: Dict[str, Dict[str, str]] = {}
        self.final_schedule: List[str] = []
        self.aborted: bool = False

    def run(self) -> None:
        print()
        for _, operation in enumerate(self.operations):
            time.sleep(1e-7)

            action = operation[0]
```

```
            tx = operation[1]

            if not self.tx_timestamp.get(tx):
                self.tx_timestamp[tx] = {
                    'start': time.time(), 'validation': float(sys.maxsize), 'end':
float(sys.maxsize)
                }

            if abs(self.tx_timestamp[tx]["end"] - sys.maxsize) > 1e-9:
                print("[ ❗ ] invalid operation on input. aborting.")
                self.aborted = True
                break

            if not self.tx_resource.get(tx):
                self.tx_resource[tx] = {}

            match action:
                case 'W':
                    resource_name = operation[2]
                    resource_value = operation[3]

                    self.tx_resource[tx][resource_name] = resource_value

                    print(
                        f'[✨] [TX-{tx}] [WRITE] {resource_name} = {
                            self.tx_resource[tx][resource_name]}'
                    )

                    self.final_schedule.append(f'W{tx}({resource_name})')

                    if not self.tx_written.get(tx):
                        self.tx_written[tx] = {resource_name}
                        continue

                    self.tx_written[tx].add(resource_name)

                case 'R':
                    resource_name = operation[2]

                    if not self.tx_resource[tx].get(resource_name):
                        self.tx_resource[tx][resource_name] = '0'

                    print(
                        f'[✨] [TX-{tx}] [READ] {resource_name} = {
                            self.tx_resource[tx][resource_name]}'
                    )

                    self.final_schedule.append(f'R{tx}({resource_name})')
```

```
                    if not self.tx_read.get(tx):
                        self.tx_read[tx] = {resource_name}
                        continue

                    self.tx_read[tx].add(resource_name)

                case 'C':
                    self.tx_timestamp[tx]["validation"] = time.time()

                    print(f'[✨] [TX-{tx}] [VALIDATE]')

                    self.final_schedule.append(f'V{tx}')

                    if not self.__validate(tx):
                        print(f'[✨] [TX-{tx}] [ROLLBACK]')

                        self.final_schedule.append(f'Rb{tx}')

                        self.__rollback(tx)
                    print(
                        f'[✨] [TX-{tx}] [FINISHED], final values: {self.tx_resource[tx]}')

                    self.final_schedule.append(f'C{tx}')

                    self.tx_timestamp[tx]["end"] = time.time()

        # print final schedule
        if not self.aborted:
            print()
            print('[✅] final schedule: ', end='')
            for i, operation in enumerate(self.final_schedule):
                print(f'{operation}', end=' ')
            print()

    def __validate(self, tx: str) -> bool:
        for timestamp_tx, timestamp_tx_value in self.tx_timestamp.items():
            if timestamp_tx != tx:
                if self.tx_timestamp[tx]["start"] < timestamp_tx_value["end"] <
self.tx_timestamp[tx]["validation"]:
                    if self.tx_written.get(timestamp_tx) and
self.tx_read[tx].intersection(self.tx_written[timestamp_tx]):
                        return False
        return True

    def __rollback(self, tx: str) -> None:
        self.tx_timestamp[tx] = {
            "start": time.time(),
```

```
            "validation": time.time(),
            "end": sys.maxsize
        }
```

Setelah program diimplementasi, dilakukan percobaan dengan *sequence* sebagai berikut: R1(B); R2(B); W1(B=2); W2(B=3); R3(B); C1; C2; C3. Program menghasilkan *schedule* sebagai berikut,



Dari *schedule* tersebut terlihat fase *read and execution* berjalan normal, namun saat mencapai fase *validation* untuk transaksi 2, terjadi konflik antara transaksi 2 dengan transaksi 1, yaitu transaksi 2 melakukan pembacaan *resource* yang ditulis oleh transaksi 1 sebelum transaksi 1 *finished* atau *commit*. Selain itu, juga terjadi konflik antara transaksi 3 dengan transaksi 1 dan 2, yaitu transaksi 3 melakukan pembacaan *resource* yang ditulis oleh transaksi 1 dan 2 sebelum kedua transaksi tersebut *finished* atau *commit*.

Sedikit catatan, untuk *resource* yang tidak di-*assign* suatu nilai, nilai *default* dari *resource* tersebut adalah 0.

c. Multiversion Timestamp Ordering Concurrency Control (MVCC)

Multiversion Timestamp Ordering merupakan salah satu dari tiga jenis *Multiversion Concurrency Control*, yaitu *Timestamp Ordering, Two-Phase Locking,* dan *Snapshot Isolation*. *Key point* dari MV *Timestamp Ordering* ini adalah adanya *versioning* untuk setiap *resource* yang terlibat. Karena adanya *versioning* untuk tiap *resource*, semua operasi *read* yang dilakukan akan selalu berhasil. Semua operasi *read* yang dilakukan akan menggunakan *resource* dengan

versi yang sesuai, yaitu memiliki nilai *Write-Timestamp* atau *W-Timestamp* kurang dari sama dengan *Timestamp* transaksi. Sedangkan operasi *write* akan membuat versi baru dari *resource* yang di-*write* dengan syarat nilai *Read-Timestamp* atau *R-Timestamp* dari *resource* tersebut kurang dari sama dengan *Timestamp* transaksi. Jika *R-Timestamp resource* lebih dari *Timestamp* transaksi, maka akan dilakukan *rollback* untuk semua transaksi yang menggunakan *resource* hasil manipulasi dari transaksi tersebut dan tentu saja juga dilakukan *rollback* untuk transaksi itu sendiri.

Pada bagian ini implementasi MV *Timestamp Ordering* menggunakan bahasa pemrograman python. Jika terjadi konflik dan harus dilakukan *rollback*, maka operasi-operasi yang di-*rollback* akan dieksekusi kembali tepat setelahnya. Berikut merupakan kode implementasi dari MV *Timestamp Ordering*,

```
class MultiversionTimestampOrderingCC:
    def __init__(self, data: List[List[str]]) -> None:
        self.operations = data
        self.tx_ts_start: Dict[str, float] = {}
        self.resources: Dict[str, Dict[int, Dict[str, str | float]]] = {}
        self.tx_recent_operation: Dict[str, List[List[str]]] = {}
        self.reads: Set[Tuple[str, str]] = set()
        self.final_schedule: List[str] = []
        self.tx_commited: List[str] = []
        self.aborted: bool = False

    def run(self) -> None:
        print()
        self.__process_operation(self.operations, False)
        if not self.aborted:
            print()
            print('[✅] final schedule: ', end='')
            for _, operation in enumerate(self.final_schedule):
                print(operation, end=' ')
            print()

    def __process_operation(self, operations: List[List[str]], rolled_back: bool):
        for _, operation in enumerate(operations):
            if self.aborted:
                break

            time.sleep(1e-7)
```

```
            action = operation[0]
            tx = operation[1]

            if tx in self.tx_commited:
                print("[ ❗ ] invalid operation on input. aborting.")
                self.aborted = True
                break

            if not self.tx_ts_start.get(tx) or rolled_back:
                self.tx_ts_start[tx] = time.time()
                if not rolled_back:
                    self.tx_recent_operation[tx] = []

            if action != 'C':
                resource_name = operation[2]
                if not self.resources.get(resource_name):
                    self.resources[resource_name] = {
                        0: {
                            'tx_producer': '0',
                            'rts': 0,
                            'wts': 0,
                            'content': '0'
                        }
                    }

            if not rolled_back:
                self.tx_recent_operation[tx].append(operation)

            match action:
                case 'W':
                    resource_name = operation[2]
                    content = operation[3]

                    for version in range(len(self.resources[resource_name]) - 1, -1, -1):
                        if float(self.resources[resource_name][version]['wts']) <
self.tx_ts_start[tx]:
                            # if rts < ts(tx): create new version for the corresponding
resource
                            if float(self.resources[resource_name][version]['rts']) >
self.tx_ts_start[tx]:
                                self.__rollback(tx)

                            # rollback condition
                            else:
                                self.resources[resource_name][version + 1] = {
```

```
                                    'tx_producer': tx,
                                    'rts': self.tx_ts_start[tx],
                                    'wts': self.tx_ts_start[tx],
                                    'content': content
                                }

                                print(f'[✨] [TX-{tx}] [WRITE] {resource_name} v{version
+ 1} = {
                                    self.resources[resource_name][version + 1]}')

                                self.final_schedule.append(
                                    f'W{tx}({resource_name})')

                                break

                            elif abs(float(self.resources[resource_name][version]['wts']) -
self.tx_ts_start[tx]) < 1e-9:
                                self.resources[resource_name][version]['content'] = content
                                self.resources[resource_name][version]['tx_producer'] = tx

                                print(f'[✨] [TX-{tx}] [WRITE] {resource_name} v{version} = {
                                    self.resources[resource_name][version]}')

                                self.final_schedule.append(
                                    f'W{tx}({resource_name})')

                                break

                    case 'R':
                        resource_name = operation[2]

                        for version in range(len(self.resources[resource_name]) - 1, -1, -1):
                            if float(self.resources[resource_name][version]['wts']) <
self.tx_ts_start[tx] or abs(float(self.resources[resource_name][version]['wts']) -
self.tx_ts_start[tx]) < 1e-9:
                                self.resources[resource_name][version]['rts'] =
self.tx_ts_start[tx]
                                self.reads.add(
                                    (tx,
str(self.resources[resource_name][version]['tx_producer'])))

                                print(f'[✨] [TX-{tx}] [READ] {resource_name} v{version} = {
                                    self.resources[resource_name][version]}')

                                self.final_schedule.append(
```

```
                            f'R{tx}({resource_name})')

                        break

            case 'C':
                print(f'[✨] [TX-{tx}] [COMMIT]')

                self.final_schedule.append(f'C{tx}')

                self.tx_commited.append(tx)

    def __rollback(self, tx: str) -> None:
        rolled_back_txs = self.__get_rolled_back_transaction(tx)
        for _, tx in enumerate(rolled_back_txs):
            print(f'[✨] [TX-{tx}] [ROLLBACK] start')
            self.final_schedule.append(f'Rb{tx}')
            self.__process_operation(self.tx_recent_operation[tx], True)
            print(f'[✨] [TX-{tx}] [ROLLBACK] end')

    def __get_rolled_back_transaction(self, tx: str) -> List[str]:
        result = [tx]
        new_reads: Set[Tuple[str, str]] = set()
        for _, read in enumerate(self.reads):
            if (read[1] == tx):
                result += self.__get_rolled_back_transaction(read[0])
            else:
                new_reads.add(read)

        self.reads = new_reads
        return result
```

Setelah program diimplementasi, dilakukan percobaan dengan *sequence* sebagai berikut: R5(X); R2(Y); R1(Y); W3(Y=4); W3(Z=5); R5(Z); R2(Z); R1(X); R4(W); W3(W); W5(Y=8); W5(Z=9); C1; C2; C3; C4; C5. Program menghasilkan *schedule* sebagai berikut,

```
[✨] [TX-5] [READ] X v0 = {'tx_producer': '0', 'rts': 1701439521.4158244, 'wts': 0, 'content': '0'}
[✨] [TX-2] [READ] Y v0 = {'tx_producer': '0', 'rts': 1701439521.4161744, 'wts': 0, 'content': '0'}
[✨] [TX-1] [READ] Y v0 = {'tx_producer': '0', 'rts': 1701439521.4164178, 'wts': 0, 'content': '0'}
[✨] [TX-3] [WRITE] Y v1 = {'tx_producer': '3', 'rts': 1701439521.416578, 'wts': 1701439521.416578, 'content': '4'}
[✨] [TX-3] [WRITE] Z v1 = {'tx_producer': '3', 'rts': 1701439521.416578, 'wts': 1701439521.416578, 'content': '5'}
[✨] [TX-5] [READ] Z v0 = {'tx_producer': '0', 'rts': 1701439521.4158244, 'wts': 0, 'content': '0'}
[✨] [TX-2] [READ] Z v0 = {'tx_producer': '0', 'rts': 1701439521.4161744, 'wts': 0, 'content': '0'}
[✨] [TX-1] [READ] X v0 = {'tx_producer': '0', 'rts': 1701439521.4164178, 'wts': 0, 'content': '0'}
[✨] [TX-4] [READ] W v0 = {'tx_producer': '0', 'rts': 1701439521.4171762, 'wts': 0, 'content': '0'}
[✨] [TX-3] [ROLLBACK] start
[✨] [TX-3] [WRITE] Y v2 = {'tx_producer': '3', 'rts': 1701439521.4176211, 'wts': 1701439521.4176211, 'content': '4'}
[✨] [TX-3] [WRITE] Z v2 = {'tx_producer': '3', 'rts': 1701439521.4177647, 'wts': 1701439521.4177647, 'content': '5'}
[✨] [TX-3] [WRITE] W v1 = {'tx_producer': '3', 'rts': 1701439521.417904, 'wts': 1701439521.417904, 'content': '0'}
[✨] [TX-3] [ROLLBACK] end
[✨] [TX-5] [ROLLBACK] start
[✨] [TX-5] [READ] X v0 = {'tx_producer': '0', 'rts': 1701439521.4180741, 'wts': 0, 'content': '0'}
[✨] [TX-5] [READ] Z v2 = {'tx_producer': '3', 'rts': 1701439521.4182618, 'wts': 1701439521.4177647, 'content': '5'}
[✨] [TX-5] [WRITE] Y v3 = {'tx_producer': '5', 'rts': 1701439521.4183795, 'wts': 1701439521.4183795, 'content': '8'}
[✨] [TX-5] [ROLLBACK] end
[✨] [TX-5] [WRITE] Z v3 = {'tx_producer': '5', 'rts': 1701439521.4183795, 'wts': 1701439521.4183795, 'content': '9'}
[✨] [TX-1] [COMMIT]
[✨] [TX-2] [COMMIT]
[✨] [TX-3] [COMMIT]
[✨] [TX-4] [COMMIT]
[✨] [TX-5] [COMMIT]

[✅] final schedule: R5(X) R2(Y) R1(Y) W3(Y) W3(Z) R5(Z) R2(Z) R1(X) R4(W) Rb3 W3(Y) W3(Z) W3(W) Rb5 R5(X) R5(Z) W5(Y) W5(Z) C1
 C2 C3 C4 C5
```

Dari *schedule* tersebut terlihat terjadi beberapa *rollback* saat dilakukan *write* W3(W) dan W5(Y=8) karena secara berturut-turut nilai *R-timestamp(W v0) > TS(T3)* dan *R-timestamp(Y v2) > TS(T5*). Setelah dilakukan *rollback*, W5(Z=9) dapat berjalan normal.

Sedikit catatan, untuk *resource* yang tidak di-*assign* suatu nilai, nilai *default* dari *resource* tersebut adalah 0.

3. Eksplorasi Recovery

a. *Write-Ahead Log*

Write-Ahead Log (WAL) merupakan metode *recovery* yang mengharuskan perubahan terhadap basis data hanya dilakukan setelah perubahan tersebut telah dicatat (*logged*) dan *log records* terkait telah ditulis pada sebuah *stable storage*. Ketika terjadi kegagalan, DBMS melakukan recovery dengan REDO atau UNDO perubahan yang dilakukan terhadap basis data sebelum terjadi kegagalan berdasarkan *log records* yang telah disimpan pada *stable storage*. REDO adalah menjalankan kembali perintah transaksi jika transaksi bersifat *complete*, sedangkan UNDO adalah membatalkan perintah transaksi jika transaksi bersifat *incomplete*. REDO menjamin *property durability* dan UNDO menjamin *property atomicity* terpenuhi.

b. *Continuous Archiving*

Continuous Archiving merupakan metode *recovery* yang menyimpan salinan data secara terus-menerus (*continuous*) pada sebuah file arsip transaksi dengan memanfaatkan *Write-Ahead Log*. Karena pengarsipan terus-menerus diperbarui, ketika terjadi kegagalan DBMS dapat melakukan *recovery* sesuai dengan titik waktu tertentu yang diinginkan oleh administrator selama terdapat arsip transaksi yang tersimpan untuk titik waktu tersebut.

c. *Point-in-Time Recovery*

Point-in-Time Recovery merupakan metode *recovery* dengan mengembalikan atau memulihkan *basis data* ke keadaan yang konsisten pada satu titik waktu tertentu. Point-in-Time Recovery memanfaatkan backup data dan backup logs yang telah disimpan oleh Write-Ahead Log dan Continuous Recovery. Ketika terjadi kegagalan, DBMS akan memasuki mode recovery dan membaca arsip WAL sampai titik waktu tertentu yang dispesifikasikan oleh administrator basis data sebagai stopping point dari recovery sehingga basis data akan kembali ke keadaan pada titik waktu tersebut.

d. Simulasi Kegagalan pada PostgreSQL

- *Setup* basis data yang diuji

Terdapat basis data bernama “pitr” berisi tabel “test_1” yang dibuat dengan perintah berikut.

```
5   CREATE TABLE test_1(id INTEGER);
6   INSERT INTO test_1 VALUES (generate_series(1,100000));
```

- *Physical Backup* terhadap klaster basis data

Sebelum melakukan *physical backup* terhadap klaster basis data, terdapat beberapa pengaturan pada **postgresql.conf** yang perlu diubah, yaitu:

1. Mengubah **wal_level** menjadi **logical**
2. Mengubah **archive_mode** menjadi **on**
3. Mengubah **archive_command** menjadi **copy "D:\\Coding\\mbd\\log\\%f" "%p"**

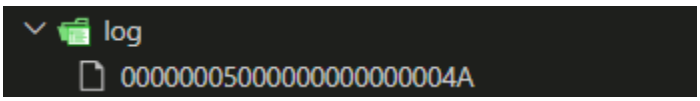
```
205   #------------------------------------------------------------------------------
206   # WRITE-AHEAD LOG
207   #------------------------------------------------------------------------------
208
209   # - Settings -
210
211   wal_level = logical         # minimal, replica, or logical
212                               # (change requires restart)
```

```
256   # - Archiving -
257
258   archive_mode = on       # enables archiving; off, on, or always
259                           # (change requires restart)
260   #archive_library = ''       # library to use to archive a WAL file
261                           # (empty string indicates archive_command should
262                           # be used)
263   archive_command = 'copy "%p" "D:\\Coding\\mbd\\log\\%f"'        # command to use to archive a WAL file
264                           # placeholders: %p = path of file to archive
265                           #               %f = file name only
266                           # e.g. 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
```

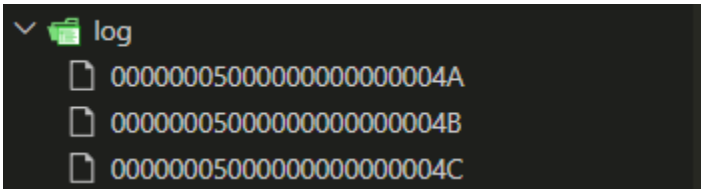
Setelah melakukan pengubahan di atas, *restart* PostgreSQL agar perubahan pengaturan dapat berlaku. Ketika PostgreSQL sudah berjalan, lakukan perintah untuk membuat tabel “test_2” berikut.

```
8   CREATE TABLE test_2(id INTEGER);
9   INSERT INTO test_2 VALUES (generate_series(1,250000));
```

Setelah menjalankan perintah di atas, akan akan terlihat muncul log records pada folder yang sudah ditentukan pada file **postgresql.conf** sebelumnya, berikut contoh log records yang muncul pada folder.



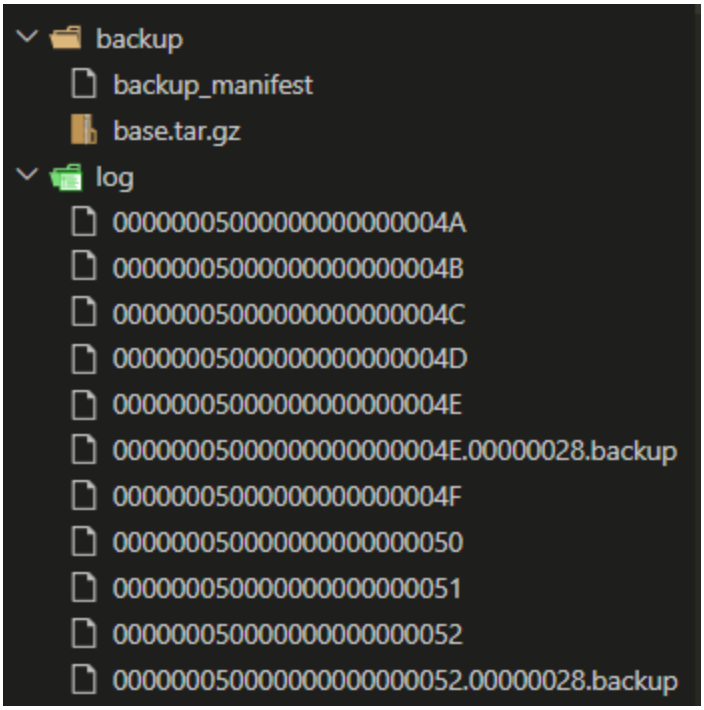
Lakukan kembali perintah membuat tabel sampai muncul beberapa log records seperti berikut.



Setelah terdapat beberapa log records, akan dilakukan *physical backup* terhadap klaster basis data dengan menggunakan perintah berikut.

```
C:\Program Files\PostgreSQL>pg_basebackup -U postgres -h localhost -p 5432 -D D:\Coding\mbd\backup -Ft -Xf -z -P
Password:
78748/78748 kB (100%), 1/1 tablespace
```

Akan muncul sebuah *physical backup* pada folder yang ditentukan sesuai perintah, sebagai contoh berikut, muncul zip file bernama base.tar.gz pada folder backup. Selain itu, akan muncul sebuah log record yang mencatat bahwa *physical backup* telah dilakukan.



log > 000000050000000000000052.00000028.backup

```
1   START WAL LOCATION: 0/52000028 (file 000000050000000000000052)
2   STOP WAL LOCATION: 0/52000100 (file 000000050000000000000052)
3   CHECKPOINT LOCATION: 0/52000060
4   BACKUP METHOD: streamed
5   BACKUP FROM: primary
6   START TIME: 2023-12-01 18:44:47 +07
7   LABEL: pg_basebackup base backup
8   START TIMELINE: 5
9   STOP TIME: 2023-12-01 18:44:48 +07
10  STOP TIMELINE: 5
```

Setelah *physical backup* terhadap klaster basis data dilakukan, jalankan beberapa perintah yang mengubah keadaan basis data sehingga terdapat perbedaan antara keadaan basis data ketika dilakukan *physical backup* dengan ketika basis data mengalami kegagalan. Langkah terakhir sebelum melakukan recovery adalah mensimulasikan kegagalan PostgreSQL dengan cara mematikan PostgreSQL.

```
pitr=# \d
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.
!?>
```

- Data sebelum kegagalan terjadi

```
pitr=# \d
         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | test_1 | table | postgres
 public | test_2 | table | postgres
 public | test_3 | table | postgres
 public | test_4 | table | postgres
(4 rows)


pitr=# select * from current_timestamp;
       current_timestamp
-------------------------------
 2023-12-01 18:50:49.680855+07
(1 row)
```

- Data saat kegagalan terjadi

```
pitr=# \d
         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | test_1 | table | postgres
 public | test_2 | table | postgres
 public | test_3 | table | postgres
 public | test_4 | table | postgres
 public | test_6 | table | postgres
 public | test_7 | table | postgres
 public | test_8 | table | postgres
 public | test_9 | table | postgres
(8 rows)


pitr=# select * from current_timestamp;
       current_timestamp
-------------------------------
 2023-12-01 18:51:43.383388+07
(1 row)
```

- Data setelah dilakukan *recovery*

```
pitr=# \d
         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | test_1 | table | postgres
 public | test_2 | table | postgres
 public | test_3 | table | postgres
 public | test_4 | table | postgres
(4 rows)


pitr=# select * from current_timestamp;
      current_timestamp
-------------------------------
 2023-12-01 19:07:39.54077+07
(1 row)
```

- Penjelasan proses *recovery*

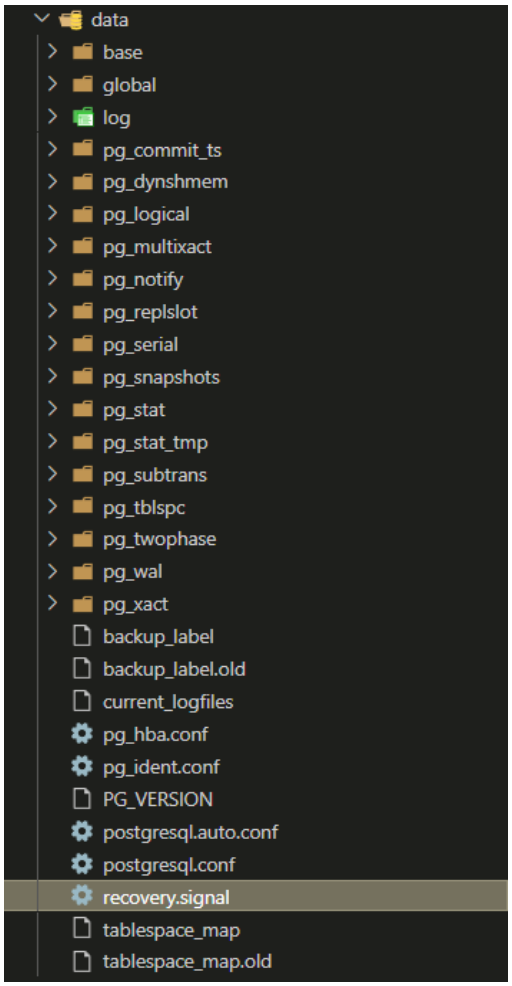
Sebelum menyalakan kembali PostgreSQL, hal yang perlu dilakukan adalah memindahkan isi dari *physical backup* yang telah dilakukan pada langkah sebelumnya ke folder data PostgreSQL. Kemudian, ubah pengaturan pada file **postgresql.conf** berikut.

1. Mengubah **restore_command** menjadi **copy "D:\\Coding\\mbd\\log\\%f" "%p"**
2. Mengubah **recovery_target_time** menjadi **2023-12-01 18:44:48 +07**

```
# - Archive Recovery -

# These are only used in recovery mode.

restore_command = 'copy "D:\\Coding\\mbd\\log\\%f" "%p"'        # command to use to restore an archived WAL file
                # placeholders: %p = path of file to restore
                #               %f = file name only
                # e.g. 'cp /mnt/server/archivedir/%f %p'
```

```
# - 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 proceed
                # (change requires restart)
recovery_target_time = '2023-12-01 18:44:48 +07'    # the time stamp up to which recovery will proceed
                # (change requires restart)
```

Untuk memasuki mode recovery, buat sebuah file kosong bernama recovery.signal pada folder data PostgreSQL seperti berikut.



Kemudian, nyalakan kembali PostgreSQL. Ketika dinyalakan kembali, PostgreSQL akan memasuki mode recovery dan akan mengembalikan basis data ke keadaan sesuai dengan *physical backup* yang telah dilakukan. Kemudian, akan muncul log PostgreSQL yang menggambarkan proses recovery sebagai berikut.

```
16 > data > log > postgresql-2023-12-01_190612.log
1   2023-12-01 19:06:12.560 +07 [11376] LOG:  database system was interrupted; last known up at 2023-12-01 18:44:47 +07
2   2023-12-01 19:06:12.932 +07 [11376] LOG:  starting point-in-time recovery to 2023-12-01 18:44:48+07
3   2023-12-01 19:06:12.968 +07 [11376] LOG:  restored log file "000000050000000000000052" from archive
4   2023-12-01 19:06:12.991 +07 [11376] LOG:  redo starts at 0/52000028
5   2023-12-01 19:06:12.992 +07 [11376] LOG:  consistent recovery state reached at 0/52000100
6   2023-12-01 19:06:13.019 +07 [11376] LOG:  restored log file "000000050000000000000053" from archive
7   2023-12-01 19:06:13.041 +07 [11376] LOG:  recovery stopping before commit of transaction 844, time 2023-12-01 18:51:33.564093+07
8   2023-12-01 19:06:13.041 +07 [11376] LOG:  pausing at the end of recovery
9   2023-12-01 19:06:13.041 +07 [11376] HINT:  Execute pg_wal_replay_resume() to promote.
```

Dapat dilihat bahwa basis data dikembalikan kepada keadaan pada titik waktu **2023-12-01 18:44:48+07** sesuai dengan titik waktu dilakukan *physical backup* dan perubahan yang dilakukan setelah titik waktu tersebut tidak dijalankan kembali. Dengan begitu, penambahan tabel “test_6”, “test_7”, “test_8”, dan “test_9” yang dilakukan pada titik waktu **2023-12-01 18:51:33.564093+07** tidak terdapat pada basis data setelah proses recovery.

4. Pembagian Kerja

NIM	Nama	Bagian
13521042	Kevin John W Hutabarat	Implementasi Concurrency Control Protocol
13521050	Naufal Syifa Firdaus	Eksplorasi Transaction Isolation
13521118	Ahmad Ghulam Ilham	Eksplorasi Recovery
13521150	I Putu Bakta Hari Sudewa	Implementasi Concurrency Control Protocol

Referensi

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts (7th ed.)*. New York, NY: McGraw-Hill Education.

PostgreSQL: Documentation: 16: 30.3. Write-Ahead Logging (WAL)

PostgreSQL: Documentation: 16: 26.3. Continuous Archiving and Point-in-Time Recovery (PITR)

PostgreSQL Point In Time Recovery (tutorialdba.com)