TUGAS BESAR

IF3140 - Manajemen Basis Data

MEKANISME CONCURRENCY CONTROL DAN RECOVERY

Disusun oleh:

Lucky Jonathan Chandra 13517136

Bryan Bernigen 13520034

Ghebyon Tohada Nainggolan 13520079

Nicholas Budiono 13520121

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

A. Eksplorasi Concurrency Control

- Serializability

Serializability adalah derajat isolasi paling tinggi, transaksi konkuren dapat dipastikan, hasil akan dipastikan sama jika transaksi dilakukan dalam urutan tertentu. Satu transaksi tidak akan overlap dengan transaksi yang lain. Ini mengartikan jika kita ingin menjalankan beberapa transaksi, meskipun dijalankan dalam urutan yang berbeda, hasil nya akan tetap sama.

- Repeatable Read

Repeatable Read adalah derajat isolasi setelah serializability, pada derajat ini, hasil query pasti akan menghasilkan output yang sama, seberapa banyak pun query tersebut dijalankan, jika ada transaksi yang melakukan pengubahan, query akan tetap bisa, jika memenuhi syarat query.

- Read Committed

Read committed adalah derajat setelah derajat isolasi repeatable read, pada derajat ini, data hanya bisa dilihat jika data tersebut sudah di commit oleh transaksi yang lain.

- Read Uncommitted

Read uncommitted adalah derajat paling bawah, setelah read committed, pada derajat ini, transaksi dapat melihat data yang belum di commit.

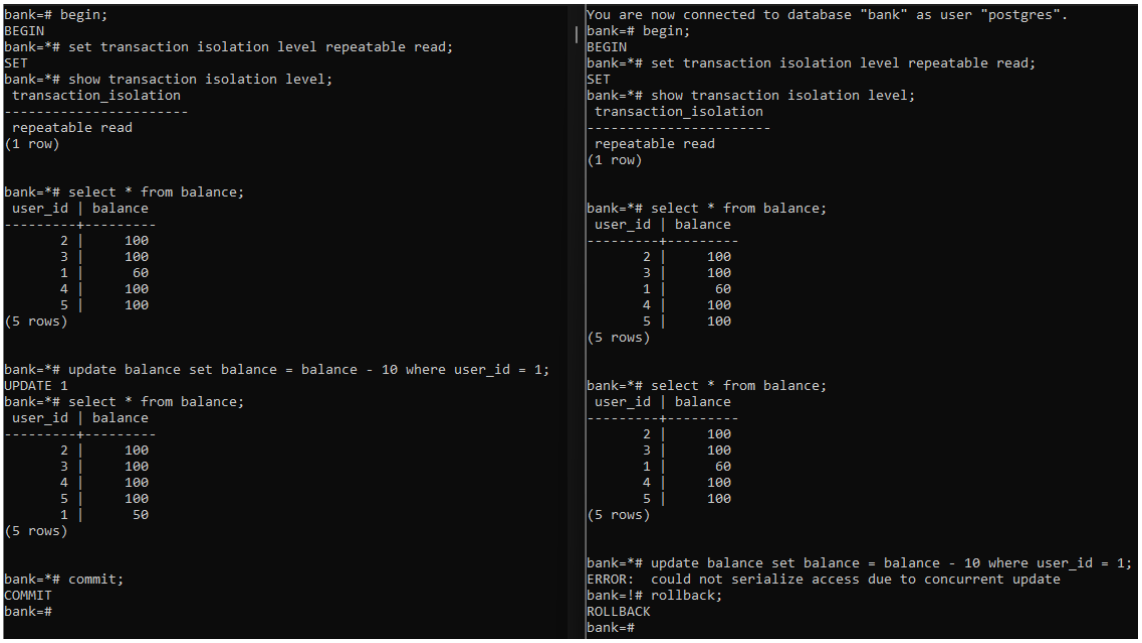
- Simulasi Read Committed

```
bank=# begin
bank-# ;
BEGIN
bank=*# set transaction isolation level read committed;

SET
bank=*# show transaction isolation level;
 transaction_isolation
-----------------------
 read committed
(1 row)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
(4 rows)


bank=*# insert into balance values(5,100);
INSERT 0 1
bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
       5 |     100
(5 rows)


bank=*# commit;
COMMIT
bank=#
```

```
bank=# begin;
BEGIN
bank=*# set transaction isolation level read committed;
SET
bank=*# show transaction isolation level;
 transaction_isolation
-----------------------
 read committed
(1 row)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
(4 rows)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
(4 rows)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
       5 |     100
(5 rows)


bank=*# D
```

Dari foto diatas, dengan derajat isolasi merupakan read committed, dapat dilihat bahwa data yang dimasukan di terminal kiri tidak tertulis di terminal kanan. Hal ini dikarenakan perubahan pada terminal kiri belum di commit. Jika sudah, maka akan dihasilkan hasil yang sama pada terminal di kanan. Permasalahan ini dinamakan phantom read, dimana hasil dari query berbeda dengan hasil dari execute sebelumnya dikarenakan oleh hasil belum di commit.

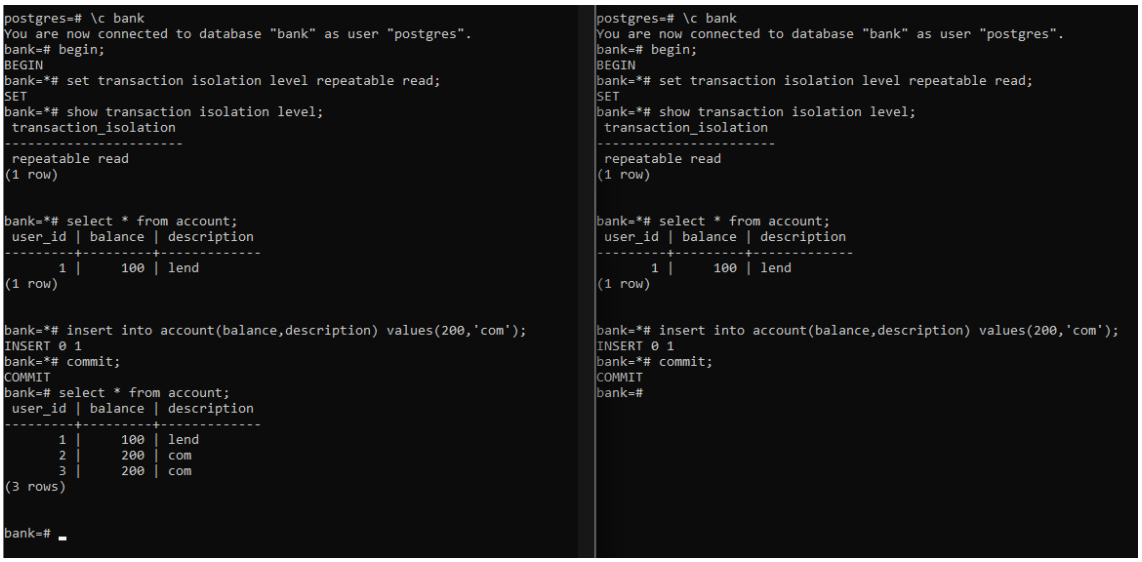
- Simulasi Repeatable Read



```
bank=# begin;
BEGIN
bank=*# set transaction isolation level repeatable read;
SET
bank=*# show transaction isolation level;
 transaction_isolation
-----------------------
 repeatable read
(1 row)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
       5 |     100
(5 rows)


bank=*# update balance set balance = balance - 10 where user_id = 1;
UPDATE 1
bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       4 |     100
       5 |     100
       1 |      50
(5 rows)


bank=*# commit;
COMMIT
bank=#
```

```
You are now connected to database "bank" as user "postgres".
bank=# begin;
BEGIN
bank=*# set transaction isolation level repeatable read;
SET
bank=*# show transaction isolation level;
 transaction_isolation
-----------------------
 repeatable read
(1 row)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
       5 |     100
(5 rows)


bank=*# select * from balance;
 user_id | balance
---------+---------
       2 |     100
       3 |     100
       1 |      60
       4 |     100
       5 |     100
(5 rows)


bank=*# update balance set balance = balance - 10 where user_id = 1;
ERROR:  could not serialize access due to concurrent update
bank=!# rollback;
ROLLBACK
bank=#
```

Dari foto di atas, terlihat bahwa derajat isolasi di kedua terminal adalah repeatable read, pada terminal di kiri, dilakukan sebuah update terhadap table, terlihat bahwa pada terminal di kanan, hasil perubahan tidak terlihat sebelum di commit, ketika sudah di commit, baru bisa dilihat, tetapi jika ingin dilakukan update di data yang sama, akan diberikan error, hal ini dikarenakan data pada user_id sudah dikurangi 10 oleh terminal di kiri, sehingga query untuk mengubah data yang sama tidak akan bisa dijalankan, yang perlu dilakukan adalah rollback.

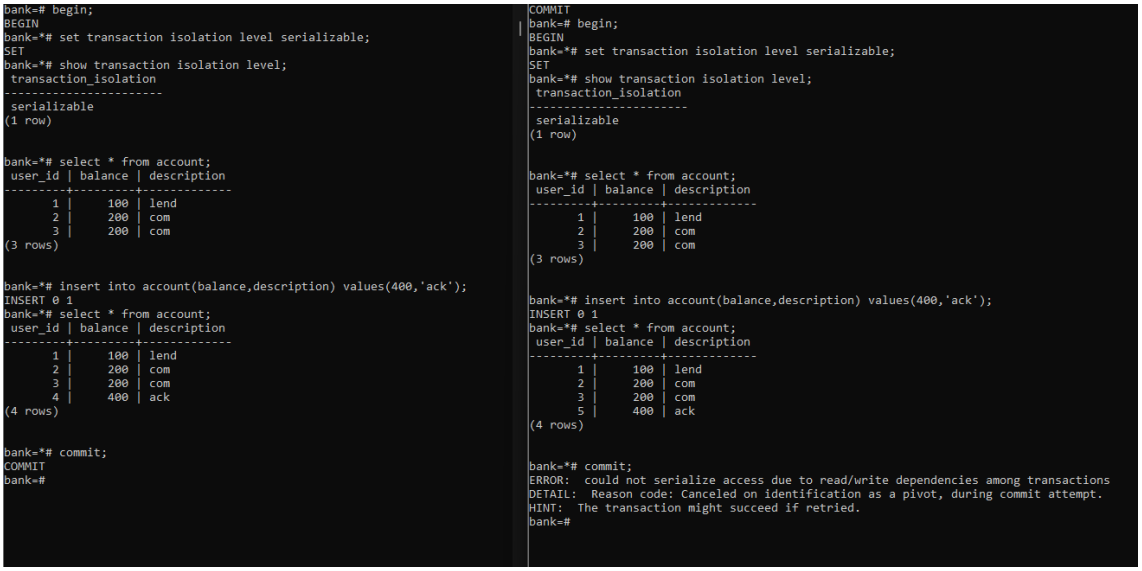


```
postgres=# \c bank
You are now connected to database "bank" as user "postgres".
bank=# begin;
BEGIN
bank=*# set transaction isolation level repeatable read;
SET
bank=*# show transaction isolation level;
 transaction_isolation
-----------------------
 repeatable read
(1 row)


bank=*# select * from account;
 user_id | balance | description
---------+---------+-------------
       1 |     100 | lend
(1 row)


bank=*# insert into account(balance,description) values(200,'com');
INSERT 0 1
bank=*# commit;
COMMIT
bank=# select * from account;
 user_id | balance | description
---------+---------+-------------
       1 |     100 | lend
       2 |     200 | com
       3 |     200 | com
(3 rows)


bank=# _
```

```
postgres=# \c bank
You are now connected to database "bank" as user "postgres".
bank=# begin;
BEGIN
bank=*# set transaction isolation level repeatable read;
SET
bank=*# show transaction isolation level;
 transaction_isolation
-----------------------
 repeatable read
(1 row)


bank=*# select * from account;
 user_id | balance | description
---------+---------+-------------
       1 |     100 | lend
(1 row)


bank=*# insert into account(balance,description) values(200,'com');
INSERT 0 1
bank=*# commit;
COMMIT
bank=#
```

Untuk mensimulasikan serializability, dibutuhkan simulasi lagi, pada foto di atas, terlihat bahwa kedua terminal ingin menambahkan data yang sama, ketika kedua terminal dijalankan, data yang masuk ke table ada dua yang ter-duplicate, hal ini mengartikan bahwa tidak ada serializability pada derajat isolasi.

- Simulasi Serializability



Pada foto di atas, terlihat bahwa derajat isolasi yang digunakan adalah serializable pada kedua terminal, dengan pengetahuan bahwa terjadi permasalahan pada derajat isolasi repeatable read, dilihat bahwa jika ingin memasukkan data yang sama, tidak akan ada masalah sebelum di commit, tetapi setelah di commit, terminal di kanan mengeluarkan error bahwa pengubahan data yang dilakukan pada terminal di kiri telah dilakukan dan pengubahan data di terminal kanan akan menghasilkan hal yang sama, pada derajat isolasi ini, dapat diketahui hal demikian dan dapat dicegah sehingga tidak akan ada duplikat data.

B. Implementasi Concurrency Control Protocol

Concurrency control protocol diimplementasikan diatas struktur dasar program tugas concurrency CPSC 438/538. Program tersebut telah menyediakan struktur data dan implementasi untuk transaction, threading, dan test case. Kami hanya perlu untuk melakukan implementasi kode program sesuai dengan arahan yang ada di README.MD. Berikut merupakan link yang kami gunakan sebagai fondasi implementasi kami:

https://github.com/banerjs/concurrency/tree/5635c7fc05f69886889eb4af2a59379515e7edba

```
------------------------------------------------
Part 1A: Simple Locking (exclusive locks only)
------------------------------------------------

Once you've looked through the code and are somewhat familiar with the overall structure and flow, you'll implement a simplified
version of two-phase locking. The protocol goes like this:

1) Upon entering the system, each transaction requests an EXCLUSIVE lock on EVERY item that it will either read or write.
2) Wait until ALL locks are granted. This might be immediately, or the transaction might have to wait for earlier transactions
to release their locks on the relevant keys.
3) Execute program logic.
4) Release ALL locks at commit/abort time.

Two additional rules:
A) Transactions must submit their entire set of lock requests ATOMICALLY. This means that if any of Txn A's lock requests is
submitted before some lock request of Txn B, then ALL of Txn A's lock requests must be submitted before ANY of Txn B's. In
practice, we do this by having a single thread do all concurrency control work.
B) Each lock must be granted to transactions in the order that they requested it. So if Txn A and Txn B both requested a lock on
the record with key "X", and A was the first to request the lock, then A must also be the first to be granted the lock.

Note that the transaction's read and write sets are always declared up front in 'Txn::readset_' and 'Txn::writeset_'. When
executing, txns are allowed to read records whose keys appear in EITHER its readset or its writeset, and it may write records
whose keys appear in writeset.

To help you get comfortable using the transaction processing framework, most of this algorithm is already implemented in
'TxnProcessor::RunScheduler1()'. Locks are requested and released at all the right times, and all necessary data structures for
an efficient lock manager are already in place. All you need to do is implement the 'WriteLock', 'Release', and 'Status' methods
in the class 'LockManagerA'.

The test file 'txn/lock_manager_test.cc' provides some rudimentary correctness tests for your lock manager implementations, but
additional tests may be added when we grade the assignment. We therefore suggest that you augment the tests with any additional
cases you can think of that the existing tests do not cover.
```

```
------------------------------------------------
Part 2: Serial Optimistic Concurrency Control (OCC)
------------------------------------------------

For OCC, you will have to implement the 'TxnProcessor::RunOCCScheduler' method. To test the correctness of your OCC
implementation, you may want to add one or more new tests to txn/txn_processor_test.cc, but these should not be submitted.

This is a simplified version of OCC compared to the one presented in the paper.

Pseudocode for the OCC algorithm to implement (in the RunOCCScheduler method):

  while (true) {
    Get the next new transaction request (if one is pending) and start it running.
    Deal with all transactions that have finished running.
  }

  Starting a transaction:
    Record start time.
    Start transaction running in its own thread.

  Dealing with a finished transaction:
    // Validation phase:
    for (each record whose key appears in the txn's read and write sets) {
      if (the record was last updated AFTER this transaction's start time) {
        Validation fails!
      }
    }
    // Commit/restart
    if (validation failed) {
      Completely restart the transaction.
    } else {
      Apply all writes.
      Mark transaction as committed.
    }
```

1. Simple Locking (Exclusive Lock Only)

Untuk menangani locking, terdapat sebuah kelas LockManager sebagai berikut

```
class LockManager {

 public:
```

```
 virtual ~LockManager() {}

 virtual bool ReadLock(Txn* txn, const Key& key) = 0;

 virtual bool WriteLock(Txn* txn, const Key& key) = 0;

 virtual void Release(Txn* txn, const Key& key) = 0;

 virtual LockMode Status(const Key& key, vector<Txn*>* owners) = 0;


protected:

 struct LockRequest {

   LockRequest(LockMode m, Txn* t) : txn_(t), mode_(m) {}

   Txn* txn_;       // Pointer to txn requesting the lock.

   LockMode mode_;  // Specifies whether this is a read or write lock request.

 };

 unordered_map<Key, deque<LockRequest>*> lock_table_;

 deque<Txn*>* ready_txns_;

 unordered_map<Txn*, int> txn_waits_;

};
```

Berikut Merupakan Penjelasan beberpa atribut dan method yang berperan penting dalam melakukan Simple Locking:
Catatan: Method yang diberi stabilo kuning merupakan method yang diimplementasikan

ReadLock() & WriteLock()	Method yang dipanggil ketika transaction merequest Lock (Keduanya sama karena Exclusive Lock Only)
Release()	Method yang dipanggil ketika melepas Lock
Status()	Method yang digunakan untuk mengecek status kepemilikan sebuah Lock
Struct LockRequest	Struktur data yang digunakan ketika sebuah transaksi merequest lock (merupakan struktur data yang dimasukan kedalam antrian lock juga)
lock_table_	Atribut yang menyimpan seluruh data permintaan Lock

ready_txns_	Transaksi yang siap untuk di eksekusi
txn_waits_	Antribut yang menyimpan data berapa banyak lock yang ditunggu oleh sebuah transaksi

Algoritma Locking In a nutshell (Untuk lengkapnya silahkan cek kode program. Harusnya komentar sudah cukup lengkap untuk memahami keseluruhan kode)

A) Request Lock
   1. Ketika ada request lock A oleh transaksi T, maka cek di lock_table apakah pernah ada yang request lock A juga
   2. Jika tidak ada, maka Lock tinggal diambil dan kasih tau lock table bahwa lock A sedang diambil transaksi T
   3. Jika pernah ada request, cek apakah lock tersebut masih dipegang oleh yang merequest.
   4. Jika sudah tidak dipegang, maka ambil lock dan kasih tau lock_table bahawa lock A sedang diambil
   5. Jika sedang dipegang, maka masukan transaksi T ke antrian menunggu lock A dan beri tahu txn_waits bahwa transaksi T sedang menunggu kunci:

B) Lepas Lock
   1. Jika transaksi T ingin melepas lock A, maka cek di lock_table apakah benar transaksi T sedang memegang lock A
   2. Jika sedang tidak memegang lock A, maka cukup keluarkan transaksi T dari antrian menunggu lock A (transaction T bukan mau melepas lock A melainkan ingin membatalkan permintaan lock A)
   3. Jika sedang memeganng lock A, maka lepas Lock A lalu cek apakah ada antrian untuk lock A (ada transaksi lain yang ingin mengambil lock A)
   4. Jika tidak ada maka selesai (cukup lepas lock di step 3)
   5. Jika ada yang menuggu, maka berikan lock ke trasaksi selanjutnya di antrian dan beri tahu txn_waits bahwa transaksi tersebut telah menerima 1 lock

i) Testing

Metode Testing:
Testing dilakukan dengan script test dan akan dilakukan Assert untuk memastikan bahwa algoritma locking sesuai dengan yang diharapkan

Test 1

```
TEST(LockManagerA_SimpleLocking) {
  deque<Txn*> ready_txns;
  LockManagerA lm(&ready_txns);
  vector<Txn*> owners;

  Txn* t1 = reinterpret_cast<Txn*>(1);
  Txn* t2 = reinterpret_cast<Txn*>(2);
```

```
  Txn* t3 = reinterpret_cast<Txn*>(3);

  // Txn 1 acquires read lock.
  lm.ReadLock(t1, 101);
  ready_txns.push_back(t1);   // Txn 1 is ready.
  EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
  EXPECT_EQ(1, owners.size());
  EXPECT_EQ(t1, owners[0]);
  EXPECT_EQ(1, ready_txns.size());
  EXPECT_EQ(t1, ready_txns.at(0));

  // Txn 2 requests write lock. Not granted.
  lm.WriteLock(t2, 101);
  EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
  EXPECT_EQ(1, owners.size());
  EXPECT_EQ(t1, owners[0]);
  EXPECT_EQ(1, ready_txns.size());

  // Txn 3 requests read lock. Not granted.
  lm.ReadLock(t3, 101);
  EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
  EXPECT_EQ(1, owners.size());
  EXPECT_EQ(t1, owners[0]);
  EXPECT_EQ(1, ready_txns.size());

  // Txn 1 releases lock.  Txn 2 is granted write lock.
  lm.Release(t1, 101);
  EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
  EXPECT_EQ(1, owners.size());
  EXPECT_EQ(t2, owners[0]);
  EXPECT_EQ(2, ready_txns.size());
  EXPECT_EQ(t2, ready_txns.at(1));

  // Txn 2 releases lock.  Txn 3 is granted read lock.
  lm.Release(t2, 101);
  EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
  EXPECT_EQ(1, owners.size());
  EXPECT_EQ(t3, owners[0]);
  EXPECT_EQ(3, ready_txns.size());
  EXPECT_EQ(t3, ready_txns.at(2));

  END;
}
```

Test 2

```
TEST(LockManagerA_LocksReleasedOutOfOrder) {
 deque<Txn*> ready_txns;
 LockManagerA lm(&ready_txns);
 vector<Txn*> owners;

 Txn* t1 = reinterpret_cast<Txn*>(1);
 Txn* t2 = reinterpret_cast<Txn*>(2);
 Txn* t3 = reinterpret_cast<Txn*>(3);
 Txn* t4 = reinterpret_cast<Txn*>(4);

 lm.ReadLock(t1, 101);   // Txn 1 acquires read lock.
 ready_txns.push_back(t1);  // Txn 1 is ready.
 lm.WriteLock(t2, 101);  // Txn 2 requests write lock. Not granted.
 lm.ReadLock(t3, 101);   // Txn 3 requests read lock. Not granted.
 lm.ReadLock(t4, 101);   // Txn 4 requests read lock. Not granted.

 lm.Release(t2, 101);    // Txn 2 cancels write lock request.

 // Txn 1 should now have a read lock and Txns 3 and 4 should be next in line.
 EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
 EXPECT_EQ(1, owners.size());
 EXPECT_EQ(t1, owners[0]);

 // Txn 1 releases lock.  Txn 2 is granted read lock.
 lm.Release(t1, 101);
 EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
 EXPECT_EQ(1, owners.size());
 EXPECT_EQ(t3, owners[0]);
 EXPECT_EQ(2, ready_txns.size());
 EXPECT_EQ(t3, ready_txns.at(1));

 // Txn 3 releases lock.  Txn 4 is granted read lock.
 lm.Release(t3, 101);
 EXPECT_EQ(EXCLUSIVE, lm.Status(101, &owners));
 EXPECT_EQ(1, owners.size());
 EXPECT_EQ(t4, owners[0]);
 EXPECT_EQ(3, ready_txns.size());
 EXPECT_EQ(t4, ready_txns.at(2));

 END;
}
```

Hasil Testing

```
igen@DESKTOP-6OJ0V3G:/mnt/d/git/ITB/temp/MBD/Tubes2/ConcurrencyControl$ make test
+ cxx txn/lock_manager_test.cc
+ cxx txn/lock_manager.cc
+ cxx txn/txn_processor.cc
+ ld bin/txn/lock_manager_test
+ cxx txn/txn_processor_test.cc
+ ld bin/txn/txn_processor_test
== bin/txn/lock_manager_test ==
[ LockManagerA_SimpleLocking ] BEGIN
[ LockManagerA_SimpleLocking ] PASS
[ LockManagerA_LocksReleasedOutOfOrder ] BEGIN
[ LockManagerA_LocksReleasedOutOfOrder ] PASS
```

ii) Analisis

Program Berhasil Melewati kedua Test dengan detail sebagai berikut
Test1:

Command	Status	Assert

T1 request XL(101)	Kepemilikan XL(101) =T1
	Jumlah transaksi ready = 1 (T1)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t1, owners[0]);`
		`EXPECT_EQ(1, ready_txns.size());`
		`EXPECT_EQ(t1, ready_txns.at(0));`

T2 request XL(101)	T2 menunggu XL(101)
	Kepemilikan XL(101) = T1
	Jumlah transaksi ready = 1 (T1)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t1, owners[0]);`
		`EXPECT_EQ(1, ready_txns.size());`

T3 request XL(101)	T3 menunggu XL(101)
	Kepemilikan XL(101) = T1
	Jumlah transasi ready = 1 (T1)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t1, owners[0]);`
		`EXPECT_EQ(1, ready_txns.size());`

T1 lepas lock XL(101)	T1 lepas XL(101) maka T2 sekarang memiliki XL(101)
	Kepemilikan XL(101) = T2
	Jumlah transaksi ready = 2 (T1, T2)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t2, owners[0]);`
		`EXPECT_EQ(2, ready_txns.size());`
		`EXPECT_EQ(t2, ready_txns.at(1));`

T2 lepas lock XL(101)	T2 lepas XL(101) maka T3 sekarang memiliki XL(101)
	Kepemilikan XL(101) = T3
	Jumlah transaksi ready = 3 (T1,T2,T3)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t3, owners[0]);`
		`EXPECT_EQ(3, ready_txns.size());`
		`EXPECT_EQ(t3, ready_txns.at(2));`

Test2:

Command	Status	Assert
T1 request XL(101)	Kepemilikan XL(101) = 1	
T2 request XL(101)	Antrian XL(101) = T2	
T3 request XL(101)	Antrian XL(101) = T2,T3	
T4 request XL(101)	Antrian XL(101) = T2,T3,T4	
T2 lepas lock XL(101)	Kepemilikan XL(101) = T1 Antrian XL(101) = T3,T3	`EXPECT_EQ(1, owners.size());` `EXPECT_EQ(t1, owners[0]);`

T1 lepas lock XL(101)	T1 lepas lock XL(101), maka T3 akan memiliki XL(101)
	Jumlah ready transaction = 2 (T1, T3)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t3, owners[0]);`
		`EXPECT_EQ(2, ready_txns.size());`
		`EXPECT_EQ(t3, ready_txns.at(1));`

T3 lepas lock XL(101)	T3 lepas lock XL(101), maka T4 akan memiliki XL(101)
	Jumlah ready transaction = 3 (T1,T3,T4)
		`EXPECT_EQ(1, owners.size());`
		`EXPECT_EQ(t4, owners[0]);`
		`EXPECT_EQ(3, ready_txns.size());`
		`EXPECT_EQ(t4, ready_txns.at(2));`

2. Serial Optimistic Concurrency Control (OCC)

Untuk menangani OCC, terdapat kelas TxnProcessor Sebagai Berikut

```
class TxnProcessor {
 public:
  explicit TxnProcessor(CCMode mode);

  ~TxnProcessor();

  void NewTxnRequest(Txn* txn);

  Txn* GetTxnResult();

 private:
  void RunScheduler();

  void RunSerialScheduler();

  void RunLockingScheduler();
```

```
  void RunOCCScheduler();

  void RunOCCParallelScheduler();

  void ExecuteTxn(Txn* txn);

  void ApplyWrites(Txn* txn);

  CCMode mode_;

  StaticThreadPool tp_;

  Storage storage_;

  int next_unique_id_;
  Mutex mutex_;

  AtomicQueue<Txn*> txn_requests_;

  deque<Txn*> ready_txns_;

  AtomicQueue<Txn*> completed_txns_;

  AtomicQueue<Txn*> txn_results_;

  LockManager* lm_;
};
```

Berikut Merupakan Penjelasan beberpa atribut dan method yang berperan penting dalam melakukan Simple Locking:
Catatan: Method yang diberi stabilo kuning merupakan method yang diimplementasikan

RunOCCScheduler()	Method yang mengimplementasikan Serial Optimistic Concurrency Control
tp_	Atribut yang menyimpan main thread (setiap transaksi OCC berjalan di thread masing-masing yang nantinya kembali ke thread utama)
txn_request_	Atribut yang menyimpan data request transaksi yang perlu dijalankan
completed_txn_	Atribut yang menyimpan data transaksi-transaksi yang sudah selesai di

	eksekusi namun belum di commit/abort
txn_results_	Atribut yang menyimpan data transaksi yang telah di commit/abort
storage_	Storage utama yang menyimpan timestamp kapan suatu data dibaca/ditulis

Algoritma OCC In a nutshell (Untuk lengkapnya silahkan cek kode program. Harusnya komentar sudah cukup lengkap untuk memahami keseluruhan kode)

1. Jika ada request transaksi baru, biarkan transaksi jalan di thread sendiri
2. Jika ada transaksi yang sudah selesai, maka cek apakah transaksi tersebut mau commit/abort
3. Jika ingin abort, biarkan transaksi abort dan catat di txn_results_
4. Jika ingin commit, maka set transaksi valid
5. Laliu cek untuk semua data yang dibaca oleh transaksi ini. Jika ada data yang dibaca oleh transaksi ini namun ternyata selama transaksi ini berlangsung ada transaksi lain yang commit dan mengubah data yang dibaca oleh transaksi ini (ditandai dengan perubahan timestamp di storage utama), maka transaksi menjadi invalid
6. Jika setelah pengecekan transaksi valid, maka commit transaksi dan ubah data di local storage sesuai data di thread transaksi ini. Lalu catat transaksi berhasil di txn_result
7. Jika setelah pengecekan transaksi invalid, maka hapus seluruh data pembacan dan penulisan yang pernah dilakukan transaksi ini. Lalu masukkan ulang transaksi ini ke txn_request agar dijalankan ulang

i) Testing

Testing dilakukan dengan script test yang akan mengecek beberapa skenario ketika banyak transaksi dilakukan secara simultan pada pada sebuah database. Skenario yang akan di test adalah skenario dimana seluruh transaksi merupakan read only, lalu skenario dimana 1% transaksi tersebut merupakan write (sehingga mungkin ada rollback), lalu 10% write, 65% hingga 100% write. Lalu hasil yang dihasilkan merupakan throughput database.

```
int main(int argc, char** argv) {
  cout << "\t\t\t    Average Transaction Duration" << endl;
  cout << "\t\t0.1ms\t\t1ms\t\t10ms\t\t100ms";
  cout << endl;

  vector<LoadGen*> lg;

  cout << "Read only" << endl;
  lg.push_back(new RMWLoadGen(10000, 10, 0, 0.0001));
  lg.push_back(new RMWLoadGen(10000, 10, 0, 0.001));
  lg.push_back(new RMWLoadGen(10000, 10, 0, 0.01));
  lg.push_back(new RMWLoadGen(10000, 10, 0, 0.1));

  Benchmark(lg);
```

```
class RMWLoadGen : public LoadGen {
 public:
  RMWLoadGen(int dbsize, int rsetsize, int wsetsize, double wait_time)
    : dbsize_(dbsize),
      rsetsize_(rsetsize),
      wsetsize_(wsetsize),
      wait_time_(wait_time) {
  }
```

```
void Benchmark(const vector<LoadGen*>& lg) {
  // Number of transaction requests that can be active at any given time.
  int active_txns = 100;
  deque<Txn*> doneTxns;

  // Set initial db state.
  map<Key, Value> db_init;
  for (int i = 0; i < 10000; i++)
    db_init[i] = 0;

  // For each MODE...
  for (CCMode mode = OCC;
       mode <= P_OCC;
       mode = static_cast<CCMode>(mode+1)) {
    // Print out mode name.
    cout << ModeToString(mode) << flush;
```

```
// For each experiment...
for (uint32 exp = 0; exp < lg.size(); exp++) {
  int txn_count = 0;

  // Create TxnProcessor in next mode.
  TxnProcessor* p = new TxnProcessor(mode);

  // Initialize data with initial db state.
  Put init_txn(db_init);
  p->NewTxnRequest(&init_txn);
  p->GetTxnResult();

  // Record start time.
  double start = GetTime();

  // Start specified number of txns running.
  for (int i = 0; i < active_txns; i++)
    p->NewTxnRequest(lg[exp]->NewTxn());

  // Keep 100 active txns at all times for the first full second.
  while (GetTime() < start + 1) {
    doneTxns.push_back(p->GetTxnResult());
    txn_count++;
    p->NewTxnRequest(lg[exp]->NewTxn());
  }

  // Wait for all of them to finish.
  for (int i = 0; i < active_txns; i++) {
    doneTxns.push_back(p->GetTxnResult());
    txn_count++;
  }

  // Record end time.
  double end = GetTime();

  // Print throughput
  cout << "\t" << (txn_count / (end-start)) << "\t" << flush;

  // Delete TxnProcessor and completed transactions.
  doneTxns.clear();
  delete p;
}
```

```
    cout << endl;
  }
}
```

Berikut Merupakan Hasil Testing

Dengan algoritma OCC salah (langsung return)

```
void TxnProcessor::RunOCCScheduler()
{
  return;
}
```

	Average Transaction Duration			
	0.1ms	1ms	10ms	100ms
Read only				
OCC	2741.83	813.741	97.3965	9.97499
1% contention				
OCC	2382.31	824.12	93.5606	9.88881
10% contention				
OCC	2366.52	823.355	95.8439	9.87528
65% contention				
OCC	2002.07	694.857	96.728	9.99775
100% contention				
OCC	2112.47	666.311	96.3226	9.95858
High contention mixed read/write				
OCC	3180.14	2308.68	785.193	92.6797

Dengan Algoritma OCC yang benar

	Average Transaction Duration			
	0.1ms	1ms	10ms	100ms
Read only				
OCC	62175.9	61896.6	9465.3	932.788
1% contention				
OCC	29408.7	33638.9	3956.02	330.495
10% contention				
OCC	23700.5	8819.25	995.693	86.5172
65% contention				
OCC	4694.77	1663	181.229	17.1921
100% contention				
OCC	3267.7	888.847	100.268	9.9969
High contention mixed read/write				
OCC	8497.28	7388.86	2176.81	843.273

ii) Analisis Algoritma

Dapat dilihat bahwa dengan algoritma OCC, throughput akan sangat besar ketika seluruh transaksi merupakan read only karena read only transaction dapat dijalankan secara simultan tanpa ada satupun yang rollback. Rollback pada OCC hanya terjadi jika ada salah satu transaksi yang write sehingga membuat transaksi lain yang berjalan secara pararrel harus rollback (jika data yang dibaca beririsan dengan yang ditulis)

Dapat dilihat juga bahwa ketika banyak write, maka throughput OCC menjadi sangat buruk karena banyak transaksi yang harus rollback. Bahkan throughput OCC dapat dikalahkan oleh algoritma tidak jelas yang hanya return saja (baseline tanpa threading dan tanpa rollback).

Eksplorasi Recovery

Write Ahead a Log

Write-Ahead Logging adalah metode standar untuk memastikan integritas data, serta juga mendukung atomicity dan durability pada database. Konsep utama dari Write Ahead Log ialah sebelum data pada filedata (tabel) diubah, perlu dilakukan pencatatan perubahan yang terjadi ke dalam log. Log ini akan berguna salah satunya apabila terjadi kerusakan pada sistem database. Database dapat di-recover menggunakan log tersebut dengan cara roll-forward (REDO) dari *checkpoint* terakhir yang dicatat.

Continuous Archiving

Continuous Archiving adalah salah satu metode peng-arsipan dengan memanfaatkan Write Ahead a Log. Sesuai dengan namanya *Continuous Archiving* bertujuan untuk melakukan pengarsipan secara terus menerus, agar arsip-arsip data tersebut dapat dimanfaatkan untuk melakukan *recovery.* Hal yang sangat menguntungkan dari menerapkan *Continuous Archiving* adalah kita dapat melakukan *recover* sebebas mungkin(banyak pilihan untuk recover ke bagian mana), selagi arsip itu tersimpan.

Point-in-Time Recovery

Point-in-Time Recovery mengacu pada pemulihan data pada saat admin memulihkan sekumpulan data dari suatu waktu di masa lampau. Pada Postgresql Point-in-Time Recovery dapat dilakukan dengan menerapkan kedua metode sebelumnya yaitu, Continuous Archiving dan Write Ahead Log. Secara umum gambaran untuk melakukan *Point-in-Time Recovery* adalah sebagai berikut, admin menentukan waktu acuan untuk melakukan *recover.* Karena Log sudah ditulis dengan menggunakan *Continuous Archiving,* maka tinggal dilakukan eksekusi ulang dari *Checkpoint*. Eksekusi ulang dilakukan satu-persatu dengan *roll-forward* hingga didapatkan log dengan waktu yang sesuai dengan yang didefinisikan oleh admin

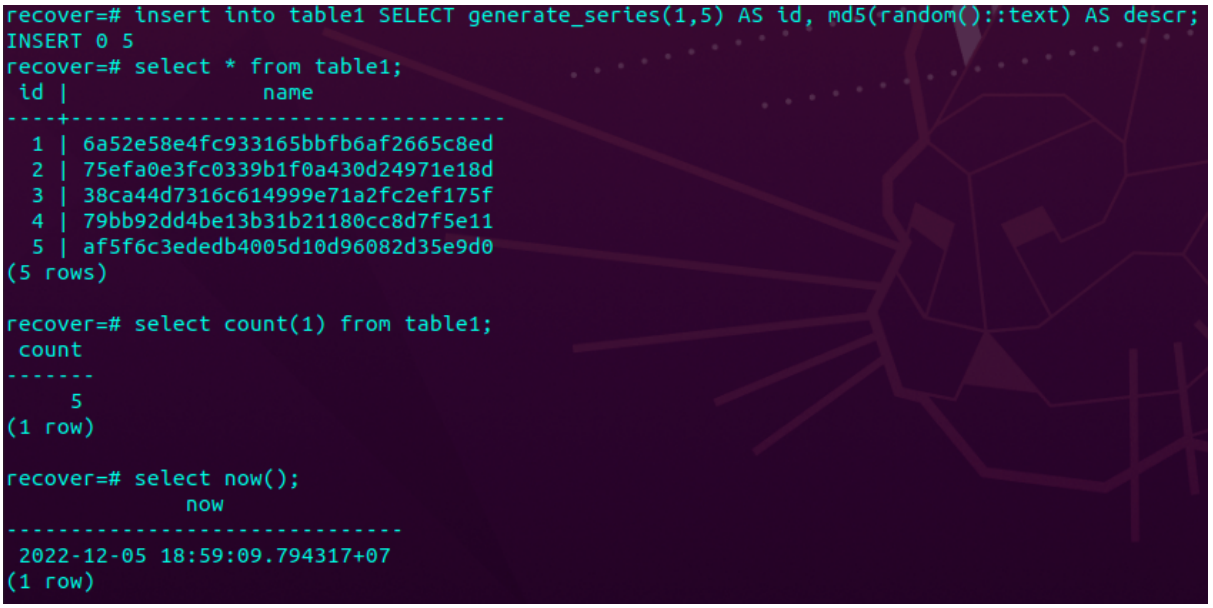
Simulasi Kegagalan pada PostgreSQL

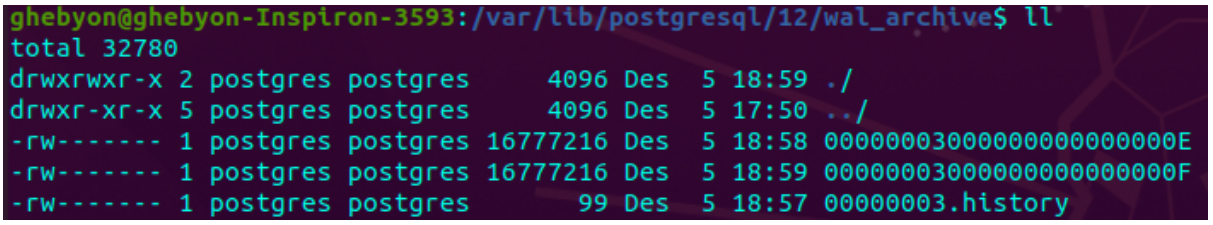
Sebelum melakukan simulasi, disiapkan terlebih dahulu folder ‘basebackup’ dan ‘wal_archive’. Kemudian ubah konfigurasi pada postgresql (pada contoh ini file konfigurasi terdapat pada path `/etc/postgresql/12/main/postgresql.conf`). Konfigurasi yang perlu diubah adalah sebagai berikut :

```
Wal_Level =replica or archive
archive_mode=on
archive_command = 'cp %p /var/lib/postgresql/12/wal_archive/%f'
archive_timeout = 60
```

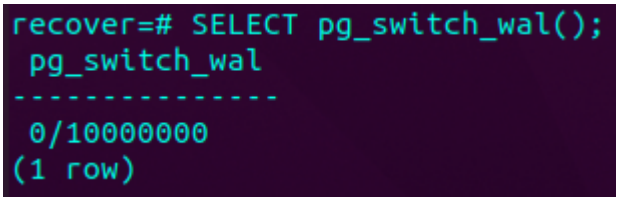
Data sebelum kegagalan terjadi

Insert 5 data pertama





Switch_wal



```
ghebyon@ghebyon-Inspiron-3593:/var/lib/postgresql/12/wal_archive$ ll
total 65552
drwxrwxr-x 2 postgres postgres     4096 Des  5 19:03 ./
drwxr-xr-x 5 postgres postgres     4096 Des  5 17:50 ../
-rw------- 1 postgres postgres 16777216 Des  5 18:58 00000003000000000000000E
-rw------- 1 postgres postgres 16777216 Des  5 18:59 00000003000000000000000F
-rw------- 1 postgres postgres 16777216 Des  5 19:02 000000030000000000000010
-rw------- 1 postgres postgres      340 Des  5 19:02 000000030000000000000010.00000028.backup
```

Sebelum dilakukan operasi selanjutnya, ditetapkan bahwa *state* saat ini adalah basebackup atau checkpoint yang. Untuk membuat checkpoint tersebut lakukan penghapusan data base backup sebelumnya dan isi dengan basebackup baru dengan command berikut

```
rm -rf /var/lib/postgresql/12/basebackup/*
sudo -u postgres pg_basebackup -D
/var/lib/postgresql/12/basebackup -Ft -P
```

Insert 5 data kedua

```
recover=# insert into table1 SELECT generate_series(6,10) AS id, md5(random()::text) AS descr;
INSERT 0 5
recover=# select * from table1;
 id |               name
----+----------------------------------
  1 | 6a52e58e4fc933165bbfb6af2665c8ed
  2 | 75efa0e3fc0339b1f0a430d24971e18d
  3 | 38ca44d7316c614999e71a2fc2ef175f
  4 | 79bb92dd4be13b31b21180cc8d7f5e11
  5 | af5f6c3ededb4005d10d96082d35e9d0
  6 | d23ec942131cc79f5d0b6895a3bc6b14
  7 | e70fda2a3b089a5d5bf2fffb89265886
  8 | 1d6ff952642ab108810f3584e9ba52b4
  9 | 467471ecca11bfe5b2ebc84d24b2230e
 10 | c28c861ea4814aee449d592be7ca97a7
(10 rows)

recover=# select count(1) from table1;
 count
-------
    10
(1 row)

recover=# select now();
              now
-------------------------------
 2022-12-05 19:03:31.543991+07
(1 row)
```

```
ghebyon@ghebyon-Inspiron-3593:/var/lib/postgresql/12/wal_archive$ ll
total 65552
drwxrwxr-x 2 postgres postgres     4096 Des  5 19:03 ./
drwxr-xr-x 5 postgres postgres     4096 Des  5 17:50 ../
-rw------- 1 postgres postgres 16777216 Des  5 18:58 00000003000000000000000E
-rw------- 1 postgres postgres 16777216 Des  5 18:59 00000003000000000000000F
-rw------- 1 postgres postgres 16777216 Des  5 19:02 000000030000000000000010
-rw------- 1 postgres postgres      340 Des  5 19:02 000000030000000000000010.00000028.backup
-rw------- 1 postgres postgres 16777216 Des  5 19:03 000000030000000000000011
-rw------- 1 postgres postgres       99 Des  5 18:57 00000003.history
```

Insert 5 data ketiga

```
recover=# insert into table1 SELECT generate_series(11,15) AS id, md5(random()::text) AS descr;
INSERT 0 5
recover=# select * from table1;
 id |               name
----+----------------------------------
  1 | 6a52e58e4fc933165bbfb6af2665c8ed
  2 | 75efa0e3fc0339b1f0a430d24971e18d
  3 | 38ca44d7316c614999e71a2fc2ef175f
  4 | 79bb92dd4be13b31b21180cc8d7f5e11
  5 | af5f6c3ededb4005d10d96082d35e9d0
  6 | d23ec942131cc79f5d0b6895a3bc6b14
  7 | e70fda2a3b089a5d5bf2fffb89265886
  8 | 1d6ff952642ab108810f3584e9ba52b4
  9 | 467471ecca11bfe5b2ebc84d24b2230e
 10 | c28c861ea4814aee449d592be7ca97a7
 11 | 022f5c0c478afaca6743cfc65974afb9
 12 | 3b8f65e42b9420c12e9a89acb6ff7527
 13 | 297836ccf1e273aefec992702c0643ce
 14 | 53ba28a38bf34215daab5f75168c554d
 15 | 41b6c571a372df6b570fd333409c5e7a
(15 rows)

recover=# select count(1) from table1;
 count
-------
    15
(1 row)

recover=# select now();
              now
-------------------------------
 2022-12-05 19:08:08.507882+07
(1 row)
```

Saat kegagalan terjadi

```
recover=# select * from table1;
FATAL:  terminating connection due to administrator command
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: Succeeded.
```

Data setelah dilakukan recovery

Karena metode recovery yang dipakai adalah Post in Time Recovery, maka tambahkan konfigurasi berikut pada file konfigurasi postgresql sebelumnya

```
restore_command = 'cp /var/lib/postgresql/12/wal_archive/%f %p'
recovery_target_time = '2022-12-05 19:03:31'
```

Recovery_target_time diisi dengan data bertipe datetime. Data tersebut akan dipakai untuk menargetkan sampai mana log akan dijalankan.

```
postgres=# \c recover;
You are now connected to database "recover" as user "postgres".
recover=# select * from table1;
 id |               name
----+----------------------------------
  1 | 6a52e58e4fc933165bbfb6af2665c8ed
  2 | 75efa0e3fc0339b1f0a430d24971e18d
  3 | 38ca44d7316c614999e71a2fc2ef175f
  4 | 79bb92dd4be13b31b21180cc8d7f5e11
  5 | af5f6c3ededb4005d10d96082d35e9d0
  6 | d23ec942131cc79f5d0b6895a3bc6b14
  7 | e70fda2a3b089a5d5bf2fffb89265886
  8 | 1d6ff952642ab108810f3584e9ba52b4
  9 | 467471ecca11bfe5b2ebc84d24b2230e
 10 | c28c861ea4814aee449d592be7ca97a7
(10 rows)
```

Recovery dilakukan dimulai dari basebackup atau checkpoint untuk dimulainya recovery. Kemudian dilakukan pembacaan log secara roll-forwad dan dilakukan eksekusi terhadap database sesuai dengan log yang dibaca. Hingga akhirnya ditemukan waktu pada log yang sesuai dengan target waktu yang diinputkan oleh admin.

Kesimpulan dan Saran

Kesimpulan

Database pada umumnya menyediakan beberapa level isolasi yang dapat diubah sesuai dengan kebutuhan pengguna. Derajat isolasi yang tinggi akan membuat data integrity terjamin namun dengan mengorbankan throughput yang relatif lebih rendah dibanding derajat isolasi rendah yang memiliki throughput tinggi namun data integrity yang kurang terjamin. Oleh karena itu user dapat menyesuaikan kebutuhannya sendiri yakni perpaduan antara throughput dan data integrity

Pada umumnya, database menggunakan derajat isolasi tinggi yang menjamin integritas data yang ada. Walaupun throughput database dengan derajat isolasi tinggi cenderung lebih rendah dibanding derajat isolasi rendah, namun ada beberapa protokol concurrency control yang dapat digunakan untuk meningkatkan throughput. Beberapa protokol tersebut adalah Simple Locking dan Serial Optimistic Concurrency Control. Dengan menerapkan protokol tersebut, throughput dapat meningkat dibandingkan level serial terutama jika kebanyakan transaksi merupakan transaksi read only.

Sebuah database tidak terbebas dari yang namanya error. Error dapat terjadi kapan saja sehingga perlu metode untuk menangani error tersebut agar data tidak hilang. Untuk menangani error tersebut, postgre menyediakan beberapa metode recovery terhadap error yakni Write-Ahead Log, Continuous Archiving, dan Point-in-Time Recovery. Write-Ahead Log merupakan metode paling standar yang biasa digunakan. Continous Archiving memanfaatkan Write-Ahead Log untuk recovery sedangkan Point-in-Time Recovery memanfaatkan keduanya untuk recovery.

Saran

Sebaiknya untuk tugas implementasi concurrency control, seluruh kelompok diberikan template github classroom seperti tugas lab sister (misal orkom atau jarkom). Dengan demikian, seluruh kelompok akan menghasilkan implementasi yang sesuai dengan harapan para asisten. Hal tersebut kami rasa perlu karena implementasi concurrency control yang cukup rumit jika benar-benar diimplementasikan dari scratch (terutama pada bagian threading di OCC/MVCC)

Pembagian Tugas

NIM	Nama	Tugas
13517136	Lucky Jonathan Chandra	Implementasi Concurrency Control Protocol
13520034	Bryan Bernigen	Implementasi Concurrency Control Protocol, Kesimpulan dan Saran
13520079	Ghebyon Tohada Nainggolan	Eksplorasi Recovery pada Postgresql
13520121	Nicholas Budiono	Eksplorasi Concurrency Control

Referensi

1. https://github.com/banerjs/concurrency/tree/5635c7fc05f69886889eb4af2a59379515e7edba
2. https://www.postgresql.org/docs/12/continuous-archiving.html
3. https://www.youtube.com/watch?v=8jYPu40x-3E
4. https://dev.to/techschoolguru/understand-isolation-levels-read-phenomena-in-mysql-postgres-c2e
5. https://www.geeksforgeeks.org/transaction-isolation-levels-dbms/