

IF3140 MANAJEMEN BASIS DATA

MEKANISME CONCURRENCY CONTROL DAN RECOVERY



K02 Kelompok 10

Anggota :

Rachel Gabriela Chen	13521044
Eugene Yap Jin Quan	13521074
Fazel Ginanda	13521098
Muhammad Zaydan Athallah	13521104

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

Daftar Isi.....	1
1. Eksplorasi Transaction Isolation.....	2
a. Serializable.....	3
b. Repeatable Read.....	3
c. Read Committed.....	4
d. Read Uncommitted.....	4
e. Simulasi Transaction Isolation.....	4
a. Simulasi Serializability.....	5
b. Simulasi Repeatable Read.....	6
c. Simulasi Read Committed.....	8
2. Implementasi Concurrency Control Protocol.....	10
a. Two-Phase Locking (2PL).....	10
b. Optimistic Concurrency Control (OCC).....	22
c. Multiversion Timestamp Ordering Concurrency Control (MVCC).....	25
3. Eksplorasi Recovery.....	32
a. Write-Ahead Log.....	32
b. Continuous Archiving.....	32
c. Point-in-Time Recovery.....	33
d. Simulasi Kegagalan pada PostgreSQL.....	33
4. Kesimpulan dan Saran.....	38
5. Pembagian Kerja.....	40
Referensi.....	41

1. Eksplorasi Transaction Isolation

Derajat isolasi pada PostgreSQL mengacu pada tingkat penguncian dan isolasi yang diterapkan pada transaksi, mempengaruhi bagaimana transaksi bersaing untuk mengakses dan memanipulasi data.

Standar SQL menetapkan empat tingkatan isolasi transaksi, yang disusun berdasarkan tingkat ketaatannya, yaitu *serializable*, *repeatable read*, *read committed*, dan *read uncommitted*. Setiap tingkatan dijelaskan dengan menentukan fenomena-fenomena yang tidak diperbolehkan terjadi dalam transaksi.

1. Dirty read

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

2. Non Repeatable read

Transaksi mengulang membaca data yang sebelumnya telah diakses, tetapi pada kali tersebut menemukan bahwa data tersebut telah mengalami perubahan oleh transaksi lain yang sudah di-*commit*.

3. Phantom read

Transaksi menjalankan ulang suatu *query* yang menghasilkan baris-baris sesuai dengan kondisi pencarian, namun pada saat dieksekusi kembali, baris-baris tersebut sudah mengalami perubahan akibat dari transaksi lain yang telah dikonfirmasi.

4. Serialization anomaly

Ketika sebuah kelompok transaksi di-*commit*, hasilnya tidak sesuai dengan hasil yang mungkin terjadi jika transaksi-transaksi tersebut dieksekusi satu per satu secara berurutan.

Tabel 1.1 Level Isolasi Transaksi

Isolation Level	<i>Dirty Read</i>	<i>Non-Repeatable Read</i>	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

a. *Serializable*

Serializable menjamin tingkat konsistensi tertinggi dengan menerapkan penguncian eksklusif pada data yang sedang diakses, mencegah transaksi lain untuk membaca atau mengubah data tersebut selama transaksi berlangsung. Hal ini menghindari berbagai anomali transaksi, seperti *non-repeatable read*, *phantom read*, dan *dirty read*, untuk memastikan hasil transaksi yang lebih konsisten dan dapat diprediksi. Namun, perlu diingat bahwa *Serializability* dapat berpotensi mempengaruhi kinerja karena transaksi mungkin saling memblokir untuk menjaga integritas data.

b. *Repeatable Read*

Repeatable Read adalah tingkat isolasi transaksi yang mencegah pembacaan tak terulang, meskipun memungkinkan terjadinya *phantom read*. Dengan menerapkan penguncian untuk memblokir akses dan perubahan data yang sedang diakses oleh transaksi lain, *Repeatable Read* memastikan bahwa transaksi saat ini tidak akan melihat hasil dari transaksi baru yang dimulai setelah transaksi saat ini dimulai. Meskipun tingkat isolasi ini memberikan tingkat konsistensi yang baik, tetapi dapat menyebabkan penguncian yang lebih intensif dibandingkan dengan tingkat isolasi yang lebih rendah sehingga mempengaruhi kinerja aplikasi.

c. Read Committed

Read Committed adalah tingkat isolasi di mana transaksi hanya dapat melihat data yang telah di-commit oleh transaksi lain. Ini memastikan bahwa transaksi tidak akan melihat perubahan yang belum dikonfirmasi, menghindari *dirty read* (membaca data yang belum di-commit). Meskipun tingkat isolasi ini menghindari anomali *dirty read* dan memberikan keseimbangan antara konsistensi dan kinerja, masih memungkinkan terjadinya *repeatable read* dan *phantom read*, di mana transaksi dapat melihat hasil yang baru dimasukkan atau dihapus oleh transaksi lain setelah transaksi saat ini dimulai.

d. Read Uncommitted

Read Uncommitted adalah tingkat isolasi transaksi yang paling rendah dalam sistem manajemen basis data, seperti PostgreSQL. Pada tingkat ini, transaksi dapat melihat data yang sedang dalam proses perubahan oleh transaksi lain, termasuk data yang belum di-commit. Hal ini membuat mungkin terjadinya *dirty read*, *repeatable read*, dan *phantom read*. Meskipun tingkat isolasi ini dapat meningkatkan kinerja sistem karena minimnya penguncian, penggunaan *Read Uncommitted* harus dipertimbangkan dengan hati-hati karena dapat menyebabkan hasil transaksi yang tidak konsisten dan menghadirkan risiko konflik data. Tingkat isolasi ini seringkali digunakan dalam skenario di mana kinerja lebih diutamakan daripada konsistensi data yang ketat.

e. Simulasi Transaction Isolation

Untuk memahami perbedaan antara berbagai tingkatan isolasi transaksi, simulasi akan dilakukan pada PostgreSQL. Proses simulasi akan melibatkan dua terminal, di mana satu terminal digunakan untuk mengeksekusi *query*, sedangkan terminal lainnya digunakan untuk memeriksa tingkat isolasi. Penting untuk dicatat bahwa level isolasi transaksi *read uncommitted* tidak akan disimulasikan pada PostgreSQL karena level tersebut tidak tersedia.

a. Simulasi *Serializability*

```
postgres=# begin;
BEGIN
postgres=# set transaction isolation level serializable;
SET
postgres=# show transaction isolation level;
transaction_isolation
-----
serializable
(1 row)

postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
(7 rows)

postgres=# insert into "Person"(cars) values(99);
INSERT 0 1
postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
(8 rows)

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

```
postgres=# begin;
BEGIN
postgres=# set transaction isolation level serializable;
SET
postgres=# show transaction isolation level;
transaction_isolation
-----
serializable
(1 row)

postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
(7 rows)

postgres=# insert into "Person"(cars) values(11);
INSERT 0 1
postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          9 | 11
(8 rows)

postgres=# 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.
postgres=#
```

Pada derajat isolasi *serializable* pada kedua terminal, tidak akan terjadi masalah sebelum dilakukan *commit*. Ketika terminal kiri melakukan *commit* tidak akan terjadi masalah dan ketika terminal kanan melakukan *commit* terminal akan menghasilkan *error* bahwa transaksi tidak dapat dilakukan. Hal ini terjadi karena masih terdapat fenomena *serializable anomaly* yaitu transaksi yang dijalankan tidak dapat dibuat seolah-olah serial sehingga mengakibatkan terjadinya *serializable failure*.

Pada simulasi ini, dapat disimpulkan bahwa level isolasi transaksi *serializable* tidak memperbolehkan keempat fenomena *dirty read*, *non-repeatable read*, *phantom read*, dan *serialization anomaly*.

b. Simulasi *Repeatable Read*

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

postgres=# select * from "Person";
 person_id | cars
-----+-----
1 | 5
2 | 10
3 | 15
4 | 120
5 | 20
6 | 55
7 | 77
(7 rows)

postgres=# update "Person" set cars = cars - 4 where person_id = 1;
UPDATE 1
postgres=# COMMIT;
COMMIT
postgres=# select * from "Person"
postgres=# ;
 person_id | cars
-----+-----
2 | 10
3 | 15
4 | 120
5 | 20
6 | 55
7 | 77
1 | 1
(7 rows)

postgres=#
```

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

postgres=# select * from "Person";
 person_id | cars
-----+-----
1 | 5
2 | 10
3 | 15
4 | 120
5 | 20
6 | 55
7 | 77
(7 rows)

postgres=# select * from "Person";
 person_id | cars
-----+-----
1 | 5
2 | 10
3 | 15
4 | 120
5 | 20
6 | 55
7 | 77
(7 rows)

postgres=# update "Person" set cars = cars - 4 where person_id = 1;
ERROR:  could not serialize access due to concurrent update
postgres=# select * from "Person";
ERROR:  current transaction is aborted, commands ignored until end of transaction block
postgres=# commit;
ROLLBACK
postgres=# select * from "Person";
 person_id | cars
-----+-----
2 | 10
3 | 15
4 | 120
5 | 20
6 | 55
7 | 77
1 | 1
(7 rows)
```

Pada terminal di kiri, dilakukan sebuah update terhadap tabel, terlihat bahwa pada terminal di kanan, hasil perubahan tidak terlihat sebelum di commit. Ketika dilakukan update pada data yang sama, akan diberikan pesan *error*, hal ini karena data sudah dikurangi 4 oleh terminal di kiri, sehingga *query* untuk mengubah data yang sama tidak akan bisa dijalankan, sehingga perlu dilakukan adalah rollback untuk menghindari *Repeatable Read*.

```

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

postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
(8 rows)

postgres=# insert into "Person"(cars) values(22);
INSERT 0 1
postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
         10 | 22
(9 rows)

postgres=# commit;
COMMIT
postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
         10 | 22
         11 | 33
(10 rows)

```

```

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

postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
(8 rows)

postgres=# insert into "Person"(cars) values(33);
INSERT 0 1
postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
         11 | 33
(9 rows)

postgres=# commit;
COMMIT
postgres=# select * from "Person";
 person_id | cars
-----+-----
          2 | 10
          3 | 15
          4 | 120
          5 | 20
          6 | 55
          7 | 77
          1 | 1
          8 | 99
         10 | 22
         11 | 33
(10 rows)

```

Ketika kedua terminal menambahkan data pada tabel dalam waktu bersamaan. Data dapat ditambahkan kedalam tabel yang menandakan bahwa derajat isolasi ini memperbolehkan *serialization anomaly*.

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

c. Simulasi *Read Committed*

```
postgres=# begin;
BEGIN
postgres=# show transaction isolation level;
transaction_isolation
-----
read committed
(1 row)

postgres=# select * from "Person";
 person_id | cars
-----+-----
          1 |    5
          2 |   10
          3 |   15
          4 |  120
          5 |   20
          6 |   55
(6 rows)

postgres=# insert into "Person"(cars) VALUES(77);
INSERT 0 1
postgres=# select * from "Person";
 person_id | cars
-----+-----
          1 |    5
          2 |   10
          3 |   15
          4 |  120
          5 |   20
          6 |   55
          7 |   77
(7 rows)

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

```
postgres=# show transaction isolation level;
transaction_isolation
-----
read committed
(1 row)

postgres=# select * from "Person";
 person_id | cars
-----+-----
          1 |    5
          2 |   10
          3 |   15
          4 |  120
          5 |   20
          6 |   55
(6 rows)

postgres=# select * from "Person";
 person_id | cars
-----+-----
          1 |    5
          2 |   10
          3 |   15
          4 |  120
          5 |   20
          6 |   55
          7 |   77
(7 rows)

postgres=#
```

Pada derajat isolasi *read committed*, data yang dimasukkan di terminal kiri tidak terlihat pada terminal kanan. Hal ini karena transaksi yang dilakukan pada terminal kiri belum di *commit*. Setelah

terminal kiri melakukan *commit* maka hasil yang dihasilkan terminal kiri akan sama dengan hasil yang dihasilkan pada terminal kanan.

Dari simulasi ini, dapat disimpulkan bahwa level isolasi transaksi *read committed* tidak memperbolehkan *dirty read*, namun memperbolehkan *non-repeatable read*, *phantom read*, dan *serialization anomaly*.

2. Implementasi Concurrency Control Protocol

Kode implementasi Concurrency Control Protocol dapat diakses pada repository <https://github.com/yuujin-Q/concurrency-control>.

a. Two-Phase Locking (2PL)

Mekanisme two-phase locking yang diimplementasikan memanfaatkan kelas LockManager:

LockManager (lock_manager.cc)
<p>LockManager merupakan kelas yang mengatur <i>key</i> (item pada database) dengan transaksi (Txn) yang sedang memegang kunci <i>key</i> tersebut. Kelas ini mengimplementasikan beberapa fungsi:</p> <ol style="list-style-type: none">1. WriteLock Fungsi ini digunakan untuk memberikan <i>EXCLUSIVE lock</i> sebuah <i>key</i> kepada sebuah transaksi. Return dari fungsi ini adalah apakah locking berhasil atau tidak.<ul style="list-style-type: none">- Jika tidak ada yang sedang memegang <i>lock</i> dari <i>key</i> tersebut, maka <i>lock</i> akan langsung di-grant dan <i>current transaction</i> disimpan pada lock table.- Jika yang sedang memegang <i>lock</i> dari <i>key</i> tersebut adalah transaksi yang sedang meminta <i>lock</i>, kita tahu bahwa <i>lock</i> yang sedang dipegang adalah <i>SHARED lock</i> sehingga kita meng-upgrade dengan cara me-release <i>SHARED lock</i> dan memberikan <i>EXCLUSIVE lock</i> kepada transaksi tersebut.- Selain kasus di atas, maka <i>lock</i> tidak diberikan2. ReadLock Fungsi ini digunakan untuk memberikan <i>SHARED lock</i> sebuah <i>key</i> kepada sebuah transaksi. Return dari fungsi ini adalah apakah locking berhasil atau tidak.<ul style="list-style-type: none">- Jika tidak ada yang sedang memegang <i>lock</i> dari <i>key</i> tersebut, maka <i>lock</i> akan langsung di-grant dan <i>current transaction</i> disimpan pada lock table.- Jika yang sedang <i>lock</i> yang sedang dipegang oleh transaksi lain adalah <i>SHARED lock</i>, maka kita bisa memberikan <i>lock</i> kepada transaksi saat yang meminta.- Selain kasus di atas, maka <i>lock</i> tidak diberikan3. Release Fungsi ini digunakan untuk melepaskan <i>lock</i> terhadap sebuah <i>key</i> yang sedang dipegang oleh transaksi tertentu.4. Status Fungsi ini digunakan untuk mengecek <i>owners</i> dari sebuah <i>lock</i> dan status dari <i>lock</i> tersebut (<i>EXCLUSIVE</i> atau <i>SHARED</i>)
<pre>bool LockManagerA::WriteLock(Txn *txn, const Key &key) { // look up if the key is being locked auto it = lock_table_.find(key); auto waiting_tx_it = txn_waits_.find(txn); if (it == lock_table_.end() it->second->empty()) // not found</pre>

```

{
    // lock the key
    LockRequest new_lock_request = LockRequest(EXCLUSIVE, txn);
    deque<LockRequest> *lock_requests = new deque<LockRequest>;
    lock_requests->push_back(new_lock_request);
    lock_table_.insert(make_pair(key, lock_requests));
    if (waiting_tx_it != txn_waits_.end()) // delete from waiting tx
    {
        txn_waits_.erase(waiting_tx_it);
    }
    return true;
}

vector<Txn *> owners = vector<Txn *>{};
LockMode mode = this->Status(key, &owners);
if (mode == SHARED && owners.size() == 1 && owners[0] == txn)
{
    this->Release(txn, key);
    deque<LockRequest> *lock_requests = it->second;
    lock_requests->push_back(LockRequest(EXCLUSIVE, txn)); // add to
lock requests in the key
    if (waiting_tx_it != txn_waits_.end()) // delete
from waiting tx
    {
        txn_waits_.erase(waiting_tx_it);
    }
    return true;
}
if (owners.size() > 0)
{
    if (owners[0] == txn && mode == EXCLUSIVE)
    {
        return true;
    }
}
return false;
}

bool LockManagerA::ReadLock(Txn *txn, const Key &key)
{
    // look up if the key is being locked

```

```

auto it = lock_table_.find(key);
auto waiting_tx_it = txn_waits_.find(txn);
if (it == lock_table_.end()) // not found
{
    // lock the key
    LockRequest new_lock_request = LockRequest(SHARED, txn);
    deque<LockRequest> *lock_requests = new deque<LockRequest>;
    lock_requests->push_front(new_lock_request);
    lock_table_.insert(make_pair(key, lock_requests));
    if (waiting_tx_it != txn_waits_.end()) // delete from waiting tx
    {

        txn_waits_.erase(waiting_tx_it);
    }
    return true;
}
if (this->Status(key, new vector<Txn *>) != EXCLUSIVE ||
it->second->empty())
{
    LockRequest new_lock_request = LockRequest(SHARED, txn);

    it->second->push_front(new_lock_request);
    if (waiting_tx_it != txn_waits_.end()) // delete from waiting tx
    {
        txn_waits_.erase(waiting_tx_it);
    }
    return true;
}
return false;
}

void LockManagerA::Release(Txn *txn, const Key &key)
{
    auto it = lock_table_.find(key);
    if (it != lock_table_.end())
    {
        deque<LockRequest> *lock_requests = it->second;
        for (auto req_it = lock_requests->begin(); req_it !=
lock_requests->end(); )
        {

```

```

        if (req_it->txn_ == txn)
        {

            if (req_it != lock_requests->begin())
            {
                auto waiting_tx_it = txn_waits_.find(txn);
                if (waiting_tx_it != txn_waits_.end())
                {
                    txn_waits_.erase(waiting_tx_it);
                }
            }
            lock_requests->erase(req_it);
        }
        else
        {
            ++req_it;
        }
    }
}

// Implement this method!
}

LockMode LockManagerA::Status(const Key &key, vector<Txn *> *owners)
{
    owners->clear();
    auto it = lock_table_.find(key);
    if (it != lock_table_.end())
    {
        deque<LockRequest> *lock_requests = it->second;
        if (it->second->empty())
        {
            return UNLOCKED;
        }
        LockMode mode;
        bool foundExclusive = false;
        for (auto lock_it = lock_requests->begin(); lock_it !=
lock_requests->end(); lock_it++)
        {
            if (lock_it->mode_ == EXCLUSIVE)
                // we dont count exclusive locks if it s not the first one in

```

```

the queue
{
    foundExclusive = true;
}
else
{
    mode = lock_it->mode_;
    owners->push_back(lock_it->txn_);
}
}
if (foundExclusive && owners->empty())
{
    owners->push_back(lock_requests->begin()->txn_);
    mode = EXCLUSIVE;
}
return mode;
}
// Implement this method!
return UNLOCKED;
}

```

Implementasi 2PL terdapat pada kelas TxnProcessor pada fungsi RunLockingScheduler

RunLockingScheduler (txn_processor.cc)

Fungsi ini memanfaatkan *threading* yang akan terus berjalan selama masih ada transaksi yang menunggu hasil pada *main process*. Untuk setiap *incoming transaction request*, akan dibuat sebuah *thread* baru untuk menjalankan transaksi tersebut agar dicapai *concurrency*. Pemrosesan sebuah transaksi dilakukan pada fungsi ProcessTxn.

ProcessTxn melakukan eksekusi dalam dua tahap, yaitu memproses readset kemudian writeset. Untuk setiap tahap, proses yang terjadi adalah:

1. Untuk setiap item yang perlu dioperasikan, akan dicek apakah permintaan *lock* berhasil.
2. Jika permintaan *lock* berhasil, maka akan dilanjutkan untuk *item* selanjutnya.
3. Jika permintaan *lock* tidak berhasil, maka *thread* akan meminta *lock* / menunggu sampai *lock* diberikan. Namun, jika pemegang *lock* adalah transaksi yang lebih tua, maka transaksi ini akan di-*rollback* untuk mencegah *deadlock*. (Mekanisme *wait die*)
4. Proses dilanjutkan hingga semua item berhasil memperoleh *lock*

Setelah semua *item* memperoleh *lock*, transaksi akan dieksekusi dan hasilnya akan diberikan kepada *main process*. Kemudian, semua *lock* akan dilepas.

Untuk setiap proses yang perlu melakukan perubahan pada lock table, dilakukan “mutex lock” dikarenakan banyak *thread* yang berjalan secara konkuren. Hal ini dilakukan agar tidak terjadi *race condition* dan semua *thread* membaca data yang benar.

```
void TxnProcessor::RunLockingScheduler()
{
    Txn *txn;
    while (tp_.Active())
    {
        // Take transaction from requests
        if (txn_requests_.Pop(&txn))
        {
            // for the transaction taken, we run it on a new thread
            // so that each transaction runs on their own thread
            tp_.RunTask(new Method<TxnProcessor, void, Txn *>(
                this,
                &TxnProcessor::ProcessTxn,
                txn));
        }
    }
}

void TxnProcessor::ProcessTxn(Txn *txn)
{
    // we process the readset
    for (set<Key>::iterator it = txn->readset_.begin();
        it != txn->readset_.end(); ++it)
    {
        mutex_.Lock();
        bool lockObtained = lm_->ReadLock(txn, *it);
        mutex_.Unlock();
        // we block the transaction while the lock is not obtained
        if (LOGGING)
        {
            printf("[%ld] Acquiring read lock for readset for key: %ld \n", txn->unique_id_, *it);
        }
        while (!lockObtained)
        {
            vector<Txn *> owners;
            mutex_.Lock();
            // we check the current owner of the lock
            lm_->Status(*it, &owners);
            if (owners.size() > 0)
            {
                // if the owner is an "older" transaction / arrives later
                if (owners[0]->unique_id_ < txn->unique_id_)
                {
                    if (LOGGING)
                        printf("[%ld] Rolling back \n", txn->unique_id_);
                    // we rollback this transaction at once
                    this->ReleaseLocks(txn);
                    it = txn->readset_.begin();
                    mutex_.Unlock();
                    break;
                }
            }
        }
    }
}
```



```

    }
}

lockObtained = lm_->ReadLock(txn, *it);
mutex_.Unlock();
};
if (LOGGING)
{
    printf("[%ld] Successfully acquired read lock [%ld]\n", txn->unique_id_, *it);
}
}

// now we process the writesets
for (set<Key>::iterator it = txn->writeset_.begin();
     it != txn->writeset_.end(); ++it)
{
    if (LOGGING)
    {
        printf("[%ld] Acquiring write lock for writeset for key: %ld\n", txn->unique_id_, *it);
    }

    mutex_.Lock();
    bool lockObtained = lm_->WriteLock(txn, *it);
    mutex_.Unlock();
    while (!lockObtained) // we block the process while the lock is not obtained
    {
        mutex_.Lock();
        vector<Txn *> owners;
        lm_->Status(*it, &owners);
        // we check the current owner of the lock
        if (owners.size() > 0)
        {
            if (owners[0]->unique_id_ < txn->unique_id_)
            {
                // rollback if the owner is a younger transaction
                if (LOGGING)
                {
                    printf("[%ld] Rolling back \n", txn->unique_id_);
                }
                this->ReleaseLocks(txn);
                it = txn->writeset_.begin();
                mutex_.Unlock();
                break;
            }
        }

        lockObtained = lm_->WriteLock(txn, *it);
        mutex_.Unlock();
    }

    if (LOGGING)
    {
        printf("[%ld] Successfully acquired write lock [%ld]\n", txn->unique_id_, *it);
    }
}

// at this point we have obtained all the lock for the txn we need so we can execute it
this->ExecuteTxn(txn);
// Commit/abort txn according to program logic's commit/abort decision.

TxnStatus status = txn->Status();
// we commit the transaction

```

```

if (status == COMPLETED_C)
{
    if (LOGGING)
    {
        printf("[!] Changing the status of transaction %ld to COMMITTED\n", txn->unique_id_);
    }
    ApplyWrites(txn);
    txn->status_ = COMMITTED;
}
else if (status == COMPLETED_A)
{
    txn->status_ = ABORTED;
}
else
{
    // Invalid TxnStatus!
    DIE("Completed Txn has invalid TxnStatus: " << status);
}

// Release read locks.
mutex_.Lock();
this->ReleaseLocks(txn);
mutex_.Unlock();

// Return result to client.
txn_results_.Push(txn);
if (LOGGING)
{
    printf("[!] Finished pusing to client\n");
}
}

void TxnProcessor::ReleaseLocks(Txn *txn)
{
    for (set<Key>::iterator it = txn->readset_.begin();
        it != txn->readset_.end(); ++it)
    {
        if (LOGGING)
        {
            printf("[%ld] Releasing lock: %ld \n", txn->unique_id_, *it);
        }
        lm_->Release(txn, *it);
    }

    for (set<Key>::iterator it = txn->writeset_.begin();
        it != txn->writeset_.end(); ++it)
    {
        if (LOGGING)
        {
            printf("[%ld] Releasing lock: %ld \n", txn->unique_id_, *it);
        }
        lm_->Release(txn, *it);
    }
}

```

```

void TxnProcessor::ExecuteTxn(Txn *txn)
{
    // Get the start time
    txn->occ_start_time_ = GetTime();

    // Read everything in from readset.
    for (set<Key>::iterator it = txn->readset_.begin();
         it != txn->readset_.end(); ++it)
    {
        // Save each read result iff record exists in storage.
        Value result;
        if (storage_->Read(*it, &result))
            txn->reads_[*it] = result;
    }

    // Also read everything in from writeset.
    for (set<Key>::iterator it = txn->writeset_.begin();
         it != txn->writeset_.end(); ++it)
    {
        // Save each read result iff record exists in storage.
        Value result;
        if (storage_->Read(*it, &result))
            txn->reads_[*it] = result;
    }

    // Execute txn's program logic.
    txn->Run();

    // Hand the txn back to the RunScheduler thread.
    completed_txns_.Push(txn);
    if (LOGGING)
    {
        printf("[!] Current completed txns count: %d\n", completed_txns_.Size());
    }
}

void TxnProcessor::ApplyWrites(Txn *txn)
{
    // Write buffered writes out to storage.
    for (map<Key, Value>::iterator it = txn->writes_.begin();
         it != txn->writes_.end(); ++it)
    {
        storage_->Write(it->first, it->second, txn->unique_id_);
    }
}

```

Eksperimen:

```

'Low contention' Read only (5 records)
2PL      [1] Acquiring read lock for readset for key: 0
[3] Acquiring read lock for readset for key: 0
[1] Successfully acquired read lock [0]
[2] Acquiring read lock for readset for key: 0
[3] Successfully acquired read lock [0]
[3] Acquiring read lock for readset for key: 1
[1] Acquiring read lock for readset for key: 1
[3] Successfully acquired read lock [1]
[3] Acquiring read lock for readset for key: 2
[3] Successfully acquired read lock [2]
[1] Successfully acquired read lock [1]
[1] Acquiring read lock for readset for key: 2
[1] Successfully acquired read lock [2]
[2] Successfully acquired read lock [0]
[2] Acquiring read lock for readset for key: 1
[!] Current completed txns count: 1
[!] Changing the status of transaction 3 to COMMITED
[3] Releasing lock: 0
[3] Releasing lock: 1
[3] Releasing lock: 2
[!] Finished pusing to client
[!] Current completed txns count: 2
[!] Changing the status of transaction 1 to COMMITED
[1] Releasing lock: 0
[1] Releasing lock: 1
[1] Releasing lock: 2
[2] Successfully acquired read lock [1]
[2] Acquiring read lock for readset for key: 2
[2] Successfully acquired read lock [2]
[!] Finished pusing to client
[!] Current completed txns count: 3
[!] Changing the status of transaction 2 to COMMITED
[2] Releasing lock: 0
[2] Releasing lock: 1
[2] Releasing lock: 2
[!] Finished pusing to client

```

Dari hasil logging di atas, dapat dilihat bahwa terdapat 3 transaksi (T1, T2, T3) yang semuanya berusaha untuk membaca 3 key, Karena semuanya hanya melakukan pembacaan, tidak ada kasus dimana terjadi *rollback* atau gagal mendapatkan *key*. Setelah semua *key* diperoleh, hasilnya di-*push* ke *client*. Dapat dilihat juga urutan yang acak karena ketiganya berjalan secara *concurrent*.

```

1834.04 [2] Acquiring write lock for writeset for key: 2
[3] Acquiring read lock for readset for key: 0
[3] Successfully acquired read lock [0]
[3] Acquiring read lock for readset for key: 2
[3] Rolling back
[3] Releasing lock: 0
[3] Releasing lock: 2
[3] Successfully acquired read lock [0]
[3] Acquiring read lock for readset for key: 2
[3] Rolling back
[3] Releasing lock: 0
[3] Releasing lock: 2
[3] Successfully acquired read lock [0]
[3] Acquiring read lock for readset for key: 2
[1] Acquiring read lock for readset for key: 1
[1] Successfully acquired read lock [1]
[2] Successfully acquired write lock [2]
[!] Current completed txns count: 1
[!] Changing the status of transaction 2 to COMMITED
[3] Rolling back
[3] Releasing lock: 0
[3] Releasing lock: 2
[3] Successfully acquired read lock [0]
[3] Acquiring read lock for readset for key: 2
[3] Rolling back
[3] Releasing lock: 0
[3] Releasing lock: 2
[3] Successfully acquired read lock [0]
[3] Acquiring read lock for readset for key: 2
[3] Rolling back
[3] Releasing lock: 0
[3] Releasing lock: 2
[3] Successfully acquired read lock [0]
[1] Acquiring read lock for readset for key: 2
[2] Releasing lock: 2
[!] Finished pusing to client
[3] Acquiring read lock for readset for key: 2
[3] Successfully acquired read lock [2]
[1] Successfully acquired read lock [2]
[!] Current completed txns count: 2
[!] Changing the status of transaction 3 to COMMITED
[3] Releasing lock: 0
[3] Releasing lock: 2
[!] Current completed txns count: 3
[!] Changing the status of transaction 1 to COMMITED
[1] Releasing lock: 1
[!] Finished pusing to client
[1] Releasing lock: 2
[!] Finished pusing to client
[3] Acquiring read lock for readset for key: 1
[3] Successfully acquired read lock [1]
[3] Acquiring read lock for readset for key: 2
[3] Successfully acquired read lock [2]

```

Dapat dilihat di atas terdapat 3 transaksi (1,2,3) yang masing-masing berusaha untuk membaca dan menulis 3 *item* (0,1,2). Terdapat *rollback* pada transaksi 3, hal ini dikarenakan transaksi 3

meminta *lock* pada *item 2* yang telah dipegang dengan *EXCLUSIVE LOCK* oleh transaksi 2 (transaksi yang lebih tua). Setelah transaksi 2 melepas *lock* pada *item 2*, barulah transaksi 3 dapat dilanjutkan. Terlihat juga bahwa transaksi 1 menunggu setelah meminta *SHARED LOCK* untuk membaca *item 2*.

Perbandingan dengan Serial

```
== bin/txn/txn_processor_test ==
```

	0.1ms	Average Transaction	Duration
		1ms	10ms
'Low contention' Read only (5 records)			
Serial	9214.84	990.282	99.8398
2PL	23060.7	3283.55	513.719
'Low contention' Read only (30 records)			
Serial	7182.3	960.366	99.5252
2PL	6209.16	3164.27	418.011
'High contention' Read only (5 records)			
Serial	9368.17	985.206	99.7512
2PL	20222.6	3289.58	473.052
'High contention' Read only (30 records)			
Serial	7209.44	922.867	99.5434
2PL	8783.24	3282.52	346.812
Low contention read-write (5 records)			
Serial	8696.4	985.391	99.7864
2PL	14499.4	3640.75	449.937
Low contention read-write (10 records)			
Serial	7837.07	964.749	99.6059
2PL	14457.2	3239.87	436.2
High contention read-write (5 records)			
Serial	9014.29	987.352	99.8314
2PL	18532.9	3207.5	446.154
High contention read-write (10 records)			
Serial	8154.94	978.344	99.7296
2PL	13633.5	3774.22	386.995
High contention mixed read only/read-write			
Serial	8907.15	1169.04	133.431
2PL	4828.33	2005.32	301.957

Dapat dilihat pada gambar di atas, dilakukan berbagai macam eksperimen dengan jumlah data yang berbeda-beda pada database. High contention berarti terdapat variasi data pada database sedikit sehingga memungkinkan kemungkinan lebih dari 1 transaksi berusaha mengakses data yang sama.

Dari hasil eksperimen, secara umum *two phase locking* menunjukkan hasil yang lebih baik

dengan rata-rata *throughput* yang lebih besar dibandingkan *serial*. Artinya, 2PL dapat memproses lebih banyak transaksi pada satu satuan waktu dibanding *serial*. Hal ini sesuai ekspektasi karena setiap transaksi diproses pada *thread* nya sendiri. Namun, dapat dilihat terdapat kasus dimana Serial memiliki throughput 8097, sedangkan 2PL memiliki throughput sebesar 4828, hal ini dikarenakan terdapat mekanisme *wait die* yang melakukan *rollback* pada 2PL sehingga terdapat banyak *rollback* yang mungkin saja tidak diperlukan.

b. Optimistic Concurrency Control (OCC)

Berikut adalah implementasi *method* yang digunakan untuk menjalankan Optimistic Concurrency Control (Serial Validation).

```
void TxnProcessor::RunOCCScheduler() {
    // Serial OCC/Validation-Based Protocol
    Txn *txn;

    // check for active transaction requests in pool
    while (tp_.Active()) {
        // get next new transaction request
        if (txn_requests_.Pop(&txn)) {
            // transaction is pending, pass to exec thread
            txn->occ_start_time_ = GetTime();

            tp_.RunTask(new Method<TxnProcessor, void, Txn *>(this, &TxnProcessor::ExecuteTxn,
            txn));
        }

        // check completed transactions (not committed/aborted)
        while (completed_txns_.Pop(&txn)) {
            bool validationFailed = false;

            // validation phase, check for transaction validity
            // check timestamp for each record whose key appears in the txn's read and write sets

            // check readset
            for (auto itr = txn->readset_.begin(); itr != txn->readset_.end(); itr++) {
                double recordUpdateTime = storage_->Timestamp(*itr);

                // check if the record was last updated AFTER this transaction's start time
                // valid condition: data_modify_time < txn_start_time
                if (recordUpdateTime > txn->occ_start_time_) {
                    // failed validation
                    validationFailed = true;
                    break;
                }
            }

            // check writeset
            for (auto itr = txn->writeset_.begin(); itr != txn->writeset_.end(); itr++) {
```

```

double recordUpdateTime = storage_->Timestamp(*itr);

// check if the record was last updated AFTER this transaction's start time
// valid condition: data_modify_time < txn_start_time
if (recordUpdateTime > txn->occ_start_time_) {
    // failed validation
    validationFailed = true;
    break;
}

// DECISION: abort/commit
if (validationFailed) {
    // ABORT transaction, RESTART transaction
    // cleanup txn
    txn->reads_.clear();
    txn->writes_.clear();
    txn->status_ = INCOMPLETE;

    // restart txn
    mutex_.Lock();
    txn->unique_id_ = next_unique_id_;
    next_unique_id_++;
    txn_requests_.Push(txn);
    mutex_.Unlock();
} else {
    // COMMIT
    // write to storage
    txn->status_ = COMPLETED_C;
    ApplyWrites(txn);

    // set as committed, push to result
    txn->status_ = COMMITTED;
    txn_results_.Push(txn);
}
}
}
}

```

Pada kode yang diimplementasikan, implementasi menjalankan secara lokal (tidak langsung mengubah penyimpanan) setiap permintaan transaksi tanpa melakukan *locking*. Selanjutnya, untuk setiap transaksi yang memiliki status *completed* (transaksi selesai dan belum dilakukan *commit* atau *abort*), sebuah transaksi divalidasi dengan waktu mulai transaksi dan dengan baris data yang diakses (*read*) atau dimodifikasi (*write*) oleh transaksi tersebut.

Proses validasi memastikan bahwa waktu mulai transaksi lebih baru daripada waktu terakhir setiap baris data yang akan diakses atau akan dimodifikasi oleh transaksi tersebut. Dengan kata

lain, suatu transaksi yang valid adalah transaksi yang hanya mengoperasikan data yang diakses atau diubah sebelum transaksi dimulai.

Implementasi OCC ini diuji berdasarkan *throughput* pada beberapa kondisi *data contention*. Berikut adalah hasil pengujian yang membandingkan performa protokol serial dan protokol OCC.

```
== bin/txn/txn_processor_test ==
```

	Average Transaction Duration		
	0.1ms	1ms	10ms
'Low contention' Read only (5 records)			
Serial	4322.3	967.415	99.3464
OCC	7606.34	1152.55	181.402
'Low contention' Read only (30 records)			
Serial	4259.17	584.2	97.9827
OCC	6909.24	1441.31	196.237
'High contention' Read only (5 records)			
Serial	7286.16	650.696	129.768
OCC	7970.53	1318.89	208.751
'High contention' Read only (30 records)			
Serial	5285.43	571.988	93.608
OCC	7674.48	1401.35	179.071
Low contention read-write (5 records)			
Serial	9415.3	1158.54	99.0631
OCC	10857.4	1830.46	170.134
Low contention read-write (10 records)			
Serial	5708.08	606.979	120.905
OCC	9887.68	1819.13	146.041
High contention read-write (5 records)			
Serial	7466.21	960.814	99.2719
OCC	3748.53	1303.08	174.413
High contention read-write (10 records)			
Serial	6304.66	946.195	98.7789
OCC	4710.29	1263.71	102.888
High contention mixed read only/read-write			
Serial	6763.48	1189.88	324.644
OCC	7936.5	1687.36	164.226

Contention menunjukkan seberapa sering *thread*, atau dalam kasus ini transaksi, yang memperebutkan akses terhadap baris data yang sama. Berdasarkan pengujian di atas, diperoleh bahwa secara umum, protokol OCC menghasilkan *throughput* yang lebih besar dibandingkan protokol serial. Dalam kasus *read only*, protokol OCC memiliki *throughput* lebih unggul karena adanya paralelisme dan tidak adanya konflik ketika tahap validasi (dikarenakan bersifat *read only*). Di sisi lain, pada kasus *read-write*, protokol OCC memiliki performa yang buruk

jika *contention* bersifat tinggi. Hal ini dikarenakan frekuensi kegagalan validasi yang berbanding lurus dengan frekuensi *contention*, sehingga memperlambat eksekusi *schedule* (pada hasil analisis terlihat pada pengujian *high contention read-write* untuk 5 dan 10 *records* dengan *delay* 0.1ms).

c. Multiversion Timestamp Ordering Concurrency Control (MVCC)

Multiversion Timestamp Ordering Concurrency Control adalah protokol *concurrency control* yang melakukan validasi transaksi berdasarkan waktu penulisan dan pembacaan pada sebuah baris data, umur transaksi, dan berdasarkan versi data yang disimpan. Mekanisme MVCC yang diimplementasikan adalah sebagai berikut.

1. Pada operasi pembacaan dan penulisan, MVCC menggunakan *lock* terhadap baris data yang dioperasikan.
2. Pada operasi pembacaan sebuah baris data, protokol MVCC akan mencari *version* dari baris data yang memiliki umur yang kurang dari atau sama dengan umur transaksi. Setelah membaca baris data, apabila transaksi memiliki waktu pembacaan yang lebih besar daripada yang tersimpan pada *version*, waktu tersebut digantikan dengan umur transaksi pembaca.
3. Pada operasi penulisan sebuah baris data, protokol MVCC terlebih dahulu melakukan validasi. Pada protokol MVCC, operasi penulisan yang sah adalah penulisan terhadap sebuah baris data yang memiliki sebuah *version* dengan umur kurang dari atau sama dengan umur transaksi penulis (atau jika baris data tidak memiliki *version*).
4. Jika operasi penulisan adalah valid, protokol MVCC akan membuat *version* baru dari baris data yang mengalami modifikasi, dengan umur (waktu penulisan) ditetapkan sebagai umur transaksi penulis.
5. Pada awal menjalankan basis data, setiap baris data diinisialisasi dengan waktu penulisan 0, dan waktu pembacaan terakhir 0.

Berikut adalah kode implementasi protokol MVCC. Kode implementasi operasi pembacaan, penulisan, dan validasi terdapat pada *method Read, Write, dan CheckWrite* (mvcc_storage.cc). Sementara itu, terdapat *method MVCCExecuteTxn* dan *RunMVCCScheduler* yang digunakan untuk melakukan *scheduling* dan manajemen *thread* untuk simulasi protokol (txn_processor.cc).

mvcc_storage.cc

```

// MVCC Read
bool MVCCStorage::Read(Key key, Value* result, int txn_unique_id) {
    // get version list for key
    auto key_versions = mvcc_data_.find(key);
    if (key_versions == mvcc_data_.end()) {
        return false;
    }

    auto version_list = key_versions->second;
    if (version_list->empty()) {
        return false;
    }

    // Iterate the version_lists and return the version whose write timestamp
    // (version_id) is the largest write timestamp less than or equal to txn_unique_id.
    auto latest_version = version_list->begin();
    for (auto itr = version_list->begin(); itr != version_list->end(); itr++) {
        if ((*itr)->version_id_ > (*latest_version)->version_id_ && (*itr)->version_id_
        <= txn_unique_id) {
            // get latest version with timestamp less or equal to txn_unique_id
            latest_version = itr;
        }
    }

    // get value and update read timestamp
    *result = (*latest_version)->value_;
    if ((*latest_version)->max_read_id_ < txn_unique_id) {
        (*latest_version)->max_read_id_ = txn_unique_id;
    }
    return true;
}

// Check whether apply or abort the write
bool MVCCStorage::CheckWrite(Key key, int txn_unique_id) {
    auto key_versions = mvcc_data_.find(key);

    // no key exist in version database, return true
    if (key_versions == mvcc_data_.end()) {
        return true;
    }

    auto version_list = key_versions->second;

    // Iterate the version_lists
    // get latest version where write timestamp less than or equal to txn_unique_id.
    auto valid_version = version_list->begin();
    for (auto itr = version_list->begin(); itr != version_list->end(); itr++) {

```

```

        if ((*itr)->version_id_ > (*valid_version)->version_id_ && (*itr)->version_id_ <=
txn_unique_id) {
            // get latest version with timestamp less or equal to txn_unique_id
            valid_version = itr;
        }
    }

    // no valid version exists
    if ((*valid_version)->version_id_ > txn_unique_id) {
        return false;
    } else {
        return true;
    }
}

// MVCC Write, call this method only if CheckWrite return true.
void MVCCStorage::Write(Key key, Value value, int txn_unique_id) {
    // create version, assume malloc succeeds
    Version* to_write = new Version;
    to_write->value_ = value;
    to_write->version_id_ = txn_unique_id;
    to_write->max_read_id_ = txn_unique_id;

    // get key versions deque
    auto key_versions = mvcc_data_.find(key);

    if (key_versions == mvcc_data_.end()) {
        // no versions exists for key, insert first version
        deque<Version*>* versions = new deque<Version*>();
        versions->push_front(to_write);
        mvcc_data_[key] = versions;
        return;
    }

    // ELSE, versions exists for key
    auto version_list = key_versions->second;
    if (version_list->empty()) {
        // redundant check, check if version_list is empty
        version_list = new deque<Version*>();
        version_list->push_front(to_write);
        return;
    }

    // // Iterate the version_lists
    // // get latest version where write timestamp less than or equal to txn_unique_id.
    auto latest_valid_version = version_list->begin();
    for (auto itr = version_list->begin(); itr != version_list->end(); itr++) {

```

```

        if ((*itr)->version_id_ > (*latest_valid_version)->version_id_ &&
(*itr)->version_id_ <= txn_unique_id) {
            // get latest version with timestamp less or equal to txn_unique_id
            latest_valid_version = itr;
        }
    }

    if ((*latest_valid_version)->version_id_ == txn_unique_id) {
        // if same timestamp, update value
        (*latest_valid_version)->value_ = value;
        delete to_write;
    } else {
        // create new version
        version_list->push_front(to_write);
    }
}
}

```

txn_processor.cc

```

void TxnProcessor::MVCCExecuteTxn(Txn* txn) {
    // Read all necessary data for this transaction from storage
    // (Note that unlike the version of MVCC from class, you should lock the key
before each read)
    // read for readset
    for (auto read_key : txn->readset_) {
        Value result;
        storage_->Lock(read_key);
        if (storage_->Read(read_key, &result)) {
            txn->reads_[read_key] = result;
        }
        storage_->Unlock(read_key);
    }

    // read for writeset
    for (auto write_key : txn->writeset_) {
        Value result;
        storage_->Lock(write_key);
        if (storage_->Read(write_key, &result)) {
            txn->reads_[write_key] = result;
        }
        storage_->Unlock(write_key);
    }

    // Execute the transaction logic (i.e. call Run() on the transaction)
    txn->Run();
}

```

```

// Acquire all locks for keys in the write_set_
// Call MVCCStorage::CheckWrite method to check all keys in the write_set_
bool passed = true;
for (auto write_key : txn->writeset_) {
    storage_>Lock(write_key);
    if (!storage_>CheckWrite(write_key, txn->unique_id_)) {
        passed = false;
        break;
    }
}

// check if writes valid
if (passed) {
    // passed, Apply the writes
    txn->status_ = COMPLETED_C;
    ApplyWrites(txn);
    txn->status_ = COMMITTED;
    txn_results_.Push(txn);
} else {
    // cleanup txn
    txn->reads_.clear();
    txn->writes_.clear();
    txn->status_ = INCOMPLETE;

    // completely restart the transaction
    mutex_.Lock();
    txn->unique_id_ = next_unique_id_;
    next_unique_id_++;
    txn_requests_.Push(txn);
    mutex_.Unlock();
}

// Release all locks for keys in the write_set_
for (auto keys : txn->writeset_) {
    storage_>Unlock(keys);
}
}

void TxnProcessor::RunMVCCScheduler() {
    // MVCC
    Txn *txn;

    // check for active transaction requests in pool
    // Pop a txn from txn_requests_, and pass it to a thread to execute.
    while (tp_.Active()) {
        // get next new transaction request
        if (txn_requests_.Pop(&txn)) {
            // transaction is pending, pass to exec thread

```

```

        tp_.RunTask(new Method<TxnProcessor, void, Txn *>(this,
&TxnProcessor::MVCCExecuteTxn, txn));
    }
}
}

```

Kode implementasi di atas diuji dan dibandingkan terhadap *throughput* protokol serial. Berikut adalah hasil pengujian yang dilakukan.

	Average Transaction Duration		
	0.1ms	1ms	10ms
'Low contention' Read only (5 records)			
Serial	3877.47	955.313	99.3028
MVCC	7473.51	1582.52	218.234
'Low contention' Read only (30 records)			
Serial	4764.48	902.803	98.8036
MVCC	5811	1314.82	123.893
'High contention' Read only (5 records)			
Serial	7425.35	968.682	99.5818
MVCC	9468.44	919.447	207.521
'High contention' Read only (30 records)			
Serial	5255.13	546.623	97.9762
MVCC	3140.31	810.135	162.472
Low contention read-write (5 records)			
Serial	5687.34	960.259	98.9766
MVCC	2491.56	958.376	168.974
Low contention read-write (10 records)			
Serial	6338.95	931.846	97.691
MVCC	8103.74	865.92	140.667
High contention read-write (5 records)			
Serial	7715.79	603.975	99.3654
MVCC	9622.68	1100.99	150.099
High contention read-write (10 records)			
Serial	6100.86	930.65	98.9922
MVCC	4715.05	1309.9	163.811
High contention mixed read only/read-write			
Serial	1998.11	1130.09	Segmentation fault
make: *** [txn/Makefile.inc:17: test-txn] Error 139			

Berdasarkan hasil pengujian, perbandingan antara *throughput* protokol serial dan protokol MVCC tidak menunjukkan tren yang bermakna. Akan tetapi, jika hasil pengujian diteliti lebih lanjut, pengujian menunjukkan bahwa performa protokol MVCC menurun seiring pertambahan jumlah *record* (lihat perbandingan antara pengujian 5 *record* dan 30 *record*). Pada pengujian, jumlah *record* yang diproses mempengaruhi jumlah terjadinya konflik validasi *version*.

Hal menarik dari pengujian di atas juga adalah *error* yang ditampilkan pada pengujian terakhir

(mixed high contention). Hal ini menunjukkan bahwa kode implementasi belum sempurna.

3. Eksplorasi Recovery

a. *Write-Ahead Log*

Write-Ahead Log (WAL) adalah metode *recovery* yang digunakan pada PostgreSQL untuk menjamin integritas data. Penerapan WAL mengimplikasikan bahwa setiap perubahan terhadap data ditulis hanya jika perubahan tersebut sudah di-log. Aturan dari *Write-Ahead-Log* adalah sebagai berikut.

- 1) Transaksi T_i memasuki *state* commit setelah *log record* $\langle T_i \text{ commit} \rangle$ ditulis ke *stable storage*.
- 2) Sebelum *log record* $\langle T_i \text{ commit} \rangle$ bisa dituliskan ke *stable storage*, semua *log record* yang terkait dengan transaksi T_i harus sudah ditulis ke *stable storage*.
- 3) Sebelum sebuah blok data di *main memory* bisa dituliskan ke basis data (di non-volatile storage), semua *log record* yang berhubungan dengan data di blok tersebut harus sudah ditulis di *stable storage*.

Penggunaan WAL mengakibatkan banyak operasi penulisan ke disk menjadi berkurang. Hal ini disebabkan hal yang perlu ditulis ke disk hanyalah file WAL, bukan setiap data yang diubah oleh suatu transaksi. Selain itu, file WAL ditulis secara sekuensial. Akibatnya, *cost* untuk melakukan sinkronisasi WAL lebih rendah daripada melakukan sinkronisasi data secara langsung.

b. *Continuous Archiving*

Continuous Archiving adalah metode *recovery* pada PostgreSQL yang mengombinasikan *backup* pada level file-system dengan backup dari file WAL. Secara singkat, *recovery* menggunakan metode ini terdiri dari proses melakukan *restore* backup yang ada pada file-system kemudian melakukan *replay* dari file WAL yang telah di-*backup* untuk mengubah *state* dari sistem menjadi *current state*. Pendekatan *recovery* dengan metode ini memberikan beberapa keuntungan sebagai berikut.

- 1) Tidak memerlukan *file system backup* secara konsisten dan sempurna karena setiap inkonsistensi akan dikoreksi oleh *log replay*.
- 2) *Continuous backup* dapat dilakukan dengan meng-*archive* file WAL secara terus-menerus atau kontinu sehingga tidak perlu dilakukan *backup* menyeluruh berkali-kali.

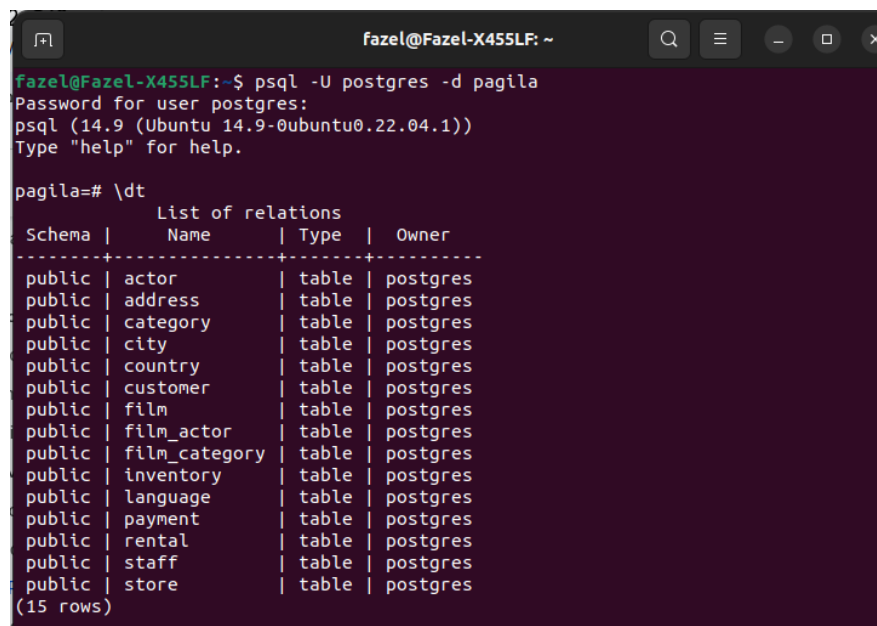
- 3) Dapat menghentikan *replay* dari entri WAL pada titik manapun dan menghasilkan *snapshot* dari basis data pada saat itu secara konsisten sehingga mendukung *point-in-time recovery*.
- 4) Dapat menyediakan *warm standby system* dengan cara menyimpan deretan file WAL secara terus-menerus ke mesin lain yang sudah memiliki file *base backup* yang sama sehingga dapat membuat salinan yang hampir mendekati *state* terkini dari basis data.

c. *Point-in-Time Recovery*

Point-in-Time Recovery (PITR) adalah metode recovery pada PostgreSQL yang memungkinkan untuk mengembalikan atau me-*restore* basis data ke suatu titik atau *state* pada waktu tertentu. Metode ini digunakan atau dijalankan bersamaan dengan *continuous archiving* yang memastikan bahwa tersedia deretan file WAL yang merekam setiap perubahan basis data. Untuk melakukan *point-in-time recovery*, diperlukan untuk menspesifikasikan *recovery target* yang berisi *date/time*, nama dari sebuah *restore point*, atau transaction ID. *Recovery target* ini digunakan sebagai dasar pengembalian atau *recovery* basis data. Artinya, basis data akan dikembalikan ke *state* dari *recovery target* tersebut.

d. Simulasi Kegagalan pada PostgreSQL

Daftar tabel pada basis data saat kondisi awal adalah sebagai berikut.



```
fazel@Fazel-X455LF: ~
fazel@Fazel-X455LF:~$ psql -U postgres -d pagila
Password for user postgres:
psql (14.9 (Ubuntu 14.9-0ubuntu0.22.04.1))
Type "help" for help.

pagila=# \dt
          List of relations
Schema | Name          | Type  | Owner
-----+-----+-----+-----
public | actor         | table | postgres
public | address       | table | postgres
public | category      | table | postgres
public | city          | table | postgres
public | country       | table | postgres
public | customer      | table | postgres
public | film          | table | postgres
public | film_actor    | table | postgres
public | film_category | table | postgres
public | inventory     | table | postgres
public | language      | table | postgres
public | payment       | table | postgres
public | rental        | table | postgres
public | staff         | table | postgres
public | store         | table | postgres
(15 rows)
```

Gambar 3.1 Daftar Tabel pada Database

Kemudian, dilakukan *physical backup* terhadap klaster basis data dengan cara meng-*copy* semua file basis data pada direktori data PostgreSQL.

```
fazel@Fazel-X455LF:~$ sudo -u postgres pg_basebackup -D ~/database_backup
fazel@Fazel-X455LF:~$
```

Gambar 3.2 Backup Database

Setelah itu, seorang user menghapus tabel payment.

```
pagila=# DROP TABLE payment;
DROP TABLE
pagila=#
```

Gambar 3.3 Penghapusan Tabel payment

Tindakan user tersebut dinyatakan sebagai kegagalan pada simulasi ini. Seharusnya tabel tersebut tidak boleh dihapus oleh user karena akan mengancam integritas basis data. Akan tetapi, hal tersebut tidak dapat dicegah berdasarkan *security* yang disediakan oleh basis data pada saat ini. Akibatnya, jumlah tabel pada daftar tabel dari basis data ini berkurang satu seperti diperlihatkan gambar di bawah ini.

```
pagila=# \dt
          List of relations
Schema |      Name      | Type  | Owner
-----+-----+-----+-----
public | actor           | table | postgres
public | address         | table | postgres
public | category        | table | postgres
public | city            | table | postgres
public | country         | table | postgres
public | customer        | table | postgres
public | film            | table | postgres
public | film_actor      | table | postgres
public | film_category   | table | postgres
public | inventory       | table | postgres
public | language        | table | postgres
public | rental          | table | postgres
public | staff           | table | postgres
public | store           | table | postgres
(14 rows)
```

Gambar 3.4 Daftar Tabel setelah Tabel payment Dihapus

Untuk memulihkan basis data ke keadaan semula, *Database Administrator* melanjutkan proses *recovery* dengan me-*restore* file yang sudah di-*backup*.

```
fazel@Fazel-X455LF:~$ sudo service postgresql stop
fazel@Fazel-X455LF:~$ sudo mv /var/lib/postgresql/14/main/pg_wal ~/
fazel@Fazel-X455LF:~$ sudo rm -rf /var/lib/postgresql/14/main
fazel@Fazel-X455LF:~$ sudo mkdir /var/lib/postgresql/14/main
fazel@Fazel-X455LF:~$ sudo cp -a ~/database_backup/. /var/lib/postgresql/14/main
fazel@Fazel-X455LF:~$ sudo chown postgres:postgres /var/lib/postgresql/14/main
fazel@Fazel-X455LF:~$ sudo chmod 700 /var/lib/postgresql/14/main
fazel@Fazel-X455LF:~$ sudo rm -rf /var/lib/postgresql/14/main/pg_wal
fazel@Fazel-X455LF:~$ sudo cp -a ~/pg_wal /var/lib/postgresql/14/main/pg_wal
fazel@Fazel-X455LF:~$ sudo nano /etc/postgresql/14/main/postgresql.conf
fazel@Fazel-X455LF:~$ sudo nano /etc/postgresql/14/main/postgresql.conf
fazel@Fazel-X455LF:~$ sudo nano /etc/postgresql/14/main/postgresql.conf
fazel@Fazel-X455LF:~$ sudo nano /etc/postgresql/14/main/postgresql.conf
fazel@Fazel-X455LF:~$ sudo touch /var/lib/postgresql/14/main/recovery.signal
fazel@Fazel-X455LF:~$ sudo service postgresql start
fazel@Fazel-X455LF:~$
```

Gambar 3.5 Restore Database

Langkah selanjutnya adalah memastikan server basis data berhasil memulihkan file WAL yang sudah di-*archive* dengan melakukan konfigurasi *recovery settings*. Konfigurasi tersebut terdapat pada file `postgresql.conf` di direktori `/etc/postgresql/14/main`. Pada simulasi ini, *recovery target* yang digunakan adalah waktu sehingga basis data akan dipulihkan ke *state* pada waktu tersebut. Untuk itu, diperlukan pemeriksaan waktu terlebih dahulu.

Berdasarkan mekanisme *recovery* pada PostgreSQL, *recovery_target_time* harus merupakan waktu setelah dilakukan *physical backup*. Oleh karena itu, waktu yang dapat menjadi *recovery_target_time* pada simulasi ini adalah waktu yang berada pada rentang sejak *physical backup* berhasil dilakukan sampai dengan sebelum seorang user menghapus tabel `payment`. Pada simulasi ini, dilakukan pemeriksaan waktu pada rentang tersebut menggunakan perintah SQL “`SELECT CURRENT_TIMESTAMP;`” yang mengembalikan waktu terkini dari PostgreSQL. Setelah dieksekusi, didapatkan `CURRENT_TIMESTAMP` bernilai `2023-12-01 17:15:20.840621`. Oleh karena itu, *recovery_target_time* yang dipilih adalah `2023-12-01 17:15:00.000000`. Waktu tersebut termasuk ke dalam rentang waktu yang sudah disyaratkan di awal karena *physical backup* terjadi beberapa menit sebelumnya.

```

fazel@Fazel-X455LF: ~
GNU nano 6.2 /etc/postgresql/14/main/postgresql.conf

restore_command = 'cp ~/database_archive/%f %p'          # command to use to res
                  # placeholders: %p = path of file to restore
                  #                               %f = file name only
                  # e.g. 'cp /mnt/server/archivedir/%f %p'
#archive_cleanup_command = ''                            # command to execute at every restartpoint
#recovery_end_command = ''                               # command to execute at completion of recovery

# - 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
                  # (change requires restart)
recovery_target_time = '2023-12-01 17:15:00.000000'      # the time stamp up to
                  # (change requires restart)
#recovery_target_xid = ''                               # the transaction ID up to which recovery will
                  # (change requires restart)
#recovery_target_lsn = ''                               # the WAL LSN up to which recovery will proceed
                  # (change requires restart)
#recovery_target_inclusive = on                          # Specifies whether to stop:
                  # just after the specified recovery target (on)

^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace  ^U Paste     ^J Justify   ^_ Go To Line

```

Gambar 3.6 Konfigurasi *Recovery Target*

Setelah itu, dilakukan restart terhadap basis data dan berhasil. Keberhasilan ini ditandai dengan isi dari log transaksi yang diperlihatkan oleh gambar berikut.

```

fazel@Fazel-X455LF: ~
2023-12-01 17:35:52.283 WIB [40050] LOG:  starting PostgreSQL 14.9 (Ubuntu 14.9-0ubuntu0.22.04.1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, 64-bit
2023-12-01 17:35:52.283 WIB [40050] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2023-12-01 17:35:52.285 WIB [40050] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-12-01 17:35:52.290 WIB [40051] LOG:  database system was interrupted; last known up at 2023-12-01 17:12:33 WIB
cp: cannot stat '~/database_archive/00000002.history': No such file or directory
2023-12-01 17:35:52.337 WIB [40051] LOG:  starting point-in-time recovery to 2023-12-01 17:15:00+07
cp: cannot stat '~/database_archive/00000001000000000000000004': No such file or directory
2023-12-01 17:35:52.344 WIB [40051] LOG:  redo starts at 0/4000028
2023-12-01 17:35:52.344 WIB [40051] LOG:  consistent recovery state reached at 0/4000100
2023-12-01 17:35:52.347 WIB [40050] LOG:  database system is ready to accept read-only connections
cp: cannot stat '~/database_archive/00000001000000000000000005': No such file or directory
2023-12-01 17:35:52.349 WIB [40051] LOG:  recovery stopping before commit of transaction 885, time 2023-12-01 17:15:30.919733+07
2023-12-01 17:35:52.349 WIB [40051] LOG:  pausing at the end of recovery
2023-12-01 17:35:52.349 WIB [40051] HINT:  Execute pg_wal_replay_resume() to promote.
fazel@Fazel-X455LF:~$

```

Gambar 3.7 Isi *Transaction Log*

Pada log transaksi di atas, tercatat *point-in-time recovery* ke 2023-12-01 17:15:00 dimulai pada jam 17:35:52. Kemudian, basis data melakukan redo pada 0/4000028. Terakhir, basis data berhasil mencapai *consistent recovery state* pada 0/4000100.

```
fazel@Fazel-X455LF:~$ psql -U postgres -d pagila
Password for user postgres:
psql (14.9 (Ubuntu 14.9-0ubuntu0.22.04.1))
Type "help" for help.

pagila=# \dt
          List of relations
Schema |      Name      | Type | Owner
-----+-----+-----+-----
public | actor           | table | postgres
public | address         | table | postgres
public | category        | table | postgres
public | city            | table | postgres
public | country         | table | postgres
public | customer        | table | postgres
public | film            | table | postgres
public | film_actor      | table | postgres
public | film_category   | table | postgres
public | inventory       | table | postgres
public | language        | table | postgres
public | payment         | table | postgres
public | rental          | table | postgres
public | staff           | table | postgres
public | store           | table | postgres
(15 rows)

pagila=#
```

Gambar 3.8 Daftar Tabel setelah dilakukan *Restore*

Pada gambar di atas, terlihat daftar tabel basis data setelah dilakukan recovery. Tabel payment yang pada saat kegagalan terjadi hilang dari basis data, sekarang telah kembali berada pada basis data. Dengan demikian, proses *recovery* berhasil mengembalikan basis data ke saat sebelum kegagalan terjadi.

4. Kesimpulan dan Saran

Tingkatan level isolasi transaksi dalam sistem basis data mempengaruhi sejauh mana transaksi dapat mempengaruhi data yang diakses oleh transaksi lain. Dari yang paling ketat, tingkat *serializable* tidak mengizinkan fenomena seperti *dirty read*, *non-repeatable read*, *phantom read*, atau *serialization anomaly* terjadi, menjalankan transaksi seolah-olah mereka satu-satunya yang ada. Pada tingkat *repeatable read*, meski mencegah *dirty read* dan *non-repeatable read*, *phantom read* masih mungkin terjadi. *Read committed* memperbolehkan *non-repeatable read*, sementara *read uncommitted* memungkinkan terjadinya semua fenomena tersebut. Tingkatan isolasi mempertimbangkan *trade-off* antara konsistensi data dan kinerja sistem, di mana tingkat yang lebih tinggi menawarkan konsistensi yang lebih besar namun dapat mengurangi kinerja, sementara tingkat yang lebih rendah dapat meningkatkan kinerja tetapi meningkatkan risiko inkonsistensi data.

Pada PostgreSQL, derajat isolasi transaksi yang dapat digunakan hanya tiga, yaitu *serializable*, *repeatable read*, dan *read committed*, dengan *default* derajat isolasi transaksi *read committed*. Derajat isolasi transaksi dapat diubah untuk setiap transaksi menggunakan *query* “SET TRANSACTION ISOLATION LEVEL <derajat isolasi>”.

Terdapat tiga algoritma yang diimplementasikan, yaitu algoritma *Two Phase Locking*, *Optimistic Concurrency Control* dan *Multi-version Concurrency Control*. Setiap algoritma tersebut memiliki kelebihan dan kekurangannya masing - masing. Algoritma *Two Phase Lock* mencegah inkonsistensi data dengan menjamin bahwa transaksi akan mengunci sumber daya yang diperlukan sebelum mengaksesnya dan melepaskan kunci setelah selesai. Namun, transaksi dapat mengalami *blocking* atau penundaan karena menunggu sumber daya yang dikunci yang berpotensi memperlambat kinerja secara keseluruhan. Algoritma *Optimistic Concurrency Control* memiliki kelebihan utama yaitu setiap transaksi dapat dieksekusi tanpa harus memperhitungkan setiap operasi yang dijalankan oleh scheduler. Namun, terdapat potensi untuk rollback yang lebih sering dan ini bisa meningkatkan latensi jika terjadi konflik secara teratur. Algoritma *Multi-version Concurrency Control* memiliki keunggulan utama dalam memungkinkan mencapai tingkat konkurensi yang tinggi karena memanfaatkan versi data yang lebih fleksibel. Namun, memerlukan ruang penyimpanan yang luas karena memerlukan pencatatan untuk setiap versi data yang disimpan.

Recovery merupakan proses mengembalikan sistem basis data ke keadaan yang konsisten setelah terjadi kegagalan sistem. Kegagalan bisa disebabkan oleh *crash* sistem akibat kesalahan perangkat keras

atau perangkat lunak, kegagalan media seperti *head crash*, atau kesalahan perangkat lunak dalam aplikasi, seperti kesalahan logis pada program yang mengakses basis data.

Dalam sistem basis data PostgreSQL, ada beberapa metode pemulihan (*recovery*) yang berbeda seperti *Write-Ahead Log*, *Continuous Archiving*, dan *Point-in-Time Recovery*. Setiap metode ini memiliki prosesnya sendiri dan terdapat keterkaitan di antara mereka. Sebagai contoh, *Continuous Archiving* menggunakan *Write-Ahead Log* dalam jalannya prosesnya. Demikian pula, *Point-in-Time Recovery* memanfaatkan *Continuous Archiving* untuk melakukan pemulihan data hingga mencapai titik waktu tertentu.

Sebaiknya untuk kedepannya referensi yang diberikan dapat ditinjau kembali, untuk implementasi menggunakan referensi template yang diberikan diperlukan cukup banyak penyesuaian terutama pada bagian *Two Phase Locking* untuk mekanisme *threading* dan konkurensi. Hal ini dapat dihindari apabila diberi tahu lebih awal atau memberikan template referensi lain yang lebih sesuai dengan implementasi tugas besar.

5. Pembagian Kerja

NIM	Nama	Bagian
13521044	Rachel Gabriela Chen	Implementasi Concurrency Control Protocol (2PL)
13521074	Eugene Yap Jin Quan	Implementasi Concurrency Control Protocol (OCC, MVCC)
13521098	Fazel Ginanda	Eksplorasi Recovery
13521104	Muhammad Zaydan Athallah	Eksplorasi Transaction Isolation

Referensi

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

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

[kunrenyale/CMSC828N_assignment2: CMSC828N_assignment2 \(github.com\)](#)