

IF3140 MANAJEMEN BASIS DATA

MEKANISME CONCURRENCY CONTROL DAN RECOVERY



K03 Kelompok 01

Anggota :

Matthew Mahendra	13521007
Syarifa Dwi Purnamasari	13521018
M Zulfiansyah Bayu Pratama	13521028
M. Malik I. Baharsyah	13521029

Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2023

Daftar Isi

Daftar Isi	1
1. Eksplorasi Transaction Isolation	2
a. Serializable	2
b. Repeatable Read	3
c. Read Committed	9
d. Read Uncommitted	10
2. Implementasi Concurrency Control Protocol	12
a. Two-Phase Locking (2PL)	12
b. Optimistic Concurrency Control (OCC)	20
c. Multiversion Timestamp Ordering Concurrency Control (MVCC)	25
3. Eksplorasi Recovery	31
a. Write-Ahead Log	31
b. Continuous Archiving	32
c. Point-in-Time Recovery	32
d. Simulasi Kegagalan pada PostgreSQL	33
4. Pembagian Kerja	37
Referensi	38

1. Eksplorasi Transaction Isolation

a. Serializable

Serializable Isolation Level adalah level isolasi transaksi yang paling ketat. Pada level ini, eksekusi transaksi serial disimulasikan untuk semua transaksi yang telah dikonfirmasi; seolah-olah transaksi telah dieksekusi satu per satu secara berurutan dan bukan secara bersamaan, sehingga memastikan bahwa transaksi adalah transaksi yang serial. Meskipun mirip dengan *Repeatable Read Level*, aplikasi yang menggunakan level ini harus siap untuk mengulangi transaksi karena kegagalan serialisasi. Pada dasarnya, level isolasi ini bekerja persis seperti *Repeatable Read Level*, tetapi juga memonitor kondisi yang dapat membuat eksekusi set transaksi serializable bersamaan berperilaku tidak konsisten dengan semua eksekusi serial (satu per satu) yang mungkin dari transaksi tersebut. Monitoring ini tidak memperkenalkan pemblokiran lebih dari yang ada dalam tingkat Repeatable Read, tetapi ada beberapa overhead untuk pemantauan, dan deteksi kondisi yang dapat menyebabkan anomali serialisasi akan memicu kegagalan serialisasi.

Simulasi untuk setiap level isolasi pada bagian ini akan menggunakan skema sebagai berikut: tabel1(id, nama, harga) dengan id dan harga adalah integer dan nama adalah varchar. Simulasi untuk level isolasi serializable akan menggunakan tabel sebagai berikut.

```
contoh=# select * from table1;
 id |  nama  | harga
----+-----+-----
  1 | Apel   |  7000
  2 | Apel   |  5000
  3 | Mangga |  4000
  4 | Mangga |  6000
(4 rows)
```

Transaksi 1	Transaksi 2
-------------	-------------

<pre> contoh=# begin isolation level serializable; BEGIN contoh=# select sum(harga) from table1 where nama='Mangga'; sum ----- 10000 (1 row) </pre>	<pre> contoh=# begin isolation level serializable; BEGIN contoh=# select sum(harga) from table1 where nama='Apel'; sum ----- 12000 (1 row) </pre>
<p>Transaksi 1 melakukan read pada kolom “Mangga” dilanjutkan dengan transaksi 2 yang melakukan read pada kolom “Apel”.</p>	

Transaksi 1	Transaksi 2
<pre> contoh=# insert into table1 values (5, 'Apel', 10000); INSERT 0 1 </pre>	<pre> contoh=# insert into table1 values (6, 'Mangga', 12000); INSERT 0 1 </pre>
<p>Transaksi 1 melakukan insert pada kolom “Apel” dilanjutkan dengan transaksi 2 yang melakukan insert juga pada kolom “Mangga”.</p>	

Transaksi 1	Transaksi 2
<pre> contoh=# 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. contoh=# </pre>	<pre> contoh=# commit; COMMIT contoh=# </pre>
<p>Transaksi 2 melakukan commit dan berhasil dilanjutkan transaksi 1 yang mencoba melakukan commit. Ketika transaksi 1 commit, muncul pesan error tersebut karena pada serializable isolation level, transaksi dieksekusi seolah-olah secara serial. Namun, karena eksekusi yang bersamaan, tidak ada urutan eksekusi serial yang konsisten sehingga untuk mengatasi inkonsistensi tersebut, salah satu transaksi harus melakukan rollback.</p>	

b. Repeatable Read

Isolasi *repeatable read* memastikan bahwa data yang dibaca untuk semua klausa adalah data sebelum semua transaksi dimulai. Dengan demikian, setiap transaksi tidak akan membaca data yang sudah diubah oleh transaksi lainnya sebelum transaksi tersebut di-*commit*, meskipun dalam transaksi yang dijalankannya tetap dapat terlihat perubahan yang terjadi. Dengan demikian, transaksi dapat mencegah terjadinya fenomena *dirty read*, *non repeatable read*, dan *phantom read* dengan memastikan bahwa setiap transaksi membaca database yang stable dari awal. Karena adanya isolasi dengan sifat seperti ini, maka transaksi yang melakukan updating harus siap

melakukan pengulangan karena kegagalan serialize, sedangkan untuk transaksi pembacaan tidak akan pernah mengalami serialize conflict.

Sebagai contoh dari isolasi level *repeatable read*, akan diberikan 2 transaksi. Transaksi pertama melakukan *rollback* sehingga efeknya dibatalkan dan transaksi dapat berjalan seperti biasa. Akan tetapi ketika transaksi pertama berhasil melakukan transaksi yang melakukan perubahan, maka transaksi yang melakukan *repeatable read* akan melakukan *rollback* dan memaksa keseluruhan transaksi untuk *rollback*. Akan tetapi pada percobaan setelah *rollback* ini, maka data awal yang dilihat adalah data hasil *commit* dari transaksi yang berhasil dijalankan.

Meskipun demikian, sebuah transaksi pembacaan mungkin perlu melakukan pembaharuan terhadap data yang dibacanya untuk memastikan bahwa sebuah transaksi sudah selesai tetapi tidak membaca detail dari data yang diubah karena membaca data revisi yang lebih awal dari control record. Untuk memastikan aturan bisnis pada level isolasi ini, diperlukan penggunaan lock yang teliti untuk transaksi yang saling konflik.

Simulasi dari tingkat isolasi ini akan menggunakan tiga kasus yaitu: dua transaksi *repeatable read* yang saling mengubah satu data yang sama, dua transaksi *repeatable read* yang membaca data berbeda, dan yang terakhir dua transaksi *repeatable read* yang salah satunya melakukan *rollback*. Untuk simulasi, diberikan tabel sebagai berikut,

```
contoh=# select * from tabel1
contoh-# ;
 id |  nama  | harga
----+-----+-----
  1 | Buku   |    30
  2 | Pensil |    20
  3 | Roti   |    25
(3 rows)
```

Gambar 1.1 Tabel1 untuk Simulasi Repeatable Read

Untuk kasus pertama,

Transaksi 1	Transaksi 2
<pre>contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 1 Buku 30 2 Pensil 20 3 Roti 25 (3 rows) contoh=# update tabel1 set harga = harga + 1 where id = 1; UPDATE 1</pre>	<pre>contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 1 Buku 30 2 Pensil 20 3 Roti 25 (3 rows) contoh=# update tabel1 set harga = harga + 1 where id = 1; </pre>
<p>Transaksi 1 menjalankan update tabel1 untuk harga += 1 pada id = 1. Transaksi berhasil dijalankan. Pada saat transaksi 1 selesai menjalankan update, transaksi 2 yang akan menjalankan update yang sama harus menunggu commit dari transaksi 1.</p>	

Transaksi 1	Transaksi 2
<pre>contoh=# commit; COMMIT</pre>	<pre>contoh=# update tabel1 set harga = harga + 1 where id = 1; ERROR: could not serialize access due to concurrent update</pre>
<p>Setelah transaksi 1 melakukan commit, transaksi 2 tidak dapat menjalankan transaksinya karena aturan dari repeatable read yang tidak membaca hasil commit dari transaksi yang lain. Karena ini, transaksi 2 akan rollback, menggagalkan transaksinya.</p>	

Transaksi 1	Transaksi 2
<pre>contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 31 (3 rows)</pre>	<pre>contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 31 (3 rows) contoh=# update tabel1 set harga = harga + 1 where id = 1; UPDATE 1 contoh=# commit; contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 32 (3 rows)</pre>

Setelah transaksi 1 melakukan commit, harga pada id = 1 berubah. Transaksi 2 yang memulai kembali transaksinya membaca dari data hasil commit transaksi 1. Setelahnya transaksi dapat dijalankan langsung dan disimpan.

Untuk kasus kedua,

Transaksi 1	Transaksi 2
<pre> contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 32 (3 rows) contoh=# insert into tabel1 (nama, harga) values ('Sate', 35); INSERT 0 1 contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 32 4 Sate 35 (4 rows) contoh=# commit; COMMIT contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 4 Sate 35 1 Buku 30 (4 rows) </pre>	<pre> contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 32 (3 rows) contoh=# update tabel1 set harga = harga-2 where id = 1; UPDATE 1 contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 (3 rows) contoh=# commit; COMMIT contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 4 Sate 35 1 Buku 30 (4 rows) </pre>
<p>Transaksi 1 melakukan insersi dan transaksi 2 melakukan updating. Tidak terjadi conflict karena dilakukan pada object yang berbeda. Akan tetapi, ketika melakukan selection selesai melakukan insersi dan updating pada masing-masing transaksi, maka dapat dilihat bahwa yang dibaca pada masing-masing transaksi adalah yang berdasarkan operasi yang sebelumnya dilakukan. Dengan demikian, kedua transaksi tidak saling berbagi hasil operasi karena data yang dibaca adalah data pada saat sebelum transaksi pertama berjalan.</p>	
Transaksi 1	Transaksi 2

```

contoh=# begin isolation level repeatable read;
BEGIN
contoh=# select * from tabel1;
id | nama | harga
---+---+---
 2 | Pensil | 20
 3 | Roti | 25
 4 | Sate | 35
 1 | Buku | 30
(4 rows)

contoh=# update tabel1 set harga=harga -5 where id = 4;
UPDATE 1
contoh=# commit;
COMMIT

```

```

contoh=# begin isolation level repeatable read;
BEGIN
contoh=# select * from tabel1;
id | nama | harga
---+---+---
 2 | Pensil | 20
 3 | Roti | 25
 4 | Sate | 35
 1 | Buku | 30
(4 rows)

contoh=# select * from tabel1;
id | nama | harga
---+---+---
 2 | Pensil | 20
 3 | Roti | 25
 4 | Sate | 35
 1 | Buku | 30
(4 rows)

contoh=# commit;
COMMIT
contoh=# select * from tabel1;
id | nama | harga
---+---+---
 2 | Pensil | 20
 3 | Roti | 25
 1 | Buku | 30
 4 | Sate | 30
(4 rows)

```

Bahkan dapat kita lihat, ketika transaksi pertama selesai melakukan transaksi dan memberikan commit, maka pada transaksi kedua, masih menggunakan data sebelum transaksi.

Untuk kasus ketiga,

Transaksi 1	Transaksi 2
<pre> contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 4 Sate 30 (4 rows) contoh=# update tabel1 set harga=harga+5 where id = 4; UPDATE 1 contoh=# rollback; ROLLBACK </pre>	<pre> contoh=# begin isolation level repeatable read; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 4 Sate 30 (4 rows) contoh=# update tabel1 set harga=harga-5 where id = 4; UPDATE 1 contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 4 Sate 25 (4 rows) contoh=# commit; COMMIT </pre>
<p>Transaksi 1 melakukan updating harga += 5 pada id = 4. Transaksi 2 melakukan updating harga -= 5 pada id = 4. Ketika transaksi 1 telah melakukan updating, maka ketika transaksi 2 ingin melakukan updating, transaksi 2 akan menunggu. Karena transaksi 1 rollback, maka update di transaksi 1 dibatalkan sehingga transaksi 2 dapat berjalan dan menyimpan hasilnya.</p>	

Selain ketiga kasus tersebut, ketika serializable tidak memungkinkan operasi yang concurrent, repeatable read memungkinkan sebagaimana simulasi berikut ini,

Transaksi 1	Transaksi 2
<pre> contoh=# begin isolation level repeatable read; BEGIN contoh=# select sum(harga) from tabel1 where id < 3; sum ---- 50 (1 row) contoh=# insert into tabel1(nama, harga) values ('Roti Panas', 50); INSERT 0 1 contoh=# commit; COMMIT </pre>	<pre> contoh=# begin isolation level repeatable read; BEGIN contoh=# select sum(harga) from tabel1 where id >= 3; sum ---- 55 (1 row) contoh=# insert into tabel1(nama, harga) values ('Roti Dingin', 55); INSERT 0 1 contoh=# commit; COMMIT </pre>
<p>Transaksi 1 melakukan insersi objek dengan harga 50 dan transaksi 2 dengan harga 55. Transaksi 2 dimulai sebelum transaksi 1 di-commit. Bila ini dilakukan pada level isolasi serializable, maka akan menghasilkan error serialisasi. Akan tetapi, pada level repeatable read, tidak dilakukan pemeriksaan serialisasi sehingga transaksi dapat di-commit. Level <i>repeatable read</i> memungkinkan untuk dilakukan eksekusi transaksi <i>concurrent</i>.</p>	

c. Read Committed

Isolasi read committed memastikan bahwa data yang dibaca untuk klausa SELECT adalah data pada state saat akan menjalankan query. Data ini diperlakukan sebagai sebuah snapshot dari database. Hal ini berbeda dengan repeatable read yang hanya membaca data untuk digunakan transaksi sebelum transaksi tersebut dimulai, i.e. data dari *non-transaction-control statement*. Dalam satu transaksi, suatu query SELECT tetap dapat melihat hasil perubahan data dalam transaksinya sendiri.

Untuk klausa UPDATE, DELETE, SELECT FOR UPDATE, dan SELECT FOR SHARE, maka ketika terjadi perubahan pada suatu data oleh sebuah transaksi lainnya, maka ketika suatu transaksi ingin melakukan perubahan terhadap data yang sama, diperlukan lock untuk menunggu commit atau sampai transaksi melakukan roll back. Jika terjadi commit, maka transaksi yang melakukan perubahan kedua akan mengabaikan data yang dihapus dan akan menerapkan operasi yang terjadi pada data yang akan diubah. Jika terjadi roll back, maka data yang diubah akan dinegasikan dan data tidak perlu diperbaharui. Untuk pencarian, diperlukan reevaluasi untuk melihat apakah klausa pencarian masih memenuhi atau tidak. Pada dasarnya, untuk setiap data yang mengalami perubahan, maka transaksi yang akan mengubah lebih lanjut data tersebut akan menerapkan operasi yang terjadi untuk mencapai data tersebut.

Karena hal ini, maka ada kemungkinan untuk membaca snapshot yang inkonsisten antar transaksi. Karena ini, untuk operasi yang membutuhkan pencarian yang kompleks tidak dimungkinkan pada level isolasi ini. Simulasi untuk level isolasi ini adalah pada kasus melakukan penghapusan data pada 1 transaksi dan transaksi lainnya mencoba update pada data tersebut.

Untuk kasus pertama,

Transaksi 1	Transaksi 2
-------------	-------------

<pre> contoh=# begin; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 4 Sate 25 (4 rows) contoh=# delete from tabel1 where id = 4; DELETE 1 </pre>	<pre> contoh=# begin; BEGIN contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 4 Sate 25 (4 rows) contoh=# update tabel1 set harga = harga * 2 where id = 4; </pre>
<p>Transaksi 1 menjalankan delesi id = 4. Transaksi 2 ingin melakukan update pada id = 4. Karena transaksi 1 telah mengakses id = 4 lebih awal, maka akan di-<i>lock</i> terlebih dahulu dan transaksi 2 menunggu.</p>	

Transaksi 1	Transaksi 2
<pre> contoh=# commit; COMMIT contoh=# </pre>	<pre> contoh=# update tabel1 set harga = harga * 2 where id = 4; UPDATE 0 contoh=# select * from tabel1; id nama harga ---+---+--- 2 Pensil 20 3 Roti 25 1 Buku 30 (3 rows) contoh=# commit contoh=# ; COMMIT </pre>
<p>Setelah transaksi 1 commit, transaksi 2 dapat menjalankan transaksinya dan tidak terjadi error. Hal ini disebabkan oleh sifat level isolasi ini yang akan membaca kembali data sebelum setiap query dalam transaksi dilakukan sebagai snapshot. Dengan demikian data yang transaksi 2 tidak akan mengalami error karena akan menyimpan snapshot dari hal yang telah di-<i>commit</i> terlebih dahulu</p>	

d. Read Uncommitted

Isolasi level ini memungkinkan sebuah transaksi untuk membaca hasil dari transaksi lainnya yang sedang berjalan tanpa perlu menunggu commit terjadi. Karena tidak perlu menunggu commit terjadi, maka dapat terjadi inkonsistensi data. Misalkan sebuah transaksi di-rollback, maka data yang ditulis oleh transaksi tersebut jika dibaca oleh transaksi lainnya akan tetap tercatat dan tidak terjadi rollback, sehingga terjadi dirty read. Karena tidak perlu menunggu commit, maka ketika sebuah transaksi melakukan perubahan (INSERT atau UPDATE) dan kemudian transaksi lain melakukan pembacaan sebelum terjadi commit, maka akan terjadi phantom read (ketika insersi) atau

non-repeatable read (ketika terjadi perubahan) ketika transaksi yang melakukan perubahan di-*commit*.

Meskipun ada lebih banyak dampak negatif yang dihasilkan dari isolasi level ini, akan tetapi, dengan tidak menunggu commit, maka read uncommitted memberikan waktu transaksi yang lebih cepat tanpa perlu menunggu commit. Namun, risikonya adalah anomali data yang sangat mungkin terjadi karena commit tidak perlu menunggu commit.

2. Implementasi Concurrency Control Protocol

a. Two-Phase Locking (2PL)

Two-Phase Locking adalah *concurrency control protocol* yang membagi sebuah transaksi menjadi fase *growing* (memberikan kunci) dan *shrinking* (melepas kunci). Transaksi ini dapat melakukan pemutakhiran terhadap *shared lock* menjadi *exclusive lock*. Karena ada kemungkinan untuk terjadi *deadlock* dalam menjalankan suatu transaksi akibat kunci yang tidak *compatible*, maka diperlukan *deadlock prevention* untuk 2PL. *Deadlock prevention* yang digunakan dapat berupa skema *wait-and-die* atau *wound-and-wait*.

Implementasi 2PL dalam tulisan ini ditulis dalam bahasa pemrograman Java dengan mengimplementasikan *automatic lock conversion* untuk pemutakhiran kunci dan skema *wait-and-die* untuk *deadlock prevention*. Kode yang digunakan adalah sebagai berikut,

```
import java.util.*;
import java.util.regex.*;

public class TwoPhaseLocking{
    private ArrayList<String> startTime;
    public Deque<String> schedule;
    public ArrayList<Deque<String>> doneTransactions;
    public Deque<String> lockTable;
    public Deque<String> waitQueue;
    public Deque<String> finalSchedule;
    private Pattern schedulePattern;

    /*
     * Class constructor
     * @param schedule Format as follows: Ri(X),Wi(X),Ci
     */
    public TwoPhaseLocking(String schedule){
        this.schedule = new
        ArrayDeque<>(Arrays.asList(schedule.split(";")));
        this.schedulePattern =
        Pattern.compile("([RW]) (\\d+)\\((\\w)\\)| (C) (\\d+)");
        this.lockTable = new LinkedList<>();
        this.finalSchedule = new LinkedList<>();
        this.waitQueue = new LinkedList<>();
        this.doneTransactions = new ArrayList<>();
        this.startTime = new ArrayList<>();

        /* Foreach unique object in schedule, initialize the timestamp
```

```

as 0 */
    for (String operation : this.schedule) {
        Matcher matcher = schedulePattern.matcher(operation);
        if(matcher.matches()){
            if(matcher.group(2) != null){
                if(!startTime.contains(matcher.group(2))){
                    startTime.add(matcher.group(2));
                }
            }else if(matcher.group(5) != null){
                if(!startTime.contains(matcher.group(5))){
                    startTime.add(matcher.group(5));
                }
            }
        }
    }

    for (int i = 0; i < 100; i++){
        this.doneTransactions.add(new LinkedList<>());
    }
}

/*
 * Checker for validation if a lock is to be given or not
 * @param operation: Operation name
 * @returns boolean: true if the lock is to be given and false if
there is conflict
 */
private boolean exclusiveLockChecker(String operation, String
object, String id) {
    if(operation.equals("R")){
        /* Check if there is an exclusive lock on the object from
the same Ti*/
        if(lockTable.contains("XL" + id + "(" + object + ")")){
            return true;
        }else{
            /* check if there exists a exclusive lock on the object
*/
            Boolean check1 = lockTable.stream().anyMatch(lock ->
lock.endsWith("(" + object + ")") && lock.startsWith("XL"));
            return check1 ? false : true;
        }
    }else if(operation.equals("W")){
        /* Check if there are any exclusive locks on the object from
the same Ti*/
        if(lockTable.contains("XL" + id + "(" + object + ")")){
            return true;
        }else{
            /* Check if there exists a exclusive lock on the
object*/
            if(lockTable.stream().anyMatch(lock ->
lock.endsWith("(" + object + ")") && lock.startsWith("XL"))){
                return false;
            }else{
                /* check if there exists a share lock where the id
is the same */
                long sharedLockCount = lockTable.stream()

```



```

    * Removes all locks that transaction-id has
    * @param id: String
    */
    private void releaseLocks(String id){
        /* Remove all locks on lockTable and sharedLocks (both are queue
with formats XLi(X) or SLi(X)) */
        System.out.println("RELEASING LOCKS ON T"+id);
        lockTable.removeIf(lock -> lock.startsWith("XL"+id));
        lockTable.removeIf(lock -> lock.startsWith("SL"+id));
    }

    /*
    * Scheduler
    * Schedule the Transaction
    * Schedule using automatic acquisition
    */
    public void scheduler(){
        while(!schedule.isEmpty()) {
            String operation = schedule.poll();
            Matcher matcher = this.schedulePattern.matcher(operation);
            if(matcher.matches()){
                if(matcher.group(1) != null){
                    String check = this.waitQueue.stream().filter(wait
-> wait.contains(matcher.group(2))).findFirst().orElse("");
                    Boolean isNotWaiting = true;
                    if(!check.isEmpty()){
                        Matcher m1 =
this.schedulePattern.matcher(check);
                        if(m1.matches()){
                            if(m1.group(1) != null)
                                isNotWaiting =
this.exclusiveLockChecker(m1.group(1), m1.group(3), m1.group(2));
                        }
                    }
                    if(isNotWaiting){
                        if(matcher.group(1).equals("R")){
                            /* Check if there is an shared lock on this
object
transaction, proceed
* If there is and belongs to this
transaction, proceed
* If there is none, add the lock and proceed
* If exists but doesn't belong to this
transaction, give the lock to this transaction
* If exists a exclusive lock, wait
*/

                            if(this.exclusiveLockChecker(matcher.group(1), matcher.group(3),
matcher.group(2))){
                                doneTransactions.get(Integer.parseInt(matcher.group(2))).add(operation);

                                if(!lockTable.contains("SL"+matcher.group(2)+"("+matcher.group(3)+")")){
                                    System.out.println("GRANTING SHARED
LOCK ON " + matcher.group(3) + " TO T"+matcher.group(2));
                                    lockTable.add("SL"+matcher.group(2)+"("+matcher.group(3)+")");
                                }
                            }
                        }
                    }
                }
            }
        }
    }

```



```

finalSchedule.add("SL"+matcher.group(2)+"("+matcher.group(3)+")");
    }
    System.out.println("READ OF " +
matcher.group(3) + " BY T" + matcher.group(2));
    finalSchedule.add(operation);
    }else{
        // Deadlock prevention using Wait and
Die scheme

if(isCurrentTransactionYounger(matcher.group(2), matcher.group(3))){
    /* Rollback */
    System.out.println("Aborting
transaction " + matcher.group(2));
finalSchedule.add("A"+matcher.group(2));

doneTransactions.get(Integer.parseInt(matcher.group(2))).add(operation);
    Iterator<String> iterator =
schedule.iterator();

    while (iterator.hasNext()) {
        String op = iterator.next();
        if
(op.contains(matcher.group(2))) {
            iterator.remove();

doneTransactions.get(Integer.parseInt(matcher.group(2))).add(op);
        }
    }

    releaseLocks(matcher.group(2));

while(!doneTransactions.get(Integer.parseInt(matcher.group(2))).isEmpty(
)){
    String op =
doneTransactions.get(Integer.parseInt(matcher.group(2))).poll();
    schedule.add(op);
}

startTime.removeIf(id->id.equals(matcher.group(2)));
    startTime.add(matcher.group(2));

    } else {
        /* Waiting */
        System.out.println("T" +
matcher.group(2) + " WAITS FOR LOCK");
        waitQueue.add(operation);
    }
}
}else if(matcher.group(1).equals("W")){
    /* Check in SL if there is already a lock or
not.
        * If there already exist and is has the same
id, remove the lock, upgrade to this lock

```

```

        * If there already exist and doesn't have
the same id, wait until it exists
        * If there is none, proceed
        */

if(this.exclusiveLockChecker(matcher.group(1), matcher.group(3),
matcher.group(2))) {

doneTransactions.get(Integer.parseInt(matcher.group(2))).add(operation);

if(!lockTable.contains("XL"+matcher.group(2)+"("+matcher.group(3)+")")) {
    System.out.println("GRANTING
EXCLUSIVE LOCK ON " + matcher.group(3) + " TO T"+matcher.group(2));

lockTable.add("XL"+matcher.group(2)+"("+matcher.group(3)+")");

finalSchedule.add("XL"+matcher.group(2)+"("+matcher.group(3)+")");
    }
    System.out.println("WRITE OF " +
matcher.group(3) + " BY T" + matcher.group(2));
    finalSchedule.add(operation);
    }else{

if(isCurrentTransactionYounger(matcher.group(2), matcher.group(3))) {
    /* Rollback */
    System.out.println("Aborting
transaction " + matcher.group(2));
    finalSchedule.add("A" +
matcher.group(2));

doneTransactions.get(Integer.parseInt(matcher.group(2))).add(operation);
    Iterator<String> iterator =
schedule.iterator();

    while (iterator.hasNext()) {
        String op = iterator.next();
        if
(op.contains(matcher.group(2))) {
            iterator.remove();

doneTransactions.get(Integer.parseInt(matcher.group(2))).add(op);
        }
    }

    releaseLocks(matcher.group(2));

while(!doneTransactions.get(Integer.parseInt(matcher.group(2))).isEmpty(
)) {
        String op =
doneTransactions.get(Integer.parseInt(matcher.group(2))).poll();
        schedule.add(op);
    }

startTime.removeIf(id->id.equals(matcher.group(2)));
    startTime.add(matcher.group(2));

```



```

        System.out.println("Masukkan schedule (diakhiri dengan ;) : ");
        Scanner scanner = new Scanner(System.in);

        TwoPhaseLocking twoPhaseLocking = new
TwoPhaseLocking(scanner.nextLine());
        twoPhaseLocking.scheduler();

        scanner.close();
    }
}

```

Schedule yang akan disimulasikan diletakkan sebagai constructor untuk kemudian memanggil metode scheduler untuk melakukan simulasi.

Protokol yang dibuat akan diujikan untuk 2 kasus yaitu kasus tidak terjadi deadlock dan terjadi deadlock. Diberikan schedule pertama yaitu R1(X);W1(X);C1;R2(X);W2(X);C2. Hasil dari schedule ini adalah,

```
SL1 (X)  R1 (X)  UL1 (X)  XL1 (X)  W1 (X)  C1  SL2 (X)  R2 (X)  UL2 (X)  XL2 (X)  W2 (X)  C2
```

Schedule kedua adalah R1(X);W1(X);W3(Z);R1(Z);R3(X);W2(Y);R1(Y);C1;R2(X);W2(X);W3(X);C3;C2. Ketika dijalankan, maka akan menghasilkan schedule,

```
SL1 (X)  R1 (X)  UL1 (X)  XL1 (X)  W1 (X)  XL3 (Z)  W3 (Z)  A3  SL1 (Z)  R1 (Z)  XL2 (Y)
W2 (Y)  A2  SL1 (Y)  R1 (Y)  C1  XL3 (Z)  W3 (Z)  SL3 (X)  R3 (X)  UL3 (X)  XL3 (X)  W3 (X)
C3  XL2 (Y)  W2 (Y)  SL2 (X)  R2 (X)  UL2 (X)  XL2 (X)  W2 (X)  C2
```

Karena terjadi kondisi deadlock yaitu transaksi 3 menunggu transaksi 1 dan sebaliknya serta transaksi 2 menunggu transaksi 1 dan sebaliknya, maka dilakukan waiting dan aborting untuk transaksi yang lebih muda untuk kemudian dijalankan ulang di akhir setelah menjadi bebas. Hasil transaksi menjadi serial. Usia transaksi menggunakan transaksi yang masuk terlebih dahulu.

b. Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control (OCC) adalah salah satu metode *concurrency control* dalam sistem basis data yang memungkinkan beberapa transaksi untuk beroperasi secara independen, tanpa memblokir/menghalangi satu sama lain, dan kemudian memeriksa apakah ada konflik ketika mereka selesai. Metode ini umumnya digunakan dalam situasi saat transaksi jarang menyebabkan konflik, dan blokade dapat dihindari untuk meningkatkan kinerja sistem.

Keuntungan dari Optimistic Concurrency Control meliputi kinerja yang lebih baik karena transaksi tidak mengunci sumber daya selama operasi *read* atau perubahan data (*write*), sehingga memungkinkan beberapa transaksi untuk berjalan secara paralel. Namun, metode ini membutuhkan deteksi dan penanganan konflik, yang bisa menjadi kompleks tergantung pada implementasinya.

Pada implementasi OCC di bawah ini, apabila terjadi konflik pada saat operasi dijalankan, maka transaksi akan di-*rollback*. Semua operasi yang telah dilakukan oleh transaksi tersebut akan dihapus dari *schedule*, dan *write set*-nya akan dikosongkan.

```
import java.util.*;
import java.util.regex.*;

public class OptimisticConcurrencyControl {
    private Map<String, Integer> startTimestamp;
    private Map<String, Integer> validationTimestamp;
    private Map<String, Integer> finishTimestamp;
    private Deque<String> schedule;
    private Deque<String> finalSchedule;
    private Pattern schedulePattern;
    private ArrayList<Deque<String>> doneTransactions;
    private Map<String, Set<String>> writeSets, readSets; // Added
write sets
    private Integer timestamp;

    public OptimisticConcurrencyControl(String schedule) {
        this.startTimestamp = new HashMap<>();
        this.validationTimestamp = new HashMap<>();
        this.finishTimestamp = new HashMap<>();
        this.schedulePattern =
Pattern.compile("([RW]) (\\d+) \\((\\w)\\) | (C) (\\d+)");
        this.finalSchedule = new ArrayDeque<>();
        this.doneTransactions = new ArrayList<>();
        this.writeSets = new HashMap<>();
        this.readSets = new HashMap<>();
        this.timestamp = 0;
    }
}
```

```

        this.schedule = new
        ArrayDeque<>(Arrays.asList(schedule.split(";")));

        /* For each unique object in the schedule, initialize the
        timestamp as 0 */
        for (String operation : this.schedule) {
            Matcher matcher = schedulePattern.matcher(operation);
            if (matcher.matches()) {
                if (matcher.group(1) != null) {
                    startTimestamp.putIfAbsent(matcher.group(2), 0);
                    validationTimestamp.putIfAbsent(matcher.group(2),
0);
                                finishTimestamp.putIfAbsent(matcher.group(2), 0);
                                writeSets.putIfAbsent(matcher.group(2), new
HashSet<>());
                                readSets.putIfAbsent(matcher.group(2), new
HashSet<>());
                } else if (matcher.group(4) != null) {
                    startTimestamp.putIfAbsent(matcher.group(5), 0);
                    validationTimestamp.putIfAbsent(matcher.group(5),
0);
                                finishTimestamp.putIfAbsent(matcher.group(5), 0);
                                writeSets.putIfAbsent(matcher.group(5), new
HashSet<>());
                                readSets.putIfAbsent(matcher.group(5), new
HashSet<>());
                }
            }
        }

        for (int i = 0; i < 10; i++) {
            doneTransactions.add(new LinkedList<>());
        }
    }

    /*
    * Validate a given transaction j against other transaction i
    * @param j, transaction id that is about to be committed
    */
    private boolean validateTransaction(String j) {
        boolean returnVal = true;
        Set<String> readSetJ = readSets.get(j);
        for (String i : this.startTimestamp.keySet()) {
            if (!i.equals(j)) {
                if(this.validationTimestamp.get(i) <
this.validationTimestamp.get(j)){
                    if (this.finishTimestamp.get(i) <
this.startTimestamp.get(j)) {
                        /* Pass */
                    } else if (this.startTimestamp.get(j) <
this.finishTimestamp.get(i) && this.finishTimestamp.get(i) <
this.validationTimestamp.get(j)) {
                        Set<String> writeSetI = writeSets.get(i);
                        if (!Collections.disjoint(writeSetI, readSetJ))
{
                            returnVal = false;

```

```

        break;
    }
    } else {
        returnVal = false;
        break;
    }
}
}
return returnVal;
}

public void scheduler() {
    while (!schedule.isEmpty()) {
        timestamp++;
        String op = schedule.poll();
        Matcher matcher = this.schedulePattern.matcher(op);
        if (matcher.matches()) {
            /* Set the start timestamp of each transaction */
            if (matcher.group(1) != null) {
                /* Read and Write */
                String transactionId = matcher.group(2);
                if
(timestamps.get(Integer.parseInt(transactionId)).isEmpty()) {
                    this.startTimestamp.put(transactionId,
timestamp);
                }

this.timestamps.get(Integer.parseInt(transactionId)).add(op);
                if (matcher.group(1).equals("W")) {
                    System.out.println("READ OF " + matcher.group(3)
+ " BY T" + matcher.group(2));

this.writeSets.get(transactionId).add(matcher.group(3));
                } else if (matcher.group(1).equals("R")) {
                    System.out.println("LOCAL WRITE OF " +
matcher.group(3) + " BY T" + matcher.group(2));

this.readSets.get(transactionId).add(matcher.group(3));
                }
                this.finalSchedule.add(op);
            } else if (matcher.group(4) != null) {
                String transactionId = matcher.group(5);
                this.validationTimestamp.put(transactionId,
timestamp);

                if (!validateTransaction(transactionId)) {
                    /* ROLL BACK ALL THE TRANSACTION FOR THE ID */
                    System.out.println("ABORTING T" +
transactionId);

                    this.finalSchedule.add("A" + transactionId);

                    Iterator<String> it = this.schedule.iterator();

timestamps.get(Integer.parseInt(transactionId)).add(op);
                    while (it.hasNext()) {
                        String operation = it.next();

```

```

        if (operation.contains(transactionId)) {
            it.remove();
        }
    }

    doneTransactions.get(Integer.parseInt(transactionId)).add(operation);
    }

    while
    (!doneTransactions.get(Integer.parseInt(transactionId)).isEmpty()) {
        String operation =
        doneTransactions.get(Integer.parseInt(transactionId)).poll();
        schedule.add(operation);
    }

    /* Empty the writeset on a given id */
    this.writeSets.get(transactionId).clear();
    this.readSets.get(transactionId).clear();
    timestamp++;
    this.startTimestamp.put(transactionId,
timestamp);

    } else {
        timestamp++;
        System.out.println("COMMITTING T"+transactionId
+ ". WRITING LOCAL WRITE TO DB");
        this.finalSchedule.add(op);
        this.finishTimestamp.put(transactionId,
timestamp);
    }
}

}

/* Print the full schedule */
for (String operation : finalSchedule) {
    System.out.print(operation + "\n");
}

}

public static void main(String[] args) {
    System.out.println("Masukkan schedule (diakhiri dengan ;) : ");
    Scanner scanner = new Scanner(System.in);
    String input = scanner.nextLine();

    OptimisticConcurrencyControl scheduler = new
OptimisticConcurrencyControl(input);
    scheduler.scheduler();

    scanner.close();

    // Contoh : R1(A);R2(A);W1(A);W2(A);C1;C2;
}
}

```


Adapun untuk menguji OCC di atas, terdapat dua contoh simulasi, yaitu simulasi *schedule* tanpa konflik dan dengan adanya konflik:

1. *Schedule* tanpa konflik:

R1(A);W1(B);C1;R2(B);W2(C);C2;R3(C);W3(C);C3. Dalam contoh ini, setiap transaksi (T1, T2, T3) melakukan operasi baca (R) dan tulis (W) pada variabel yang berbeda (A, B, C). Tidak ada tumpang tindih operasi baca-tulis atau tulis-tulis antar transaksi. Setiap transaksi kemudian di-commit tanpa ada konflik. Keluaran/hasil dari *schedule* tersebut adalah sebagai berikut:

R1 (A) W1 (B) C1 R2 (B) W2 (C) C2 R3 (C) W3 (C) C3
--

2. *Schedule* dengan adanya konflik:

R1(X);R2(X);W1(X);W2(X);W3(X);C1;C2;C3. Dalam contoh *schedule* ini, hasil keluaran yang diberikan adalah sebagai berikut:

R1 (X) R2 (X) W1 (X) W2 (X) W3 (X) C1 A2 C3 R2 (X) W2 (X) C2
--

Pertama-tama, *schedule* dimulai dengan membaca (R) dari T1, R2(X), W1(X), W2(X), dan W3(X), dan kemudian melakukan commit (C1). Namun, terjadi *rollback* pada T2. Hal ini disebabkan oleh adanya konflik. T2 melakukan *rollback* karena terdapat *overlap* dengan T1. Setelah *rollback*, dilakukan *commit* terlebih dahulu oleh T3 (C3) yang kemudian dilanjutkan *read* kembali (R2(X)), *write* (W2(X)), dan *commit* (C2) untuk T2.

Hasil tersebut menunjukkan bagaimana skema Optimistic Concurrency Control (OCC) bekerja dengan mendeteksi konflik dan melakukan *rollback* pada transaksi yang terlibat dalam konflik, kemudian melanjutkan dengan transaksi yang dapat di-commit.

c. Multiversion Timestamp Ordering Concurrency Control (MVCC)

Multiversion Timestamp Ordering Concurrency Control (MVCC) adalah mekanisme *concurrency control* yang digunakan dalam sistem manajemen basis data (DBMS) untuk

mengelola fitur akses bersama (*shared access*) ke data dan memastikan konsistensi dalam *multi user environment*. Metode ini sangat umum digunakan dalam sistem yang mendukung *transactions*, di mana beberapa pengguna dapat membaca (*read*) dan menulis (*write*) data secara bersamaan.

Keuntungan penggunaan MVCC di antaranya adalah peningkatan *concurrency* dan pengurangan *traffic* atas *resources* yang diakses secara bersamaan oleh banyak pengguna. Mekanisme ini memungkinkan isolasi yang lebih tinggi antar transaksi, karena masing-masing transaksi melakukan operasi terhadap salinan (versi) data mereka sendiri. Namun, perlu diketahui bahwa pengelolaan setiap versi data memerlukan implementasi yang hati-hati untuk mengendalikan *overhead* penyimpanan yang terkait dengan pemeliharaan versi-versi tersebut.

Berikut adalah implementasi MVCC menggunakan bahasa Java:

```
import java.util.*;
import java.util.regex.*;

public class MVCC {
    private Map<String, List<Map<String, Object>>> versionMap;
    private Deque<String> schedule;
    private Deque<String> finalSchedule;
    private Deque<Map<String, Object>> sequence;
    private Pattern schedulePattern;
    private Integer counter;
    private int[] transactionCounter = new int[10];

    public MVCC(String schedule) {
        this.schedulePattern =
        Pattern.compile("([RW])(\\d+)\\((\\w)\\)| (C) (\\d+)");
        this.finalSchedule = new ArrayDeque<>();
        this.versionMap = new HashMap<>();
        this.counter = 0;
        for (int i = 0; i < transactionCounter.length; i++) {
            this.transactionCounter[i] = i;
        }
        this.schedule = new
        ArrayDeque<>(Arrays.asList(schedule.split(";")));
        this.sequence = new ArrayDeque<>();
    }

    public void scheduler() {
        while (!schedule.isEmpty()) {
            String op = schedule.poll();
            Matcher matcher = this.schedulePattern.matcher(op);
            if (matcher.matches()) {
                if (matcher.group(1) != null) {
                    String transaction = matcher.group(2);
```

```

        String object = matcher.group(3);
        if (matcher.group(1).equals("R")) {
            read(op, transaction, object);
        } else if (matcher.group(1).equals("W")) {
            write(op, transaction, object);
        }
    } else if (matcher.group(4) != null) {
        /* Commit */
    }
}

}

}

public void read(String op, String transaction, String object) {
    if (!this.versionMap.containsKey(object)) {
        this.versionMap.put(object, new ArrayList<>());
    }

    int maxIndex = this.getMaxVersionWrite(object);

    if (maxIndex >= this.versionMap.get(object).size()) {
        this.versionMap.get(object).add(new HashMap<>());
    }

    int maxWrite = (int)
this.versionMap.get(object).get(maxIndex).getOrDefault("timestampW", 0);
    int maxRead = (int)
this.versionMap.get(object).get(maxIndex).getOrDefault("timestampR", 0);
    int maxVersion = (int)
this.versionMap.get(object).get(maxIndex).getOrDefault("version", 0);
    HashMap<String, Object> entry = new HashMap<>();
    entry.put("transaction", transaction);
    entry.put("object", object);
    entry.put("op", op);
    entry.put("timestamp", new int[]{maxRead,
this.transactionCounter[Integer.parseInt(transaction)]});
    entry.put("version", 0);
    this.sequence.add(entry);

    if (this.transactionCounter[Integer.parseInt(transaction)] >
maxRead) {
        this.versionMap.get(object).get(maxIndex).put("timestampR",
this.transactionCounter[Integer.parseInt(transaction)]);
        this.versionMap.get(object).get(maxIndex).put("timestampW",
maxWrite);
    }

    System.out.println(op + " Read " + object + " at version " +
maxVersion + " Timestamp " + object +
        " now is (" +
this.versionMap.get(object).get(maxIndex).get("timestampR") + ", " +
this.versionMap.get(object).get(maxIndex).get("timestampW") + ")");

    this.finalSchedule.add(op);
    this.counter++;
}

```

```

public void write(String op, String transaction, String object) {
    if (!this.versionMap.containsKey(object)) {
        this.versionMap.put(object, new ArrayList<>());
    }

    int maxIndex = this.getMaxVersionWrite(object);

    if (maxIndex >= this.versionMap.get(object).size()) {
        this.versionMap.get(object).add(new HashMap<>());
    }

    int maxWrite = (int)
this.versionMap.get(object).get(maxIndex).getOrDefault("timestampW", 0);
    int maxRead = (int)
this.versionMap.get(object).get(maxIndex).getOrDefault("timestampR", 0);
    int maxVersion = (int)
this.versionMap.get(object).get(maxIndex).getOrDefault("version", 0);

    if (this.transactionCounter[Integer.parseInt(transaction)] <
maxRead) {
        Map<String, Object> entry = new HashMap<>();
        entry.put("transaction", transaction);
        entry.put("object", object);
        entry.put("op", op);
        entry.put("timestamp", new int[]{maxRead,
this.transactionCounter[Integer.parseInt(transaction)]});
        entry.put("version", maxVersion);
        this.sequence.add(entry);
        this.rollback(transaction);
    } else if
(this.transactionCounter[Integer.parseInt(transaction)] < maxWrite) {
        this.versionMap.get(object).get(maxIndex).put("timestampW",
this.transactionCounter[Integer.parseInt(transaction)]);
        this.versionMap.get(object).get(maxIndex).put("timestampR",
maxRead);
        Map<String, Object> entry = new HashMap<>();
        entry.put("transaction", transaction);
        entry.put("object", object);
        entry.put("op", op);
        entry.put("timestamp", new int[]{maxRead,
this.transactionCounter[Integer.parseInt(transaction)]});
        entry.put("version", maxVersion);
        this.sequence.add(entry);
        this.counter++;
    } else {
        this.versionMap.get(object).get(maxIndex).put("timestampW",
this.transactionCounter[Integer.parseInt(transaction)]);
        this.versionMap.get(object).get(maxIndex).put("timestampR",
maxRead);
        this.versionMap.get(object).get(maxIndex).put("version",
this.transactionCounter[Integer.parseInt(transaction)]);
        System.out.println(op + " Write " + object + " at version "
+ this.transactionCounter[Integer.parseInt(transaction)] + " Timestamp "
+ object + " now is (" + maxRead + ", " +
this.transactionCounter[Integer.parseInt(transaction)] + ")");
    }
}

```

```

        this.finalSchedule.add(op);
        this.counter++;
    }
}

public int getMaxVersionWrite(String object) {
    int maxWTimestamp = 0;
    int maxIndex = 0;

    for (int i = 0; i < this.versionMap.get(object).size(); i++) {
        int timestampW = (int)
this.versionMap.get(object).get(i).getOrDefault("timestampW", 0);
        if (timestampW > maxWTimestamp) {
            maxWTimestamp = timestampW;
            maxIndex = i;
        }
    }

    return maxIndex;
}

public void rollback(String transactionId) {
    List<Map<String, Object>> txSequence = new ArrayList<>();
    for (Map<String, Object> entry : this.sequence) {
        if (entry.get("transaction").equals(transactionId)) {
            txSequence.add(entry);
            this.schedule.remove(entry.get("op"));
        }
    }
    for (Map<String, Object> entryy : txSequence) {
        this.schedule.addLast((String) entryy.get("op"));
    }
    this.transactionCounter[Integer.parseInt(transactionId)] =
this.counter;
    System.out.println("Transaction " + transactionId + " rolled
back. Assigned new timestamp: " +
this.transactionCounter[Integer.parseInt(transactionId)] + ".");
}

public static void main(String[] args) {
    MVCC mvcc = new
MVCC ("R5 (A) ;R2 (B) ;R1 (B) ;W3 (B) ;W3 (C) ;R5 (C) ;R2 (C) ;R1 (A) ;R4 (D) ;W3 (D) ;W5
(B) ;W5 (C) ");
    mvcc.scheduler();
}
}

```

Adapun pengujian MVCC dilakukan dengan memasukkan *schedule* R5(A);R2(B);R1(B);W3(B);W3(C);R5(C);R2(C);R1(A);R4(D);W3(D);W5(B);W5(C). dengan hasil keluarannya adalah sebagai berikut:

```
R5(A) Read A at version 0 Timestamp A now is (5, 0)
R2(B) Read B at version 0 Timestamp B now is (2, 0)
R1(B) Read B at version 0 Timestamp B now is (2, 0)
W3(B) Write B at version 3 Timestamp B now is (2, 3)
W3(C) Write C at version 3 Timestamp C now is (0, 3)
R5(C) Read C at version 3 Timestamp C now is (5, 3)
R2(C) Read C at version 3 Timestamp C now is (5, 3)
R1(A) Read A at version 0 Timestamp A now is (5, 0)
R4(D) Read D at version 0 Timestamp D now is (4, 0)
Transaction 3 rolled back. Assigned new timestamp: 9.
W5(C) Write C at version 5 Timestamp C now is (5, 5)
W3(D) Write D at version 9 Timestamp D now is (4, 9)
```

Pertama-tama, transaksi R5(A) membaca data A pada versi 0 dan mencatat timestamp aktualnya. Kemudian, R2(B) dan R1(B) membaca data B pada versi 0 dan mencatat timestampnya. W3(B) menuliskan data B dengan versi baru (versi 3) dan mencatat *timestamp*, sedangkan W3(C) menuliskan data C dengan versi 3 dan mencatat *timestamp*-nya. R5(C) dan R2(C) membaca data C pada versi 3 dan mencatat timestampnya.

Selanjutnya, R1(A) kembali membaca data A pada versi 0, dan R4(D) membaca data D pada versi 0, semuanya mencatat timestamp sesuai. Namun, saat transaksi W5(B) mencoba menuliskan data B, terjadi konflik dengan transaksi W3(B) yang telah menuliskan versi baru (versi 3). Oleh karena itu, transaksi W5(B) dibatalkan (rolled back) dan diberikan *timestamp* baru (versi 5).

Saat transaksi W3(D) mencoba menuliskan data D, terdeteksi bahwa terdapat konflik dengan transaksi R4(D) yang telah membaca data D pada versi 0. Sebagai hasilnya, transaksi W3(D) juga dibatalkan dan diberikan timestamp baru (versi 9) sesuai dengan urutan transaksi yang telah dieksekusi sejauh ini.

Hasil akhirnya mencerminkan strategi MVCC dalam mengelola konflik dan menciptakan versi baru untuk setiap tulisan. Transaksi yang mengalami konflik dibatalkan dan diberikan timestamp baru agar tetap konsisten dengan aturan pembacaan tertinggi (*highest-read*) dan penulisan tertinggi (*highest-write*) pada setiap objek data.

3. Eksplorasi Recovery

a. *Write-Ahead Log*

Write-Ahead Log (WAL) adalah salah satu metode standar untuk memastikan integritas data sehingga keandalan dan konsistensi data dalam sebuah basis data dapat dijaga. Konsep utama dari WAL adalah setiap perubahan pada data harus di-*write* hanya setelah perubahan tersebut telah dicatat (*logged*), yaitu setelah catatan (*log*) WAL yang menjelaskan perubahan tersebut diambil dan disimpan secara permanen. Dengan mengikuti prosedur WAL, halaman data tidak perlu disimpan ke *disk* setiap kali transaksi selesai, karena apabila terjadi kegagalan, basis data dapat dipulihkan menggunakan *log*. Perubahan apa pun yang belum diterapkan dapat diulang dari catatan (*log*) WAL. Proses ini disebut pemulihan *roll-forward*, atau juga dikenal sebagai *redo*.

Penggunaan WAL menghasilkan jumlah tulisan *disk* yang signifikan lebih sedikit, karena hanya berkas WAL yang perlu di-*flush* ke *disk* untuk menjamin bahwa suatu transaksi sudah terjadi, bukan setiap berkas data yang diubah oleh transaksi. Berkas WAL ditulis secara berurutan, sehingga biaya sinkronisasi WAL jauh lebih rendah daripada biaya pengosongan halaman data. Hal ini tentu menjadi sebuah pilihan unggul untuk server yang menangani banyak transaksi kecil dengan akses ke banyak bagian berbeda dalam basis data. Selain itu, ketika server memproses banyak transaksi kecil secara bersamaan, satu *fsync* dari berkas WAL bisa jadi cukup untuk menyelesaikan banyak transaksi.

WAL memungkinkan dukungan *on-line backup* dan *point-in-time recovery*. Pengarsipan data WAL dapat mendukung pengembalian (*revert*) ke setiap waktu tertentu yang dicakup oleh data WAL dengan memasang salinan cadangan fisik sebelumnya dari basis data dan memutar kembali WAL sejauh waktu yang diinginkan. Sebagai tambahan, cadangan fisik tidak harus menjadi *snapshot* instan dari keadaan basis data. Jika dibuat selama beberapa periode waktu, memutar kembali WAL untuk periode tersebut akan memperbaiki inkonsistensi internal.

b. Continuous Archiving

Continuous archiving adalah metode penyimpanan secara terus-menerus dari berkas *log* transaksi Write-Ahead Log (WAL) ke lokasi cadangan eksternal. Metode ini membuat salinan cadangan yang dapat digunakan untuk pemulihan (*recovery*) setiap saat. Continuous Archiving memiliki fitur penuh untuk pemulihan hingga titik tertentu dalam waktu. Pemulihan dapat dilakukan dengan memanfaatkan salinan berkas *log* transaksi dan berkas data dari cadangan yang telah disimpan. Tujuan utamanya adalah memberikan kemampuan pemulihan dengan memastikan keandalan data dalam keadaan gagal atau rusak.

Untuk dapat melakukan *continuous archiving* (atau biasa disebut sebagai “*online backup*” oleh beberapa vendor basis data), diperlukan rangkaian terus-menerus (*continuous sequence*) dari berkas arsip WAL yang mencakup atribut waktu setidaknya sejauh waktu mulai dari cadangan basis data. Jadi, sebelum mengambil cadangan basis data, sebaiknya prosedur untuk arsip berkas WAL perlu disiapkan dan diuji terlebih dahulu.

c. Point-in-Time Recovery

Point-in-Time Recovery (PITR) adalah kemampuan pemulihan basis data PostgreSQL ke suatu titik waktu tertentu, bukan hanya ke titik pemulihan terakhir. Metode ini dapat berguna dalam situasi saat basis data perlu dikembalikan ke kondisi sebelum suatu peristiwa tertentu terjadi, misalnya, kesalahan pengguna atau kerusakan data.

Proses PITR melibatkan penggunaan Continuous Archiving (yang menggunakan Write-Ahead Log/WAL) untuk membuat salinan cadangan transaksi secara terus-menerus. Dengan menggunakan cadangan ini, basis data dapat dipulihkan ke titik waktu yang diinginkan. Berikut adalah langkah-langkah umum untuk melakukan Point-in-Time Recovery:

- Pastikan bahwa Continuous Archiving sudah diaktifkan dan berkas log transaksi (WAL) diarsipkan secara terus-menerus.
- Pastikan bahwa berkas log transaksi yang diarsipkan mencakup periode waktu yang mencakup titik pemulihan yang diinginkan.
- Hentikan server PostgreSQL jika masih berjalan.

- Salin seluruh direktori data cluster ke lokasi sementara jika diperlukan.
- Hapus semua berkas dan direktori di dalam direktori data cluster serta root direktori tablespaces yang digunakan.
- Kembalikan berkas basis data dari cadangan berkas sistem dengan memperhatikan kepemilikan dan izin yang benar.
- Hapus berkas di dalam pg_wal/ yang berasal dari cadangan berkas sistem dan mungkin sudah tidak relevan.
- Jika terdapat berkas segmen WAL yang tidak diarsip, salin ke pg_wal/.
- Atur pengaturan pemulihan dalam postgresql.conf.
- Buat berkas recovery.signal di dalam direktori data cluster.
- Jalankan server untuk memulai proses pemulihan. Server akan masuk ke mode pemulihan dan membaca berkas WAL yang telah diarsipkan.
- Periksa isi basis data untuk memastikan pemulihan hingga keadaan yang diinginkan. Jika tidak sesuai, ulangi kembali proses pemulihan dari langkah 1.
- Jika semuanya telah sesuai dengan keadaan yang diinginkan, izinkan pengguna untuk terhubung kembali dengan mengembalikan pg_hba.conf ke keadaan normal.

Dengan melakukan langkah-langkah di atas, basis data PostgreSQL dapat dipulihkan ke titik waktu tertentu menggunakan Continuous Archiving dan Write-Ahead Log. Proses tersebut memberikan fleksibilitas dalam manajemen pemulihan dan memastikan keandalan dan konsistensi data.

d. Simulasi Kegagalan pada PostgreSQL

Sebelum simulasi kegagalan data adalah sebagai berikut,

```
contoh=# insert into tabel1(nama,harga) values ('Laptop', 200);
INSERT 0 1
contoh=# select * from tabel1;
 id |  nama  | harga
-----+-----+-----
  2 | Pensil |    20
  3 | Roti   |    25
  1 | Buku   |    30
  5 | Pisang |    30
  6 | Laptop |   200
(5 rows)
```

Gambar 3.1 Data sebelum kegagalan

Untuk memulai simulasi kegagalan, akan dibuat *backup* terlebih dahulu dari cluster database. Aturlah postgresql.conf untuk mengatur setting sebagai berikut agar continuous archiving dapat berjalan,

```
wal_level = replica
archive_mode = on
archive_command = 'copy "%p" "C:\\Program
Files\\PostgreSQL\\15\\archivedir\\%f"' # Harap sesuaikan untuk UNIX
based system
```

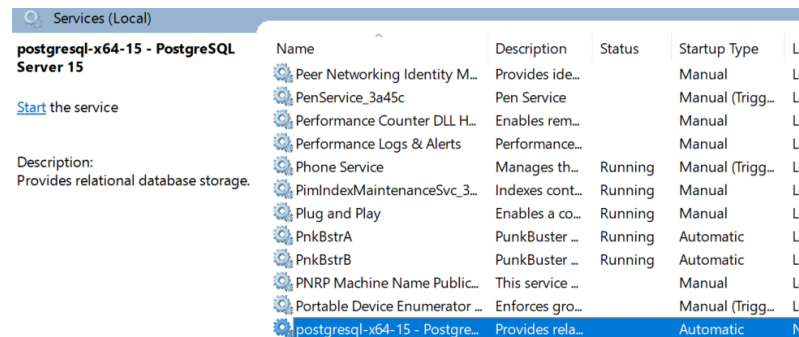
Selanjutnya, pastikan bahwa DBMS menuliskan WAL dengan memasukkan perintah berikut

```
psql -U postgres -c "select pg_switch_wal()"
```

Buatlah backup dengan menggunakan pg_basebackup

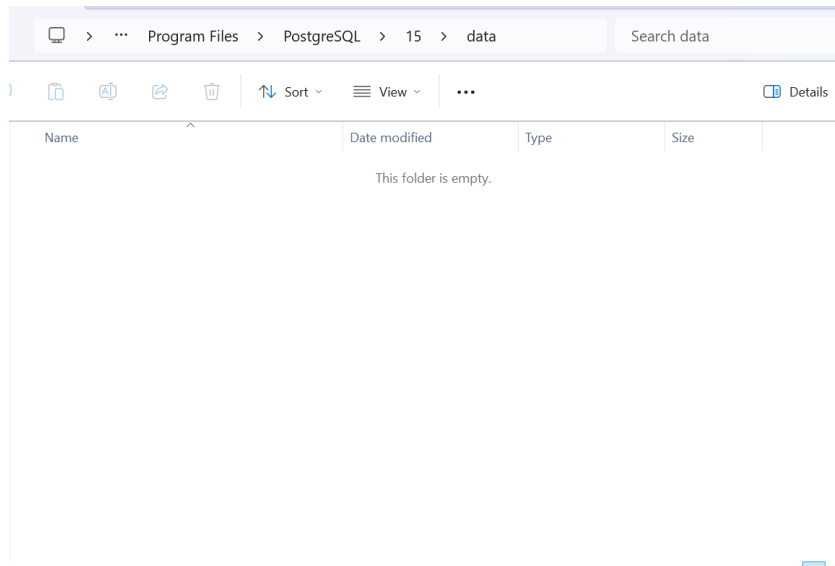
```
pg_basebackup -U postgres -Ft -D "C:/Program
Files/PostgreSQL/15/archivedir" #sesuaikan direktori
```

Akan dihasilkan 3 file pada direktori yang dituliskan yaitu base.tar, pg_wal.tar, dan backup_manifest. Selanjutnya akan disimulasikan kegagalan dengan mematikan service PostgreSQL dan melakukan data *corrupting* pada folder data.



The screenshot shows the Windows Services console with the 'Services (Local)' window open. The 'postgresql-x64-15 - PostgreSQL Server 15' service is selected. The service is currently 'Running' and has an 'Automatic' startup type. The description of the service is 'Provides relational database storage.' The table below lists the services visible in the console.

Name	Description	Status	Startup Type	Path
Peer Networking Identity M...	Provides ide...	Stopped	Manual	C:\Program Files\Microsoft SQL Server\15\...
PenService_3a45c	Pen Service	Stopped	Manual (Trigg...	C:\Program Files\Microsoft SQL Server\15\...
Performance Counter DLL H...	Enables rem...	Stopped	Manual	C:\Program Files\Microsoft SQL Server\15\...
Performance Logs & Alerts	Performance...	Stopped	Manual	C:\Program Files\Microsoft SQL Server\15\...
Phone Service	Manages th...	Running	Manual (Trigg...	C:\Program Files\Microsoft SQL Server\15\...
PimIndexMaintenanceSvc_3...	Indexes cont...	Running	Manual	C:\Program Files\Microsoft SQL Server\15\...
Plug and Play	Enables a co...	Running	Manual	C:\Program Files\Microsoft SQL Server\15\...
PnkBstrA	PunkBuster ...	Running	Automatic	C:\Program Files\Microsoft SQL Server\15\...
PnkBstrB	PunkBuster ...	Running	Automatic	C:\Program Files\Microsoft SQL Server\15\...
PNRP Machine Name Public...	This service ...	Stopped	Manual	C:\Program Files\Microsoft SQL Server\15\...
Portable Device Enumerator ...	Enforces gro...	Stopped	Manual (Trigg...	C:\Program Files\Microsoft SQL Server\15\...
postgresql-x64-15 - Postgre...	Provides rela...	Running	Automatic	C:\Program Files\Microsoft SQL Server\15\...



Gambar 3.2 Menghentikan Database dan Mengosongkan Folder Data

Saat mencoba query yang sama, maka data yang dihasilkan adalah sebagai berikut,

```
contoh=# select * from tabel1;
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
!?!> select * from tabel1;
You are currently not connected to a database.
```

Gambar 3.3 Menghentikan Database dan Mengosongkan Folder Data

Untuk melakukan *recovery*, akan dilakukan ekstraksi *base.tar* ke folder *data* dan *pg_wal.tar* ke folder *pg_wal* sebagai upadirektori dari *data*. Ekstraksi harus dilakukan secara sekuensial.

Setelahnya masuk kembali ke *postgresql.conf* untuk menambahkan setting berikut,

```
restore_command = 'copy "C:\\Program
Files\\PostgreSQL\\15\\archivedir\\%f" %p'
```

Tambahkan pula file *recover.signal* pada folder *data* untuk memulai mode *recovery*.

Setelahnya mulai kembali service PostgreSQL. Uji cobakan mengakses tabel atau database yang ada. Jika berhasil, maka sukses terjadi recovery. Terlihat pada gambar 3.3 bahwa data yang tersedia adalah sama dengan data pada gambar 3.1, menyatakan data berhasil dikembalikan.

```
PS C:\Users\Matthew> psql -U postgres contoh
Password for user postgres:
psql (15.4)
WARNING: Console code page (437) differs from Windows code page (1252)
        8-bit characters might not work correctly. See psql reference
        page "Notes for Windows users" for details.
Type "help" for help.

contoh=# select * from tabel1;
 id | nama  | harga
----+-----+-----
  2 | Pensil |    20
  3 | Roti   |    25
  1 | Buku   |    30
  5 | Pisang |    30
  6 | Laptop |   200
(5 rows)
```

Gambar 3.3 Contoh Recovery yang Sukses beserta Perintah Backup

4. Pembagian Kerja

NIM	Nama	Bagian
13521007	Matthew Mahendra	<ul style="list-style-type: none">- Eksplorasi Transaction Isolation: Repeatable Read, Read Committed, Read Uncommitted, dan simulasinya untuk repeatable read dan read committed.- Implementasi Concurrency Control Protocol: Two Phase Locking, OCC- Eksplorasi Recovery: Simulasi Kegagalan pada PostgreSQL
13521018	Syarifa Dwi Purnamasari	<ul style="list-style-type: none">- Eksplorasi Transaction Isolation: Serializable dan simulasinya, Simulasi Repeatable Read- Implementasi Concurrency Control Protocol: OCC, MVCC
13521029	M. Malik I. Baharsyah	<ul style="list-style-type: none">- Eksplorasi Recovery: Write-Ahead Log, Continuous Archiving
13521028	M. Zulfiansyah Bayu Pratama	<ul style="list-style-type: none">- Implementasi Concurrency Control Protocol: MVCC- Eksplorasi Recovery: Point-in-Time Recovery

Referensi

PostgreSQL 15.5 documentation. PostgreSQL Documentation. (2023, November 9).
<https://www.postgresql.org/docs/15/index.html>

Silberschatz, A., Korth, H. F., Sudarshan. (2019, March). *Database System Concept, Seventh Edition*. New York: McGraw-Hill.