



Sistem Basis Data

KONTROL KONKURENSI

Alif Finandhita, S.Kom

Pengendalian Konkurensi

- ⦿ Protokol berbasis-penguncian
- ⦿ Protokol berbasis-pembatasan waktu
- ⦿ Protokol berbasis-validasi
- ⦿ Penanganan Deadlock

Protokol berbasis penguncian

- Penguncian adalah salah satu mekanisme pengendalian akses konkuren terhadap sebuah item data
- Item data dapat dikunci dengan dua cara :
 1. *exclusive (X) mode*. Item data dapat dibaca (read) dan diubah(write) dengan sama baik. Penguncian terhadap data x membutuhkan instruksi **lock-X**.
 2. *shared (S) mode*. Item data hanya dapat dibaca (read). Untuk menshare kan data digunakan perintah **lock-S**.
- Penguncian dibutuhkan untuk mengelola proses konkuren. Transaksi dapat diproses setelah ada jaminan.

- Tabel kemungkinan penguncian

	S	X
S	true	false
X	false	false

- Sebuah transaksi terkadang membutuhkan jaminan penguncian pada saat mengakses item data supaya tertutup terhadap transaksi yang lain
- Beberapa transaksi dapat men-share sebuah item, tetapi jika beberapa transaksi menahan secara eksklusif pada sebuah item maka tidak ada transaksi lain yang dapat melakukan penguncian pada item tersebut.
- Jika sebuah penguncian tidak diperoleh, transaksi yang diminta akan dibuat menunggu sampai penguncian yang dilakukan transaksi lain dilepas.

- ◉ Contoh penerapan penguncian pada pentransferan dana dari B ke A :

```
T1: lock-X(B);  
    read (B);  
    B ← B - 500  
    write (B)  
    unlock(B);  
    lock-X(A);  
    read (A);  
    A ← A + 500  
    write (A)  
    unlock(A);
```

- ◉ Transaksi T2 yang akan menampilkan total saldo kedua rekening:

```
T2: lock-S(A);  
    read (A);  
    unlock(A);  
    lock-X(B);  
    read (B);  
    write (B)  
    unlock(B);  
    display(A+B)
```

- ◉ Sebuah **locking protocol** adalah sekumpulan aturan dalam sebuah transaksi yang memanggil dan melepas penguncian. Protokol locking akan membatasi penjadwalan yang ada.

Kemungkinan pada protokol-

- Sehubungan dengan sebagian jadwal penguncian

T_3	T_4
lock-X(B) read(B) $B := B - 50$ write(B)	
lock-X(A)	lock-S(A) read(A) lock-S(B)

- Baik T_3 maupun T_4 tidak menyelesaikan kondisi **lock-S(B)** mengakibatkan T_4 menunggu T_3 untuk melepaskan penguncian terhadap B , sementara eksekusi **lock-X(A)** mengakibatkan T_3 menunggu T_4 melepaskan penguncian terhadap A .
- Kondisi ini disebut **deadlock**.
 - Untuk mengatasi masalah ini T_3 atau T_4 harus di roll back dan melepaskan kunci.

Kemungkinan pada protokol penguncian

- ◉ Deadlock selalu mungkin terjadi dalam protokol lock.
- ◉ **Starvation** juga mungkin terjadi jika pengendalian akses konkuren tidak baik.
Contoh :
 - Sebuah transaksi mungkin dapat menunggu X-lock pada sebuah item, sementara transaksi lain pada urutan membutuhkan S-lock pada item yang sama.
 - Transaksi lain yang sama berulang-ulang melakukan roll back sampai dengan deadlock.
- ◉ Pengelolaan konkurensi dapat dirancang untuk menghindari starvation.

Protokol Penguncian 2-Phase

- ⦿ Aturan ini menjamin terjadinya conflict-serializable .
- ⦿ Phase 1: Fase bertumbuh (Growing Phase)
 - Transaksi dapat melakukan sejumlah penguncian
 - Transaksi belum melepaskan satupun penguncian
- ⦿ Phase 2: Fase pelepasan (Shrinking Phase)
 - Transaksi mungkin melepas kunci
 - Transaksi belum melakukan penguncian yang baru
- ⦿ Titik dalam schedule dimana transaksi tersebut telah mendapatkan penguncian akhir disebut lockpoint transaksi.

Protokol Penguncian 2-Phase

- ⦿ Locking dua fase *tidak menjamin* terjadinya deadlock
- ⦿ **strict two-phase locking**. Dengan mekanisme ini dikehendaki bahwa semua penguncian dengan mode *exclusive* dari sebuah transaksi harus tetap dipegang hingga transaksi berada dalam status berhasil sempurna (committed).
- ⦿ **Rigorous two-phase locking** yang menghendaki semua penguncian (exclusive maupun share) tetap diterapkan hingga transaksi committed.

Konversi Penguncian

- Penguncian 2-Phase dengan konversi penguncian:
 - Phase Pertama:
 - ★ Dapat menggunakan **lock-S** pada data
 - ★ Dapat menggunakan **lock-X** pada data
 - ★ Dapat konversi **lock-S** menjadi **lock-X** (**upgrade**)
 - Second Phase:
 - ★ Dapat membuka **lock-S**
 - ★ Dapat membuka **lock-X**
 - ★ Dapat konversi **lock-X** menjadi **lock-S** (**downgrade**)
- Protokol ini menjamin serialisabilitas dari suatu transaksi. Tapi masih bergantung kepada programmer untuk menambahkan berbagai macam instruksi penguncian.

• Konversi penguncian

T_5 : read (a_1)
read (a_2)

...

read (a_n)
write (a_1)

T_6 : read (a_1)
read (a_2)
write ($a_1 + a_2$)

- Jika menerapkan Locking Dua fase, maka T_5 harus mengunci a_1 dalam mode exclusive. Akibatnya semua eksekusi konkuren dari kedua transaksi menjadi eksekusi serial.
- Jika T_5 melakukan penguncian dengan mode exclusive di saat penulisan a_1 , maka kondisi konkurensi akan lebih baik, karena T_5 dan T_6 dapat mengakses a_1 dan a_2 secara simultan
- Peningkatan penguncian dari share menjadi exclusive disebut **upgrade** dan sebaliknya disebut **downgrade**

Konversi Penguncian

T_5	T_6
Lock-S (a1)	Lock-S (a1)
Lock-S (a2)	Lock-S (a2)
Lock-S (a3)	
Lock-S (a4)	
	Unlock (a1)
	Unlock (a2)
Lock-S (an)	
Upgrade (a1)	

Akuisisi otomatis dari penguncian

- Transaksi T_i menjalankan operasi read/write standar tanpa ada prosedur penguncian.
- Operasi **read(D)** akan dijalankan :
 - if T_i sudah mengunci D
then
 - read(D)
 - else**
 - begin**
 - if diperlukan tunggu s/d tidak ada transaksi lain me- **lock-X** pada D
 - lakukan T_i **lock-S** pada D ;
 - read(D)
 - end**

Akuisisi otomatis dari penguncian

- Proses **write**(D) :

if T_i telah **lock-X** pada D

then

 write(D)

else

begin

 jika perlu tunggu s.d tidak ada transaksi lain yang memiliki lock pada D ,

 jika T_i telah **lock-S** pada D

then

upgrade lock pada D ke **lock-X**

else

 perintahkan T_i me- **lock-X** pada D

 write(D)

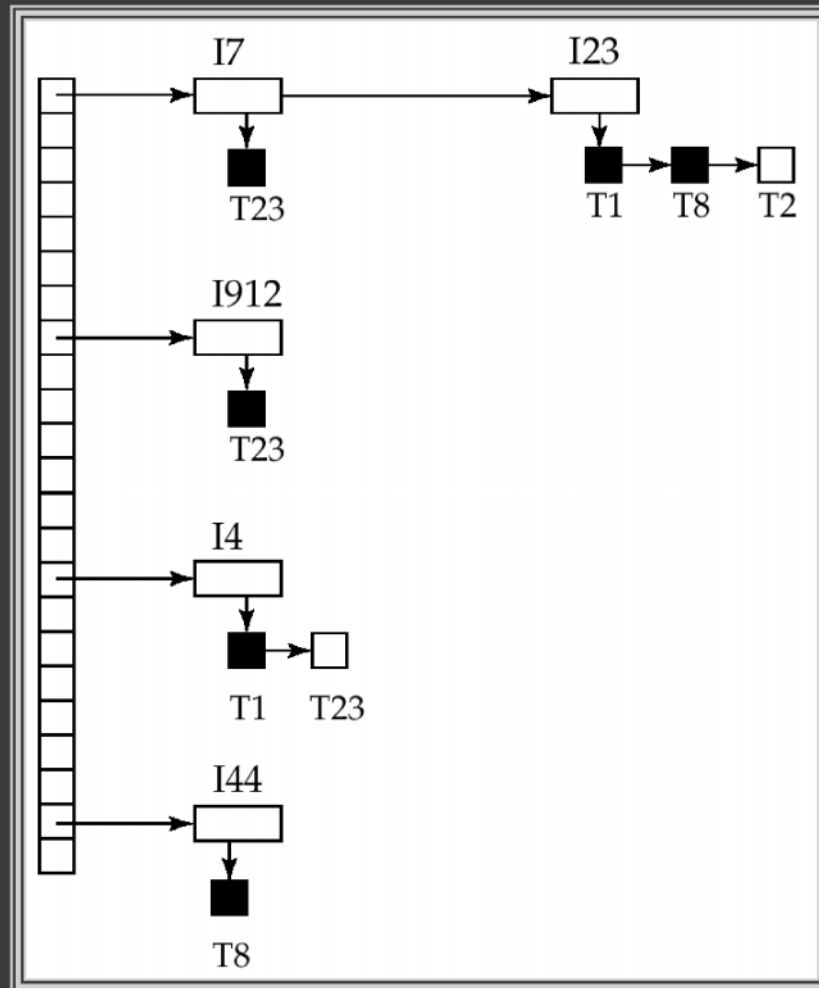
end;

- Semua penguncian akan dilepas setelah transaksi *committed* atau *abort*

Implementasi Penguncian

- **Sebuah Lock manager** dapat diterapkan sebagai sebagian dari proses yang melayani permintaan lock dan unlock
- lock manager menjawab permintaan lock dengan mengirimkan pesan penguncian (atau pesan melakukan roll back dalam kasus deadlock)
- Transaksi yang minta akan menunggu sampai dijawab
- lock manager merawat struktur data yang disebut **lock table** untuk menjamin penguncian record dan menunda permintaan
- lock table selalu diterapkan sebagai tabel indeks yang ada di memory pada nama data yang di lock

Lock Table

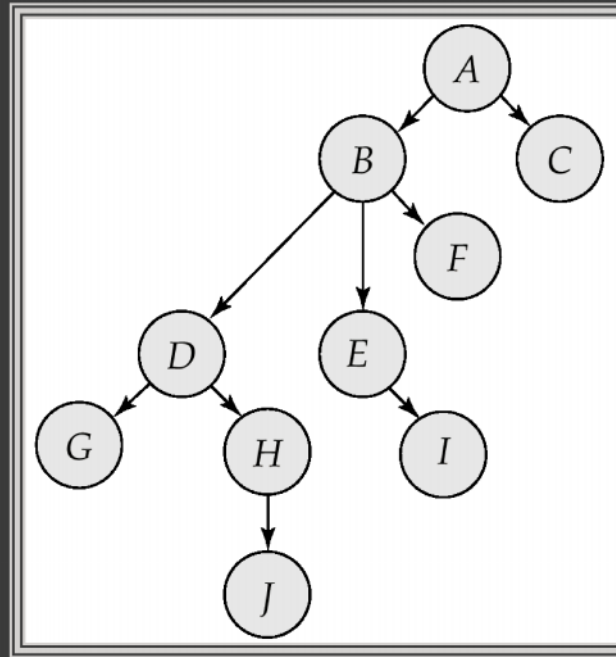


- Kotak hitam tandanya sedang mengunci, sedang yang putih menunggu permintaan
- Lock table juga mencatat jenis penguncian
- Permintaan baru ditambahkan diakhir antrian permintaan untuk item data, dan menjadi jaminan terhadap semua penguncian terakhir
- Permintaan Unlock akan menghapus lock, dan kemudian permintaan akan memeriksa apakah bisa dilakukan sekarang
- Jika transaksi batal, semua proses tunggu dihapus
 - lock manager akan menjaga daftar kejadian lock setiap transaksi secara efisien

Protokol berbasis Graph

- ⦿ Protokol berbasis Graph adalah alternatif dalam two-phase locking
- ⦿ Memberikan sebagian permintaan → pada himpunan $\mathbf{D} = \{d_1, d_2, \dots, d_h\}$ semua item data.
 - Jika $d_i \rightarrow d_j$ maka semua transaksi yang mengakses d_i dan d_j harus mengakses d_i lebih dahulu sebelum mengakses d_j .
 - Akibatnya himpunan \mathbf{D} dapat dipandang sebagai *database graph*.
- ⦿ *Tree-protocol* dalam graph protocol.

Tree Protocol



- Hanya mengizinkan exclusive lock.
- Penguncian pertama oleh T_i mungkin terhadap beberapa item data. Setelah itu, sebuah data Q dapat dikunci oleh T_i hanya jika parent dari Q saat ini di kunci oleh T_i .
- Item data mungkin di-unlocked beberapa kali.

Tree Protocol

- tree protocol menjamin conflict serializability dengan membebaskan dari deadlock.
- Unlocking terjadi lebih cepat diakhir tree-locking protocol dibanding two-phase locking protocol.
 - Waktu tunggu lebih pendek, dan meningkat dalam konkurensi
 - protocol bebas deadlock, tidak perlu rollback
 - Pembatalan transaksi dapat mengakibatkan penumpukan rollback.
- Bagaimanapun, dalam penguncian dengan protokol tree dapat terjadi, sebuah transaksi mengunci item data yang tidak diakses.
 - memperkuat locking, dan menambah waktu tunggu
 - bisa berkurang dalam konkurensi
- Penjadwalan yang tidak mungkin dibawah two-phase locking menjadi mungkin dibawah tree protocol.

Timestamp-Based Protocols

- setiap transaksi di tandai waktu kehadirannya dalam sistem. Jika transaksi yang lama T_i mempunyai time-stamp $TS(T_i)$, transaksi yang baru T_j diberi time-stamp $TS(T_j)$ dimana $TS(T_i) < TS(T_j)$.
- Skema ini menjamin serializability dengan memilih sebuah urutan diantara setiap pasangan transaksi.
- Untuk menerapkan skema ini, diterapkan dua nilai timestamp pada setiap item data Q :
 - **W-timestamp**(Q) yang menunjukkan nilai *timestamp* terbesar dari setiap transaksi yang berhasil menjalankan operasi **write**(Q).
 - **R-timestamp**(Q) yang menunjukkan nilai timestamp terbesar dari setiap transaksi yang berhasil menjalankan operasi **read**(Q).

Timestamp-Based Protocols (Cont.)

- ⦿ Timestamp akan terus diperbarui ketika ada perintah baru **read** dan **write** yang dieksekusi.
- ⦿ Untuk transaksi T_i yang menjalankan operasi **read(Q)**
 - Jika $TS(T_i) \leq \mathbf{W}\text{-timestamp}(Q)$, maka T_i perlu membaca kembali nilai Q yang ditulis. Karena itu, operasi **read** ini akan ditolak dan T_i akan di rolled back.
 - Jika $TS(T_i) \geq \mathbf{W}\text{-timestamp}(Q)$, maka operasi **read** dieksekusi, dan $\mathbf{R}\text{-timestamp}(Q)$ diisi dengan nilai terbesar diantara $\mathbf{R}\text{-timestamp}(Q)$ dan $TS(T_i)$.

Timestamp-Based Protocols

(Cont.)

• Untuk transaksi T_i yang menjalankan operasi **write**(Q).

- Jika $TS(T_i) < R\text{-timestamp}(Q)$, maka nilai Q yang baru yang dihasilkan T_i adalah nilai yang tidak akan dimanfaatkan lagi, dan sistem berasumsi bahwa nilai tersebut tidak pernah dihasilkan. Karena itu, operasi **write** ditolak dan transaksi T_i di- roll back.
- Jika $TS(T_i) < W\text{-timestamp}(Q)$, maka berarti T_i sedang berusaha melakukan penulisan nilai Q yang kadaluwarsa. Karena itu, operasi **write** akan ditolak dan T_i di- roll back.
- Kecuali itu, operasi **write** dieksekusi, dan $W\text{-timestamp}(Q)$ diberi nilai baru yang sama dengan $TS(T_i)$.

Contoh penggunaan Protocol

Sebagian jadwal item data dengan transaksi yang mempunyai timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
read(Y)	read(Y)	write(Y) write(Z)		read(X)
read(X)	read(X) abort	write(Z) abort		read(Z)
				write(Y) write(Z)

Correctness of Timestamp-Ordering Protocol

- Protokol timestamp-ordering menjamin conflict serializability jika prosesnya mengikuti urutan:



- Protokol ini menjamin konkurensi terbebas dari deadlock, karena tidak ada transaksi yang harus menunggu.

⦿ Permasalahan yang mungkin terjadi dengan timestamp-ordering protocol:

- Misalkan T_i dibatalkan, tapi T_j sudah membaca item data yang ditulis T_i
- Maka T_j harus dibatalkan; jika T_j untuk commit terlebih dahulu, maka jadwal tidak recoverable.
- Lebih jauhnya lagi, transaksi apapun yang telah membaca item data yang ditulis oleh T_j harus dibatalkan
- Hal ini dapat menyebabkan cascading rollback

⦿ Solusi :

- Semua transaksi harus dilakukan secara terstruktur
- Semua transaksi yang diproses membentuk aksi atomik; tidak ada transaksi yang boleh dieksekusi ketika transaksi lainnya sedang melakukan proses write
- Transaksi yang dibatalkan diulang dengan timestamp baru

Validation-Based Protocol

- Eksekusi dari transaksi T_i selesai dalam tiga tahap.
 1. **Read dan eksekusi**: Transaksi T_i melakukan operasi **write** hanya pada variabel lokal temporer tanpa melakukan perubahan ke basis data aktual
 2. **Validasi**: Transaksi T_i membentuk uji validasi untuk menentukan apakah transaksi tersebut dapat melakukan penyalinan / pengubahan ke basis data dari variabel lokal temporer yang nilainya diperoleh dari operasi write tanpa menyebabkan pelanggaran serializability.
 3. **Write** : Jika fase validasi transaksi T_i berhasil, maka perubahan sesungguhnya dilakukan ke basis data. Jika validasi tidak berhasil, maka T_i akan di-roll back.
- Semua fase dalam eksekusi transaksi konkuren dapat terjadi pada waktu bersamaan.
- Disebut juga **optimistic concurrency control**

Validation-Based Protocol

- Setiap transaksi T_i akan memiliki 3 timestamp
 - **Start**(T_i) : waktu dimana T_i memulainya eksekusinya
 - **Validation**(T_i): waktu dimana T_i selesai melakukan Fase pembacaan dan memulai fase validasi
 - **Finish**(T_i) : waktu dimana T_i menyelesaikan fase penulisan
- Urutan serializability ditentukan dengan teknik pengurutan timestamp dengan menggunakan nilai timestamp validation (T_i), oleh karena itu nilai $TS(T_i) = \text{Validation}(T_i)$.

Uji validasi untuk transaksi T_j

- Jika untuk semua transaksi T_i dengan $TS(T_i) < TS(T_j)$ salah satu dari dua kondisi berikut harus dapat dipenuhi :
 - **finish**(T_i) < **start**(T_j) , karena T_i menyelesaikan eksekusinya sebelum T_j dimulai
 - **start**(T_j) < **finish**(T_i) < **validation**(T_j) dan himpunan item data yang ditulis T_i tidak beririsan dengan himpunan item data yang dibaca oleh T_j .

kemudian validasi T_j dikatakan berhasil, jika tidak validasi gagal dan T_j di batalkan.

- *Justification*: Either first condition is satisfied, and there is no overlapped execution, or second condition is satisfied and
 1. operasi write oleh T_j jangan dilakukan sampai dengan operasi read dari T_i selesai.
 2. operasi write dari T_i jangan mempengaruhi operasi reads T_j jika T_j tidak melakukan operasi read terhadap operasi write yang dilakukan T_i .

Jadwal yang dihasilkan oleh Validasi

- Contoh skedul yang menggunakan validation

T_{14}	T_{15}
read(B)	read(B) <i>B:- B-50</i> read(A) <i>A:- A+50</i>
read(A) <i>(validate)</i> display (A+B)	<i>(validate)</i> write (B) write (A)

Penanganan Deadlock

- Ada dua transaksi sebagai berikut :

T_1 : write (X)
 write(Y)

T_2 : write(Y)
 write(X)

- Penjadwalan dengan deadlock

T_1	T_2
lock-X on X write (X) wait for lock-X on Y	lock-X on Y write (X) wait for lock-X on X

- ⦿ Sistem dikatakan deadlock bilamana ada lebih dari satu transaksi berada dalam keadaan saling tunggu untuk melakukan akses terhadap sebuah item data.
- ⦿ ***Pencegahan Deadlock dapat dilakukan dengan dua metode berikut :***
 - Transaksi harus mengunci semua item data sebelum memulai eksekusi.
 - Mengijinkan sistem memasuki kondisi deadlock dan kemudian berusaha untuk mengatasinya dengan memanfaatkan skema pendeteksian dan pemulihan deadlock.

Strategi pencegahan deadlock

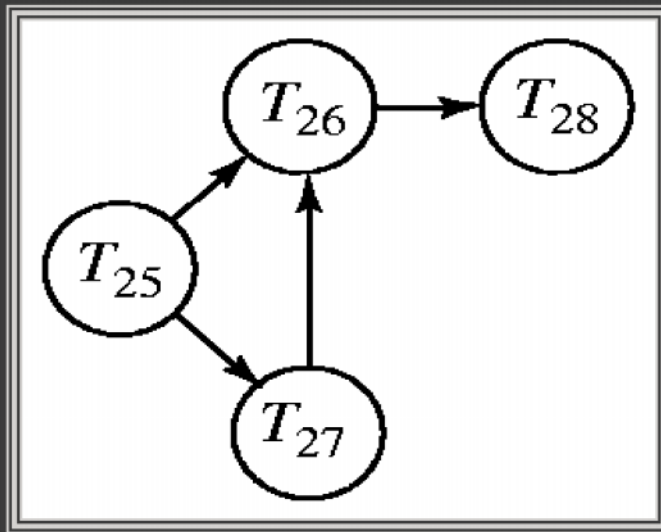
- ◉ Ada dua skema pendekatan dalam mencegah terjadinya deadlock yang menggunakan timestamp.
- ◉ **wait-die** — non-preemptive
 - Ketika transaksi T_i membutuhkan sebuah item data yang sedang dipegang oleh T_j , T_i dibolehkan menunggu hanya jika ia memiliki timestamp yang lebih kecil dari T_j (T_i lebih dahulu dari T_j). Jika tidak, T_i akan dibatalkan.
- ◉ **wound-wait** — preemptive
 - Merupakan lawan dari skema pertama. Ketika transaksi T_i membutuhkan item data yang sedang dipegang oleh T_j , T_i diperbolehkan menunggu jika ia memiliki timestamp yang lebih besar dari pada T_j (T_i datang belakangan). Jika tidak, T_j akan dibatalkan.

Strategi pencegahan deadlock

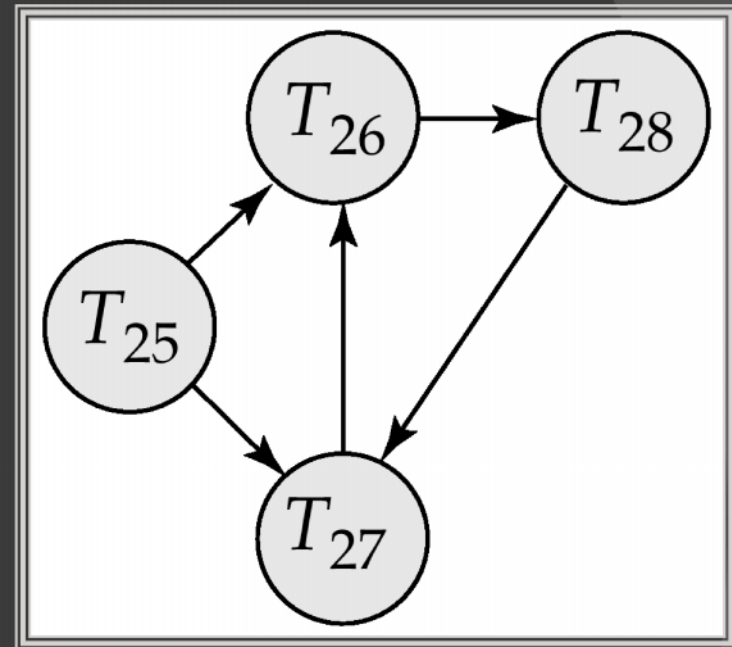
- ⦿ Dalam kedua skema *wait-die* dan *wound-wait* , transaksi yang di- roll back akan memulai dengan timestamp asal yang selalu meningkat, sehingga suatu saat transaksi yang di roll back akan memiliki nilai timestamp terkecil. Oleh karena itu tidak akan di roll back terus menerus.
- ⦿ Skema berbasis Timeout :
 - Sebuah transaksi yang membutuhkan penguncian akan menunggu selama batas waktu yang ditentukan.
 - Dengan demikian deadlock tidak akan terjadi
 - Sederhan dalam penerepannya , tetapi memungkinkan terjadinya stravasion, misalnya adanya transaksi yang memang lama mengeksekusinya.

Deteksi Deadlock

- Deadlock dapat dijelaskan dengan graph *wait-for*, yang terdiri atas pasangan $G = (V, E)$,
 - V mewakili sekumpulan simpul (semua transaksi dalam sistem)
 - E mewakili sekumpulan busur; setiap elemen dalam himpunan simpul merupakan pasangan $T_i \rightarrow T_j$.
- Jika $T_i \rightarrow T_j$ ada dalam E , maka ada busur berarah dari transaksi T_i ke transaksi T_j , yang menunjukkan bahwa transaksi T_i sedang menunggu transaksi T_j untuk melepaskan penguncian terhadap item data yang dibutuhkan.
- Ketika T_i membutuhkan item data yang sedang dipegang oleh T_j , maka busur $T_i \rightarrow T_j$ ditambahkan kedalam graph *wait-for*. Busur ini akan dihapuskan hanya jika ketika transaksi T_j telah melepaskan item data yang dipegangnya yang dibutuhkan oleh T_i .
- Ebuah deadlock akan terjadi jika dan hanya jika dalam graph *wait-for* terdapat siklus. Perlu mengelola graph ini secara periodik dengan algoritma untuk memeriksa ada tidaknya siklus dalam graph tersebut.



Graph Wait-for tanpa siklus



Graph Wait-for dengan siklus

Deadlock Recovery

⦿ Ketika deadlock terdeteksi :

- Jalankan proses roll back pada satu atau beberapa transaksi untuk lepas dari kondisi deadlock. Pilih transaksi dengan resiko minimum.
- Rollback – tentukan pula sejauhmana transaksi harus roll back
 - Total rollback: Batalkan transaksi dan muali dari awal.
 - Lebih efektif jika proses roll back dijalankan hanya sejauh diperlukan supaya terlepas dari kondisi deadlock.
- Starvation dapat terjadi jika sebuah transaksi selalu dijadikan korban yang akan dikenai jika ada proses deadlock. Caranya dengan melibatkan jumlah proses roll back yang dialami sebuah transaksi sebagai faktor biaya