Sistem Basis Data

MANAJEMEN TRANSAKSI

MANAJEMEN TRANSAKSI

- Konsep Transaksi
- State Transaksi
- Implementasi Atomik dan Durabilitas
- Eksekusi Konkuren
- Serializability
- Recoverability
- Implementasi Isolasi
- Definisi Transaksi di SQL
- Tes Serializability

Konsep Transaksi

- Transaksi adalah sebuah unit eksekusi dari program yang mengakses dan memungkinkan update berbagai macam tipe data.

- Biasanya suatu transaksi diinisialisasikan oleh program user yang ditulis dalam bahasa pemrograman atau manipulasi data tingkat tinggi (sebagai contoh, SQL, C/C++), yang dibatasi oleh statement (pemanggilan fungsi) dalam bentuk begin transaction dan end transaction.

- Transaksi terdiri dari semua operasi yang dieksekusi diantara begin transaction dan end transaction.

Konsep Transaksi (2)

- Pada saat eksekusi transaksi, database bisa saja menjadi tidak konsisten. Namun pada saat transaksi sampai pada level commited, maka databasenya harus konsisten.

- Dua hal utama yang mungkin akan dihadapi pada saat melakukan transaksi :
  - Terjadinya berbagai macam kegagalan, yang bisa disebabkan karena kegagalan hardware, system crash, dll
  - Eksekusi konkuren (secara bersama) yang melibatkan banyak transaksi

Konsep Transaksi (3)

- Untuk memastikan integritas data tetap terjaga dan transaksi dapat berjalan dengan baik, maka sistem database harus menjaga properti – properti yang terdapat di dalam transaksi.

- Properti – properti di dalam transaksi ini dikenal dengan istilah Properti ACID (Atomicity, Consistency, Isolation, Durability).

- Properti ACID memastikan perilaku yang dapat diprediksi dan menguatkan peran transaksi sebagai konsep all or nothing yang didesain untuk mengurangi manajemen load ketika ada banyak variabel.

Konsep Transaksi (4)

- Properti ACID :
- Atomicity. Transaksi dilakukan sekali dan sifatnya atomik, artinya merupakan satu kesatuan tunggal yang tidak dapat dipisah – laksanakan pekerjaannya sampai selesai atau tidak sama sekali
- Consistency. Jika basis data awalnya dalam keadaan konsisten maka pelaksanaan transaksi sendirinya juga harus meninggalkan basis data tetap dalam status konsisten
- Isolation. Isolasi memastikan bahwa secara bersamaan eksekusi transaksi terisolasi dari yang lain
- Durability. Begitu transaksi telah dilaksanakan (di-commit) maka perubahan yang dilakukan tidak akan hilang dan tetap terjaga (durable), sekalipun ada kegagalan sistem.

Konsep Transaksi (4)

- Transaksi mengakses data dengan operasi :
- read(X), mentransfer data item X dari database ke local buffer yang dimiliki oleh transaksi yang mengeksekusi operasi pembacaan (read).
- write(X), mentransfer data item X dari local buffer dari aksi transaksi yang mengeksekusi perintah penulisan kembali ke database (write)

Konsep Transaksi (5)

- Contoh implementasi transaksi, misalkan transaksi transfer uang sebesar $50 dari rekening A ke rekening B, maka transaksi tersebut dapat didefinisikan sebagai berikut :

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

Konsep Transaksi (6)

Berdasarkan contoh, ditinjau dari kebutuhan properti ACID-nya, maka :

- Kebutuhan Konsistensi (Consistency Requierements) :

Total jumlah rekening A + B harus tetap, tidak berubah setelah proses eksekusi transaksi.

- Kebutuhan Atomik (Atomicity Requeirements) :

Jika transaksi gagal diantara tahap ke-3 dan tahap ke-6, maka sistem harus memastikan bahwa perubahan yang terjadi tidak disimpan ke database, atau akan terjadi inkonsistensi data. Dengan kata lain, selesaikan transaksi atau tidak sama sekali.

Konsep Transaksi (7)

- Kebutuhan Durabilitas (Durability) :

Pada saat eksekusi transaksi selesai dilaksanakan, dan user yang melakukan transaksi sudah diberitahu bahwa transfer yang dilakukannya sukses, maka harus dipastikan bahwa tidak ada kesalahan sistem yang akan terjadi yang menyebabkan hilangnya data yang berkaitan dengan proses transfer tersebut .

- Kebutuhan Isolasi (Isolation) :

Jika diantara tahap ke-3 dan tahap ke-6 ada transaksi lain yang disisipkan, maka akan dapat menyebabkan inkonsistensi terhadap database (jumlah rekening A+B bisa jadi berkurang dari yang seharusnya). Untuk menghindari hal itu, maka transaksi bisa dieksekusi secara serial.

Tapi walaubagaimanapun, eksekusi banyak transaksi secara bersama – sama (konkuren) memiliki banyak keuntungan.

State Transaksi

- Supaya transaksi benar – benar sukses dipenuhi (successfully completion), maka sebuah transaksi harus berada di dalam salah satu state sebagai berikut :
  - Active
  - Partially committed
  - Failed
  - Aborted
  - Commited

- State tersebut dapat direpresentasikan dalam model transaksi abstrak yang sederhana

State Transaksi (2)

- Active

merupakan initial state, transaksi tetap berada pada state ini pada saat proses eksekusi

- Partially Committed

Setelah statement final telah dieksekusi

- Failed

Setelah ditelusuri bahwa eksekusi normal tidak dapat diproses kembali

- Aborted

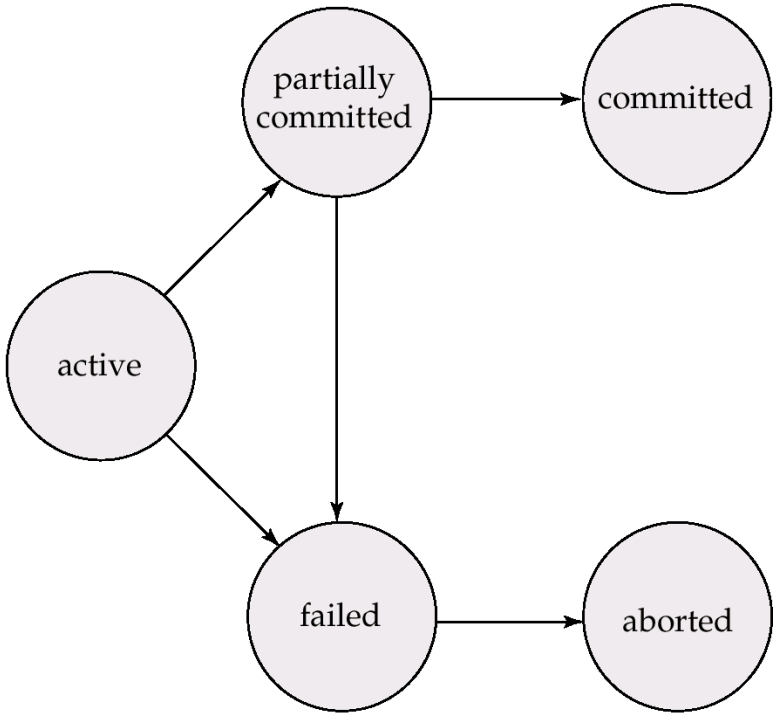
Setelah transaksi di-rolled back dan database direstore ke kondisi awal sebelum transaksi dimulai

- Committed

Setelah transaksi sukses dipenuhi

State Transaksi (3)

- Diagram state yang menggambarkan proses transaksi :



State Transaksi (4)

- Transaksi dimulai dalam keadaan state *active.* Pada saat menyelesaikan statement terakhirnya, transaksi masuk ke kondisi state *partially commited.*
- Dalam keadaan state tersebut, transaksi telah selesai dieksekusi, tapi masih mungkin untuk dibatalkan (*aborted*) hanya jika transaksi memasuki state *aborted*, karena output yang sesungguhnya masih berada di tempat penyimpanan sementara/main memory, dan karenanya kesalahan pada hardware dapat mempengaruhi kesuksesan dari penyelesaian transaksi
- Sistem Basis Data kemudian menuliskan informasi yang dibutuhkan (*write*) ke dalam disk, bahwa perubahan yang dilakukan oleh transaksi dapat dibuat kembali pada saat sistem restart jika terjadi kegagalan pada sistem.
- Pada saat informasi terakhir dituliskan, maka transaksi masuk ke kondisi state *commited*.

State Transaksi (5)

- Sebuah transaksi akan memasuki kondisi state *failed* jika setelah sistem melakukan pemeriksaan, transaksi tidak dapat lagi diproses dengan eksekusi normal (misal karena kerusakan hardware atau kesalahan logika).
- Jika berada di dalam kondisi tersebut, maka transaksi harus di *rolled back,* dan kemudian selanjutnya memasuki kondisi state *aborted* (pembatalan transaksi).
- Pada titik ini, sistem memiliki dua opsi, ulangi transaksi (***restart*** *the transaction*) dan hapus transaksi (***kill*** *the transaction*)

State Transaksi (6)

- Opsi pada state *aborted* :
  - Sistem dapat mengulang transaksi (**restart** the transaction), tapi hanya jika transaksi dibatalkan yang bisa terjadi karena adanya kerusakan software atau hardware yang tidak diciptakan melalui logika internal dari transaksinya.
  - Sistem dapat menghapus transaksi (**kill** the transaction). Biasanya terjadi karena adanya kesalah logika internal yang dapat diperbaiki hanya dengan menulis kembali program aplikasinya, atau karena inputan yang tidak baik, atau karena data yang diinginkan tidak ada di database.

Implementasi Atomik (Atomicity) dan Durabilitas (Durability)

- Komponen manajemen recovery dari sistem basis data dapat mendukung atomisitas dan durabilitas dengan berbagai macam skema.
- Salah satu skema yang dikenal adalah shadow copy,
- *Shadow copy* adalah salah satu skema yang digunakan untuk mendukung atomisitas dan durabilitas pada transaksi dengan membuat salinan / copy dari database yang ada.

Implementasi Atomik (Atomicity) dan Durabilitas (Durability) – (2)

Skema *shadow-database :*

- Mengasumsikan bahwa hanya satu transaksi yang aktif dalam waktu bersamaan (simpel tapi tidak efisien).
- Mengasumsikan bahwa database secara sederhana merupakan sebuah file di dalam disk.
- Sebuah pointer yang bernama *db-pointer* digunakan di dalam disk yang selalu mengarah ke salinan konsisten database tersebut
- Sebelum transaksi mengupdate database, salinan untuk database tersebut dibuat terlebih dahulu sepenuhnya

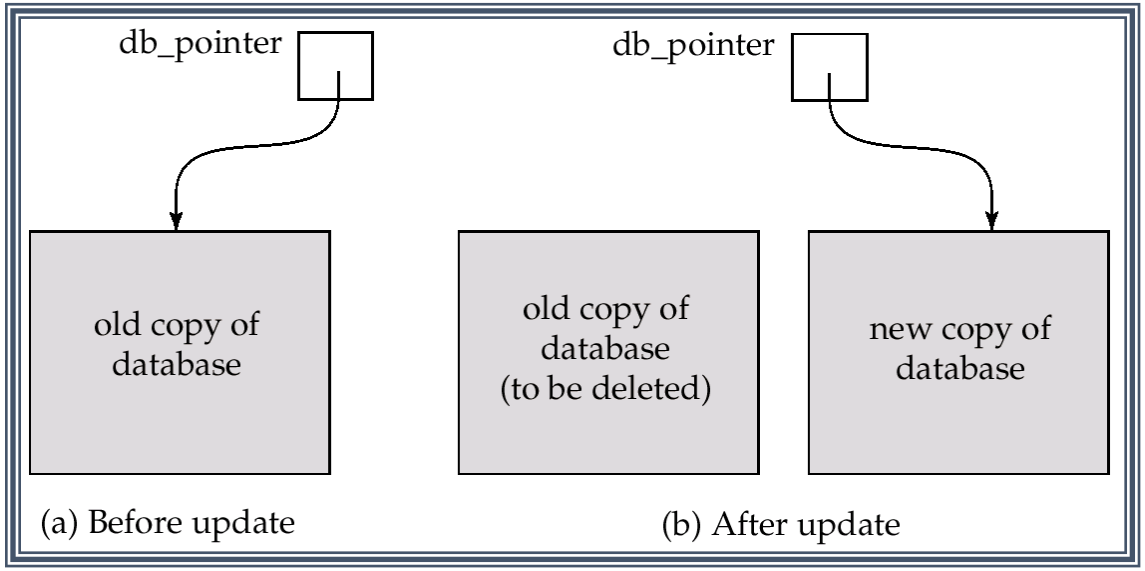
Implementasi Atomik (Atomicity) dan Durabilitas (Durability) – (3)

Skema *shadow-database :*

- Semua update dilakukan di salinan database yang baru, dan *db_pointer* akan mengarah ke salinan baru tersebut dengan catatan semua transaksi telah mencapai state *partial committed* dan semua update pagesnya telah di-*flush* ke dalam disk.
- Salinan database tersebut kemudian menjadi database utama, dan database yang lama dapat dihapus.
- Jika transaksi gagal, maka *db_pointer* akan kembali mengarah ke database lama, dan salinan dari database yang telah dibuat tersebut dapat dihapus.

Implementasi Atomik (Atomicity) dan Durabilitas (Durability) – (4)

Skema *shadow-database :*



- Asumsi transaksi tidak gagal
- Berguna untuk teks editor, tapi tidak efisien untuk database berukuran besar

Eksekusi Konkurensi

- Pada eksekusi konkurensi, banyak transaksi dimungkinkan untuk diproses secara bersama – sama dalam suatu sistem.
- Memperbolehkan banyak transaksi untuk mengupdate data secara konkuren akan menyebabkan beberapa komplikasi dengan konsistensi data.
- Untuk memastikan bahwa konsistensi tetap terjaga dengan baik pada eksekusi konkuren, membutuhkan usaha yang lebih kuat. Akan lebih mudah jika transaksi berjalan secara serial.

Eksekusi Konkurensi (2)

- Keuntungan konkurensi :
  - Meningkatkan utilitas disk dan prosesor
  - Mengurangi rata – rata waktu respon transaksi
- Dalam konkurensi dikenal dengan konsep penjadwalan *(schedule)* yang akan membantu mengidentifikasi eksekusi agar konsistensi datanya tetap terjaga.
- Sistem Basis Data harus mengontrol interaksi diantara transaksi konkuren supaya transaksi – transaksi tersebut tidak menghancurkan konsistensi basis datanya (data menjadi tidak konsisten).

Eksekusi Konkurensi - Schedule

- Penjadwalan / Schedule merupakan urutan yang mengindikasikan urutan kronologis instruksi yang mana dari transaksi konkuren yang akan dieksekusi.
- Sebuah jadwal merupakan bagian dari suatu transaksi yang harus terdiri dari semua instruksi yang ada di dalamnya.
- Sebuah jadwal harus menjaga urutan instruksi yang muncul di setiap transaksi.

Eksekusi Konkurensi – Schedule (2)

- Misal T_1 adalah transaksi transfer $50 dari A ke B, dan T_2 adalah transfer 10% dari jumlah rekening A ke B. Penjadwalan berikut adalah pada transaksi serial, dimana T_1 dulu yang diselesaikan baru diikuti oleh T_2

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Penjadwalan 1 di text book

Eksekusi Konkurensi – Schedule (3)

- Contoh penjadwalan selanjutnya, dengan transaksi yang sama, tetapi dalam bentuk transaksi konkurensi, tapi ekivalen dengan penjadwalan sebelumnya.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Penjadwalan 3 pada Text book

Eksekusi Konkurensi – Schedule (3)

- Pada contoh penjadwalan yang pertama dan yang kedua, jumlah rekening A + B sebelum dan sesudah transaksi sama, tapi tidak dengan contoh penjawalan konkurensi berikut :

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Penjadwalan 4 pada text book

Serializability

- Sistem basis data harus dapat mengontrol eksekusi konkurensi dari suatu transaksi untuk memastikan database tetap terjaga konsistensinya.
- Asumsi dasar, bahwa setiap transaksi harus tetap menjaga konsistensi database.
- Eksekusi secara serial pada suatu transaksi dapat menjaga database tetap konsisten.
- Sebuah jadwal (yang mungkin konkuren), serializable jika setara dengan penjadwalan secara serial. Pada bagian ini ada dua bentuk ekivalensi penjadwalan *(Schedule Equivalence) : Conflict* dan *View Serializability.*
- Pada serializability kita abaikan operasi selain dari instruksi **read** dan **write**, serta diasumsikan bahwa transaksi dapat melakukan perhitungan terhadap data di dalam buffer lokal diantara instruksi **read** dan **write**.

Conflict Serializability

- Instruksi I_i dan I_j masing – masing dari transaksi T_i dan T_j , konflik jika dan hanya jika ada beberapa item data yang sama (Q) diakses secara bersama – sama baik oleh I_i dan I_j , serta setidaknya salah satu dari instruksi melakukan operasi *write (Q)*.
  1. Jika I_i = **read(**Q**)** dan I_j = **read(**Q), maka I_i dan I_j tidak konflik
  2. Jika I_i = **read(**Q**)** dan I_j = **write(**Q), maka I_i dan I_j konflik
  3. Jika I_i = **write(**Q**)** dan I_j = **read(**Q), maka I_i dan I_j konflik
  4. Jika I_i = **write(**Q**)** dan I_j = **write(**Q), maka I_i dan I_j konflik

- Secara intuitif, konflik diantara I_i dan I_j memaksakan perintah logika temporal diantara keduanya.

- Jika I_i dan I_j merupakan instruksi dari transaksi berbeda dan tidak konflik, maka urutan instruksi I_i dan I_j dapat ditukar sehingga menghasilkan jadwal baru dengan hasil yang tetap sama.

Conflict Serializability (2)

- Jika jadwal S dapat ditransformasikan menjadi jadwalan S'oleh serangkaian pertukaran instruksi yang non-konflik, maka bisa dikatakan S dan S' adalah **conflict equivalent**.
- Bisa dikatakan bahwa jadwal *S* adalah **conflict serializable** jika conflict equivalent terhadap penjadwalan serial.
- Contoh penjadwalan yang tidak **conflict serializable**

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Conflict Serializability (3)

- Penjadwalan 3 di bawah bisa ditransformasikan ke penjadwalan 1, penjadwalan serial dimana T_2 diikuti T_1 , dengan serangkaian penukaran instruksi yang tidak konflik. Oleh karena itu penjadwalan di bawah ini **conflict serializability**

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

View Serializability

- Misalkan *S* dan *S'* adalah dua jadwal dengan serangkaian transaksi yang sama. *S* dan *S'* adalah **view equivalent**.

- *S* dan *S'* adalah **view equivalent** jika memenuhi kondisi sebagai berikut :
  1. Untuk masing – masing data item *Q* , jika transaksi T_i membaca inisialisasi nilai pada *Q* di dalam jadwal *S*, maka transaksi T_i yang ada di jadwal *S'* juga harus membaca inisialisasi nilai pada *Q*.
  2. Untuk masing – masing data item *Q,* jika transaksi T_i mengeksekusi *read(Q)* di dalam jadwal *S,* dan jika nilai yang dihasilkan oleh operasi *write(Q)* dieksekusi oleh transaksi T_j , maka operasi *read(Q)* pada transaksi T_i yang harus ada di jadwal *S'* juga membaca nilai Q yang dihasilkan oleh operasi *write(Q)* yang sama pada transaksi T_j
  3. Untuk masing – masing data item *Q,* transaksi yang sampai ke tahap akhir operasi *write(Q)* di dalam jadwal *S*

- Seperti yang terlihat, view serializability juga dilihat berdasarkan pada operasi read dan write saja

View Serializability (2)

- Sebuah jadwal S adalah **view seralizable ,** setara dengan penjadwalan serial.
- Setiap penjadwalan **conflict serializable** juga **view serializable**
- Penjadwalan di bawah ini (penjadwalan 9 di text book) adalah penjadwalan yang view serializable tapi tidak conflict serializable

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

View Serializability (3)

- Penjadwalan di bawah (penjadwalan 9 di text book) menghasilkan outcome yang sama sebagai jadwal serial (T_1, T_2)

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Recoverability

- **Recoverable Schedule** – Jika transaksi T_j membaca sebuah item data yang sebelumnya ditulis oleh transaksi T_i , maka operasi commit T_i muncul sebelum operasi commit T_j .
- Jadwal berikut (jadwal 11 di text book) tidak recoverable jika T_9 commit segera setelah read

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- Jika T_8 dibatalkan, maka T_9 bisa jadi akan membaca state database inkonsisten. Maka dengan demikian database harus memastikan bahwa jadwal dapat dipulihkan jika terjadi sesuatu hal.

Recoverability (2)

- **Cascading Rollback –** kesalahan pada satu transaksi yang yang dapat berpengaruh pada serangkaian transaksi lainnya sehingga keseluruhan transaksi akan di-rollback.

- Misalkan pada penjadwalan berikut dimana belum ada transaksi yang di-commit, sehingga jika terjadi masalah bisa dipulihkan

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- Jika T_{10} transaksinya fail, maka T_{11} dan T_{12} juga harus di rollback. Jika tidak demikian, maka akan menyebabkan banyaknya pekerjaan yang tidak akan terselesaikan.

Recoverability (3)

- **Cascadeless Schedule** – cascading rollback tidak dapat terjadi, untuk setiap pasang transaksi T_i dan T_j dimana T_j membaca sebuah item data yang sebelumnya ditulis oleh T_i dan operasi commit pada T_i muncul sebelum operasi read pada T_j.

- Setiap cascadeless schedule juga dapat dipulihkan (recoverable);

Implementasi Isolasi

- Jadwal harus conflict atau view serializable, dan recoverable, demi terjaganya konsistensi database dan juga sebaiknya cascadeless.
- Sebuah kebijakan dimana hanya satu transaksi yang dapat dieksekusi dalam satu waktu yang menghasilkan penjadwalan serial, tapi minim konkurensi.
- Beberapa skema hanya mengijinkan jadwal conflict-serializable untuk di-generate, dimana yang skema lainnya ada yang memungkinkan jadwal view-serializable yang tidak conflict-serializable.

Definisi Transaksi di SQL

- DML (Data Manipulation Language) harus menyertakan sebuah konstruksi untuk menspesifikasikan serangkaian aksi yang meliputi suatu transaksi.
- Di dalam SQL, transaksi dimulai secara implisit.
- Transaksi di SQL diakhiri oleh salah satu dari statement sebagai berikut :
  - **commit work -** commit transaksi yang sedang berlangsung dan memulai transaksi yang baru
  - **rollback work –** pembatalan terhadap transaksi yang sedang berlangsung

Level Konsistensi di SQL 92

- **Serializable –** default
- **Repeatable read –** hanya record yang dicommit yang dibaca, pembacaan secara berulang untuk record yang sama harus dapat menghasilkan nilai yang sama. Bagaimanapun, transaksi mungkin tidak serializable - bisa menemukan beberapa record yang ditambahkan oleh suatu transaksi tapi tidak menemukan record lainnya.
- **Read commited –** hanya record yang dicommit yang akan dibaca , tetap commit untuk pembacaan record secara berulang yang menghasilkan nilai yang berbeda.
- **Read uncommited –** record yang tidak dicommit dibaca

Testing Serializability

- Pada saat mendesain skema kontrol konkurensi, kita harus tunjukan bahwa jadwal yang dibuat oleh skema tersebut adalah serializable.
- Terdapat metode simpel dan efisien untuk menentukan conflict serializability dari suatu jadwal.
- Misalkan sebuah jadwal S. Kita dapat membuat suatu grafik langsung yang diberi nama grafik preseden (**presedence graph**).

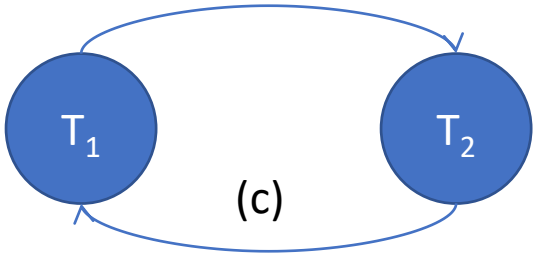
Testing Serializability (2)

- Grafik preseden terdiri dari sepasang G = (V,E), dimana V adalah serangkaian simpul dan E adalah serangkaian tepian / busur.
- Serangkaian simpul terdiri dari semua transaksi yang berperan serta di dalam penjadwalan. Serangkaian tepian / busur terdiri dari semua bentuk T_i -> T_j untuk masing – masing dari ketiga kondisi berikut :
  1. T_i eksekusi write(Q) sebelum T_j eksekusi read(Q)
  2. T_i eksekusi read(Q) sebelum T_j eksekusi write(Q)
  3. T_i eksekusi write(Q) sebelum T_j eksekusi write(Q)
- Jika bentuk T_i -> T_j ada di dalam grafik preseden, maka di setiap jadwal S' serial yang ekivalen ke jadwal S, T_i harus muncul sebelum T_j

Testing Serializability (3)

- Sebagai contoh, grafik preseden untuk penjadwalan 1 digambarkan di (a), terdiri dari bentuk dasar T_1 -> T_2 , dimana semua instruksi T_1 dieksekusi sebelum instruksi pertama pada T_2 dieksekusi. Begitu juga sebaliknya untuk contoh yang digambarkan di (b).

Testing Serializability (3)



- Grafik preseden untuk jadwal 4, terdiri dari bentuk T_1 -> T_2 , karena T_1 mengeksekusi read(A) sebelum T_2 mengeksekusi write(A). Grafik ini jg terdiri dari bentuk T_2 -> T_1 , karena T_2 mengeksekusi read(B) sebelum T_1 eksekusi write(B).

- Jika grafik preseden untuk S membentuk suatu lingkaran, maka jadwal S tidak conflict serializable. Jika sebaliknya, maka jadwal S conflict serializable