

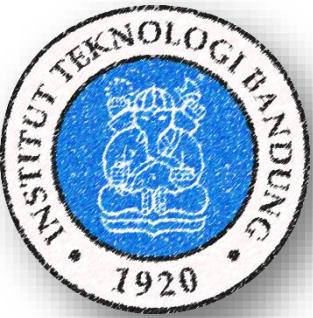
# IF3140 Manajemen Basis Data

## Concurrency Control

Slide diambil dari: Konsep Sistem Basis Data, 6<sup>th</sup> edisi - Bab 15 © Silberschatz, Korth,  
Sudarshan



# Tujuan

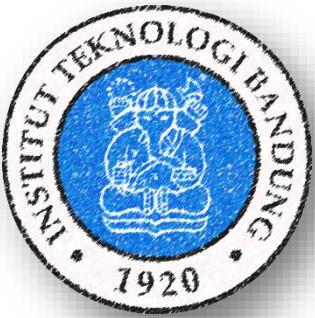


- Mahasiswa mampu:
  - Jelaskan pengaruh tingkat isolasi yang berbeda pada mekanisme kontrol konkurensi
  - Pilih tingkat isolasi yang tepat untuk mengimplementasikan protokol transaksi yang ditentukan



# Kandungan

- Protokol Berbasis Kunci
- Protokol Berbasis Stempel Waktu
- Protokol Berbasis Validasi
- Perincian Ganda
- Skema Multiversi
- Sisipkan dan Hapus Operasi
- Konkurensi dalam Struktur Indeks



# Protokol Berbasis Kunci

- Kunci adalah mekanisme untuk mengontrol akses bersamaan ke item data
- Item data dapat dikunci dalam dua mode:
  1. *eksklusif (X) mode*. Item data bisa dibaca sekaligus tertulis. X-lock diminta menggunakan **kunci-X** petunjuk.
  2. *bersama (S) mode*. Item data hanya bisa dibaca. S-lock adalah diminta menggunakan **kunci-S** petunjuk.
- Permintaan kunci dibuat untuk manajer kontrol konkurensi. Transaksi dapat dilanjutkan hanya setelah permintaan dikabulkan.



# Protokol Berbasis Kunci (Lanjutan)

- **Matriks kompatibilitas kunci**

	S	X
S	true	false
X	false	false

- Suatu transaksi dapat diberikan kunci pada suatu item jika kunci yang diminta kompatibel dengan kunci yang sudah dipegang pada item tersebut oleh transaksi lain
- Sejumlah transaksi dapat menahan kunci bersama pada suatu item,
  - tetapi jika ada transaksi yang memiliki eksklusif pada item tersebut, tidak ada transaksi lain yang dapat mengunci item tersebut.
- Jika kunci tidak dapat diberikan, transaksi yang meminta dibuat untuk menunggu sampai semua kunci yang tidak kompatibel yang dipegang oleh transaksi lain telah dilepaskan. Kunci tersebut kemudian diberikan.



# Lock Based Protocols (Cont.)

- Contoh transaksi yang melakukan penguncian:

*T<sub>2</sub>: kunci-S ( SEBUAH);*

*Baca ( SEBUAH);*

*membuka kunci( SEBUAH);*

*kunci-S ( B);*

*Baca ( B);*

*membuka kunci( B);*

*tampilan ( A + B)*

- Mengunci seperti di atas tidak cukup untuk menjamin serialisasi - jika *SEBUAH* dan *B* dapatkan pembaruan di antara waktu membaca *SEBUAH* dan *B*, jumlah yang ditampilkan akan salah.
- SEBUAH **protokol penguncian** adalah seperangkat aturan yang diikuti oleh semua transaksi saat meminta dan melepaskan kunci. Protokol penguncian membatasi rangkaian kemungkinan jadwal.



# Kesalahan Protokol Berbasis Kunci

- Pertimbangkan jadwal parsial

$T_3$	$T_4$
lock-x ( $B$ ) read ( $B$ ) $B := B - 50$ write ( $B$ )	lock-s ( $A$ ) read ( $A$ ) lock-s ( $B$ )
lock-x ( $A$ )	

- Tidak keduanya  $T_3$  maupun  $T_4$  dapat membuat kemajuan - mengeksekusi **kunci-S (  $B$  )** menyebab  $T_4$  menunggu untuk  $T_3$  untuk membuka kunci  $B$ , saat menjalankan **kunci-X ( SEBUAH )** menyebab  $T_3$  menunggu untuk  $T_4$  untuk membuka kunci *SEBUAH*.
- Situasi seperti itu disebut a **jalan buntu** .
  - Untuk menangani kebuntuan salah satu  $T_3$  atau  $T_4$  harus digulung kembali dan kuncinya dilepaskan.



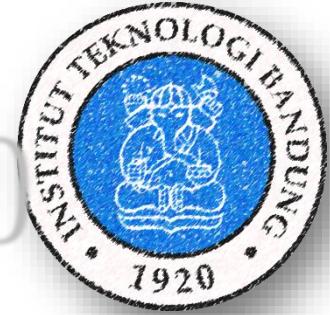
## Kesalahan Protokol Berbasis Kunci (Lanjutan.)

- Potensi kebuntuan ada di sebagian besar protokol penguncian. Kebuntuan adalah kejahatan yang diperlukan.
- **Kelaparan** juga dimungkinkan jika manajer kontrol konkurensi dirancang dengan buruk. Sebagai contoh:
  - Sebuah transaksi mungkin menunggu X-lock pada suatu item, sementara urutan transaksi lainnya meminta dan diberikan S-lock pada item yang sama.
  - Transaksi yang sama berulang kali dibatalkan karena kebuntuan.
- Manajer kontrol konkurensi dapat dirancang untuk mencegah kelaparan.



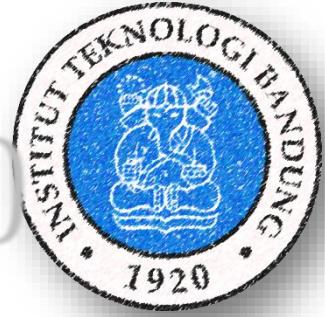
# Protokol Penguncian Dua Fase

- Ini adalah protokol yang memastikan jadwal konflik-serial.
- Fase 1: Fase Tumbuh
  - transaksi dapat memperoleh kunci
  - transaksi mungkin tidak membuka kunci
- Fase 2: Fase Penyusutan
  - transaksi dapat membuka kunci
  - transaksi mungkin tidak mendapatkan kunci
- Protokol menjamin serialisasi. Dapat dibuktikan bahwa transaksi dapat diserialkan sesuai dengan urutannya **poin kunci** ( yaitu titik di mana transaksi memperoleh kunci terakhirnya).



# Protokol Penguncian Dua Fase (Lanjutan)

- Penguncian dua fase *tidak* memastikan kebebasan dari kebuntuan
- Roll-back bertingkat dimungkinkan dengan penguncian dua fase. Untuk menghindarinya, ikuti protokol yang dimodifikasi yang disebut **penguncian dua fase yang ketat**. Di sini, transaksi harus menahan semua kunci eksklusifnya hingga berhasil / dibatalkan.
- **Penguncian dua fase yang ketat** bahkan lebih ketat: di sini *semua* kunci ditahan sampai melakukan / membatalkan. Dalam protokol ini, transaksi dapat diserialkan dalam urutan yang mereka komit.



## Protokol Penguncian Dua Fase (Lanjutan)

- Mungkin ada konflik jadwal serial yang tidak dapat diperoleh jika penguncian dua fase digunakan.
- Namun, dengan tidak adanya informasi tambahan (misalnya, memesan akses ke data), penguncian dua fase diperlukan untuk kemampuan bersambung konflik dalam pengertian berikut:

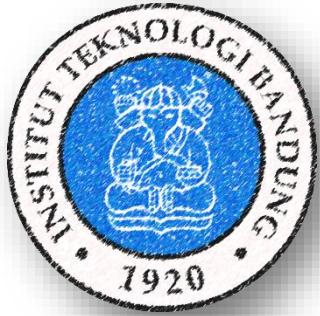
Diberikan transaksi  $T_{saya}$  yang tidak mengikuti penguncian dua fase, kita dapat menemukan transaksi  $T_j$  yang menggunakan penguncian dua fase, dan jadwal untuk  $T_{saya}$  dan  $T_j$  itu bukan konflik serializable.



# Kunci Konversi

- Penguncian dua fase dengan konversi kunci:
  - Fase pertama:
    - bisa mendapatkan kunci-S pada item
    - bisa mendapatkan kunci-X pada item
    - dapat mengubah kunci-S menjadi kunci-X (peningkatan)
  - Tahap Kedua:
    - bisa melepaskan kunci-S
    - dapat melepaskan kunci-X
    - dapat mengubah kunci-X menjadi kunci-S (penurunan versi)
- Protokol ini menjamin serialisasi. Namun tetap mengandalkan programmer untuk memasukkan berbagai instruksi penguncian.

# Akuisisi Otomatis Kunci



- Sebuah transaksi  $T_{saya}$  mengeluarkan instruksi baca / tulis standar, tanpa panggilan penguncian eksplisit.
- Operasi **Baca(  $D$  )** diproses sebagai:

jika  $T_{saya}$  memiliki kunci  $D$

kemudian

Baca(  $D$  )

lain dimulai

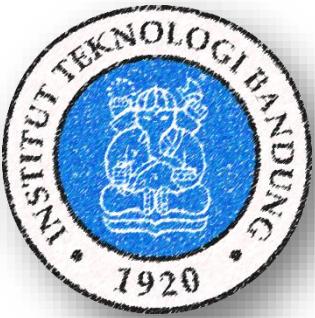
jika perlu tunggu sampai tidak ada transaksi lain

yang memiliki a **kunci-X** di  $D$

hibah  $T_{saya}$  sebuah **kunci-S** di  $D$ ;

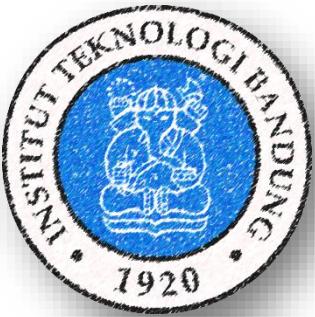
Baca(  $D$  )

akhir



# Akuisisi Otomatis Kunci (Lanjutan.)

- **menulis(  $D$  )** diproses sebagai:
  - jika  $T_{saya}$  mempunyai sebuah **kunci-X** di  $D$ 
    - kemudian**
      - menulis(  $D$  )
    - lain dimulai**
      - jika perlu tunggu sampai tidak ada trans lain. memiliki kunci apa pun  $D$ ,
      - jika  $T_{saya}$  mempunyai sebuah **kunci-S** di  $D$ 
        - kemudian**
          - meningkatkan mengunci  $D$  untuk **lock-X**
        - lainnya**
          - hibah  $T_{saya}$  sebuah **kunci-X** di  $D$
          - menulis(  $D$  )
  - Semua kunci dilepaskan setelah komit atau batal



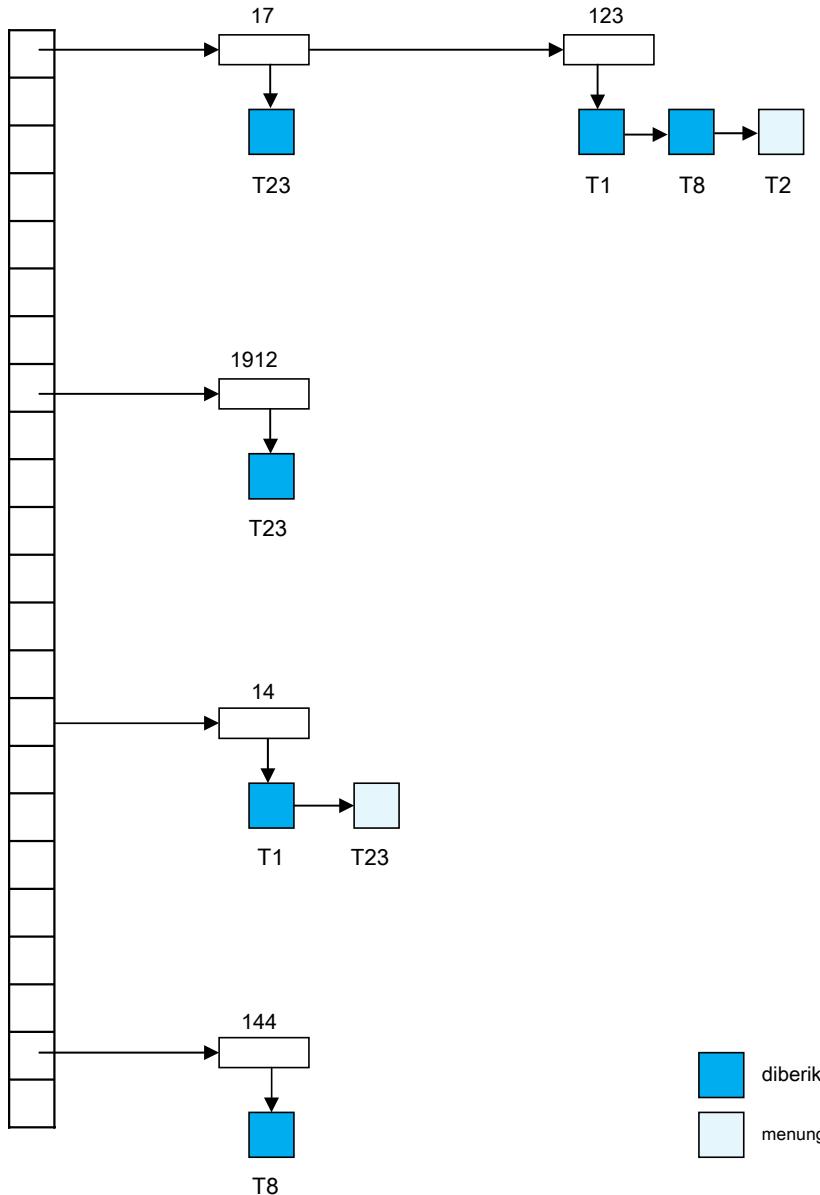
# Penerapan Locking

- SEBUAH **manajer kunci** dapat diimplementasikan sebagai proses terpisah di mana transaksi mengirim permintaan kunci dan buka kunci
- Manajer kunci membalas permintaan kunci dengan mengirimkan pesan pemberian kunci (atau pesan yang meminta transaksi untuk memutar kembali, jika terjadi kebuntuan)
- Transaksi yang meminta menunggu sampai permintaannya dijawab
- Manajer kunci memelihara struktur data yang disebut **meja kunci** untuk merekam kunci yang diberikan dan permintaan yang menunggu keputusan
- Tabel kunci biasanya diimplementasikan sebagai tabel hash dalam memori yang diindeks pada nama item data yang dikunci



# Meja Kunci

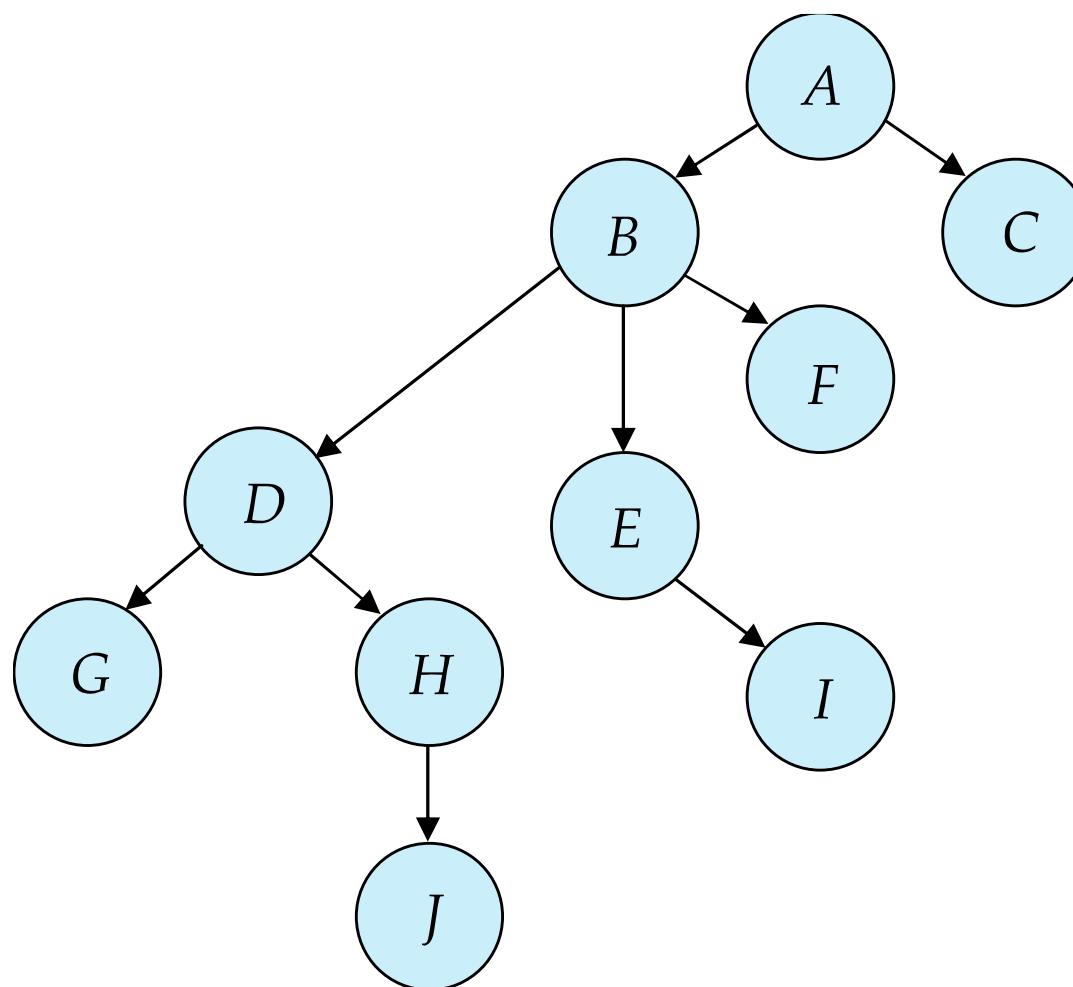
- Persegi panjang hitam menunjukkan kunci yang diberikan, sedangkan persegi berwarna putih menunjukkan permintaan menunggu
  - Tabel kunci juga mencatat jenis kunci yang diberikan atau diminta
  - Permintaan baru ditambahkan ke akhir antrian permintaan untuk item data, dan diberikan jika itu kompatibel dengan semua kunci sebelumnya
  - Permintaan buka kunci mengakibatkan permintaan dihapus, dan permintaan selanjutnya diperiksa untuk melihat apakah sekarang dapat diberikan
  - Jika transaksi dibatalkan, semua permintaan transaksi yang menunggu atau dikabulkan akan dihapus
    - manajer kunci dapat menyimpan daftar kunci yang dipegang oleh setiap transaksi, untuk mengimplementasikannya secara efisien





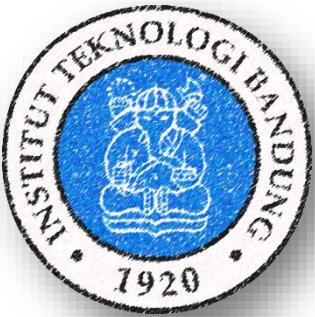
# Protokol Berbasis Grafik

- Protokol berbasis grafik adalah alternatif dari penguncian dua fase
- Menerapkan pemesanan parsial → di lokasi syuting  $D = \{ d_1, d_2, \dots, d_h \}$  dari semua item data.
  - Jika  $d_{saya} \rightarrow d_j$  lalu setiap transaksi mengakses keduanya  $d_{saya}$  dan  $d_j$  harus mengakses  $d_{saya}$  sebelum mengakses  $d_j$ .
  - Menyiratkan bahwa set  $D$  sekarang dapat dilihat sebagai grafik asiklik terarah, yang disebut a *grafik database*.
  - Itu *protokol pohon* adalah jenis protokol grafik yang sederhana.



## Protokol Pohon

- Hanya kunci eksklusif yang diperbolehkan.
- Kunci pertama oleh  $T_{saya}$  mungkin ada pada item data apa pun. Selanjutnya, sebuah data  $Q$  dapat dikunci oleh  $T_{saya}$  hanya jika orang tua  $Q$  saat ini dikunci oleh  $T_{saya}$ .
- Item data dapat dibuka setiap saat.
- Item data yang telah dikunci dan dibuka oleh  $T_{saya}$  tidak dapat dikembalikan lagi oleh  $T_{saya}$



# Protokol Berbasis Grafik (Lanjutan)

- Protokol pohon memastikan serialisasi konflik serta kebebasan dari kebuntuan.
- Buka kunci mungkin terjadi lebih awal dalam protokol penguncian pohon daripada di protokol penguncian dua fase.
  - waktu tunggu yang lebih singkat, dan peningkatan konkurensi
  - protokol bebas dari kebuntuan, tidak diperlukan rollback
- Kekurangan
  - Protokol tidak menjamin pemulihan atau kebebasan bertingkat
    - Perlu memperkenalkan dependensi commit untuk memastikan pemulihan
  - Transaksi mungkin harus mengunci item data yang tidak mereka akses.
    - peningkatan penguncian overhead, dan waktu tunggu tambahan
    - potensi penurunan konkurensi
- Jadwal yang tidak memungkinkan dalam penguncian dua fase dimungkinkan dalam protokol pohon, dan sebaliknya.



# Deadlock Handling

- Pertimbangkan dua transaksi berikut:

$T_1$ : menulis (  $X$  )

$T_2$ : menulis(  $Y$  )

menulis(  $Y$  )

menulis(  $X$  )

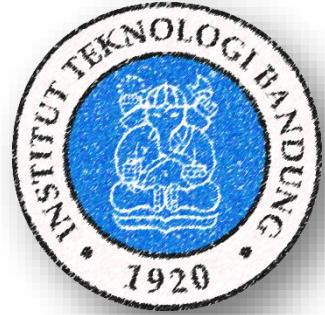
- Jadwalkan dengan kebuntuan

$T_1$	$T_2$
<b>lock-X on A</b> write (A)	<b>lock-X on B</b> write (B) wait for <b>lock-X on A</b>
wait for <b>lock-X on B</b>	

# Deadlock Handling

- Sistem menemui jalan buntu jika ada satu set transaksi sehingga setiap transaksi di set menunggu transaksi lain di set.
- **Pencegahan kebuntuan** protokol memastikan bahwa sistem akan melakukannya *tidak pernah* masuk ke dalam keadaan kebuntuan. Beberapa strategi pencegahan:
  - Mengharuskan setiap transaksi mengunci semua item datanya sebelum memulai eksekusi (pra-deklarasi).
  - Menerapkan pemesanan parsial dari semua item data dan mensyaratkan bahwa transaksi dapat mengunci item data hanya dalam urutan yang ditentukan oleh urutan parsial (protokol berbasis grafik).





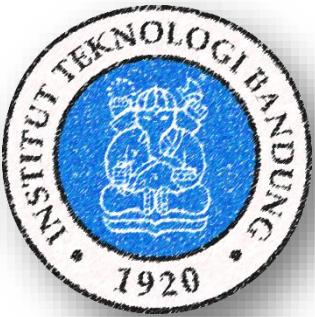
# Lebih Banyak Strategi Pencegahan Kebuntuan

- Skema berikut menggunakan stempel waktu transaksi untuk pencegahan kebuntuan saja.
- **tunggu-mati** skema - non-preemptive
  - transaksi yang lebih lama mungkin menunggu yang lebih muda untuk merilis item data. Transaksi yang lebih muda tidak pernah menunggu yang lebih tua; mereka digulung kembali.
  - transaksi bisa mati beberapa kali sebelum memperoleh item data yang dibutuhkan
- **tunggu luka** skema - preemptive
  - transaksi lama *luka* (memaksa rollback) transaksi yang lebih muda alih-alih menunggu. Transaksi yang lebih muda mungkin menunggu yang lebih tua.
  - mungkin lebih sedikit rollback daripada *tunggu-mati* skema.



## Pencegahan deadlock (Lanjutan.)

- Keduanya dalam *tunggu-mati* dan masuk *tunggu luka* skema, transaksi yang dibatalkan dimulai ulang dengan stempel waktu aslinya. Transaksi yang lebih lama dengan demikian lebih diutamakan daripada yang lebih baru, dan karenanya kelaparan dapat dihindari.
- **Skema Berbasis Batas Waktu :**
  - transaksi menunggu kunci hanya untuk jangka waktu tertentu. Setelah itu, waktu tunggu habis dan transaksi dibatalkan.
  - jadi jalan buntu tidak mungkin terjadi
  - mudah diimplementasikan; tapi kelaparan mungkin terjadi. Juga sulit untuk menentukan nilai interval waktu tunggu yang baik.

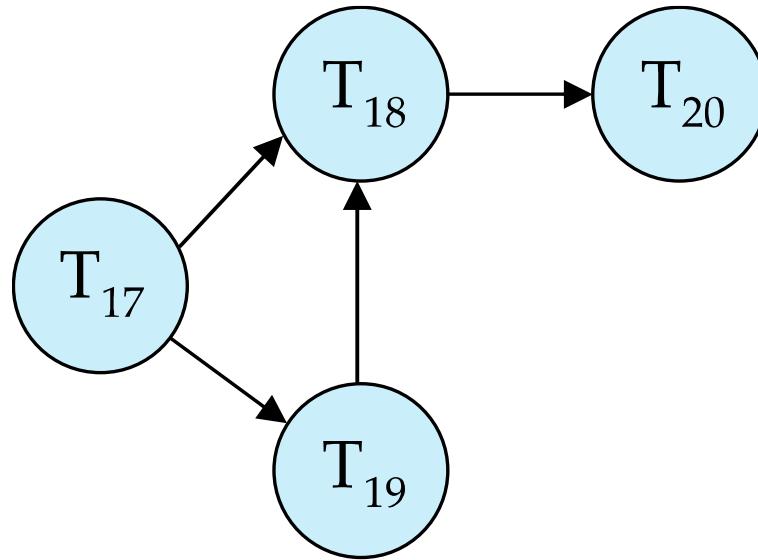


# Deteksi Kebuntuan

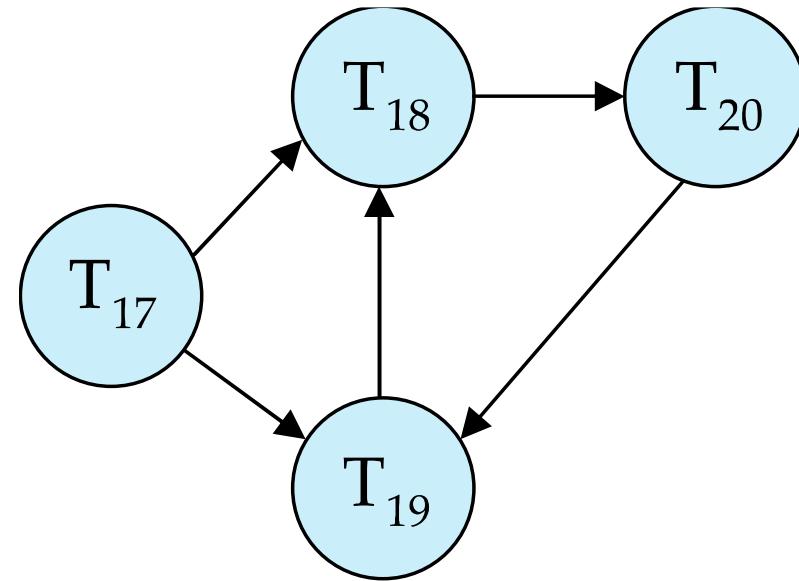
- Kebuntuan dapat digambarkan sebagai a *tunggu grafik*, yang terdiri dari sepasang  $G = (VE)$ ,
  - $V$ . adalah satu set simpul (semua transaksi dalam sistem)
  - $E$  adalah satu set tepi; setiap elemen adalah pasangan terurut  $T_{saya} \rightarrow T_j$ .
- Jika  $T_{saya} \rightarrow T_j$  masuk  $E$ , lalu ada tepi terarah dari  $T_{saya}$  untuk  $T_j$ , menyiratkan itu  $T_{saya}$  sedang menunggu  $T_j$  untuk merilis item data.
- Kapan  $T_{saya}$  meminta item data yang saat ini dipegang oleh  $T_j$ , lalu tepinya  $T_{saya} T_j$  dimasukkan ke dalam grafik tunggu. Tepi ini dihilangkan hanya jika  $T_j$  tidak lagi memegang item data yang dibutuhkan oleh  $T_{saya}$ .
- Sistem berada dalam status kebuntuan jika dan hanya jika grafik tunggu memiliki siklus. Harus menjalankan algoritme deteksi kebuntuan secara berkala untuk mencari siklus.



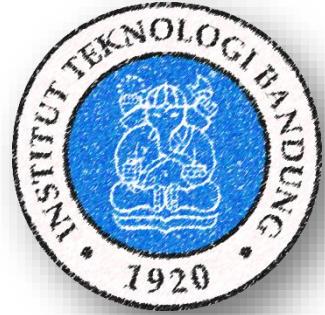
# Deteksi Kebuntuan (Lanjutan)



Grafik tunggu tanpa siklus



Tunggu grafik dengan siklus



# Pemulihan Kebuntuan

- Saat kebuntuan terdeteksi:
  - Beberapa transaksi harus dibatalkan (dijadikan korban) untuk memecahkan kebuntuan. Pilih transaksi tersebut sebagai korban dengan biaya minimum.
  - Rollback - menentukan seberapa jauh transaksi rollback
    - **Kembalikan total** : Batalkan transaksi lalu mulai ulang.
    - Lebih efektif untuk membatalkan transaksi hanya sejauh yang diperlukan untuk memecahkan kebuntuan.
  - Kelaparan terjadi jika transaksi yang sama selalu dipilih sebagai korban. Sertakan jumlah pengembalian dalam faktor biaya untuk menghindari kelaparan

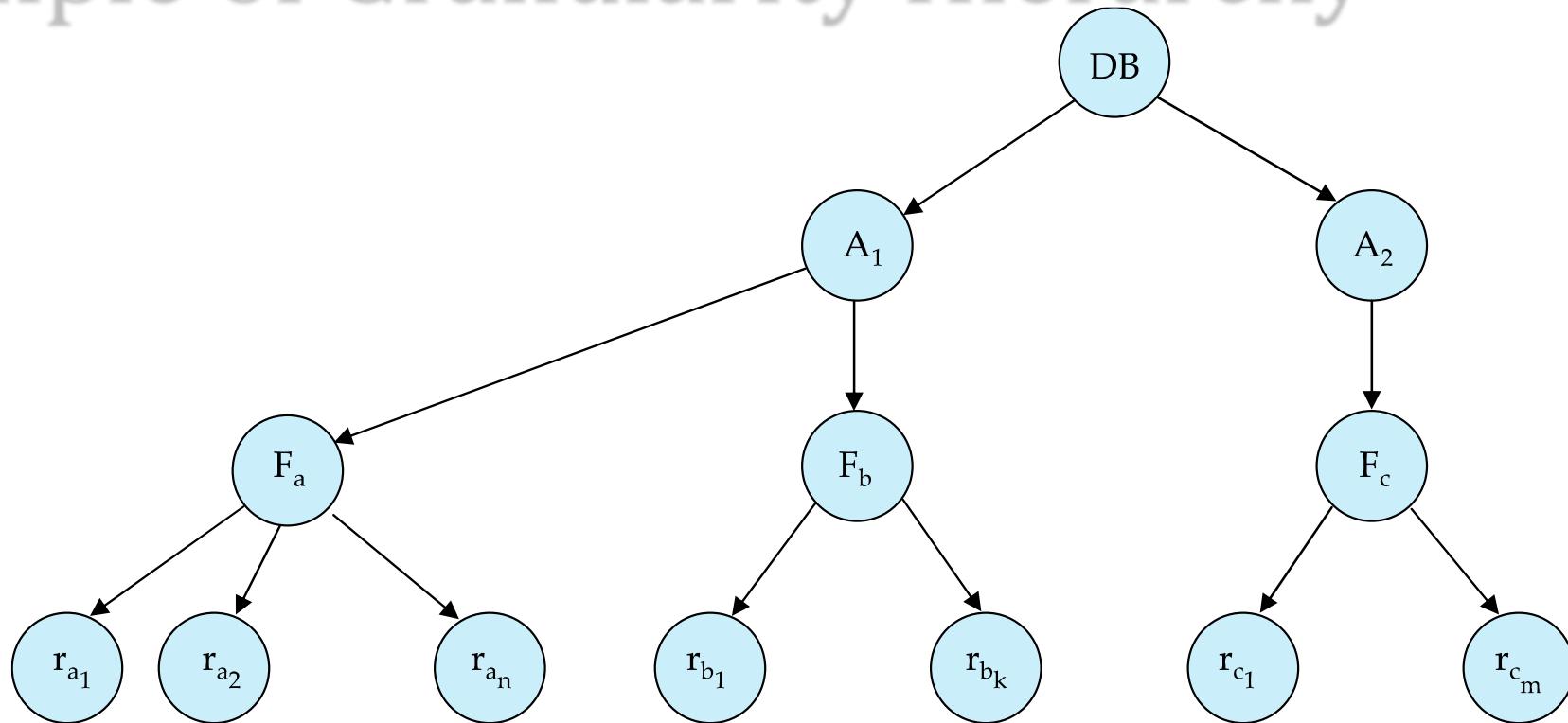


# Perincian Ganda

- Izinkan item data memiliki berbagai ukuran dan tentukan hierarki perincian data, dengan perincian kecil bertumpuk di dalam yang lebih besar
- Dapat direpresentasikan secara grafis sebagai pohon (tapi jangan bingung dengan protokol treelocking)
- Ketika sebuah transaksi mengunci simpul di pohon *secara eksplisit*, Itu *secara implisit* mengunci semua turunan node dalam mode yang sama.
- **Granularitas penguncian** (level di pohon tempat penguncian dilakukan):
  - **perincian halus** ( lebih rendah di pohon): konkurenси tinggi, overhead penguncian tinggi
  - **perincian kasar** ( lebih tinggi di pohon): overhead penguncian rendah, konkurenси rendah

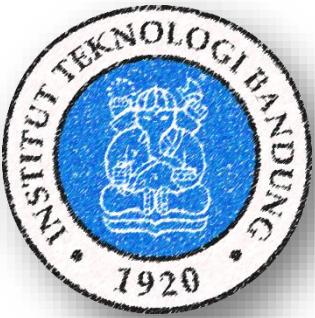


# Contoh Granularitas Hierarki



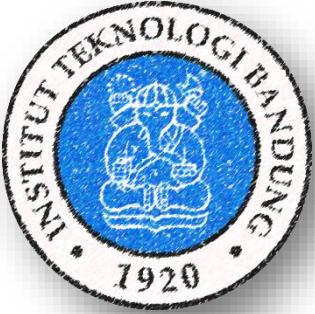
Levelnya, mulai dari level paling kasar (teratas) adalah

- *database*
- *daerah*
- *mengajukan*
- *merekam*



# Mode Kunci Niat

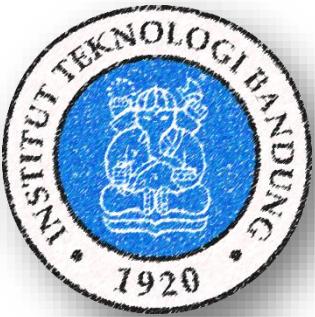
- Selain mode kunci S dan X, ada tiga mode kunci tambahan dengan beberapa perincian:
  - ***berbagi niat*** (IS): menunjukkan penguncian eksplisit di tingkat yang lebih rendah dari pohon tetapi hanya dengan kunci bersama.
  - ***niat-eksklusif*** (IX): menunjukkan penguncian eksplisit di tingkat yang lebih rendah dengan kunci eksklusif atau bersama
  - ***dibagikan dan eksklusif niat*** (SIX): subtree yang di-root oleh node tersebut dikunci secara eksplisit dalam mode bersama dan penguncian eksplisit dilakukan di tingkat yang lebih rendah dengan kunci mode eksklusif.
- kunci niat memungkinkan node tingkat yang lebih tinggi untuk dikunci dalam mode S atau X tanpa harus memeriksa semua node turunan.



# Matriks Kompatibilitas dengan Mode Intention Lock

- Matriks kompatibilitas untuk semua mode kunci adalah:

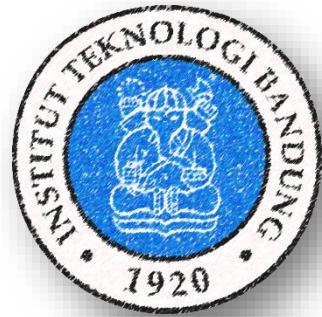
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



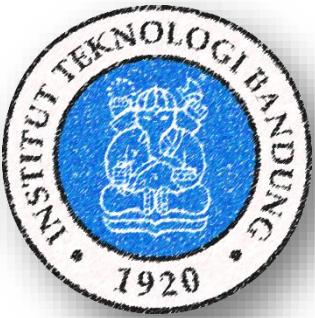
# Skema Penguncian Granularitas Ganda

- Transaksi  $T_{saya}$  dapat mengunci node  $Q$ , menggunakan aturan berikut:
  1. Matriks kompatibilitas kunci harus diperhatikan.
  2. Akar pohon harus dikunci terlebih dahulu, dan dapat dikunci dalam mode apapun.
  3. Sebuah node  $Q$  bisa dikunci oleh  $T_{saya}$  dalam mode S atau IS hanya jika induk dari  $Q$  saat ini dikunci oleh  $T_{saya}$  dalam mode IX atau IS.
  4. Sebuah node  $Q$  bisa dikunci oleh  $T_{saya}$  dalam mode X, SIX, atau IX hanya jika induk dari  $Q$  saat ini dikunci oleh  $T_{saya}$  dalam mode IX atau SIX.
  5.  $T_{saya}$  dapat mengunci node hanya jika sebelumnya tidak membuka node (yaitu,  $T_{saya}$  adalah dua-tahap).
  6.  $T_{saya}$  dapat membuka sebuah node  $Q$  hanya jika tidak ada anak dari  $Q$  saat ini dikunci oleh  $T_{saya}$ .
- Perhatikan bahwa kunci diperoleh dalam urutan akar-ke-daun, sedangkan kunci dilepaskan dalam urutan daun-ke-akar.
- **Kunci eskalasi perincian** : jika ada terlalu banyak kunci pada tingkat tertentu, alihkan ke kunci S atau X dengan perincian yang lebih tinggi

# Timestamp-Based Protocols



- Setiap transaksi diberikan stempel waktu saat memasuki sistem. Jika dua transaksi  $T_{saya}$  memiliki cap waktu TS ( $T_{saya}$ ), transaksi baru  $T_j$  diberi stempel waktu TS ( $T_j$ ) sedemikian rupa sehingga  $TS(T_i) < TS(T_j)$ .
- Protokol mengelola eksekusi bersamaan sehingga stempel waktu menentukan urutan kemampuan berseri.
- Untuk memastikan perilaku tersebut, protokol memelihara setiap data  $Q$  dua nilai cap waktu:
  - **Stempel waktu W ( $Q$ )** adalah cap waktu terbesar dari setiap transaksi yang dijalankan **menulis**( $Q$ ) berhasil.
  - **R-timestamp ( $Q$ )** adalah cap waktu terbesar dari setiap transaksi yang dijalankan **Baca**( $Q$ ) berhasil.



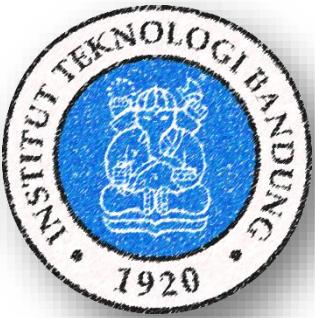
## Protokol Berbasis Stempel Waktu (Lanjutan)

- Protokol pengurutan stempel waktu memastikan bahwa setiap konflik **Baca** dan **menulis** operasi dijalankan dalam urutan stempel waktu.
- Misalkan transaksi  $T_{saya}$  masalah a **Baca( Q)**
  1. Jika  $TS(T_{saya}) \cdot W$ - cap waktu (  $Q$ ), kemudian  $T_{saya}$  perlu membaca nilai  $Q$  itu tadi sudah ditimpa.
    - Oleh karena itu, **Baca** operasi ditolak, dan  $T_{saya}$  digulung kembali.
  2. Jika  $TS(T_{saya}) \cdot W$ - cap waktu (  $Q$ ), lalu **Baca** operasi dijalankan, dan R-cap waktu (  $Q$ ) diatur ke **maks** ( R-timestamp (  $Q$ ),  $TS(T_{saya})$ ).



## Protokol Berbasis Stempel Waktu (Lanjutan)

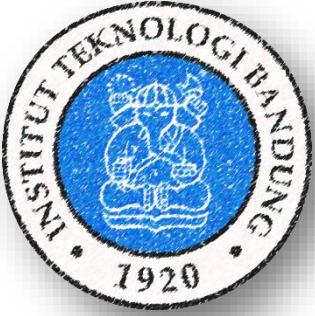
- Misalkan transaksi itu  $T_{saya}$  masalah **menulis**(  $Q$  ).
  1. Jika  $TS ( T_i ) < R$ -timestamp (  $Q$  ), lalu nilai  $Q$  bahwa  $T_{saya}$  memproduksi dulu diperlukan sebelumnya, dan sistem berasumsi bahwa nilai itu tidak akan pernah ada diproduksi.
    - Oleh karena itu, **menulis** operasi ditolak, dan  $T_{saya}$  digulung kembali.
  2. Jika  $TS ( T_i ) <$ Stempel waktu  $W ( Q )$ , kemudian  $T_{saya}$  mencoba menulis nilai yang sudah usang dari  $Q$ .
    - Oleh karena itu, ini **menulis** operasi ditolak, dan  $T_{saya}$  digulung kembali.
  3. Jika tidak, file **menulis** operasi dijalankan, dan  $W$ -timestamp (  $Q$  ) diatur ke  $TS ( T_{saya} )$ .



# Contoh Penggunaan Protokol

Jadwal parsial untuk beberapa item data untuk transaksi dengan cap waktu 1, 2, 3, 4, 5

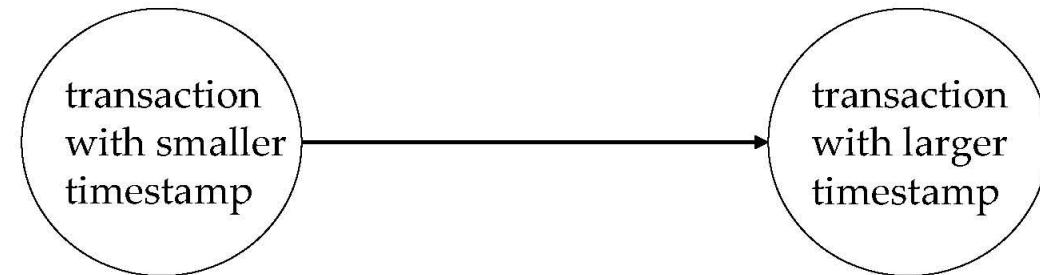
$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
				read (X)
read (Y)	read (Y)			
		write (Y) write (Z)		
	read (Z) abort			read (Z)
read (X)			read (W)	
		write (W) abort		
				write (Y) write (Z)



# Ketepatan

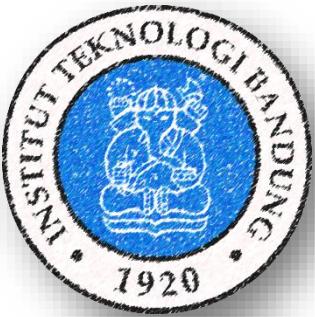
## Protokol Pemesanan Stempel Waktu

- Protokol pengurutan stempel waktu menjamin kemampuan serial karena semua busur dalam grafik prioritas berbentuk:



Dengan demikian, tidak akan ada siklus dalam grafik prioritas

- Protokol timestamp memastikan kebebasan dari kebuntuan karena tidak ada transaksi yang menunggu.
- Tetapi jadwalnya mungkin tidak bebas kaskade, dan bahkan mungkin tidak dapat dipulihkan.



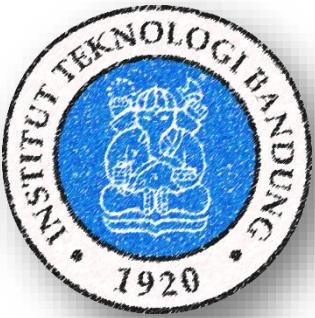
# Recoverability dan Cascade Freedom

- Masalah dengan protokol pemesanan stempel waktu:
  - Seharusnya  $T_{saya}$  dibatalkan, tapi  $T_i$  telah membaca item data yang ditulis oleh  $T_{saya}$
  - Kemudian  $T_i$  harus membatalkan; jika  $T_i$  telah diizinkan untuk melakukan lebih awal, jadwalnya tidak dapat dipulihkan.
  - Selanjutnya, setiap transaksi yang telah membaca item data yang ditulis oleh  $T_i$  harus membatalkan
  - Hal ini dapat menyebabkan rollback berjenjang --- yaitu, rangkaian rollback
- Solusi 1:
  - Sebuah transaksi disusun sedemikian rupa sehingga semua penulisannya dilakukan pada akhir pemrosesannya
  - Semua penulisan transaksi membentuk aksi atom; tidak ada transaksi yang dapat dieksekusi saat transaksi sedang ditulis
  - Transaksi yang dibatalkan dimulai ulang dengan stempel waktu baru
- Solusi 2: Bentuk penguncian terbatas: tunggu data dimasukkan sebelum membacanya
- Solusi 3: Gunakan dependensi komit untuk memastikan pemulihan

# Thomas' Write Rule



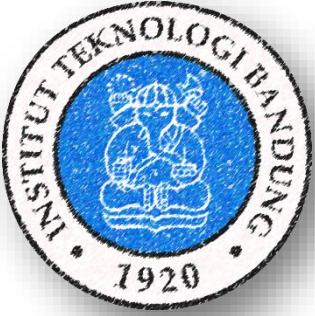
- Versi modifikasi dari protokol pengurutan stempel waktu yang usang **menulis** operasi dapat diabaikan dalam keadaan tertentu.
- Kapan  $T_{saya}$  mencoba untuk menulis item data  $Q$ , jika  $TS(T_i) < \text{Stempel waktu } W(Q)$ , kemudian  $T_{saya}$  mencoba menulis nilai usang dari  $\{Q\}$ .
  - Daripada memutar kembali  $T_{saya}$  seperti yang akan dilakukan oleh protokol pemesanan stempel waktu, ini **{ menulis}** operasi dapat diabaikan.
- Jika tidak, protokol ini sama dengan protokol pemesanan stempel waktu.
- Aturan Tulis Thomas memungkinkan potensi konkurensi yang lebih besar.
  - Mengizinkan beberapa jadwal yang dapat diserialkan yang tidak dapat dibuat bersambung konflik.



# Lihat Serializability

- Membatasi  $S$  dan  $S'$  menjadi dua jadwal dengan set transaksi yang sama.  $S$  dan  $S'$  adalah **lihat setara** jika tiga kondisi berikut terpenuhi, untuk setiap item data  $Q$ ,
1. Jika dalam jadwal  $S$ , transaksi  $T_{saya}$  membaca nilai awal  $Q$ , lalu sesuai jadwal  $S'$  juga transaksi  $T_{saya}$  harus membaca nilai awal  $Q$ .
  2. Jika dalam jadwal  $S$  transaksi  $T_{saya}$  dieksekusi **Baca( Q)**, dan nilai itu dihasilkan oleh transaksi  $T_j$  (jika ada), lalu sesuai jadwal  $S'$  juga transaksi  $T_{saya}$  harus membaca nilai  $Q$  yang diproduksi oleh yang sama **menulis( Q)** pengoperasian transaksi  $T_j$ .
  3. Transaksi (jika ada) yang melakukan final **menulis( Q)** beroperasi sesuai jadwal  $S$  juga harus melakukan final **menulis( Q)** beroperasi sesuai jadwal  $S'$ .

Seperti yang bisa dilihat, kesetaraan tampilan juga didasarkan murni pada **membaca** dan **menulis** sendirian.



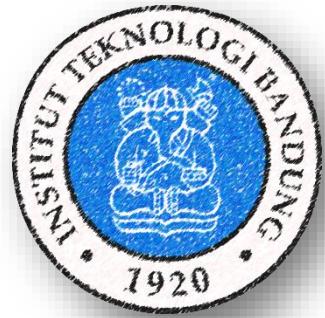
# View Serializability (Cont.)

- Sebuah jadwal  $S$  adalah **lihat serializable** jika dilihat setara dengan jadwal serial.
- Setiap konflik jadwal serializable juga melihat serializable.
- Di bawah ini adalah jadwal yang dapat diserialkan tapi *tidak* konflik dapat diserialkan.

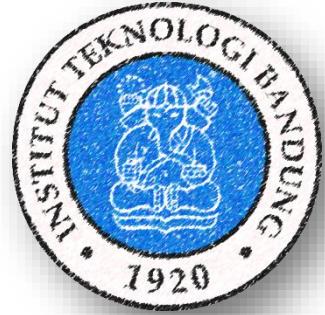
$T_3$	$T_4$	$T_6$
read( $Q$ )		
write( $Q$ )	write( $Q$ )	write( $Q$ )

- Jadwal serial apa yang setara dengan di atas?
- Setiap melihat jadwal serializable yang tidak konflik memiliki serializable **menulis buta**.

# Tes untuk View Serializability



- Uji grafik prioritas untuk kemampuan berseri konflik tidak dapat digunakan secara langsung untuk menguji kemampuan berseri tampilan.
  - Ekstensi untuk menguji kemampuan serialisasi tampilan memiliki biaya eksponensial dalam ukuran grafik prioritas.
- Masalah memeriksa apakah jadwal adalah tampilan serializable jatuh di kelas *NP*- menyelesaikan masalah.
  - Dengan demikian adanya algoritma yang efisien *sangat* tidak sepertinya.
- Namun algoritma praktis itu hanya memeriksa beberapa **kondisi yang cukup** untuk tampilan serializability masih bisa digunakan.



# Pengertian Lain tentang Kemampuan Serial

- Jadwal di bawah ini menghasilkan hasil yang sama dengan jadwal serial  $\langle T_1, T_5 \rangle$ , namun tidak setara konflik atau pandangan setara dengannya.

$T_1$	$T_5$
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( $A$ )

- Menentukan kesetaraan tersebut membutuhkan analisis operasi selain membaca dan menulis.
  - Konflik operasi, kunci operasi



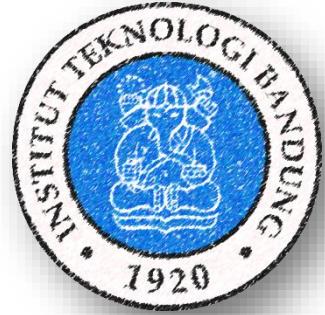
# Protokol Berbasis Validasi

- Eksekusi transaksi  $T_{saya}$  dilakukan dalam tiga tahap.
  1. **Fase baca dan eksekusi:** Transaksi  $T_{saya}$  menulis hanya untuk variabel lokal sementara
  2. **Tahap validasi:** Transaksi  $T_{saya}$  melakukan " uji validasi " untuk menentukan apakah variabel lokal dapat ditulis tanpa melanggar serializability.
  3. **Fase menulis:** Jika  $T_{saya}$  divalidasi, pembaruan diterapkan ke database; jika tidak,  $T_{saya}$  digulung kembali.
- Tiga fase dari transaksi yang dijalankan secara bersamaan dapat disisipkan, tetapi setiap transaksi harus melalui tiga fase dalam urutan itu.
  - Asumsikan untuk kesederhanaan bahwa fase validasi dan penulisan terjadi bersamaan, secara atomik dan serial
    - Yaitu, hanya satu transaksi yang menjalankan validasi / penulisan pada satu waktu.
  - Juga disebut sebagai **kontrol konkurensi yang optimis** karena transaksi dijalankan sepenuhnya dengan harapan semua akan berjalan dengan baik selama validasi



## Protokol Berbasis Validasi (Lanjutan)

- Setiap transaksi  $T_{saya}$  memiliki 3 cap waktu
  - Mulai ( $T_{i_saya}$ ): saat  $T_{saya}$  memulai eksekusinya
  - Validasi ( $T_{validasi}$ ): saat  $T_{saya}$  memasuki fase validasinya
  - Selesai ( $T_{f_saya}$ ): saat  $T_{saya}$  menyelesaikan fase menulisnya
- Urutan kemampuan serial ditentukan oleh stempel waktu yang diberikan pada waktu validasi, untuk meningkatkan konkurensi.
  - Jadi  $TS(T_{saya})$  diberi nilai Validasi ( $T_{validasi}$ ).
- Protokol ini berguna dan memberikan derajat konkurensi yang lebih besar jika kemungkinan konflik rendah.
  - karena urutan serializability tidak diputuskan sebelumnya, dan
  - relatif sedikit transaksi yang harus dibatalkan.



# Uji Validasi Transaksi $T_j$

- Jika untuk semua  $T_{saya}$  dengan TS ( $T_i < TS < T_j$ ) salah satu dari kondisi berikut berlaku:
  - **selesai(**  $T_i$  **) < Mulailah(**  $T_j$  **)**
  - **Mulailah(**  $T_j$  **) < selesai(**  $T_i$  **) < validasi(**  $T_j$  **)** dan kumpulan item data yang ditulis oleh  $T_{saya}$  tidak bersinggungan dengan kumpulan item data yang dibaca  $T_j$ .maka validasi berhasil dan  $T_j$  bisa dilakukan. Jika tidak, validasi gagal dan  $T_j$  dibatalkan.
- *Pembenaran:* Kondisi pertama terpenuhi, dan tidak ada eksekusi yang tumpang tindih, atau kondisi kedua terpenuhi dan
  - tulisan dari  $T_j$  tidak memengaruhi pembacaan  $T_{saya}$  sejak mereka terjadi setelahnya  $T_{saya}$  telah menyelesaikan pembacaannya.
  - tulisan dari  $T_{saya}$  tidak memengaruhi pembacaan  $T_j$  sejak  $T_j$  tidak membaca item apa pun yang ditulis oleh  $T_{saya}$ .

# Jadwal Dibuat oleh Validation



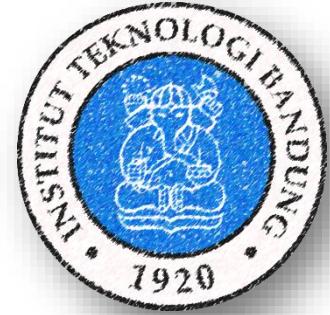
- Contoh jadwal yang dihasilkan menggunakan validasi

$T_{25}$	$T_{26}$
read ( $B$ )	read ( $B$ ) $B := B - 50$ read ( $A$ ) $A := A + 50$
read ( $A$ ) $\langle validate \rangle$ display ( $A + B$ )	$\langle validate \rangle$ write ( $B$ ) write ( $A$ )

# Skema Multiversi

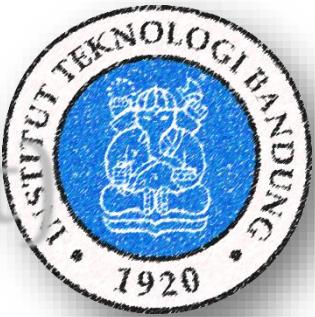


- Skema multiversi mempertahankan item data versi lama untuk meningkatkan konkurensi.
  - Pengurutan Stempel Waktu Multiversi
  - Penguncian Dua Fase Multiversi
- Masing-masing berhasil **menulis** hasil dalam pembuatan versi baru dari item data yang ditulis.
- Gunakan stempel waktu untuk melabeli versi.
- Ketika sebuah **Baca( Q)** operasi dikeluarkan, pilih versi yang sesuai  $Q$  berdasarkan stempel waktu transaksi, dan mengembalikan nilai versi yang dipilih.
- **Baca s** tidak perlu menunggu karena versi yang sesuai segera dikembalikan.



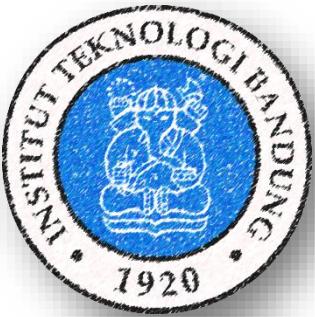
# Pengurutan Stempel Waktu Multiversi

- Setiap item data  $Q$  memiliki urutan versi  $< Q_1, Q_2, \dots, Q_m >$ . Setiap versi  $Q_k$  berisi tiga bidang data:
  - **Kandungan** -- nilai versi  $Q_k$ .
  - **Stempel waktu W** ( $Q_k$ ) - stempel waktu transaksi yang membuat (menulis) versi  $Q_k$
  - **R-timestamp** ( $Q_k$ ) - stempel waktu terbesar dari transaksi yang berhasil membaca versi  $Q_k$ .
- saat melakukan transaksi  $T_{saya}$  membuat versi baru  $Q_k$  dari  $Q$ ,  $Q_k$ 's W-timestamp dan R-timestamp diinisialisasi ke TS ( $T_{saya}$ ).
- Stempel waktu R dari  $Q_k$  diperbarui setiap kali transaksi  $T_j$  membaca  $Q_k$ , dan  $TS(T_j) > R$ -timestamp ( $Q_k$ ).



# Pengurutan Stempel Waktu Multiversi (Lanjutan)

- Misalkan transaksi itu  $T_{saya}$  masalah a **Baca( Q)** atau **menulis( Q)** operasi. Membatasi  $Q_k$  menunjukkan versi  $Q$  yang stempel waktu tulisnya adalah stempel waktu tulis terbesar kurang dari atau sama dengan TS (  $T_{saya}$ ).
  1. Jika transaksi  $T_{saya}$  masalah a **Baca( Q)**, maka nilai yang dikembalikan adalah konten versi  $Q$ . k.
  2. Jika bertransaksi  $T_{saya}$  masalah a **menulis( Q)**
    1. jika TS (  $T_i$ ) < R-timestamp (  $Q_k$ ), lalu transaksi  $T_{saya}$  digulung kembali.
    2. jika TS (  $T_i$ ) = Stempel waktu W (  $Q_k$ ), isi dari  $Q_k$  ditimpas
    3. lain versi baru dari  $Q$  dibuat.
- Perhatikan itu
  - Membaca selalu berhasil
  - Sebuah tulisan oleh  $T_{saya}$  ditolak jika ada transaksi lain  $T_j$  yang (dalam urutan serialisasi yang ditentukan oleh nilai stempel waktu) harus dibaca  
 $T_{saya}'$ s tulis, telah membaca versi yang dibuat oleh transaksi yang lebih lama dari  $T_{saya}$ .
- Protokol menjamin serialisasi



# Penguncian Dua Fase Multiversi

- Membedakan antara transaksi hanya baca dan transaksi pembaruan
- *Perbarui transaksi* dapatkan kunci baca dan tulis, dan tahan semua kunci hingga akhir transaksi. Artinya, pembaruan transaksi mengikuti penguncian dua fase yang ketat.
  - Masing-masing berhasil **menulis** hasil dalam pembuatan versi baru dari item data yang ditulis.
  - setiap versi item data memiliki stempel waktu tunggal yang nilainya diperoleh dari penghitung **ts-counter** yang bertambah selama pemrosesan komit.
- *Transaksi hanya baca* diberi stempel waktu dengan membaca nilai saat ini dari **ts-counter** sebelum mereka memulai eksekusi; mereka mengikuti protokol pengurutan stempel waktu multiversi untuk melakukan pembacaan.



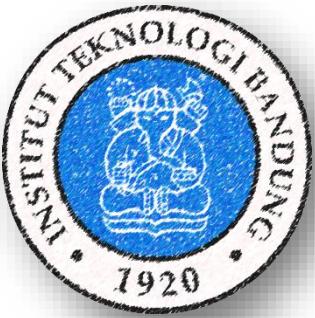
# Penguncian Dua Fase Multiversi (Lanjutan)

- Ketika transaksi pembaruan ingin membaca item data:
  - itu mendapatkan kunci bersama di atasnya, dan membaca versi terbaru.
- Ketika ingin menulis item
  - itu memperoleh kunci X; lalu membuat versi baru dari item tersebut dan menyetel stempel waktu versi ini ke •.
- Saat memperbarui transaksi  $T_{saya}$  selesai, pemrosesan komit terjadi:
  - $T_{saya}$  menyetel stempel waktu pada versi yang dibuatnya **ts-counter + 1**
  - $T_{saya}$  kenaikan **ts-counter** dengan 1
- Transaksi hanya baca yang dimulai setelahnya  $T_{saya}$  kenaikan **ts-counter** akan melihat nilai diperbarui oleh  $T_{saya}$ .
- Transaksi hanya baca yang dimulai sebelumnya  $T_{saya}$  meningkatkan **ts-counter** akan melihat nilai sebelum update oleh  $T_{saya}$ .
- Hanya jadwal serial yang dapat diproduksi.

# MVCC: Masalah Implementasi

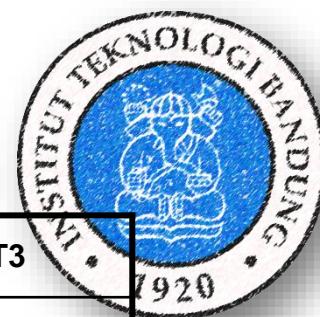


- Pembuatan beberapa versi meningkatkan overhead penyimpanan
  - Tupel ekstra
  - Ruang ekstra di setiap tupel untuk menyimpan informasi versi
- Namun, versi dapat dikumpulkan dari sampah
  - Misalnya jika Q memiliki dua versi Q5 dan Q9, dan transaksi aktif terlama memiliki timestamp > 9, maka Q5 tidak akan pernah diperlukan lagi



# Snapshot Isolation

- Motivasi: Kueri dukungan keputusan yang membaca data dalam jumlah besar memiliki konflik konkurensi dengan transaksi OLTP yang memperbarui beberapa baris
  - Hasil kinerja yang buruk
- Solusi 1: Berikan logika “ foto ” status database untuk membaca transaksi saja, transaksi baca-tulis menggunakan penguncian normal
  - Penguncian multiversi 2 fase
  - Bekerja dengan baik, tetapi bagaimana sistem mengetahui bahwa transaksi itu hanya baca?
- Solusi 2: Berikan snapshot status database untuk setiap transaksi, pembaruan saja menggunakan penguncian 2 fase untuk menjaga dari pembaruan bersamaan
  - Masalah: berbagai anomali seperti update yang hilang dapat terjadi
  - Solusi parsial: tingkat isolasi snapshot (slide berikutnya)
    - Diusulkan oleh Berenson et al, SIGMOD 1995
    - Varian diimplementasikan di banyak sistem database
      - Misalnya Oracle, PostgreSQL, SQL Server 2005



# Snapshot Isolation

- T1 transaksi yang dijalankan dengan Snapshot Isolation
  - mengambil snapshot dari data yang dilakukan di awal
  - selalu membaca / mengubah data dalam snapshotnya sendiri
  - pembaruan transaksi bersamaan tidak terlihat ke T1
  - menulis T1 selesai ketika itu dilakukan
  - **Aturan pemenang komit pertama:**
    - Komit hanya jika tidak ada transaksi bersamaan lainnya yang telah menulis data yang ingin ditulis T1.

Pembaruan bersamaan tidak terlihat  
 Pembaruan sendiri terlihat Bukan  
 pelaku pertama X

Kesalahan serialisasi, T2 digulung kembali

T1	T2	T3
W (Y: = 1) Melakukan		
	Mulailah R (X) $\rightarrow$ 0 R (Y) $\rightarrow$ 1	
		W (X: = 2) W (Z: = 3) Melakukan
	R (Z) $\rightarrow$ 0 R (Y) $\rightarrow$ 1 W (X: = 3) Commit-Req Menggugurkan	



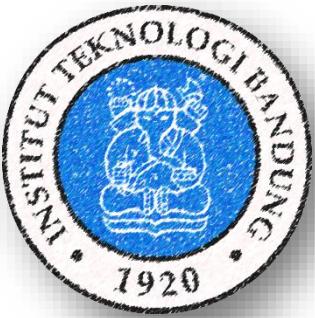
# Snapshot Dibaca

- Pembaruan serentak tidak terlihat untuk pembacaan snapshot

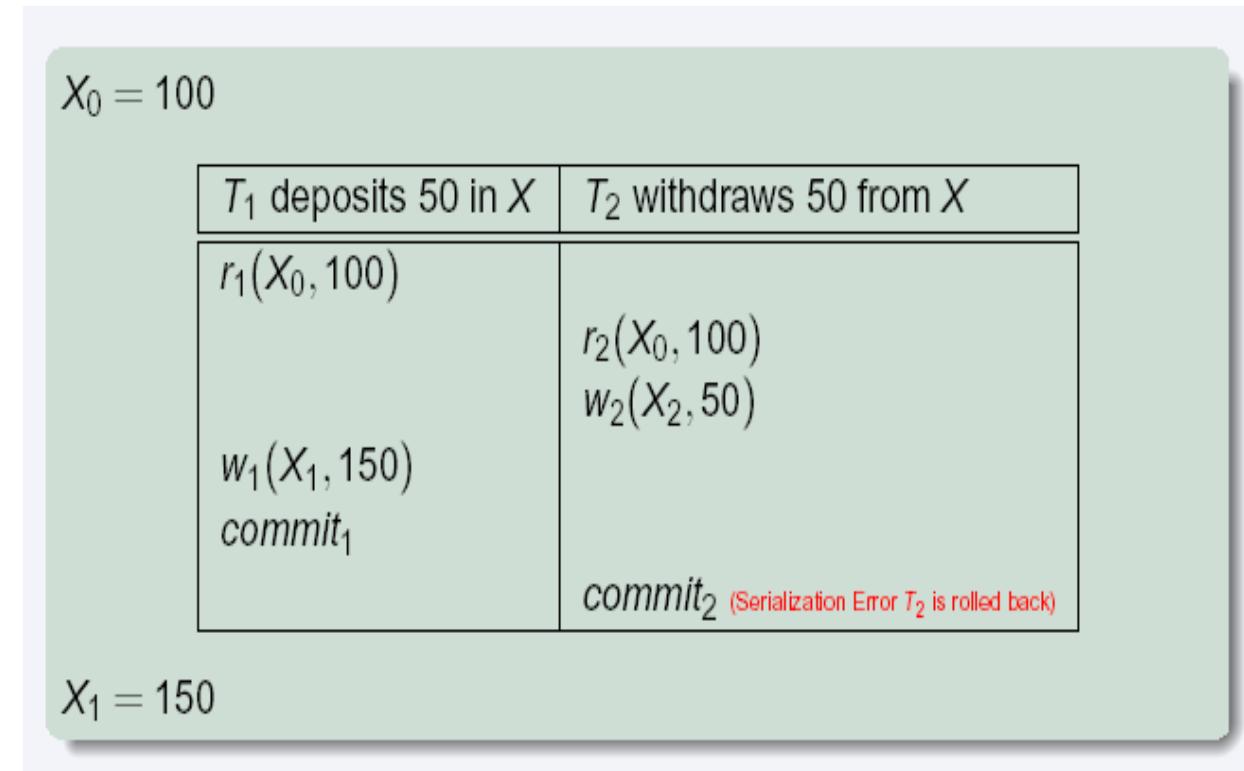
$$X_0 = 100, Y_0 = 0$$

$T_1$ deposits 50 in $Y$	$T_2$ withdraws 50 from $X$
$r_1(X_0, 100)$ $r_1(Y_0, 0)$	$r_2(Y_0, 0)$ $r_2(X_0, 100)$ $w_2(X_2, 50)$
$w_1(Y_1, 50)$ $r_1(X_0, 100)$ (update by $T_2$ not seen) $r_1(Y_1, 50)$ (can see its own updates)	$r_2(Y_0, 0)$ (update by $T_1$ not seen)

$$X_2 = 50, Y_1 = 50$$



# Snapshot Menulis: Pemenang Pertama

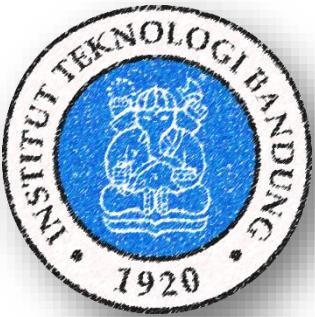


- Varian: “ **Pembaru-menang pertama** ”
  - Periksa pembaruan bersamaan saat penulisan terjadi dengan mengunci item
    - Tapi kunci harus ditahan sampai semua transaksi bersamaan selesai
  - (Oracle menggunakan ini ditambah beberapa fitur tambahan)
  - Hanya berbeda ketika terjadi keguguran, jika tidak setara



# Manfaat SI

- Membaca adalah *tidak pernah* diblokir,
  - dan juga tidak 't memblokir aktivitas txns lainnya
- Performanya mirip dengan Read Committed
- Menghindari anomali biasa
  - Tidak ada bacaan kotor
  - Tidak ada pembaruan yang hilang
  - Tidak ada pembacaan yang tidak dapat diulang
  - Pilihan berdasarkan predikat dapat diulang (tidak ada hantu)
- Masalah dengan SI
  - SI tidak selalu memberikan eksekusi yang dapat bersambung
    - Dapat diserialkan: di antara dua txns bersamaan, yang satu melihat efek yang lain
    - Dalam SI: tidak ada yang melihat efek dari yang lain
  - Hasil: Batasan integritas dapat dilanggar



# Snapshot Isolation

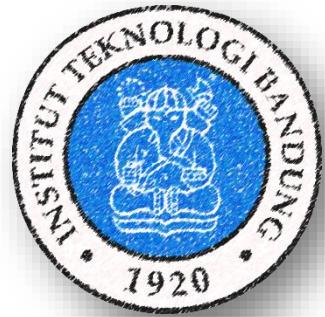
- Misalnya masalah dengan SI
  - T1:  $x := y$
  - T2:  $y := x$
  - Awalnya  $x = 3$  dan  $y = 17$ 
    - Eksekusi serial:  $x = ??$ ,  $y = ??$
    - jika kedua transaksi dimulai pada waktu yang sama, dengan isolasi snapshot:  $x = ??$ ,  $y = ??$
- Dipanggil **menulis miring**
- Kemiringan juga terjadi dengan sisipan
  - Misalnya:
    - Temukan nomor pesanan maksimal di antara semua pesanan
    - Buat pesanan baru dengan nomor pesanan = max sebelumnya + 1

# Anomali Isolasi Foto



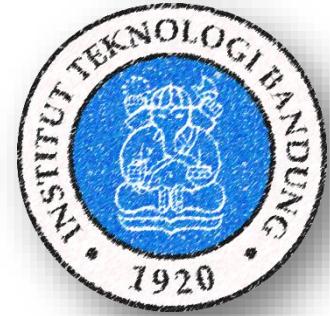
- SI merusak kemampuan serial ketika txns dimodifikasi *berbeda* item, masing-masing berdasarkan keadaan sebelumnya dari item yang dimodifikasi lainnya
  - Tidak terlalu umum dalam praktiknya
    - Misalnya, tolok ukur TPC-C berjalan dengan benar di bawah SI
    - ketika txns bentrok karena mengubah data yang berbeda, biasanya ada juga item bersama yang keduanya juga dimodifikasi (seperti jumlah total) sehingga SI akan membatalkan salah satunya
  - Tapi memang terjadi
    - Pengembang aplikasi harus berhati-hati menulis miring
- SI juga dapat menyebabkan anomali transaksi hanya-baca, di mana transaksi hanya-baca mungkin melihat keadaan yang tidak konsisten bahkan jika pembaruan dapat diserialkan
  - Kami menghilangkan detail
- Menggunakan snapshot untuk memverifikasi integritas kunci utama / asing dapat menyebabkan ketidakkonsistensi
  - Pemeriksaan kendala integritas biasanya dilakukan di luar snapshot

# SI Di Oracle dan PostgreSQL



- **Peringatan** : SI digunakan ketika tingkat isolasi diatur ke serializable, oleh Oracle, dan versi PostgreSQL sebelum 9.1
  - PostgreSQL ' Implementasi SI (versi sebelum 9.1) dijelaskan dalam Bagian 26.4.1.3
  - Oracle mengimplementasikan “ pembaru pertama menang ” aturan (varian dari “ pelaku pertama menang ”)
    - Pemeriksaan penulis secara bersamaan dilakukan pada saat penulisan, bukan pada waktu pengikatan
    - Memungkinkan transaksi dibatalkan lebih awal
    - Oracle dan PostgreSQL <9.1 tidak mendukung eksekusi serializable yang sebenarnya
  - PostgreSQL 9.1 memperkenalkan protokol baru yang disebut "Serializable Snapshot Isolation" (SSI)
    - Yang menjamin serializability yang benar termasuk menangani pembacaan predikat (datang)

# SI Di Oracle dan PostgreSQL

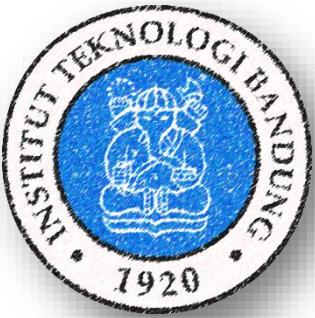


- Dapat menghindari SI untuk kueri tertentu dengan menggunakan **pilih .. untuk update** di Oracle dan PostgreSQL
  - Misalnya,
    1. pilih maks ( nomor pesanan) dari pesanan untuk pembaruan
    2. membaca nilai ke dalam maxorder variabel lokal
    3. masukkan ke dalam pesanan (maxorder + 1,...)
  - Select for update (SFU) memperlakukan semua data yang dibaca oleh kueri seolah-olah itu juga diperbarui, mencegah pembaruan serentak
  - Tidak selalu memastikan serialisasi karena fenomena hantu dapat terjadi (datang)
- Di PostgreSQL versi <9.1, SFU mengunci item data, tetapi melepaskan kunci ketika transaksi selesai, bahkan jika transaksi bersamaan lainnya aktif
  - Tidak persis sama dengan SFU di Oracle, yang menyimpan kunci sampai semuanya
  - transaksi bersamaan telah selesai



# Sisipkan dan Hapus Operasi

- Jika penguncian dua fase digunakan:
  - SEBUAH **menghapus** Operasi dapat dilakukan hanya jika transaksi yang menghapus tupel memiliki kunci eksklusif pada tupel yang akan dihapus.
  - Transaksi yang memasukkan tupel baru ke dalam database diberi kunci mode-X pada tupel tersebut
- Penyisipan dan penghapusan dapat mengarah ke **fenomena hantu** .
  - Transaksi yang memindai hubungan
    - (misalnya, temukan jumlah saldo dari semua akun di Perryridge) dan transaksi yang memasukkan tupel dalam relasi
    - (mis., masukkan akun baru di Perryridge)  
(secara konseptual) konflik meskipun tidak mengakses tupel yang sama.
  - Jika hanya kunci tuple yang digunakan, jadwal yang tidak dapat diserialkan dapat terjadi
    - Misalnya transaksi scan tidak melihat akun baru, tetapi membaca beberapa tuple lain yang ditulis oleh transaksi pembaruan



# Sisipkan dan Hapus Operasi (Lanjutan)

- Transaksi yang memindai relasi membaca informasi yang mengindikasikan tupel apa yang dikandung relasi, sementara transaksi yang memasukkan tupel memperbarui informasi yang sama.
  - Konflik harus dideteksi, misalnya dengan mengunci informasi.
- Satu solusi:
  - Kaitkan item data dengan relasi, untuk merepresentasikan informasi tentang tupel apa yang dikandung relasi.
  - Transaksi yang memindai relasi memperoleh kunci bersama di item data,
  - Transaksi memasukkan atau menghapus tupel memperoleh kunci eksklusif pada item data. (Catatan: kunci pada item data tidak bertentangan dengan kunci pada tupel individu.)
- Protokol di atas menyediakan konkurensi yang sangat rendah untuk penyisipan / penghapusan.
- Protokol penguncian indeks memberikan konkurensi yang lebih tinggi sekaligus mencegah fenomena bayangan, dengan mengharuskan kunci pada keranjang indeks tertentu.



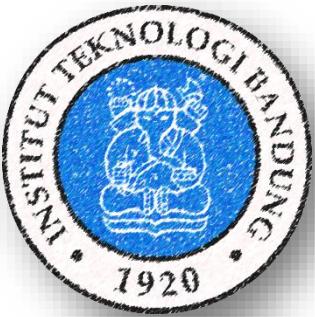
# Protokol Penguncian Indeks

- Protokol penguncian indeks:
  - Setiap relasi harus memiliki setidaknya satu indeks.
  - Sebuah transaksi dapat mengakses tupel hanya setelah menemukannya melalui satu atau lebih indeks pada relasinya
  - Sebuah transaksi  $T_{saya}$  yang melakukan pencarian harus mengunci semua node daun indeks yang diaksesnya, dalam S-mode
    - Sekalipun node daun tidak berisi tupel apa pun yang memenuhi pencarian indeks (mis. Untuk kueri rentang, tidak ada tupel dalam daun dalam rentang)
    - Sebuah transaksi  $T_{saya}$  yang menyisipkan, memperbarui atau menghapus tupel  $t_{saya}$  dalam suatu hubungan  $r$ 
      - harus memperbarui semua indeks menjadi  $r$
      - harus mendapatkan kunci eksklusif di semua node daun indeks yang terpengaruh oleh penyisipan / pembaruan / penghapusan
  - Aturan protokol penguncian dua fase harus diperhatikan
- Menjamin bahwa fenomena hantu menang ' t terjadi

# Penguncian Tombol Berikutnya



- Protokol penguncian indeks untuk mencegah hantu harus mengunci seluruh daun
  - Dapat mengakibatkan konkurensi yang buruk jika ada banyak sisipan
- Alternatif: untuk pencarian indeks
  - Kunci semua nilai yang memenuhi pencarian indeks (nilai pencarian yang cocok, atau termasuk dalam rentang pencarian)
  - Juga kunci nilai kunci berikutnya dalam indeks
  - Mode kunci: S untuk pencarian, X untuk menyisipkan / menghapus / memperbarui
- Memastikan bahwa kueri rentang akan bertentangan dengan sisipan / hapus / pembaruan
  - Terlepas dari mana yang terjadi lebih dulu, selama keduanya bersamaan



# Konkurensi dalam Struktur Indeks

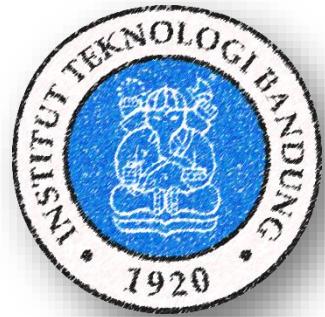
- Indeks tidak seperti item database lain yang satu-satunya tugas mereka adalah membantu mengakses data.
- Struktur indeks biasanya sangat sering diakses, lebih dari item database lainnya.
  - Memperlakukan struktur indeks seperti item database lainnya, misalnya dengan penguncian 2 fase dari node indeks dapat menyebabkan konkurensi rendah.
  - Ada beberapa protokol konkurensi indeks di mana kunci pada node internal dilepaskan lebih awal, dan tidak dalam mode dua fase.
    - Dapat diterima untuk memiliki akses bersamaan yang tidak dapat diserialkan ke indeks selama keakuratan indeks dipertahankan.
      - Secara khusus, nilai eksak terbaca di simpul internal a Pohon B + tidak relevan selama kita mendarat di simpul daun yang benar.



# Konkurensi dalam Struktur Indeks (Lanjutan)

- Contoh protokol konkurensi indeks:
- Menggunakan **kepiting** alih-alih penguncian dua fase pada simpul pohon B +, sebagai berikut. Selama pencarian / penyisipan / penghapusan:
  - Pertama kunci node root dalam mode bersama.
  - Setelah mengunci semua turunan yang diperlukan dari sebuah node dalam mode bersama, lepaskan kunci pada node tersebut.
  - Selama penyisipan / penghapusan, tingkatkan kunci simpul daun ke mode eksklusif.
  - Saat pemisahan atau penggabungan memerlukan perubahan pada induk, kunci induk dalam mode eksklusif.
- Protokol di atas dapat menyebabkan kebuntuan yang berlebihan
  - Pencarian turun kebuntuan pohon dengan pembaruan naik pohon
  - Dapat membatalkan dan memulai kembali pencarian, tanpa mempengaruhi transaksi
- Protokol yang lebih baik tersedia; lihat Bagian 16.9 untuk salah satu protokol tersebut, protokol pohon B-link
  - Intuisi: lepaskan kunci pada orang tua sebelum memperoleh kunci pada anak
    - Dan menangani perubahan yang mungkin terjadi antara pelepasan kunci dan perolehan

# Tingkat Konsistensi yang Lemah



- **Konsistensi derajat dua** : berbeda dari penguncian dua fase karena Slock dapat dilepaskan kapan saja, dan kunci dapat diperoleh kapan saja
  - X-locks harus ditahan sampai akhir transaksi
  - Serializability tidak dijamin, programmer harus memastikan bahwa tidak ada status database yang salah akan terjadi
- **Stabilitas kursor** :
  - Untuk pembacaan, setiap tupel dikunci, dibaca, dan kunci segera dilepaskan
  - X-locks ditahan sampai akhir transaksi
  - Kasus khusus konsistensi derajat-dua



# Tingkat Konsistensi yang Lemah dalam SQL

- SQL memungkinkan eksekusi yang tidak dapat diserialkan
  - **Dapat diserialkan** : adalah defaultnya
  - **Bacaan berulang** : memungkinkan hanya catatan yang berkomitmen untuk dibaca, dan mengulangi pembacaan harus mengembalikan nilai yang sama (jadi kunci baca harus dipertahankan)
    - Namun, fenomena bayangan tidak perlu dicegah
      - T1 mungkin melihat beberapa record dimasukkan oleh T2, tetapi mungkin tidak melihat record lain disisipkan oleh T2
  - **Baca berkomitmen** : sama dengan konsistensi tingkat dua, tetapi kebanyakan sistem menerapkannya sebagai stabilitas cursor
  - **Baca tanpa komitmen** : memungkinkan bahkan data yang tidak terikat untuk dibaca
- Dalam banyak sistem database, read commit adalah tingkat konsistensi default
  - harus secara eksplisit diubah menjadi serializable saat diperlukan
    - setel tingkat isolasi dapat diserialkan



# Transaksi di seluruh Interaksi Pengguna

- Banyak aplikasi membutuhkan dukungan transaksi di seluruh interaksi pengguna
  - Bisa 't gunakan penguncian
  - Mengenakan 't ingin memesan koneksi database per pengguna
- Kontrol konkurensi tingkat aplikasi
  - Setiap tupel memiliki nomor versi
  - Nomor versi catatan transaksi saat membaca tuple
    - **Pilih r.balance, r.version menjadi: A,: versi dari r dimana acctId = 23**
  - Saat menulis tuple, periksa apakah nomor versi saat ini sama dengan versi saat tupel dibaca
    - **memperbarui r set r.balance = r.balance +: deposit dimana acctId = 23 dan r.version =: versi**
- Setara dengan **kontrol konkurensi yang optimis tanpa memvalidasi set baca**
- Digunakan secara internal dalam sistem Hibernate ORM, dan secara manual di banyak aplikasi
- Penomoran versi juga dapat digunakan untuk mendukung pemeriksaan kemenangan pelaku pertama dari isolasi snapshot
  - Tidak seperti SI, pembacaan tidak dijamin berasal dari satu snapshot

# Akhir Bab Chapter

Terima kasih kepada Alan Fekete dan Sudhir Jorwekar atas contoh Isolasi Snapshot

