IF3140 Manajemen Basis Data
Concurrency Kontrol

Slide diambil dari: Sistem Basis Data Konsep, 6 th Edisi - Bab 15 © Silberschatz, Korth, Sudarshan

tujuan

- Siswa dapat:
  - Menjelaskan efek dari tingkat isolasi yang berbeda pada mekanisme kontrol konkurensi
  - Pilih tingkat isolasi yang tepat untuk menerapkan protokol transaksi tertentu

kadar

- Protokol kunci Berbasis
- Protokol timestamp Berbasis
- Protokol validasi Berbasis
- beberapa Granularity
- Multiversion Skema
- Insert dan Operasi Delete
- Concurrency di Struktur Indeks

Protokol kunci Berbasis

- Sebuah kunci adalah mekanisme untuk mengontrol akses bersamaan ke item data
- Data barang yang bisa dikunci dalam dua mode:
  1. *eksklusif Mode (X).* item data dapat baik membaca serta tertulis. X-lock diminta menggunakan **lock-X** petunjuk.
  2. *bersama (S) modus.* item data hanya dapat dibaca. S-kunci adalah diminta menggunakan **lock-S** petunjuk.
- permintaan kunci yang dibuat untuk manajer concurrency-control. Transaksi dapat dilanjutkan hanya setelah permintaan dikabulkan.

Kunci Berbasis Protokol (Cont.)

- **matriks kunci-kompatibilitas**

	S	X
S	true	false
X	false	false

- Sebuah transaksi dapat diberikan kunci pada item jika kunci diminta kompatibel dengan kunci yang sudah diselenggarakan pada item dengan transaksi lainnya
- Setiap jumlah transaksi dapat menahan kunci bersama pada item,
  - tetapi jika transaksi memegang eksklusif pada item ada transaksi lain mungkin memegang kunci apapun pada item.
- Jika kunci tidak dapat diberikan, transaksi meminta dibuat untuk menunggu sampai semua kunci tidak kompatibel dipegang oleh transaksi lain telah dirilis. kunci ini kemudian diberikan.

Kunci Berbasis Protokol (Cont.)

- Contoh transaksi melakukan penguncian:

  T_2: **lock-S (** *SEBUAH);*

  Baca baca (*SEBUAH);*

  membuka kunci(*SEBUAH);*

  lock-S (*B);*

  Baca baca (*B);*

  membuka kunci(*B);*

  display (*A + B)*

- Mengunci seperti di atas tidak cukup untuk menjamin serializability - jika *SEBUAH* dan *B* mendapatkan update di-antara membaca dari *SEBUAH* dan *B,* jumlah yang ditampilkan akan salah.
- SEBUAH **protokol penguncian** adalah seperangkat aturan diikuti oleh semua transaksi sementara meminta dan melepaskan kunci. Mengunci protokol membatasi set jadwal mungkin.

Perangkap Protokol Lock Berbasis

- 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)	

- juga tidak T_3 maupun T_4 dapat membuat kemajuan - mengeksekusi **lock-S (** *B)* penyebab T_4 menunggu untuk T_3 untuk melepaskan kunci pada *B,* sementara mengeksekusi **lock-X (** *SEBUAH)* penyebab T_3 menunggu untuk T_4 untuk melepaskan kunci pada *SEBUAH*.
- situasi seperti ini disebut **jalan buntu .**
  - Untuk menangani salah satu kebuntuan T_3 atau T_4 harus digulung kembali dan kunci dirilis.

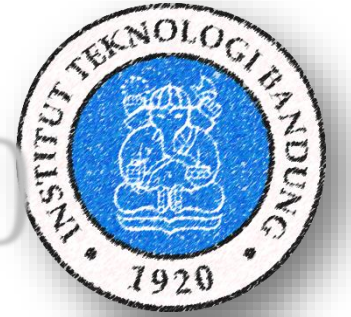
Perangkap Protokol Lock Berbasis (Cont.)

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

Dua-Tahap Locking Protocol

- Ini adalah protokol yang menjamin jadwal konflik-serializable.
- Tahap 1: Tumbuh fase
  - transaksi dapat memperoleh kunci
  - transaksi mungkin tidak melepaskan kunci
- Tahap 2: Menyusut fase
  - transaksi dapat melepaskan kunci
  - transaksi mungkin tidak mendapatkan kunci
- The menjamin protokol serializability. Hal ini dapat dibuktikan bahwa transaksi dapat serial di urutan mereka **poin kunci** (yaitu titik di mana transaksi yang diperoleh kunci akhir).

Dua-Tahap Locking Protocol (Cont.)



- penguncian dua-fase *tidak* memastikan kebebasan dari kebuntuan
- Cascading roll-kembali mungkin di bawah penguncian dua fase. Untuk menghindari hal ini, ikuti protokol dimodifikasi disebut **penguncian dua-fase yang ketat** . Berikut transaksi harus memegang semua kunci eksklusif sampai itu melakukan / dibatalkan.
- **penguncian dua-tahap ketat** bahkan ketat: di sini *semua* kunci diadakan sampai komit / batalkan. Dalam protokol ini transaksi dapat serial dalam urutan di mana mereka komit.

Dua-Tahap Locking Protocol (Cont.)

- Ada dapat jadwal serializable konflik yang tidak dapat diperoleh jika penguncian dua-fase digunakan.
- Namun, dengan tidak adanya informasi tambahan (misalnya, memerintahkan akses ke data), penguncian dua-tahap yang dibutuhkan untuk serializability konflik dalam arti berikut:

Mengingat 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 yang tidak serializable konflik.

kunci Konversi

- penguncian dua-fase dengan konversi kunci:
  - Fase pertama:
    - bisa mendapatkan kunci-S pada item
    - bisa mendapatkan kunci-X pada item
    - dapat mengkonversi kunci-S untuk kunci-X (upgrade)
  - Tahap Kedua:
    - dapat melepaskan kunci-S
    - dapat melepaskan kunci-X
    - dapat mengkonversi kunci-X untuk kunci-S (downgrade)
- Hal ini menjamin protokol serializability. Tapi masih mengandalkan programmer untuk memasukkan berbagai instruksi penguncian.

Otomatis Akuisisi Kunci

- transaksi T_{saya} masalah standar membaca / menulis instruksi, tanpa panggilan penguncian eksplisit.
- Operasi **Baca baca(** *D)* diproses sebagai:

jika T_{saya} memiliki kunci pada *D*

kemudian

Baca baca(*D)*

lain mulai

jika menunggu perlu sampai ada transaksi

lain memiliki **lock-X** di *D*

hibah T_{saya} Sebuah **lock-S** di *D;*

Baca baca(*D)*

akhir

Otomatis Akuisisi Kunci (Cont.)

- **menulis(** *D)* diproses sebagai:

  jika *T saya* mempunyai sebuah **lock-X** di *D*

  kemudian

  menulis(*D)*

  lain mulai

  jika menunggu perlu sampai tidak lain trans. memiliki kunci apapun pada *D,*

  jika *T saya* mempunyai sebuah **lock-S** di *D*

  kemudian upgrade mengunci *D* untuk **lock-X**

  yang lain

  hibah *T saya* Sebuah **lock-X** di *D*

  menulis(*D)*

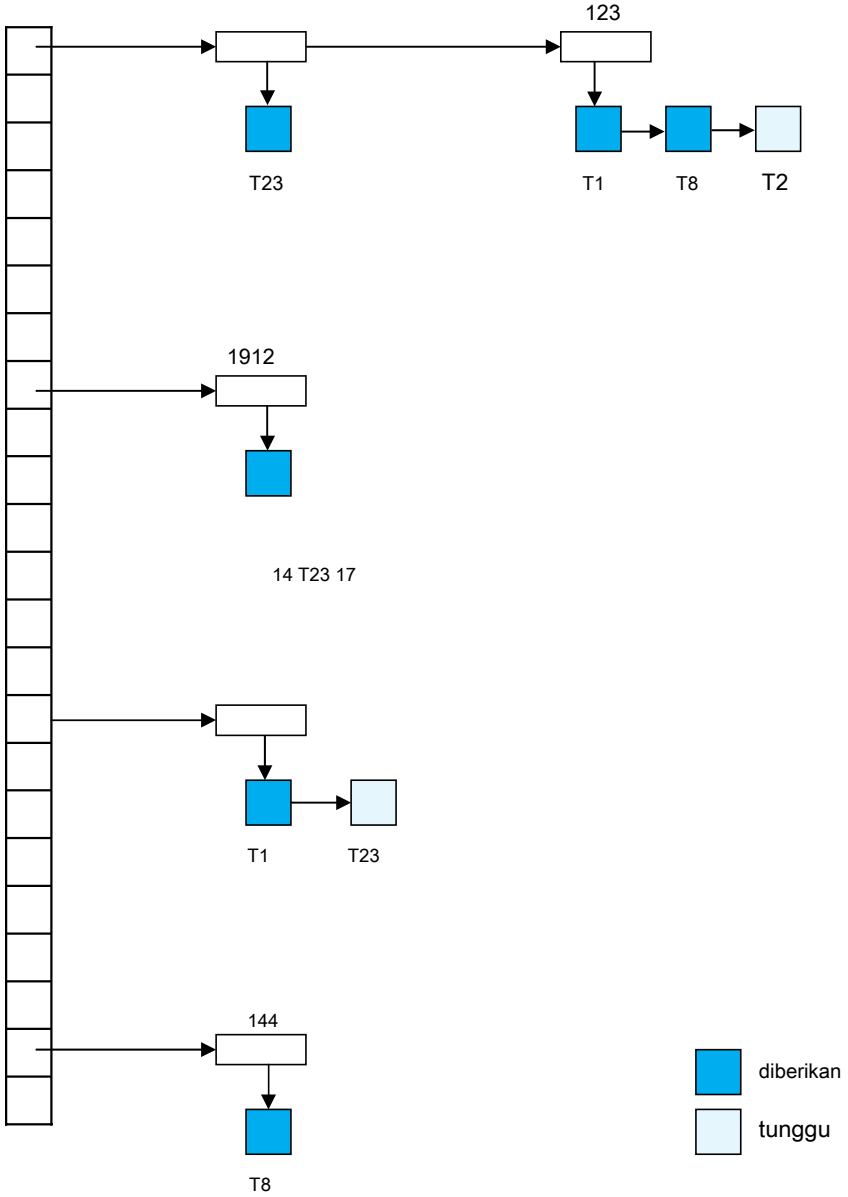
  akhir;

- Semua kunci dilepaskan setelah melakukan atau batalkan

Pelaksanaan Mengunci

- SEBUAH **manajer kunci** dapat diimplementasikan sebagai proses terpisah yang transaksi mengirim permintaan kunci dan membuka
- Manajer kunci balasan untuk permintaan kunci dengan mengirimkan pesan kunci hibah (atau pesan yang meminta transaksi roll kembali, dalam kasus kebuntuan)

- Transaksi meminta menunggu sampai permintaannya dijawab
- Manajer kunci mempertahankan struktur data yang disebut **tabel kunci** untuk merekam diberikan kunci dan permintaan yang tertunda
- Tabel kunci biasanya diimplementasikan sebagai di memori tabel hash diindeks pada nama dari item data yang terkunci

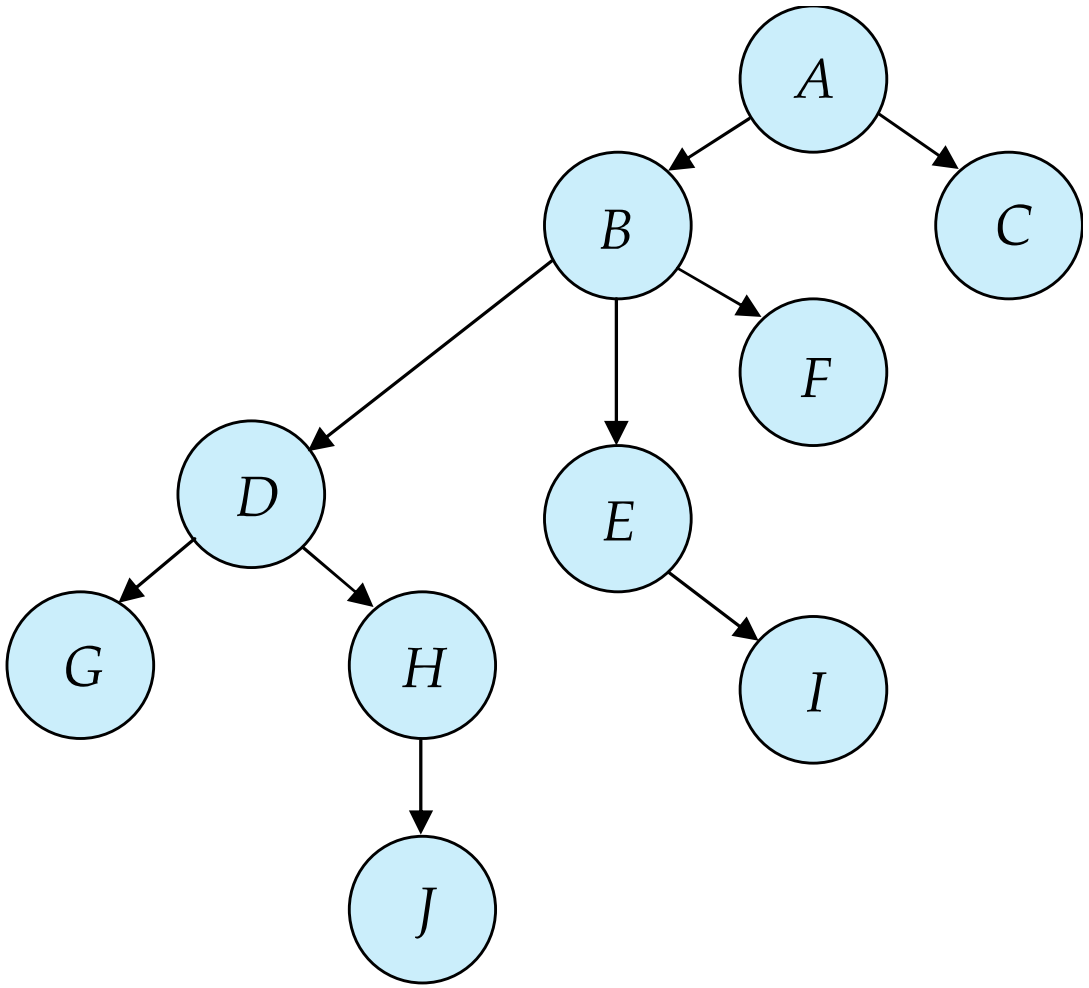


Lock Table

- Hitam persegi panjang menunjukkan kunci Memang, yang putih menunjukkan permintaan menunggu
- Tabel kunci juga mencatat jenis kunci diberikan atau diminta
- permintaan baru ditambahkan ke akhir antrian permintaan untuk item data, dan diberikan jika kompatibel dengan semua kunci sebelumnya
- Permintaan membuka mengakibatkan permintaan yang dihapus, dan permintaan kemudian diperiksa untuk melihat apakah mereka kini dapat diberikan
- Jika dibatalkan transaksi, semua permintaan tunggu atau diberikan transaksi dihapus
  - manajer kunci dapat menyimpan daftar kunci yang dipegang oleh setiap transaksi, untuk menerapkan ini secara efisien

Protokol Grafik Berbasis

- protokol berbasis grafik merupakan alternatif untuk penguncian dua-fase
- Memaksakan pemesanan parsial • di set **D = {** $d_1, d_2, ..., d_h$**}** semua item data.
  - **Jika** d_{saya} **•** d_j **maka setiap transaksi yang mengakses kedua** d_{saya} **dan** d_j **keharusan akses** d_{saya} **sebelum mengakses** d_j.
  - Menyiratkan bahwa himpunan **D** mungkin sekarang dilihat sebagai grafik asiklik diarahkan, yang disebut *grafik basis data.*
- Itu *pohon-protokol* adalah jenis sederhana protokol grafik.



tree Protocol

- Hanya kunci eksklusif diperbolehkan.
- Pertama kunci dengan T_{saya} mungkin pada setiap item data. Selanjutnya, data Q dapat dikunci dengan T_{saya} hanya jika induk Q saat ini dikunci oleh T_{saya}.
- Data item dapat dibuka setiap saat.
- Item data yang telah terkunci dan dibuka oleh T_{saya} tidak kemudian dapat relocked oleh T_{saya}

Grafik Berbasis Protokol (Cont.)

- Protokol pohon serializability Memastikan konflik serta kebebasan dari kebuntuan.

- Unlocking dapat terjadi sebelumnya dalam protokol pohon-mengunci dari dalam protokol dua tahap penguncian.
  - waktu tunggu lebih pendek, dan peningkatan concurrency
  - protokol adalah kebuntuan bebas, tidak ada rollbacks diperlukan
- Kerugian
  - Protokol tidak menjamin kebebasan pemulihan atau cascade
    - Perlu untuk memperkenalkan berkomitmen dependensi untuk memastikan pemulihan
  - Transaksi mungkin harus mengunci item data bahwa mereka tidak melakukan akses.
    - meningkat mengunci biaya overhead, dan waktu tunggu tambahan
    - potensi penurunan konkurensi
- Jadwal tidak mungkin di bawah penguncian dua-tahap yang mungkin di bawah protokol pohon, dan sebaliknya.

Penanganan deadlock

- Mempertimbangkan dua transaksi berikut:

  T_1: menulis (*X)*

  menulis(*Y)*

  T_2: menulis(*Y)*

  menulis(*X)*

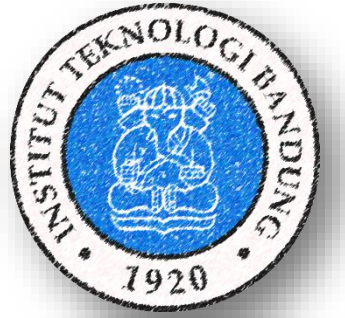
- Jadwal dengan kebuntuan

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Penanganan deadlock

- Sistem menemui jalan buntu jika ada satu set transaksi tersebut bahwa setiap transaksi di set menunggu untuk transaksi lain di set.
- ***pencegahan deadlock*** protokol memastikan bahwa sistem akan *tak pernah* masuk ke dalam keadaan deadlock. Beberapa strategi pencegahan:
  - Mengharuskan setiap transaksi mengunci semua item data sebelum dimulai eksekusi (predeclaration).
  - Memaksakan pemesanan parsial dari semua item data dan mengharuskan transaksi dapat mengunci item data hanya dalam urutan yang ditentukan oleh urutan parsial (protokol berbasis grafik).

Lebih Deadlock Strategi Pencegahan



- Skema berikut menggunakan cap waktu transaksi untuk kepentingan pencegahan kebuntuan saja.
- **menunggu-die** Skema - non-preemptive
  - transaksi yang lebih tua mungkin menunggu satu muda ke item rilis data. transaksi yang lebih muda tidak pernah menunggu lebih tua; mereka digulung kembali sebagai gantinya.
  - transaksi mungkin mati beberapa kali sebelum mengambilnya diperlukan item data
- **luka-tunggu** Skema - preemptive
  - transaksi yang lebih tua *luka (* Pasukan rollback) transaksi muda bukannya menunggu untuk itu. transaksi yang lebih muda mungkin menunggu yang lebih tua.
  - mungkin sedikit rollbacks dari *menunggu-die* skema.

pencegahan Deadlock (Cont.)

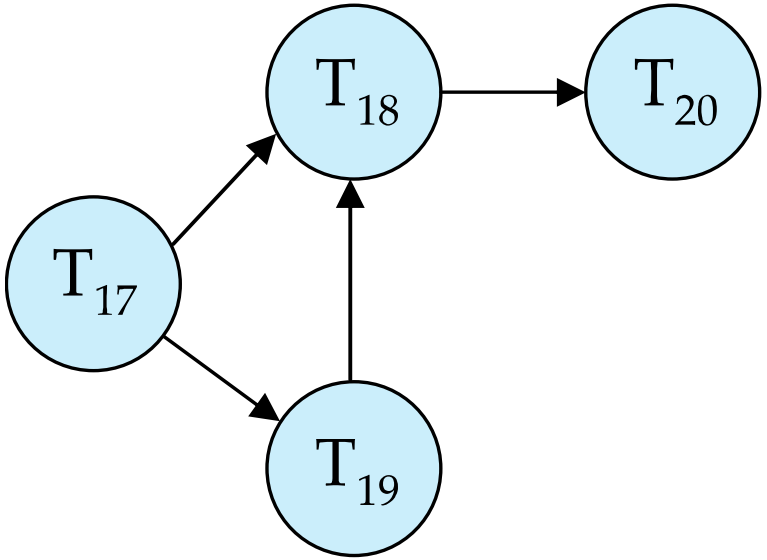
- baik dalam *menunggu-die* dan masuk *luka-tunggu* skema, transaksi kembali digulung-restart dengan timestamp aslinya. transaksi yang lebih tua sehingga didahului dengan orang-orang baru, dan kelaparan yang karenanya dihindari.

- Skema Timeout Berbasis :
  - transaksi menunggu kunci hanya untuk jumlah waktu tertentu. Setelah itu, menunggu kali dan transaksi yang terguling kembali.
  - sehingga deadlock tidak mungkin
  - sederhana untuk diterapkan; tapi kelaparan mungkin. Juga sulit untuk menentukan nilai yang baik dari interval timeout.

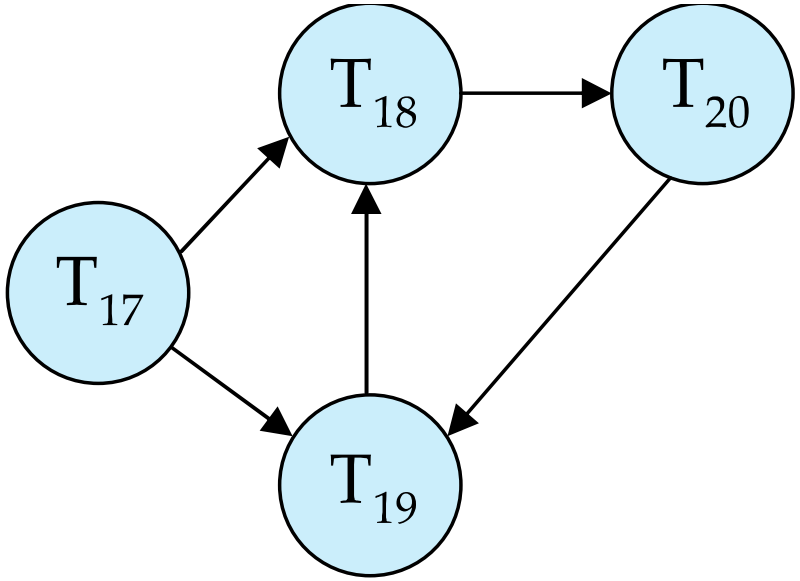
deadlock Detection

- Deadlock dapat digambarkan sebagai *menunggu-untuk grafik,* yang terdiri dari sepasang *G =* (*VE),*
  - *V* adalah seperangkat simpul (semua transaksi dalam sistem)
  - *E* adalah seperangkat tepi; setiap elemen adalah pasangan terurut $T_{saya} \bullet T_j$.
- Jika $T_{saya} \bullet T_j$ adalah di *E,* maka ada tepi diarahkan dari T_{saya} untuk T_j, menyiratkan bahwa T_{saya} menunggu T_j untuk merilis item data.
- Kapan T_{saya} permintaan item data saat ini ditahan oleh T_j, maka tepi $T_{saya}\ T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback : Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

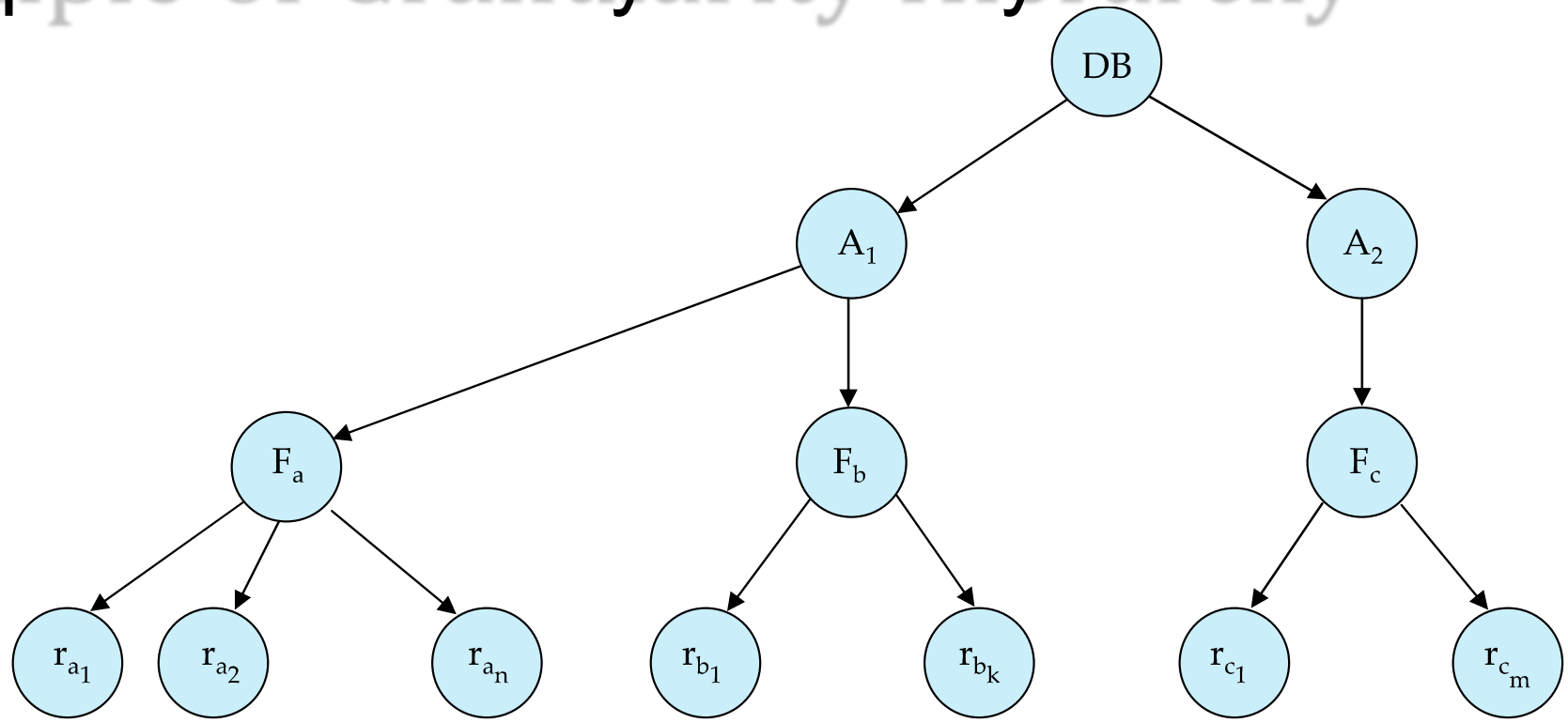
Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones

- Can be represented graphically as a tree (but don't confuse with treelocking protocol)

- When a transaction locks a node in the tree *explicitly,* it *implicitly* locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity (** lower in tree): high concurrency, high locking overhead
  - **coarse granularity (** higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

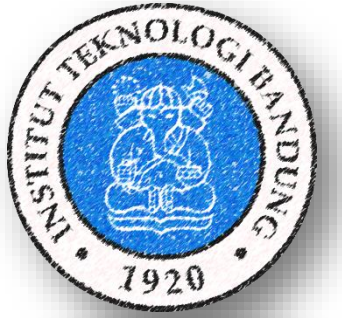
- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared (*** IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive (*** IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive (*** SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

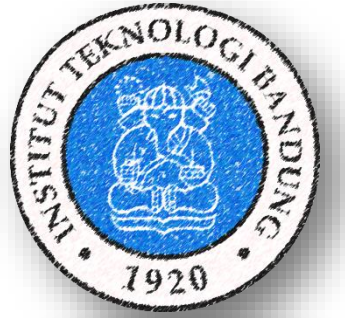
	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme



- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation** : in case there are too many locks at a particular level, switch to higher granularity S or X lock

Timestamp-Based Protocols



- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned timestamp TS(T_j) such that TS(T_i) < TS(T_j).

- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp(** Q**)** is the largest time-stamp of any transaction that executed **write(** Q**)** successfully.
  - **R-timestamp(** Q**)** is the largest time-stamp of any transaction that executed **read(** Q**)** successfully.

Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read(** Q**)**
  1. If TS(T_j) • **W-** timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If TS(T_j) • **W-** timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max(** R-timestamp(Q), TS(T_j)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write(** Q**).**
  1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
     - Hence, the **write** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

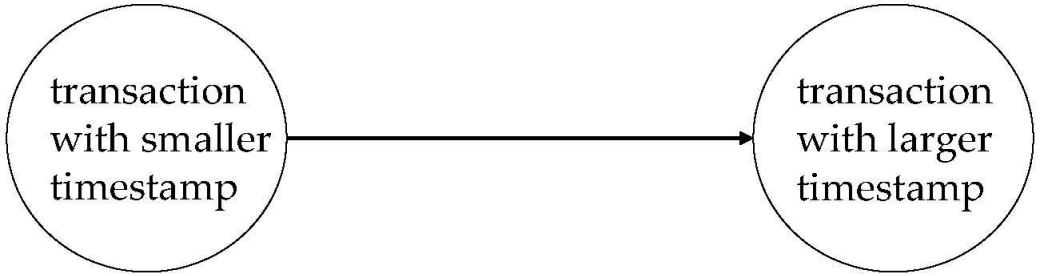
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 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)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas ' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of { Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this { **write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,

1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
2. If in schedule S transaction T_i executes **read(** Q**),** and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write(** Q) operation of transaction T_j.
3. The transaction (if any) that performs the final **write(** Q**)** operation in schedule S must also perform the final **write(** Q**)** operation in schedule S'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

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

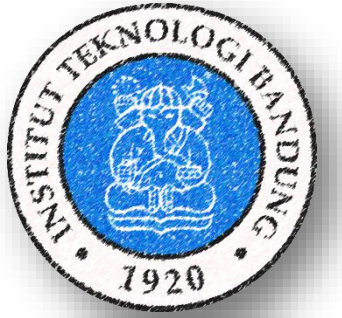
- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes** .

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.

- The problem of checking if a schedule is view serializable falls in the class of *NP*- complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Other Notions of Serializability



- The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

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)

- Determining such equivalence requires analysis of operations other than read and write.
  - Operation-conflicts, operation locks

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase:** Transaction T_i writes only to temporary local variables
  2. **Validation phase:** Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
  3. **Write phase:** If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish(** T_i) < **start(** T_j)
  - **start(** T_j) < **finish(** T_i) < **validation(** T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification:* Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B$ 50
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

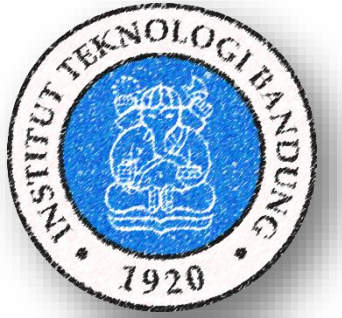
Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.

- Use timestamps to label versions.
- When a **read(** *Q)* operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.

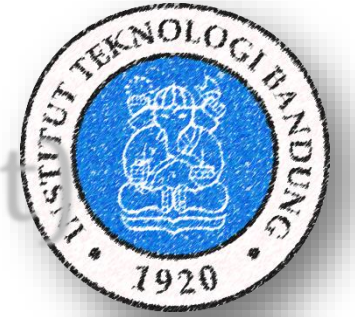
- **read** s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering



- Each data item Q has a sequence of versions $< Q_1, Q_2, ..., Q_m >$. Each version Q_k contains three data fields:
  - **Content --** the value of version Q_k.
  - **W-timestamp(** Q_k**)** – timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp(** Q_k**)** – largest timestamp of a transaction that successfully read version Q_k
- when a transaction T_i creates a new version Q_k of Q, Q_k's W- timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering (Cont)

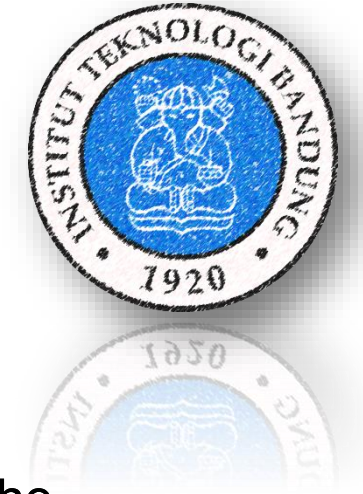


- Suppose that transaction T_i issues a **read(** Q**)** or **write(** Q**)** operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

  1. If transaction T_i issues a **read(** Q**),** then the value returned is the content of version Q_k.
  2. If transaction T_i issues a **write(** Q**)**
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. else a new version of Q is created.

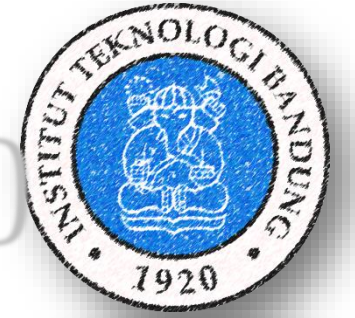
- Observe that
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read
    T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability

Multiversion Two-Phase Locking



- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Multiversion Two-Phase Locking (Cont.)



- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to •.

- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter +** 1
  - T_i increments **ts-counter** by 1
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.

- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

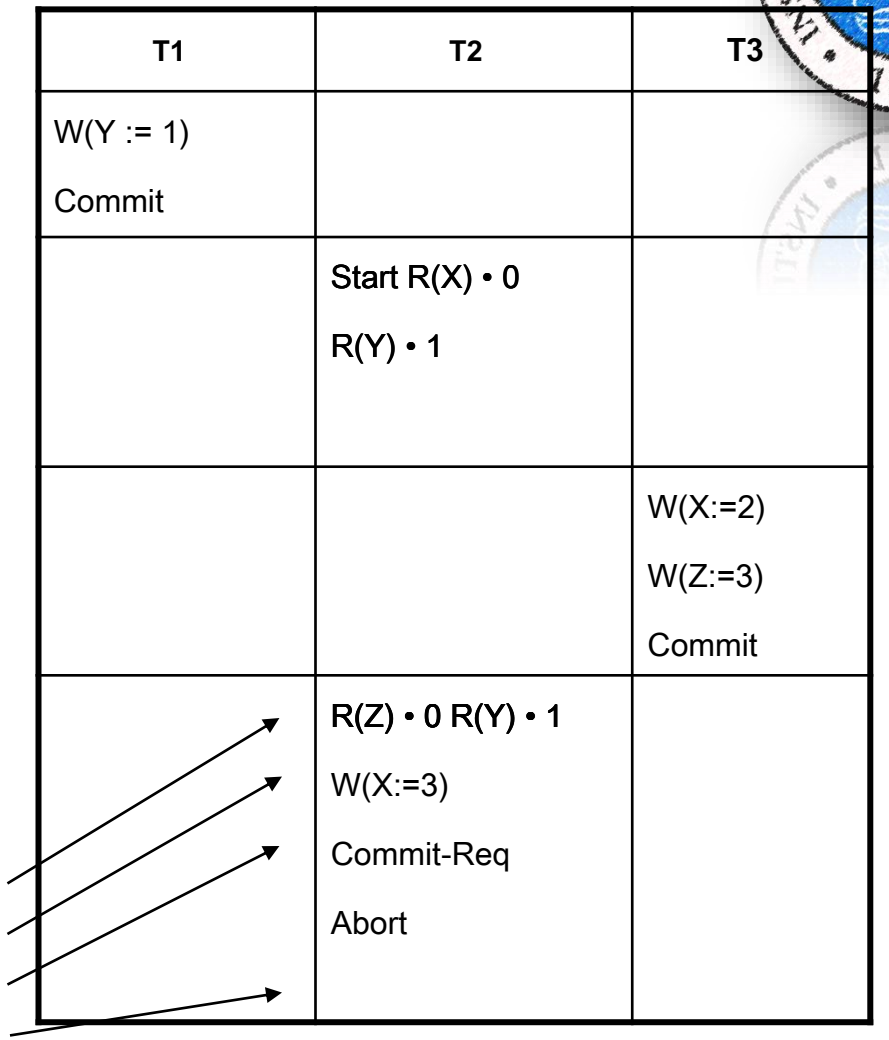
- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Give logical “ snapshot ” of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- Solution 2: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule:**
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.



Concurrent updates not visible

Own updates are visible Not

first-committer of X

Serialization error, T2 is rolled back

Snapshot Read

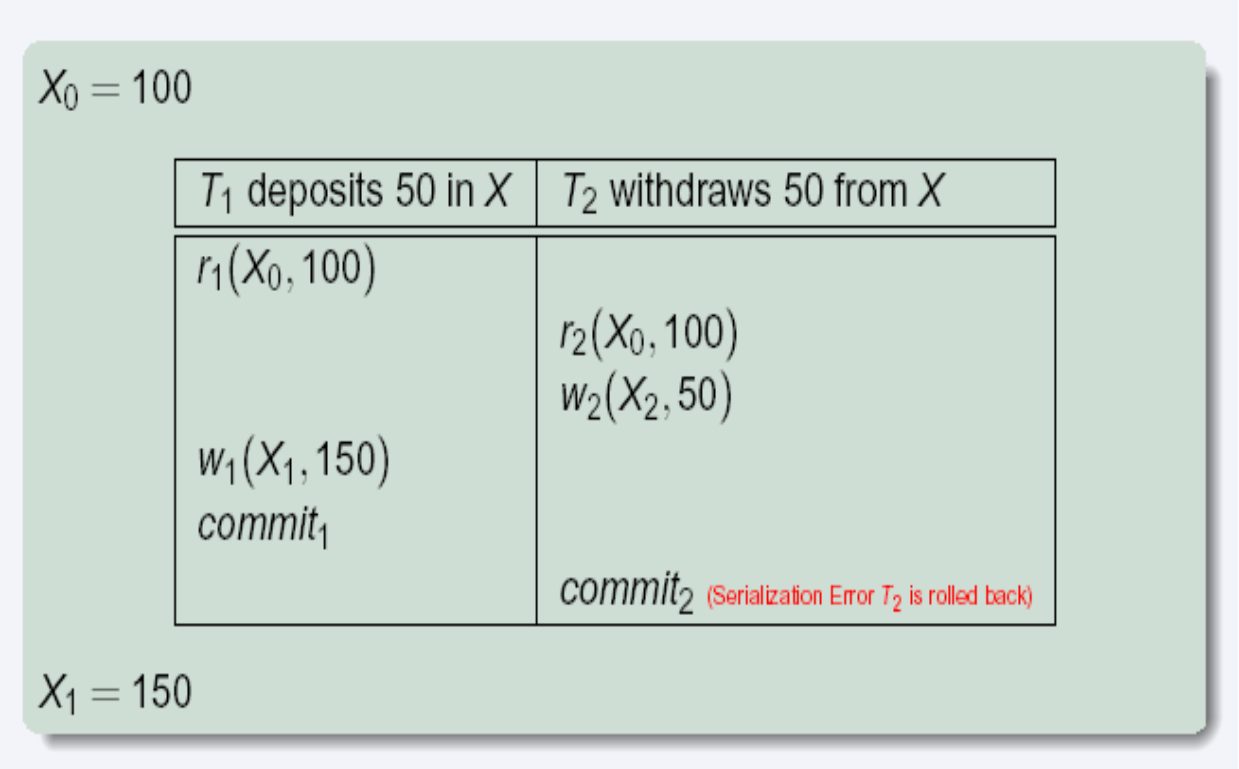
- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

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 Write: First Committer Wins



- Variant: " **First-updater-wins** "
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

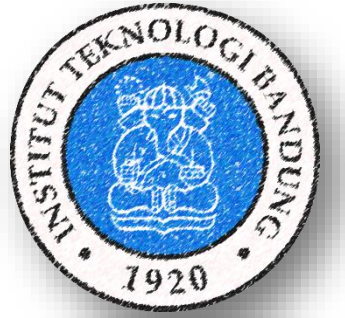
Benefits of SI

- Reading is *never* blocked,
  - and also doesn ’ t block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- E.g. of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

Snapshot Isolation Anomalies



- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot

SI In Oracle and PostgreSQL

- **Warning :** SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1
  - PostgreSQL ' s implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
  - Oracle implements " first updater wins " rule (variant of " first committer wins ")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
    - Oracle and PostgreSQL < 9.1 do not support true serializable execution
  - PostgreSQL 9.1 introduced new protocol called "Serializable Snapshot Isolation" (SSI)
    - Which guarantees true serializabilty including handling predicate reads (coming up)

SI In Oracle and PostgreSQL

- Can sidestep SI for specific queries by using **select .. for update** in Oracle and PostgreSQL
  - E.g.,

    1. select max(orderno) **from** orders **for update**

    2. read value into local variable maxorder

    3. insert into orders (maxorder+1, …)
  - Select for update (SFU) treats all data read by the query as if it were also updated, preventing concurrent updates
  - Does not always ensure serializability since phantom phenomena can occur (coming up)
- In PostgreSQL versions < 9.1, SFU locks the data item, but releases locks when the transaction completes, even if other concurrent transactions are active
  - Not quite same as SFU in Oracle, which keeps locks until all
  - concurrent transactions have completed

Insert and Delete Operations

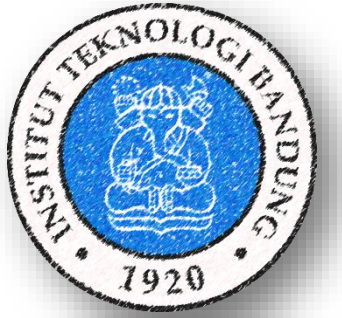
- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple

- Insertions and deletions can lead to the **phantom phenomenon** .
  - A transaction that scans a relation
    - (e.g., find sum of balances of all accounts in Perryridge) and a

    transaction that inserts a tuple in the relation
    - (e.g., insert a new account at Perryridge)

    (conceptually) conflict in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - E.g. the scan transaction does not see the new account, but reads some other tuple written by the update transaction

Insert and Delete Operations (Cont.)



- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

Index Locking Protocol

- Index locking protocol:
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
    - must update all indices to r
    - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won ' t occur

Next-Key Locking

- Index-locking protocol to prevent phantoms required locking entire leaf
  - Can result in poor concurrency if there are many inserts
- Alternative: for an index lookup
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures that range queries will conflict with inserts/deletes/updates
  - Regardless of which happens first, as long as both are concurrent

Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a B+-tree are irrelevant so long as we land up in the correct leaf node.

Concurrency in Index Structures (Cont.)

- Example of index concurrency protocol:
- Use **crabbing** instead of two-phase locking on the nodes of the B+-tree, as follows. During search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node.
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- Better protocols are available; see Section 16.9 for one such protocol, the B-link tree protocol
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

Weak Levels of Consistency

- **Degree-two consistency** : differs from two-phase locking in that Slocks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur
- **Cursor stability** :
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable** : is the default
  - **Repeatable read** : allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed** : same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted** : allows even uncommitted data to be read
- In many database systems, read committed is the default consistency level
  - has to be explicitly changed to serializable when required
    - **set isolation level serializable**

Transactions across User Interaction

- Many applications need transaction support across user interactions
  - Can ' t use locking
  - Don ' t want to reserve database connection per user
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - **select** r.balance, r.version **into :** A, :version
      from r **where** acctId =23
  - When writing tuple, check that current version number is same as the version when tuple was read
    - **update** r **set** r.balance = r.balance + :deposit
      where acctId = 23 **and** r.version = :version
- Equivalent to **optimistic concurrency control without validating read set**
- Used internally in Hibernate ORM system, and manually in many applications
- Version numbering can also be used to support first committer wins check of snapshot isolation
  - Unlike SI, reads are not guaranteed to be from a single snapshot

End of Chapter

Thanks to Alan Fekete and Sudhir Jorwekar for Snapshot Isolation examples