ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 10η: Transactions - part 3

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

Locking on Hierarchies of DB Elements

- Assume there is a tree structure to the data
  - Hierarchy of lockable elements (relations, tuples, attributes)
  - Data organized in a tree (e.g., a B+-tree)
- Locking schemes seen so far perform poorly in such cases
- 3 levels of DB elements:
  - Relations are the largest lockable elements
  - Each relation comprises one or more blocks
  - Each block contains one or more tuples
- Need a new type of lock, called a warning lock

Warning Locks

- Ordinary locks: S (shared), X (exclusive)
- Warning locks denote the intention to obtain a lock
  - IS: intention to obtain a shared lock
  - IX: intention to obtain an exclusive lock
- Rules:
  - To place a S or X lock, start at the root of the hierarchy
  - If at the element that we want to lock, request S or X lock
  - If the element is further down the hierarchy, place a warning of the appropriate kind at the current node
  - Proceed to the appropriate child node and repeat until the desired node is reached

Warning Locks

- Compatibilities:
  - An IS on a node N is only incompatible with an X lock on N
  - An IX on a node N is incompatible with S and X on N
  - Potential conflicts that may arise with IS/IS, IS/IX, IX/IS and IX/IX will be resolved at a lower level
  - An S on N is compatible with IS and S
  - An X on N is incompatible with every other type of lock
- Can only lock existing items, but not items that might later be inserted.
- To handle insertions / deletions:
  - Get exclusive lock on A before deleting A
  - At insert A operation by Ti,Ti is given exclusive lock on A

Phantoms

- Phantoms are tuples that should have been locked but they weren't because they didn't exist at the time the locks were granted
- Example:
  - relation R (E#,name,…)
  - constraint: E# is key
  - use tuple locking

R	E#	Name	….
o1	55	Smith	
o2	75	Jones	

Example

T1: Insert <99,Gore,…> into R

T2: Insert <99,Bush,…> into R

T_1	T_2
$S_1(o_1)$	$S_2(o_1)$
$S_1(o_2)$	$S_2(o_2)$
Check Constraint	Check Constraint
⋮	⋮
Insert o_3[99,Gore,..]	
	Insert o_4[99,Bush,..]

Example

Solution relies on using multiple granularities: before insertion at node N, lock parent of N in exclusive mode

T_1	T_2
$X_1(R)$	
	$X_2(R)$ wait
Check Constraint	
Insert o_3[99,Gore,..]	
U(R)	
	$X_2(R)$
	Check Constraint
	tuple with e# 99 already exists

Tree Protocols

- Data elements may be organized in a tree-shaped structure (due to their inter-link pattern)
  - e.g., data organized in a B-tree
- Despite the fact that DB elements are disjoint, independent pieces of data, the only way to access a particular node in a tree-shaped structure is through the parent node
- Different opportunities for applying locking policies along paths to data elements arise
  - 2PL is too rigid in such cases
  - assuming that the granularity of locking is that of tree nodes, concurrent access to the B-tree is close to impossible
  - each transaction starts at the root and locks all nodes in a path; root cannot be unlocked until all locks are obtained

Tree Protocols

- More problems with 2PL
  - inserts or deletes may cause changes to the root node; exclusive or update locks must be used for the root node
  - read-only transactions will be able to execute concurrently
  - more opportunities would arise if the lock to the root could be released early
  - … but this could violate 2PL and serializability
- Need specialized protocols for tree-structured data
- These protocols may violate 2PL
- Must exploit the order of access to nodes for ensuring serializability

Rules of Access to Tree-Structured Data

- The Tree Protocol
  - assumptions: single lock mode; consistent transactions; legal schedules
  - Rules:

1. The first lock of any transaction may be at any tree node
2. Subsequent locks are granted to a transaction only if the transaction already holds a lock on the parent node
3. Nodes can be unlocked at any time
4. Nodes that are locked by a transaction may not be relocked (even if there still exists a lock on the node's parent)

Does it Work?

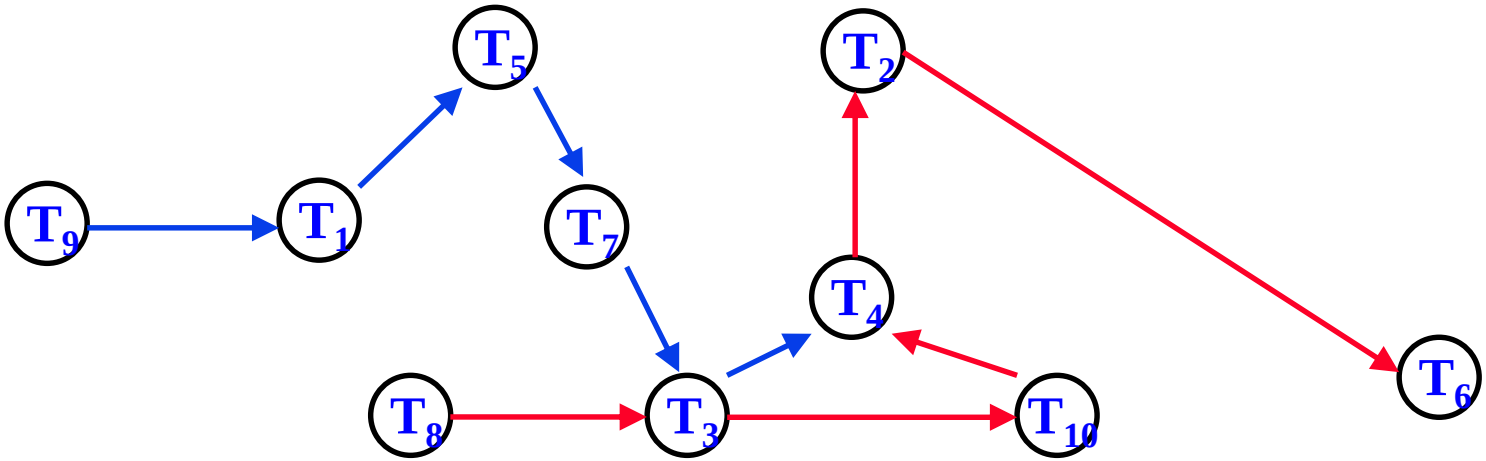
- It enforces a serial order in the execution of transactions:
  - if in schedule S, Ti and Tj lock a common node and Ti locks it first, then Ti precedes Tj
- A precedence graph can be constructed to represent these precedence relations among transactions in a schedule
- If it is acyclic, any topological order of the transactions is an equivalent serial schedule
- Claim: schedules following the tree protocol always produce acyclic precedence graphs
  - if two transactions lock several common elements, then they are all locked in the same order
  - if Ti locks the root before Tj, then Ti locks every node that is common to Ti and Tj before Tj does.

Ensuring Serializability

- We can prove that, for every schedule S following the tree protocol, there is some serial order equivalent to S
- Proof sketch: (by induction on the number of tree nodes)
  - Base case: 1 node (root); follows by the previous observations
  - Induction hypothesis: there is a serial order for all the transactions that lock nodes in any subtree containing more than one node
  - Induction: Must merge the serial orders of the subtrees. Consider for each subtree the set of transactions that lock one or more nodes in the subtree. These sets have as common elements the transactions that lock the root. These transactions lock every common node in the same order they lock the root. Build a serial order for the entire set by starting with the transactions that lock the root in the appropriate order. The rest only need to be placed consistently with the serial order of their subtrees.

Ensuring Serializability

- Example: Assume there are 10 transactions, T_1, T_2, …, T_{10} of which T_1, T_3, T_4, T_6 lock the root in this order. Assume that the root node has two children nodes: first node is locked by the transactions with an odd index and T_4 and the second is locked by the transactions with an even index and T_3.
  - Order in the 1st subtree: T_9, T_1, T_5, T_7, T_3 ,T_4
  - Order in the 2nd subtree: T_8, T_3, T_{10} , T_4, T_2, T_6



Timestamp-based Concurrency Control

- Basic ideas:
  - each transaction is assigned a timestamp
  - the timestamps of the transactions that last read and write each DB element are recorded
  - these values are compared for ensuring that the serial schedule according to the transactions' timestamps is equivalent to the actual schedule
- An optimistic approach: it assumes that no non-serializable will appear; problems are fixed only when some violation occurs
  - fixing problems involves only abort and restart
- Locking-based methods are pessimistic (preventive)
- Optimistic scheduling is better than locking when many of the transactions are read-only

Timestamps

- Each transaction T is assigned a unique timestamp TS(T)
- Timestamps are assigned in ascending order when transactions begin
- Timestamps are generated by:
  - using the system clock
  - a counter maintained by the scheduler: counter is incremented by 1 each time a transaction starts
- Scheduler must maintain a table of active transactions and their respective timestamps

Timestamps

- In order to use timestamps for concurrency control, each DB element is associated with two timestamps and an additional bit:
  - RT(X): read time of X = highest timestamp of a transaction that has read X
  - WT(X): write time of X = highest timestamp of a transaction that has written X
  - C(X): commit bit for X = 1 iff the most recent transaction to write X has already committed
    - This bit is used in order to avoid a situation where some transaction T reads data written by a transaction U that later aborts (dirty read on uncommited data)

Problems

- Scheduler assumes that the timestamp order of transactions is also the serial order in which they must appear to execute
- Scheduler must check that whenever a read / write occurs, what happens could have happened if each transaction executed instantaneously at the moment of its timestamp; if not problems may occur:
  - Read too late: T tries to read X, but WT(X) indicates that X was written after T executed, i.e., TS(T) < WT(X). T must not read X and the scheduler must abort T.
  - Write too late: T tries to write X, but RT(X) indicates that some other transaction should have read the value written by T but it read some other value (WT(X) < TS(T) < RT(X))

More Problems

- T reads X and X was last written by U; TS(U) < TS(T); What if after T reads X (the value written by U), U aborts?
  - must delay T's read until U commits or aborts
  - can check C(X) to determine whether U has committed
- TS(T) < TS(U) and U writes X first; when T tries to write, do nothing
  - Thomas write rule: writes can be skipped when a write with a later write-time is already in place
- Policy: when T writes element X, the write is considered tentative and may be undone if T aborts; C(X) is set to 0 and the scheduler makes a copy of the old value of X and its previous WT(X)

Timestamp-based Scheduling Rules

1. If the scheduler receives a request by transaction T to read X then:
   a. If TS(T) >= WT(X) then
      i. If C(X)=1, grant the request. If TS(T) > RT(X), set RT(X):=TS(T); otherwise do not change RT(X)
      ii. If C(X)=0, delay T until C(X) becomes 1 or the transaction that wrote X aborts
   b. If TS(T) < WT(X) then abort T and restart it with a new, higher timestamp

Timestamp-based Scheduling Rules

2. If the scheduler receives a request by transaction T to write X then:
   a. If TS(T) >= RT(X) and TS(T) >= WT(X) then
      i. Write the new value for X
      ii. Set WT(X):=TS(T)
      iii. Set C(X):=0
   b. If TS(T) >= RT(X) but TS(T) < WT(X) then there is already a later value for X.
      i. If C(X)=1, ignore the write by T
      ii. If C(X)=0, delay T

Timestamp-based Scheduling Rules

3. If the scheduler receives a request by transaction T to commit then, all the DB elements written by T must be found and their commit bit must be set to 1
4. If the scheduler receives a request by transaction T to abort or T must be rolled back then, any transaction waiting on X that T wrote must repeat its attempt to read or write

- Variations of timestamp based scheduling that use multiple versions of the DB can be used when DB elements are disk blocks or pages

Timestamp-based Scheduling vs Locking

- Timestamping outperforms locking in situations where either the majority of transactions are read-only or conflicting operations are scarce
- Locking performs better in high-conflict situations:
  - Locking will delay transactions as they wait for locks; even if deadlock occurs, one of the transactions will be rolled back
  - But, if rollbacks are frequent, even more delay will be incurred
- Commercial systems follow a middle-of-the-road approach:
  - Transactions are divided in read-only and read/write
  - Read/write are executed in 2PL; read-only are executed using timestamp-based concurrency control

Validation-based Concurrency Control

- Another optimistic concurrency control strategy:
  - Transactions are allowed to access data without locks
  - Check whether transactions have executed in a serializable manner
  - Just before a transaction starts to write values of DB elements, it goes through a validation phase where the sets of elements it has read and will write are compared with the write sets of other transactions
- Transactions execute in 3 phases
  - Read
  - Validate
  - Write

Validation-based Concurrency Control

- Each transaction that successfully validates may be thought as executing at the moment that it validates
- A validation-based scheduler can use the serial order implied by the validation times of transactions in order to determine whether the transactions' behaviors are consistent with it
- Scheduler maintains three sets:
  - START: set of transactions that have started but not yet completed validation; START(T):time at which T started
  - VAL: set of transactions that have been validated but not yet finished the writing phase; VAL(T):time at which T validated
  - FIN: set of transactions that have completed phase 3; FIN(T): time at which T finished

Validation Rules

- RS(T): the set of DB elements that T reads
- WS(T): the set of DB elements that T writes
- When the validation of T is attempted:
  1. Compare RS(T) with WS(U) for every U s.t. FIN(U) > START(T) and check whether RS(T) $\cap$ WS(U) = $\varnothing$
  2. Compare WS(T) with WS(U) for every U s.t. FIN(U) > VAL(T) check whether WS(T) $\cap$ WS(U) = $\varnothing$

Comparison of Concurrency Control Mechanisms

- Storage Utilization:
  - Locking: space required in the lock table is proportional to the number of DB elements locked
  - Timestamps:
    - Naïve approach: store read and write times for all DB elements
    - Timestamps earlier than that of the earliest active transaction do not matter; only the timestamps for the recently accessed elements are needed
  - Validation: read/write sets and timestamps for each active transaction
  - Timestamping and validation may use a little more space than locking

Comparison of Concurrency Control Mechanisms

- Delay Effects: depend on the degree of interaction, i.e., the likelihood that a transaction will access an element that is also being accessed by another transaction
  - Locking delays transactions but avoids rollbacks (except in deadlock situations) even when interaction is high. Timestamps and validation do not delay transactions but may cause them to rollback, which in turn results in more delays
  - If interaction is low, neither timestamps nor validation will cause many rollbacks and may be preferable to locking
  - If rollback becomes necessary, timestamps discover problems earlier than validation (which permits transactions to perform all work before deciding whether it needs to be rolled back)

View Serializability

- Conflict serializability guarantees serializability independently of what transactions actually do
- But…. it is a quite strong condition on schedules of concurrent execution
- A weaker notion that still guarantees serializability is view serializability
- The main difference between conflict and view serializability appears in situations where a transaction writes a value that no other transaction reads (but possibly writes later)
  - under view serializability such a write action can be repositioned in the schedule (might be prohibited under conflict serializability)

Definitions

- Let S_1 and S_2 be two schedules involving the same set of transactions;
  - let T_I be a hypothetical transaction that writes initial values for each DB element read by any transaction in the schedule;
  - let T_F be a hypothetical transaction that reads every element written by the transactions after the schedule ends
  - for every $R_i(A)$, we can find the $W_j(A)$ that most closely precedes it; transaction T_j is called the source of the read action
- S_1 and S_2 are called view equivalent if for every read action in S_1 its source is the same in S_2, and vice versa.
- A schedule that is view equivalent to a serial schedule is called view serializable

Example

- Consider the following schedule S:

T_1:				$R_1(A)$			$W_1(B)$		
T_2:		$R_2(B)$	$W_2(A)$					$W_2(B)$	
T_3:					$R_3(A)$				$W_3(B)$

  - only the value of B written by T_3 is read by T_F
  - S is not conflict serializable
  - sources for read actions:
    - source of $R_2(B)$ is T_I
    - source of $R_1(A)$ and $R_3(A)$ is T_2
    - source of $R_F(A)$ is T_2
    - source of $R_F(B)$ is T_3

- S is view serializable;
- equivalent serial schedule: (T_2 , T_1 , T_3)

Testing View Serializability

- Polygraphs: a generalization of precedence graphs
  - a node for each transaction and additional nodes for the two hypothetical transactions
  - an arc from T_j to T_i for each action $R_i(X)$ with source T_j
  - if T_j is the source of $R_i(X)$ and T_k is another transaction that writes X, T_k must appear either before T_j or after T_i . This is denoted by two arcs in the graph (can choose one of the two).

    Special cases:
    - if T_j is T_I, then the arc from T_i to T_k is introduced
    - if T_i is T_F, then the arc from T_k to T_j is introduced

Τέλος Ενότητας

Χρηματοδότηση

•Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.

•Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.

•Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημείωμα αδειοδότησης

•Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση, Όχι Παράγωγο Έργο 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό και τα οποία αναφέρονται μαζί με τους όρους χρήσης τους στο «Σημείωμα Χρήσης Έργων Τρίτων».

[1] http://creativecommons.org/licenses/by-nc-nd/4.0/

•Ως **Μη Εμπορική** ορίζεται η χρήση:

–που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο

–που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο

–που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

•Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

.

Σημείωμα Αναφοράς