Chapter 15 : Concurrency Control

Database System Concepts, 6th Ed.

Outline

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity

Lock-Based Protocols
Πρωτόκολλα Βασισμένα στο κλείδωμα

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

1. *exclusive (X) mode (Αποκλειστικό κλείδωμα)*. Data item can be both read as well as written.
   X-lock is requested using **lock-X** instruction.

2. *shared (S) mode (Κοινόχρηστο κλείδωμα)*. Data item can only be read.
   S-lock is requested using **lock-S** instruction.

- Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted (παραχωρείται) a lock on an item if the requested lock is **compatible with locks already held on the item** by other transactions
- Any number of transactions can hold shared locks on an item,
  - But if any transaction holds an **exclusive** on the item **no other transaction may hold any lock** on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A*+*B*)

- Locking as above is not sufficient to guarantee **serializability** — if ***A*** and ***B*** get **updated in-between** the **read of *A* and *B***, the displayed sum would be wrong.
- A **locking protocol (πρωτόκολλο κλειδώματος)** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules (χρονοδιαγραμμάτων).

The Two-Phase Locking Protocol
Πρωτόκολλο κλειδώματος Δυο Φάσεων

- This protocol **ensures conflict-serializable schedules** (χρονοδιαγράμματα σειριοποιήσιμα ως προς τις διενέξεις).
- Phase 1: Growing Phase (Φάση ανάπτυξης)
  - Transaction may obtain locks
  - Transaction **may not** release locks
- Phase 2: Shrinking Phase (φάση σύμπτυξης)
  - Transaction may release locks
  - Transaction **may not** obtain locks
- The protocol assures serializability (σειριοποιησιμότητα διένεξης). It can be proved that the transactions can be serialized in the order of their **lock points (σημεία κλειδώματος)** (i.e., the point where a transaction acquired its final lock = **this is the end of the growing phase of the transaction**).

The Two-Phase Locking Protocol (Cont.)
Πρωτόκολλο κλειδώματος Δυο Φάσεων

- There can **be conflict serializable schedules** that cannot be obtained if **two-phase locking is used**.
  - Cause serializability is achieved in the order of the **lock points of the transaction (= the end of the growing phase)**
- However, in the absence of extra information (e.g., ordering of access to data), **two-phase locking** is needed for conflict serializability in the following sense:
  - Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Lock Conversions
Μετατροπές Κλειδωμάτων

- Two-phase locking with lock conversions(κλείδωμα δυο φάσεων με μετατροπές κλειδωμάτων) – **allows more concurrency**:
  - First Phase (Growing Phase):
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade-αναβάθμιση)
  - Second Phase (Shrinking Phase):
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade - υποβάθμιση)
- This protocol **assures serializability**. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks
Σχήμα αυτόματου κλειδώματος - Read

- This scheme automatically creates lock according to read / write requests.
- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

    if T_i has a lock on *D*

    then

    read(*D*)

    else begin

    if necessary **wait until no other transaction has** a **lock-X** on *D*

    grant T_i a **lock-S** on *D*;

    read(*D*)

    end

Automatic Acquisition of Locks (Cont.) Σχήμα αυτόματου κλειδώματος - Write

- **write**(*D)* is processed as:

 if T_i has a **lock-X** on *D*
 then
 write(*D*)
 else begin
 if necessary **wait until no other transaction has any lock on *D***,
 if T_i has a **lock-S** on *D*
 then
 upgrade lock on *D* to **lock-X**
 else
 grant T_i a **lock-X** on *D*
 write(*D*)
 end;

- All **locks are released after commit or abort**

Deadlocks
Αδιέξοδες Καταστάσεις

- Consider the partial schedule

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*)	

- Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock (αδιέξοδο)**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlocks (Cont.) Αδιέξοδες Καταστάσεις

- Two-phase locking ***does not*** **ensure freedom from deadlocks** (see previous example).
- In addition to deadlocks, there is a possibility of **starvation (Επ' αόριστο αναμονή).**
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

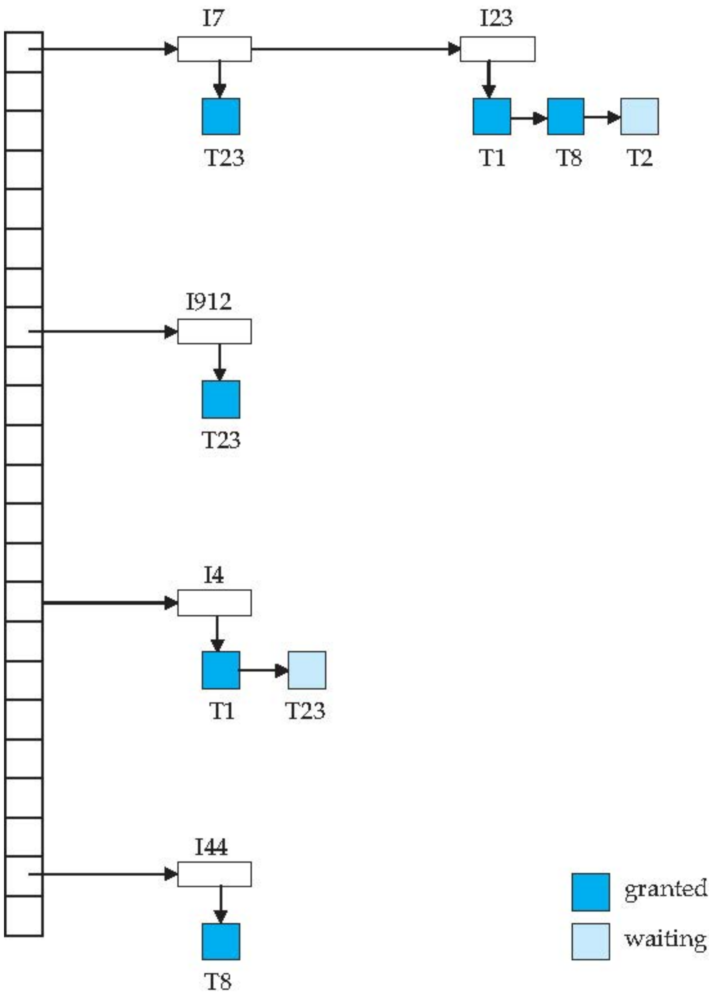
Deadlocks (Cont.)
Αδιέξοδες Καταστάσεις

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a **possibility of cascading roll-backs**.
- **Avoiding cascading roll-backs**:
  - Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.
  - **Rigorous two-phase locking** is even stricter. Here, *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Implementation of Locking (Χειρισμός Κλειδωμάτων)

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock **grant messages-μηνύματα παραχώρησης** (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table (πίνακας κλειδωμάτων)** to record **granted locks (παραχωρημένα κλειδώματα)** and **pending requests (αιτήματα σε αναμονή).**
- The lock table is usually implemented as an in-memory **hash table** indexed on the **name of the data item** being locked

Lock Table (Πίνακας Κλειδωμάτων)



- Hash index on name of the data item
- **Dark blue** rectangles indicate granted locks **(παραχωρημένα κλειδώματα)**; **light blue** indicate waiting requests**(αιτήματα σε αναμονή).**
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If **transaction aborts**, all **waiting or granted requests of the transaction** are deleted
  - lock manager may also keep a list of locks held by **each transaction**, to implement this efficiently

Deadlock Handling (Χειρισμός Κλειδωμάτων)

- System is deadlocked if **there is a set of transactions** such that **every transaction in the set** is waiting for **another transaction in the set**.
- ***Deadlock prevention*** (Αποτροπή Αδιέξοδης Κατάστασης) protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration- προ-δήλωση).
  - Impose **partial ordering of all data items** and require that a transaction can lock data items only in the order specified by the partial order. (After a lock on a specific item the transaction can not require to lock items ordered before that one)

More Deadlock Prevention Strategies (Στρατηγικές Αποτροπής Αδιέξοδης Κατάστασης)

- Following schemes use transaction timestamps for the sake of deadlock prevention alone. **They consider the older as more important!**
- **wait-die(αναμονή-τερματισμός)** scheme — non-preemptive (τεχνική χωρίς αντικατάσταση)
  - **older transaction may wait** for younger one to release data item. (older means smaller timestamp) Younger transactions never. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** (τραυματισμός-αναμονή) scheme — preemptive (τεχνική αντικατάστασης)
  - older transaction *wounds* (forces rollback) of a younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

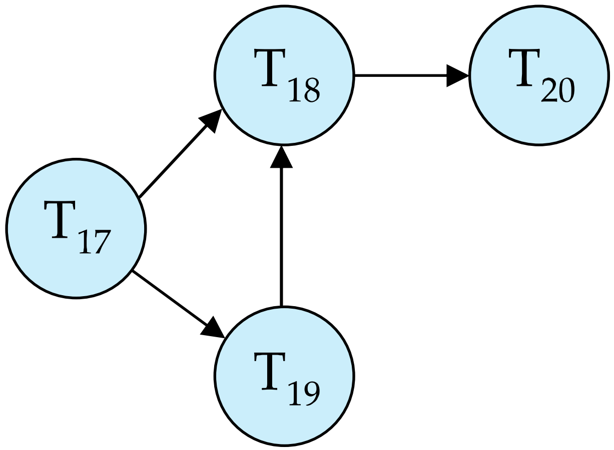
Deadlock prevention (Cont.) (Στρατηγικές Αποτροπής Αδιέξοδης Κατάστασης)

- Both in *wait-die* and in *wound-wait* schemes, **a rolled back transactions is restarted with its original timestamp**. Older transactions thus have precedence over newer ones, and **starvation is hence avoided**.
- **Another Strategy - Timeout-Based Schemes (Λήξη Χρόνου Κλειδωμάτων):**
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is **rolled back** and restarted,
  - Thus, deadlocks are not possible
  - simple to implement; but **starvation (Επ' αόριστο αναμονή) is possible**. Also difficult to determine good value of the timeout interval.

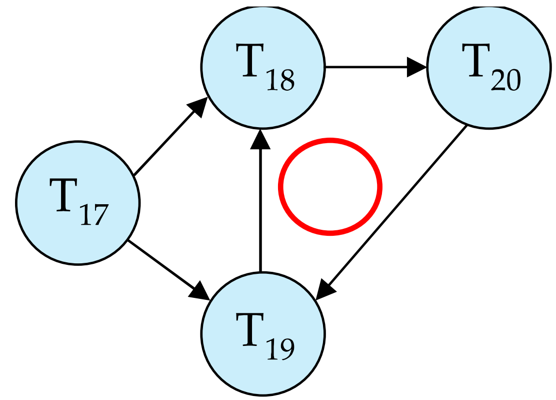
Deadlock Detection
Εντοπισμός Αδιέξοδης Κατάστασης

- For systems with no deadlock prevention mechanisms.
- Deadlocks can be described as a *wait-for graph (γράφημα αναμονής)*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that **T_i is waiting for T_j to release a data item**.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ 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
T17 waits for T18, T19
T18 waits for T20
T19 waits for T18



Wait-for graph with a cycle

Deadlock Recovery
Αποκατάσταση από Αδιέξοδη Κατάσταση

- 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 (Αναίρεση) -- determine how far to roll back transaction
    - Total rollback (Συνολκή αναίρεση συναλλαγής): Abort the transaction and then restart it.
    - Partial rollback (Μερική αναίρεση συναλλαγής) More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation (Επ' αόριστο αναμονή) happens if same transaction is always chosen as victim. **Include the number of rollbacks in the cost factor to avoid starvation.**

End of Module 15