Durham University

# Advanced Databases

## Concurrency Control Techniques I

**Dr. George Mertzios**
**Michaelmas Term**

george.mertzios@durham.ac.uk
Room 2066, MCS Building
Tel: 42 429

# Course Outline

- Enhanced Entity-Relationship (EER) Model
- Semistructured Databases - XML
- XML Data Manipulation - XPath, XQuery
- Transactions and **Concurrency Control**
- Distributed Transactions
- Distributed Concurrency Control

# Schedules

- Serial schedule: a schedule where the operations of any two transactions are not interleaved

- Non-serial schedule: a schedule where the operations of some transactions are interleaved

- A non-serial schedule is (conflict) serializable if:
  – it produces a database state that can be produced by some serial execution of the same transactions

- How can we achieve serializability in practice?

# Concurrency control

Two main approaches to ensure consistency when executing transactions concurrently:

- Conservative (pessimistic) methods:
  - actively avoid conflicts
  - delay (or restart) transactions when they are in conflict

- Optimistic methods:
  - assume that transactions are rarely in conflict
  - check for conflicts just before the transaction commits (i.e. at the end)

# Concurrency control

- Two main conservative methods:
  - locking
  - timestamping

- Lock: when a transaction accesses the database, the “lock” denies access to other transactions, to prevent incorrect results

The most widely used method to *ensure* serializability

# The locking method

- A transaction *T* can keep two types of locks:
  - <u>shared lock</u> (or <u>read lock</u>):
    - *T* is allowed only to read some data item
    - any other transaction can only read this item
  - <u>exclusive lock</u> (or <u>write lock</u>):
    - *T* is allowed to read and write on some data item
    - any other transaction has no access to this item
- Terminology:
  - a transaction requests a lock to the DBMS
  - the DBMS grants the lock; otherwise the transaction waits
  - a transaction releases a lock on a data item (the item “unlocks”)

# Rules for locks

- When a transaction needs to access a data item, it requests:
  - a shared lock for read only access
  - an exclusive lock for read and write access
- If the item is currently *not* locked by another transaction, the lock will be granted
- If the item is currently *locked*, the DBMS checks compatibility between requested lock / existing lock:
  - a shared lock is requested on an item already locked by a shared lock $\Longrightarrow$ the new lock is granted
  - otherwise: the transaction must wait until the existing lock is released

# Rules for locks

- A transaction holds a lock until it explicitly releases it:
  - during its execution, or
  - when it terminates, i.e.
    - when it commits, or
    - when it aborts
- The effects of a write operation are made visible:
  - only when the exclusive lock is released (to ensure the Isolation property)
- Some systems permit a lock to be:
  - upgraded from a shared lock to an exclusive lock, or
  - downgraded from an exclusive lock to a shared lock

(to increase more the concurrency $\Longrightarrow$ efficiency)

# The locking method

- Can we guarantee serializability by just using locks?

NO !

write_lock ($T_9$, $bal_x$)

unlock ($T_9$, $bal_x$)

write_lock ($T_9$, $bal_y$)

unlock ($T_9$, $bal_y$)

| $T_9$ | $T_{10}$ |
|---|---|
| begin_transaction | |
| read(**bal$_x$**) | |
| **bal$_x$** = **bal$_x$** + 100 | |
| write(**bal$_x$**) | begin_transaction |
| | read(**bal$_x$**) |
| | **bal$_x$** = **bal$_x$** *1.1 |
| | write(**bal$_x$**) |
| | read(**bal$_y$**) |
| | **bal$_y$** = **bal$_y$** *1.1 |
| | write(**bal$_y$**) |
| read(**bal$_y$**) | commit |
| **bal$_y$** = **bal$_y$** − 100 | |
| write(**bal$_y$**) | |
| commit | |

write_lock ($T_{10}$, $bal_x$)

unlock ($T_{10}$, $bal_x$)

write_lock ($T_{10}$, $bal_y$)

unlock ($T_{10}$, $bal_y$)

still not serializable ! (why?)

# The locking method

- Can we guarantee serializability by just using locks?

Suppose initially: $bal_x$=100, $bal_y$=400

If First $T_9$, then $T_{10}$:

- $bal_x$=220
  $bal_y$=330

($bal_x$ + $bal_y$ = 550)

First $T_{10}$, then $T_9$:

- $bal_x$=210
  $bal_y$=340

($bal_x$ + $bal_y$ = 550)

| $T_9$ | $T_{10}$ |
|---|---|
| begin_transaction | |
| read($bal_x$) | |
| $bal_x = bal_x + 100$ | |
| write($bal_x$) | begin_transaction |
| | read($bal_x$) |
| | $bal_x = bal_x * 1.1$ |
| | write($bal_x$) |
| | read($bal_y$) |
| | $bal_y = bal_y * 1.1$ |
| | write($bal_y$) |
| read($bal_y$) | commit |
| $bal_y = bal_y - 100$ | |
| write($bal_y$) | |
| commit | |

But in this schedule:

- $bal_x$=<u>220</u>
  $bal_y$=<u>340</u>

($bal_x$ + $bal_y$ = <u>560</u>)

<u>Thus:</u>

not the same as any serial schedule!

still not serializable ! (why?)

# The locking method

- To guarantee serializability, we need an additional protocol controlling the positioning of locks

- Two phase locking (2PL):
  - for every single transaction, all locking operations occur before all unlocking operations

- Two phases for every transaction:
  - growing phase
    - acquire all needed locks / no unlock
  - shrinking phase
    - release the locks / no new lock

# Two phase locking (2PL)

- Prevent the lost update problem:
  - the request of $T_1$ for an exclusive lock waits until the exclusive lock is released by $T_2$

| Time | $T_1$ | $T_2$ | $\mathbf{bal_x}$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | write_lock($\mathbf{bal_x}$) | 100 |
| $t_3$ | write_lock($\mathbf{bal_x}$) | read($\mathbf{bal_x}$) | 100 |
| $t_4$ | WAIT | $\mathbf{bal_x} = \mathbf{bal_x} + 100$ | 100 |
| $t_5$ | WAIT | write($\mathbf{bal_x}$) | 200 |
| $t_6$ | WAIT | commit/unlock($\mathbf{bal_x}$) | 200 |
| $t_7$ | read($\mathbf{bal_x}$) | | 200 |
| $t_8$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | | 200 |
| $t_9$ | write($\mathbf{bal_x}$) | | 190 |
| $t_{10}$ | commit/unlock($\mathbf{bal_x}$) | | 190 |

# Two phase locking (2PL)

- Prevent the dirty data problem:
  - the request of $T_3$ for an exclusive lock waits until the exclusive lock is released by $T_4$
  - this happens only after the rollback of $T_4$ is completed

| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | write_lock($bal_x$) | 100 |
| $t_3$ | | read($bal_x$) | 100 |
| $t_4$ | begin_transaction | $bal_x = bal_x + 100$ | 100 |
| $t_5$ | write_lock($bal_x$) | write($bal_x$) | 200 |
| $t_6$ | WAIT | rollback/unlock($bal_x$) | 100 |
| $t_7$ | read($bal_x$) | | 100 |
| $t_8$ | $bal_x = bal_x - 10$ | | 100 |
| $t_9$ | write($bal_x$) | | 90 |
| $t_{10}$ | commit/unlock($bal_x$) | | 90 |

# Two phase locking (2PL)

- Prevent the inconsistent analysis problem:

exclusive locks

read locks

| Time | $T_5$ | $T_6$ | $\mathbf{bal_x}$ | $\mathbf{bal_y}$ | $\mathbf{bal_z}$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | write_lock($\mathbf{bal_x}$) | | 100 | 50 | 25 | 0 |
| $t_4$ | read($\mathbf{bal_x}$) | read_lock($\mathbf{bal_x}$) | 100 | 50 | 25 | 0 |
| $t_5$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | WAIT | 100 | 50 | 25 | 0 |
| $t_6$ | write($\mathbf{bal_x}$) | WAIT | 90 | 50 | 25 | 0 |
| $t_7$ | write_lock($\mathbf{bal_z}$) | WAIT | 90 | 50 | 25 | 0 |
| $t_8$ | read($\mathbf{bal_z}$) | WAIT | 90 | 50 | 25 | 0 |
| $t_9$ | $\mathbf{bal_z} = \mathbf{bal_z} + 10$ | WAIT | 90 | 50 | 25 | 0 |
| $t_{10}$ | write($\mathbf{bal_z}$) | WAIT | 90 | 50 | 35 | 0 |
| $t_{11}$ | commit/unlock($\mathbf{bal_x}$, $\mathbf{bal_z}$) | WAIT | 90 | 50 | 35 | 0 |
| $t_{12}$ | | read($\mathbf{bal_x}$) | 90 | 50 | 35 | 0 |
| $t_{13}$ | | sum = sum + $\mathbf{bal_x}$ | 90 | 50 | 35 | 90 |
| $t_{14}$ | | read_lock($\mathbf{bal_y}$) | 90 | 50 | 35 | 90 |
| $t_{15}$ | | read($\mathbf{bal_y}$) | 90 | 50 | 35 | 90 |
| $t_{16}$ | | sum = sum + $\mathbf{bal_y}$ | 90 | 50 | 35 | 140 |
| $t_{17}$ | | read_lock($\mathbf{bal_z}$) | 90 | 50 | 35 | 140 |
| $t_{18}$ | | read($\mathbf{bal_z}$) | 90 | 50 | 35 | 140 |
| $t_{19}$ | | sum = sum + $\mathbf{bal_z}$ | 90 | 50 | 35 | 175 |
| $t_{20}$ | | commit/unlock($\mathbf{bal_x}$, $\mathbf{bal_y}$, $\mathbf{bal_z}$) | 90 | 50 | 35 | 175 |

# Two phase locking (2PL)

- It can be formally proved that:
  - if every transaction in a schedule follows the two-phase locking (2PL) protocol, then the schedule is always conflict serializable

- Still:
  - problems can occur by early release of locks

- The "cascading rollback" problem:
  - a transaction rollbacks after long time
  - this can cause a pile up of rollbacks

$\Longrightarrow$ inefficient database !

# The cascading rollback problem

| Time | $T_{14}$ | $T_{15}$ | $T_{16}$ |
|---|---|---|---|
| $t_1$ | begin_transaction | | |
| $t_2$ | write_lock($bal_x$) | | |
| $t_3$ | read($bal_x$) | | |
| $t_4$ | read_lock($bal_y$) | | |
| $t_5$ | read($bal_y$) | | |
| $t_6$ | $bal_x = bal_y + bal_x$ | | |
| $t_7$ | write($bal_x$) | | |
| $t_8$ | unlock($bal_x$) | begin_transaction | |
| $t_9$ | ⋮ | write_lock($bal_x$) | |
| $t_{10}$ | ⋮ | read($bal_x$) | |
| $t_{11}$ | ⋮ | $bal_x = bal_x + 100$ | |
| $t_{12}$ | ⋮ | write($bal_x$) | |
| $t_{13}$ | ⋮ | unlock($bal_x$) | |
| $t_{14}$ | ⋮ | ⋮ | |
| $t_{15}$ | rollback | ⋮ | |
| $t_{16}$ | | ⋮ | begin_transaction |
| $t_{17}$ | | ⋮ | read_lock($bal_x$) |
| $t_{18}$ | | rollback | ⋮ |
| $t_{19}$ | | | rollback |

# The cascading rollback problem

- Solution: the <u>rigorous</u> two-phase locking (2PL)
  - release all locks at the end of every transaction (when it *commits*)

- With rigorous 2PL:
  - the transactions are serializable in the order they commit
  - no cascading rollback !

- Another variant: the <u>strict</u> two-phase locking (2PL)
  - release all write locks at the end of every transaction (when it *commits*)
  - read locks can be released earlier

# Deadlocks

- <u>Deadlocks:</u> another problem with the locking method
  - two (or more) transactions wait for each other

    $\Longrightarrow$ they can wait for ever
  - this happens also with the 2PL protocol!
  - also when all locks are released at the end of transactions!

| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($\mathbf{bal_x}$) | begin_transaction |
| $t_3$ | read($\mathbf{bal_x}$) | write_lock($\mathbf{bal_y}$) |
| $t_4$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | read($\mathbf{bal_y}$) |
| $t_5$ | write($\mathbf{bal_x}$) | $\mathbf{bal_y} = \mathbf{bal_y} + 100$ |
| $t_6$ | write_lock($\mathbf{bal_y}$) | write($\mathbf{bal_y}$) |
| $t_7$ | WAIT | write_lock($\mathbf{bal_x}$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |

The DBMS has to recognize and resolve the deadlock

# Handling deadlocks

## Two general techniques:

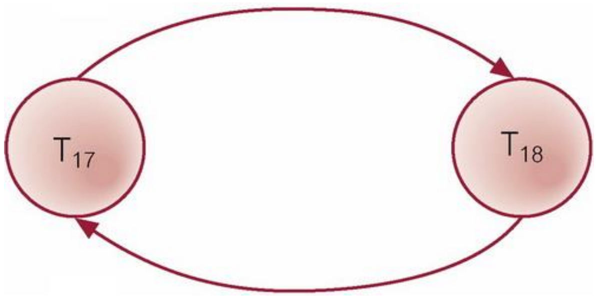
- Timeouts:
  - a transaction requests a lock only for a system-defined (maximum) period of time
- After that point:
  - the DBMS assumes there is a deadlock
  - the request times out
  - the transaction rolls back and restarts
- a simple solution
- used by several commercial DBMSs

# Handling deadlocks

Two general techniques:

- Deadlock detection:
  - construct the wait-for graph $G = (N, E)$
  - nodes $N$: one node for each transaction $T_i$
  - a directed edge $T_i \rightarrow T_j$ whenever transaction $T_i$ is waiting for a lock that is kept by transaction $T_j$

- Theorem: a deadlock exists if and only if the wait-for graph contains a directed cycle



# Handling deadlocks

- Once a deadlock is detected:
  - the DBMS aborts (rolls back) a transaction
  - check for a deadlock too often
    - $\Longrightarrow$ large computational overhead (slower database)
  - check for a deadlock too rarely
    - $\Longrightarrow$ deadlocks may be undetected for long periods

Important parameters for recovery:

- How far to roll back a transaction?
  - simplest solution: undo all changes (restart)
  - more efficient: possibly roll back only a part of the transaction

# Handling deadlocks

Other important parameters for recovery:

- Choice of the deadlock “victim”:
  - which transaction to abort?
  - the choice affects efficiency of the database
- General criteria:
  - how long a transaction has been running?
    - better to abort a “short” transaction
  - how many data items did it update so far?
    - better to abort transactions that made few changes
  - how many data items does it still have to update?
    - better not to abort transactions that have few more data to update
    - difficult for the DBMS to know in advance

# Handling deadlocks

Other important parameters for recovery:

- Avoiding “starvation”:
  - starvation occurs when a specific transaction is always chosen as the “victim”
  - this transaction can never complete

- Common solution:
  - store how many times every transaction has been aborted
  - when an upper limit is reached, use different selection criteria

# Granularity of data items

- Further locking parameter: granularity
  - how “large” is the “data item” to be locked each time?

- Hierarchy of granularity:
  - entire database – coarsest size of a data item
  - file (relation / table)
  - page (section of physical disk where relations are stored)
  - record (tuple of relation)
  - field value (cell of a tuple) – finest size of a data item

# Granularity of data items

- Coarser data item size
  - large locked items / fewer locks requested

  $\Longrightarrow$ lower degree of concurrency permitted

- Finer data item size
  - small locked items / more locks requested

  $\Longrightarrow$ more locking information needs to be stored

- Granularity affects efficiency:
  - best item size depends on the nature of transactions

    (i.e. which / how many data items needed per transaction)

# Concurrency control

- Next lecture:
  - timestamping
  - the alternative method to guarantee serializability
  - no locks $\Longrightarrow$ no deadlocks
- Locking method:
  - make conflicting transactions wait
- Timestamping method:
  - roll back and restart conflicting transactions

# Summary of the Lecture

- Concurrency control methods:
  - the locking method
    - shared / exclusive locks
  - two phase locking (2PL)
    - growing / shrinking phase
  - use 2PL to prevent:
    - the lost update problem
    - the dirty data problem
    - the inconsistent analysis problem
  - the cascading rollback problem
  - deadlocks
    - timeouts
    - deadlock detection