

Database Technology

Topic 10: Concurrency Control

Olaf Hartig

olaf.hartig@liu.se

Goal

- Preserve **Isolation** of the ACID properties



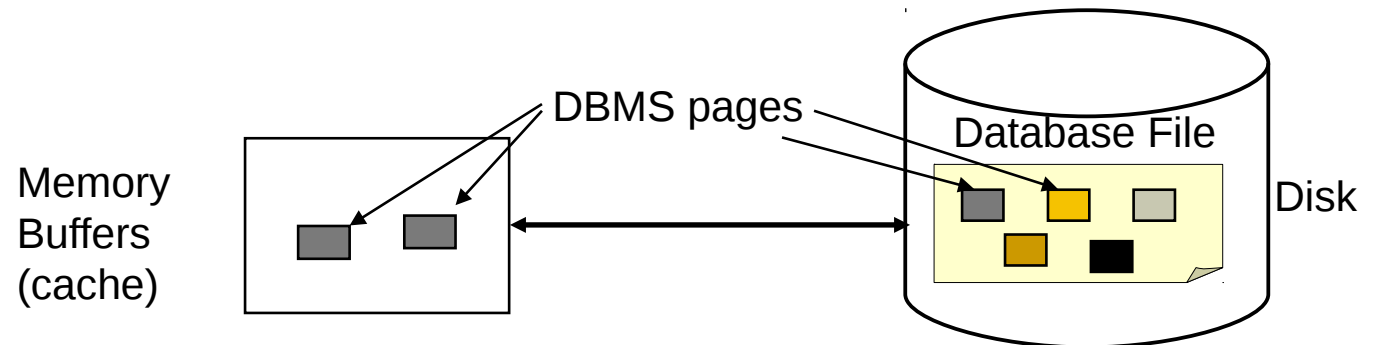
Transaction Processing Model

Simple Database Model

- **Database**: simply, a collection of named items
- Granularity (size) of these data items is unimportant
 - May be a field, a tuple, or a file block, etc
 - Transaction processing concepts are independent of granularity

Basic Operations

- `read_item(X)`: reads item X into a program variable (for simplicity, assume that the variable is also named X)
- `write_item(X)`: write the value of program variable X into the database item named X
- These operations take some amount of time to execute
- Basic unit of data transfer between the disk and the computer main memory is a file block/page



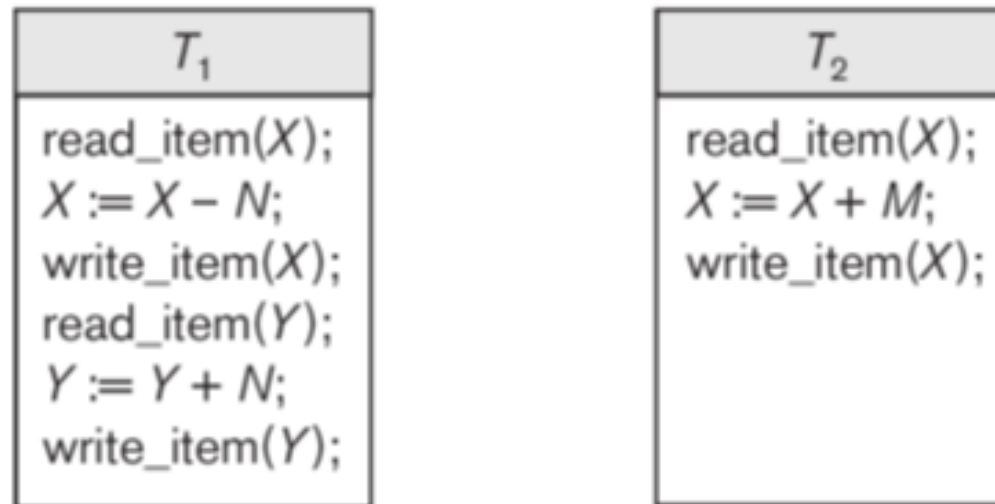
Steps of Read / Write Operations

- `read_item(X)` consists of the following steps:
 1. Find address of the file block that contains item X
 2. Copy the file block into a buffer in main memory (if the block is not already in main memory)
 3. Copy item X from the buffer to the program variable X
- `write_item(X)` consists of the following steps:
 1. Find address of the file block that contains item X
 2. Copy the file block into a buffer in main memory (if the block is not already in main memory)
 3. Copy item X from the program variable named X into its correct location in the buffer
 4. Store the updated block from the buffer back to disk (either immediately or at some later point in time)

Transaction Notation

- Focus on read and write operations
 - For instance, $w_5(Z)$ means that transaction 5 writes data item Z
- b_i and e_i specify transaction boundaries (begin and end)
 - i specifies a unique transaction identifier (TID)

- Example:



- T_1 : $b_1, r_1(X), w_1(X), r_1(Y), w_1(Y), e_1$
- T_2 : $b_2, r_2(X), w_2(X), e_2$

Initial Concepts

Schedule

- Sequence of interleaved operations from multiple TAs

- Example:

	at ATM window #1	at ATM window #2
1	read_item(savings);	
2	savings = savings - \$100;	
3		read_item(checking);
4	write_item(savings);	
5	read_item(checking);	
6		checking = checking - \$20;
7		write_item(checking);
8	checking = checking + \$100;	
9	write_item(checking);	
10		dispense \$20 to customer;

– $S: b_1, r_1(s), b_2, r_2(c), w_1(s), r_1(c), w_2(c), w_1(c), e_1, e_2$

Quiz

What can be concluded from the following schedule?

..., $r_3(\text{EMPLOYEE})$, b_4 , $w_2(\text{STUDENT})$, ...

A: Some employee has read a student record.

B: A transaction has read some data and then written it back.

C: At least three transactions were running concurrently.

D: All of the above.

E: None of the above.

Serial Schedules

- Definition: a schedule is *serial* if the operations of any TA are executed directly one after the other
 - i.e., no interleaving of operations from different TAs
- Characteristics:
 - Serial schedules trivially guarantee the isolation property
 - For n transactions, there are $n!$ serial schedules
 - Each of them produces a correct result (assuming the consistency preservation property)
 - However, not all of them might produce the *same* result
 - For instance, If two people try to reserve the last seat on a plane, only one gets it. The serial order determines which one. The two orderings have different results, but either one is correct.

Serial Schedules (cont'd)

Serial schedules are *not feasible* for performance reasons:

- Long transactions force other transactions to wait
- When a transaction is waiting for disk I/O or any other event, system cannot switch to other transaction
- Solution: allow *some* interleaving
(without sacrificing correctness!)

Acceptable Interleavings

(Serializability)

Conflicts

- Executing some operations in a different order causes a different outcome
 - ... $r_1(X), w_2(X), \dots$ vs. ... $w_2(X), r_1(X), \dots$
 T_1 will read a different value for X
 - ... $w_1(Y), w_2(Y), \dots$ vs. ... $w_2(Y), w_1(Y), \dots$
value for Y after both operations will be different
- Note that two read operations do not have this issue
 - ... $r_1(Z), r_2(Z), \dots$ vs. ... $r_2(Z), r_1(Z), \dots$
both TAs read the same value of Z

Conflicts and Equivalence

Definition: Two operations **conflict** if

1. they access the same data item X ,
2. they are from two different transactions, and
3. at least one of them is a write operation.

Definition: Two schedules are **conflict equivalent** if the relative order of *any two conflicting operations* is the same in both schedules.

Example:

$S1: b_1, r_1(s), b_2, r_2(c), w_1(s), r_1(c), w_2(c), w_1(c), e_1, e_2$

$S2: b_1, r_1(s), r_1(c), b_2, r_2(c), w_1(s), w_2(c), w_1(c), e_2, e_1$

Serializability

Definition: A schedule with n transactions is **serializable** if it is conflict equivalent to *some* serial schedule of the same n transactions.

- Serializable schedule “correct” because equivalent to some serial schedule, and any serial schedule acceptable
 - Transactions see data as if they were executed serially
 - Transactions leave DB state as if they were executed serially (hence, serializable schedules will leave the database in a consistent state)
- Efficiency achievable through interleaving and concurrent execution

Testing Serializability

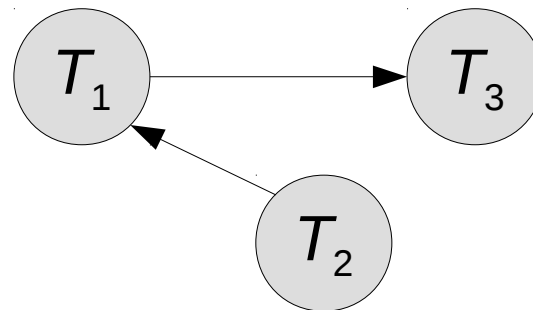
- Construct a *serialization graph* for the schedule
 - Node for each transaction in the schedule
 - Direct edge from T_i to T_j if some read or write operation in T_i appears before a conflicting operation in T_j
- A schedule is serializable if and only if its serialization graph has no cycles

Example

- Consider the following schedule

$S: b_1, r_1(X), b_2, r_2(Y), w_1(X), b_3, w_2(Y), e_2, r_1(Y), r_3(X), e_3, w_1(Y), e_1$

- Serialization graph of S :



- No cycles! Hence, S is serializable.
 - Equivalent to the following serial schedule:

$S': b_2, r_2(Y), w_2(Y), e_2, b_1, r_1(X), w_1(X), r_1(Y), w_1(Y), e_1, b_3, r_3(X), e_3$

T_2 T_1 T_3

Quiz

Remember

- If the initial value of checking is \$500, what value does it have after the following interleaved execution completes?

	at ATM window #1	at ATM window #2
1	read_item(savings);	
2	savings = savings - \$100;	
3		read_item(checking);
4	write_item(savings);	
5	read_item(checking);	
6		checking = checking - \$20;
7		write_item(checking);
8	checking = checking + \$100;	
9	write_item(checking);	
10		dispense \$20 to customer;

A: \$480

B: \$500

C: \$580

D: \$600

– S: $b_1, r_1(s), b_2, r_2(c), w_1(s), r_1(c), w_2(c), w_1(c), e_1, e_2$

Key Question

Can we make sure that we only get serializable schedules?

Locking Techniques for Concurrency Control

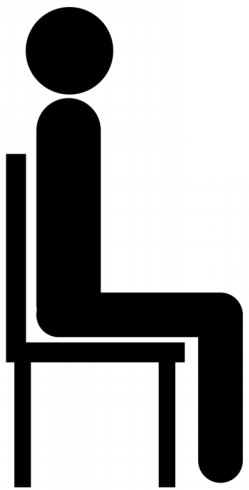
Database Locks

- Locks can be used to ensure that conflicting operations cannot occur
- **Exclusive lock** for writing, **shared lock** for reading
 - Transaction cannot read item without first getting a shared or an exclusive lock on it
 - Transaction cannot write item without first getting exclusive lock on it



Database Locks (cont'd)

- Request for lock may cause transaction to **block** (wait) because write lock is exclusive
 - Any lock on X (read or write) cannot be granted if some other transaction holds write lock on X
 - Write lock on X cannot be granted if some other transaction holds *any* lock on X

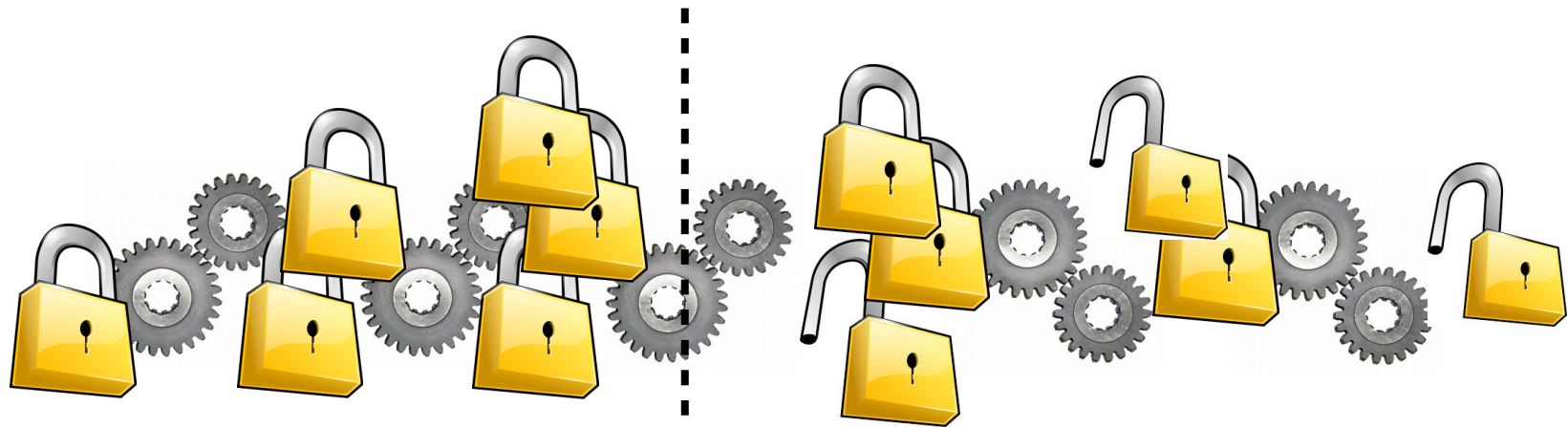


- Blocked transactions are unblocked and granted the requested lock when conflicting transaction(s) release their lock(s)



Two-Phase Locking (2PL)

Definition: A transaction follows the two-phase locking (2PL) protocol if *all* of its `read_lock()` and `write_lock()` operations come before its first `unlock()` operation.



- A transaction that follows the 2PL protocol has an *expansion phase* and a *shrinking phase*
- If all transactions in a schedule follow the 2PL protocol, then the schedule is serializable



Deadlock

- Two or more transactions wait for one another to unlock some data item
 - T_i waits for T_j waits for ... waits for T_n waits for T_i
- Deadlock prevention:
 - Conservative 2PL protocol: Wait until you can lock all the data to be used beforehand
 - Wait-die
 - Wound-wait
 - No waiting
 - Cautious waiting
- Deadlock detection:
 - Wait-for graph
 - timeouts

Starvation

- A transaction is not executed for an indefinite period of time while other transactions are executed normally
 - e.g., T waits for write lock and other TAs repeatedly grab read locks before all read locks are released
- Starvation prevention:
 - First-come-first-served waiting scheme
 - Wait-die
 - Wound-wait
 - etc.

Summary

Summary

- Characterizing schedules based on serializability
 - Serial and non-serial schedules
 - Conflict equivalence of schedules
 - Serialization graph
- Two-phase locking
 - Guarantees conflict serializability
 - Possible problems: deadlocks and starvation

www.liu.se