Concurrency Control

• Transaction = Sequence of Operations
• ACID = Atomic, Consistent, Isolated, Durable
• Interleaving of transactions can break Isolation
• Various concurrency-control mechanisms ensure Isolation
Example of Interfering Transactions

• Transfer $100 from Checking to Saving:
  1) \( c := \text{read(checking)} \)
  2) \( \text{write(checking, } c - 100) \)
  3) \( s := \text{read(savings)} \)
  4) \( \text{write(savings, } s + 100) \)

• Print the combined balance:
  a) \( x := \text{read(checking)} \)
  b) \( y := \text{read(savings)} \)
  c) \( \text{print } x + y \)
- Bad: 1 2 a b c 3 4
- Good: 1 2 3 4 a b c
- Good: a b c 1 2 3 4
- Good: a 1 b 2 3 c 4
- How can we distinguish the Good from the Bad?
Abstract View of Transactions

• Transfer from checking to saving:
  – $r_1(c)$, $w_1(c)$, $r_1(s)$, $w_1(s)$

• Calculate the combined balance:
  – $r_2(c)$, $r_2(s)$

• Bad: $r_1(c)$, $w_1(c)$, $r_2(c)$, $r_2(s)$, $r_1(s)$, $w_1(s)$

• Good: $r_1(c)$, $w_1(c)$, $r_1(s)$, $w_1(s)$, $r_2(c)$, $r_2(s)$

• Good: $r_2(c)$, $r_2(s)$, $r_1(c)$, $w_1(c)$, $r_1(s)$, $w_1(s)$

• Good: $r_2(c)$, $r_1(c)$, $r_2(s)$, $w_1(c)$, $r_1(s)$, $w_1(s)$

• Good: $r_2(c)$, $r_1(c)$, $r_2(s)$, $w_1(c)$, $r_1(s)$, $w_1(s)$
Serializable Histories

- History is *serial* if there is no interleaving:
  - \(w_1(x), w_1(y), r_2(x), w_2(y)\)
  - Can be abbreviated as \(T_1, T_2\)

- History is *serializable* if it has the same effect as a serial history

- For instance, these histories have the same effect:
  1. \(w_2(x), r_1(x), r_1(y), w_2(z)\)
  2. \(w_2(x), w_2(z), r_1(x), r_1(y)\)

- Since (2) is serial \(T_2, T_1\), then (1) is serializable
Equivalence between Histories

- What does it mean to say that two histories have the “same effect”?

- Suppose H and H’ are two histories that result from interleaving the transactions $T_1, \ldots, T_k$

- Then H and H’ are equivalent if:
  - Each read operation in H returns the same value as the corresponding read operation in H’
  - For each variable $x$ that is written to, the final write to $x$ in H and the final write to $x$ in H’ are the same operation
Transaction Manager Must Ensure Serializability

- Since serializable history has the same effect as a serial and hence correct history:
  - Serializable = Correct

- The TM accepts read and write operations $r_i(x)$, $w_j(y)$ from various transactions

- To ensure that the history is serializable, it may:
  - Delay some of the operations
  - Abort some transactions

- How to determine if a history is serializable?
Conflicting Operations

- Suppose $o_i(x)$ is an operation in $T_i$, and $o_j(x)$ is in $T_j$, where $i \neq j$

- Because they operate on the same database element, $o_i(x)$ and $o_j(x)$ may “conflict”

- The operations conflict if $o_i(x)$, $o_j(x)$ has a different effect than $o_j(x)$, $o_i(x)$

- I.e., they conflict if they don’t commute

- Hence: $w_i(x)$ conflicts with both $w_j(x)$ and $r_j(x)$, while $r_i(x)$ and $r_j(x)$ do not conflict
The Serialization Graph

- Let H be a history for $T_1, \ldots, T_k$
- Directed graph with a node for each $T_i$
- Suppose that $o_i(x)$ and $o_j(x)$ conflict
- If $o_i(x)$ precedes $o_j(x)$ in H, edge goes from $T_i$ to $T_j$
- Otherwise edge goes from $T_j$ to $T_i$
- Example: $H = r_1(x), w_2(x), w_2(y), r_1(y)$
- SG has edge from $T_1$ to $T_2$, and from $T_2$ to $T_1$
**Serializability Theorem**

- Let H be a history for $T_1, \ldots, T_k$
- Let SG be the serialization graph for H
- If SG has no cycles, then:
  1. H is serializable
  2. A serial equivalent to H is given by any topological sort of $T_1, \ldots, T_k$
- Example: $r_1(x), r_3(z), w_3(x), w_1(y), w_2(y), w_2(z)$
- SG: $T_1 \rightarrow T_3, T_3 \rightarrow T_2, T_1 \rightarrow T_2$
- Serializable, with one topsort: $T_1 \ T_3 \ T_2$
Locking

• Transactions acquire read locks and write locks on data items before using them

• Read locks are shared, i.e., can be many read locks on the same data item

• Write locks are exclusive, i.e., no other locks (read or write) can be held

• Transaction manager maintains a lock table, and delays operations that are blocked by locks
Two-Phase Locking (2PL)

• Standard method for TM to ensure serializability

• The 2PL condition:
  – Once a transaction releases a lock, then it cannot acquire any new locks

• So there are two phases:
  1. Lock acquisition
  2. Lock releasing

• The *lockpoint* is the moment when the last lock is acquired
Illustration of 2PL

• $T_1$: $r_1(c), w_1(c), r_1(s), w_1(s), w_1(t)$

• $T_2$: $r_2(c), r_2(s)$

• History, including locking operations:
  $rl_1(c), r_1(c), wl_1(c), w_1(c), rl_2(c)$ (delayed), $rl_1(s), r_1(s), wl_1(s), w_1(s), wl_1(t), ul_1(c), rl_2(c), r_2(c), rl_2(s), r_2(s), ul_1(s), ul_2(c), ul_2(s), w_1(t), ul_1(t)$

• History of read and write operations:
  $r_1(c), w_1(c), r_1(s), w_1(s), r_2(s), r_2(s), w_1(t)$

• SG is acyclic: $T_1 \rightarrow T_2$
**Incorrectness without 2PL**

- $T_1$: $r_1(c) \; w_1(c) \; r_1(s) \; w_1(s) \; w_1(t)$
- $T_2$: $r_2(c) \; r_2(s)$
- If locks are released right after the data is used, then any interleaving—including incorrect ones—can be achieved
- But this violates 2PL, because, e.g.,
  - $wl_1(x) \; w_1(x) \; ul_1(x) \; wl_2(x) \; w_2(x) \; ul_2(x)$
  - Not 2PL because $wl_2(x)$ comes after $ul_2(x)$
Theorem: 2PL Guarantees Serializability

Proof.

1. Suppose the transactions are $T_1, \ldots, T_k$, and that H is an execution of them that can be obtained by 2PL

2. Suppose that $T_i \rightarrow T_j$ in the SG

3. This means that some $o_i(x)$ and $o_j(x)$ conflict, and $o_i(x)$ came before $o_j(x)$ in H

4. Then $T_i$ and $T_j$ both performed on operation on $x$, the operations conflicted, and $T_i$ did its operation first
5. Hence \( T_i \) held a lock on \( x \) before \( T_j \) held a lock on \( x \)
6. Because operations conflicted, these locks could not be held at the same time
7. Hence \( T_i \) must have released the lock on \( x \) before \( T_j \) acquired the lock on \( x \)
8. Hence, \( \text{lockpoint}(T_i) < \text{lockpoint}(T_j) \)
9. So, what we’ve shown is that \( T_i \rightarrow T_j \) implies that \( \text{lockpoint}(T_i) < \text{lockpoint}(T_j) \)
10. Therefore, a cycle in the SG would imply the impossible condition that \( \text{lockpoint}(T_i) < \text{lockpoint}(T_j) \)
11. Hence, SG is acyclic, and so the history is serializable
Deadlocking with 2PL

• $T_1 = w_1(x) \, w_1(y)$

• $T_2 = w_2(y) \, w_2(x)$

• A deadlocking history:
  1. $wl_1(x)$
  2. $w_1(x)$
  3. $wl_2(y)$
  4. $w_2(y)$
  5. $wl_1(y)$ (blocked)
  6. $wl_2(y)$ (blocked, deadlock)
The Wait-Graph

- TM can break deadlocks if it constructs the wait-graph
- Nodes are transactions, edges when one transaction waits for a lock held by another
- Cycles indicate deadlock
- The TM breaks cycles by aborting a sufficient number of transactions
Cascading Aborts

• Suppose \( T_1 \) reads a variable that \( T_2 \) wrote, and then \( T_2 \) aborts
• Then \( T_1 \) must also abort
• I.e., the abort of \( T_2 \) “cascades” to \( T_1 \)
• This becomes messy, because aborts can potentially cascade through long chains of transactions
**Strict 2PL**

- A stronger form of 2PL
- The strict 2PL condition:
  - Locks are only released after transaction commits
- Eliminates the problem of cascading aborts
- Throughput may be reduced, though, because locks are held for longer
SQL Support for Transactions

- Transaction is started implicitly, by executing an SQL statement
- Transaction is ended explicitly by issuing either a COMMIT or a ROLLBACK command
- If there was a COMMIT:
  - For every ASSERTION that was declared as DEFERRABLE (in the SQL DDL stage), and therefore was not being checked during the transaction, it is now automatically checked
– If there is a failure of such an assertion, the COMMIT is automatically converted to a ROLLBACK