Database Management Systems 2010/11

– Chapter 6: Transactions –

J. Gamper

- Transaction Concept
- ACID Properties
- Atomicity and Durability
- Concurrent Execution
- Serializability
- Recoverability
- Isolation

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Transaction Concept

- **Transaction:** A logical unit of program execution (i.e.,a sequence of actions) that accesses and possibly updates various data items.It includes one or more DB access operations (insertion,deletion,modification,retrieval)
- For a transaction we have:
  - A transaction must see a consistent database
  - During transaction executio the DB may be inconsistent.
  - When the transaction is committed, the DB must be consistent
- Two main issues to deal with:
  - Various failures, e.g., hardware failures and system crashes
  - Concurrent execution of multiple transacctions

ACID Properties

- To preserve integrity of data, a transaction must meet the **ACID** properties:
  - **Atomicity:** A transaction's changes to the state are atomic, i.e., either all operations of the transaction are properly reflected in the DB or none are (⇒ recovery manager)
  - **Consistency**: A transaction is a correct transformation of a state.The actions (taken as a group) do not violate any of the integrity constraints associated with the state. (⇒ application programs and integrity checker)
  - **Isolation:** Although multiple transactions may execute concurrently, it appears to each transaction that all other transactions are either executed before or after. (⇒ concurrency manager)
  - **Durability:** After a transaction completes (commits) successfully, the changes it has made to the database persist, even if there are system failures. (⇒ recovery manager)

ACID Properties ...

- **Example:** Transaction to transfer \$50 from account A to account B:
  1. read(A)
  2. A := A - 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)

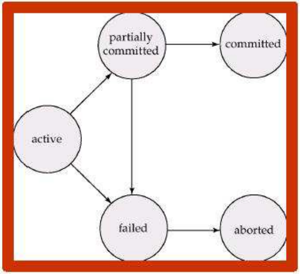
- ACID properties:
  - **Consistency requirement:** the sum of A and B is unchanged by the execution of the transaction.
  - **Atomicity requirement:** if the transaction fails after step 3 and before step 6, the system should ensure that the updates are not reflected in the DB, else an inconsistency will result.

ACID Properties ...

- **Example (contd.)**
  - **Durability requirement:** once the user has been notified that the transaction has completed (i.e., the $50 are transferred), the updates to the DB by the transaction must persist despite failures.
  - **Isolation requirement:** if between steps 3 and 6, another transaction is allowed to access the partially updated DB, it will see an inconsistent DB (the sum $A + B$ will be less than it should be). This might result in an inconsistent DB state after the completion of both transaction, if e.g. the second transaction performs updates on A and B.
    - These problems can be avoided trivially by running transactions serially, i.e., one after the other.
    - However, executing multiple transactions concurrently has significant benefits in performance.

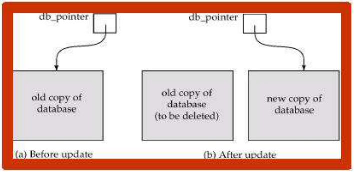
Transaction State

- **Active:** (initial state) The transaction stays in this state during execution.
- **Partially committed:** After the final statement has been executed.
- **Failed:** After the discovery that normal execution can no longer proceed. completion.
- **Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
  - Kill the transaction



Atomicity and Durability

- The **recovery manager** of a DBMS implements the support for atomicity and durability.
- **Shadow-database** scheme (assume that only one transaction is active at a time and disks do not fail):
  - A pointer *db_pointer* points to current consistent copy of DB.
  - All updates are made on a **shadow copy** of the DB, and *db_pointer* is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages are flushed to disk.
  - If transaction fails, old copy pointed to by *db_pointer* is used.



- Extremely inefficient for large DB (copy of entire DB)
  - Useful for text editors

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction throughput: one transaction can use the CPU while another reads from or writes to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- Basic assumption:
  - Each transaction preserves DB consistency.
  - Serial execution of a set of transactions preserves DB consistency.
- The concurrency control system restricts the interactions between concurrent transactions in order to preserve the DB consistency.
  - Not all concurrent schedules ensure consistency!

Schedules

- **Schedule:** Sequence of instructions from a set of concurrent transactions that indicate the chronological order in which these instructions are executed.
  - Must consist of all instructions of all transactions.
  - Must preserve the order of instructions within each individual transaction.
- **Serial schedule:** The transactions execute one after the other.
  - One transaction is completely finished before another transaction starts.

Schedules . . .

- **Example:** Consider the following transactions:
  - T_1 transfer \$50 from A to B
  - T_2 transfer 10% of the balance from A to B.

- The following is a serial schedule, in which T_1 is followed by T_2.
- Integrity constraint: Sum of A + B is preserved

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedules ...

- **Example:** (contd.)
- The following schedule is not a serial schedule, but it is **equivalent** to the previous one.
- In both schedules the sum of A + B is preserved.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedules ...

- **Example:** (contd.)
- The following concurrent schedule does not preserve the value of the sum A + B.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- **Serializable schedule:** A schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence:
  - conflict serializability
  - view serializability
- We ignore operations other than **read** and **write** instructions:
  - We assume that transactions may perform arbitrary computations on data in local buffers in between **reads** and **writes**.
  - Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability

- Instructions I_i and I_j of transactions T_i and T_j **conflict** iff there exists a data item Q accessed by I_i and I_j and at least one of these instructions is a write operation on Q, i.e.,
  - $I_i = read(Q)$ and $I_j = write(Q)$
  - $I_i = write(Q)$ and $I_j = read(Q)$
  - $I_i = write(Q)$ and $I_j = write(Q)$
- $I_i = read(Q)$ and $I_j = read(Q)$ do not conflict since the order in which the two instructions are executed does not matter.

Conflict Serializability ...

- **Transformation** of schedules to generate equivalent schedules
- Assume a schedule S and two consecutive instructions I_i and I_{i+1}.
  Instructions I_i and I_{i+1} can be **swapped** if they are **non-conflicting**, i.e., if
  - both are read instructions, or
  - they refer to different DB items, or
  - one of them is not a DB operation (e.g., not read or write)

Conflict Serializability ...

- **Conflict equivalent schedules:** If a schedule S can be transformed into a schedule S' by a series of nonconflicting swaps of instructions then S and S' are conflict equivalent.
- **Conflict serializable schedule:** A schedule S is conflict serializable iff it is conflict equivalent to a serial schedule.

Conflict Serializability ...

- **Example:** A schedule that is not conflict serializable

T_3	T_4
$read(Q)$	
	$write(Q)$
$write(Q)$	

- It is not possible to swap instructions in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$ or the serial schedule $\langle T_4, T_3 \rangle$.

Conflict Serializability ...

- **Example:** The following schedule is conflict serializable, since it can be transformed into $\langle T_1, T_2 \rangle$ by nonconflicting swaps.

T_1	T_2
read(A)	
write(A)	
	write(A)
	write(A)
read(B)	
write(B)	
	write(B)
	write(B)

View Serializability

- **View equivalent schedules:** Let S and S' be two schedules over the same set of transactions $T_1, T_2, \ldots, T_n$. S and S' are view equivalent if the following three conditions are met for all data items Q:
  1. If T_i reads the initial value of Q in S ($read(Q)$), then T_i must in S' also read the initial value of Q.
  2. If T_i executes $read(Q)$ in S and that value of Q was produced by T_j (if any), then T_i must in S' also read the value of Q that was produced by T_j .
  3. The transaction (if any) that performs the final $write(Q)$ in S must perform the final $write(Q)$ in S'.

- **View serializable schedule:** A schedule S is view serializable if it is view equivalent to a serial schedule.

Conflict Serializability ...

- **Example:** The following schedule is view-serializable (to $\langle T_3, T_4, T_6 \rangle$) but not conflict serializable by nonconflicting swaps.

T_3	T_4	T_6
$read(Q)$		
	$write(Q)$	
$write(Q)$		
		$write(Q)$

- Generally, we have:
  - Every conflict serializable schedule is also view serializable
  - Every view serializable schedule that is not conflict serializable has **blind writes**, i.e., write operations without having performed a read operation.

Other Notions of Serializability

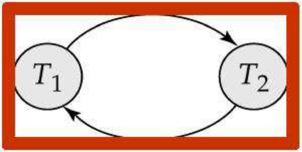
- The schedule given below produces the same outcome as the serial schedule $\langle T_1, T_5 \rangle$, yet it is not conflict equivalent or view equivalent to it.

T_1	T_5
$read(A)$	
$A := A - 50$	
$write(A)$	
	$read(B)$
	$B := B - 10$
	$write(B)$
$read(B)$	
$B := B + 50$	
$write(B)$	
	$read(A)$
	$A := A + 10$
	$write(A)$

- Determining such equivalence requires to analyse operations other than read and write.

Testing for Conflict Serializability

- Consider a schedule S of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph (conflict graph):** A directed graph with a node T_i for each transaction and with an edge $T_i \rightarrow T_j$ iff one of the following conditions holds:
  - T_i executes $write(Q)$ before T_j executes $read(Q)$
  - T_i executes $read(Q)$ before T_j executes $write(Q)$
  - T_i executes $write(Q)$ before T_j executes $write(Q)$



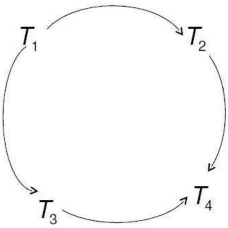
- A schedule is **conflict serializable** if and only if its precedence graph is **acyclic**.

Testing for Conflict Serializability ...

- **Example:** A conflict serializable schedule with 5 transactions and precedence graph

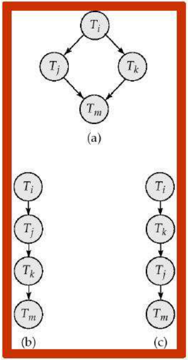
T1	T2	T3	T4	T5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				

Precedence graph



Testing for Conflict Serializability . . .

- Cycle-detection algorithms exist which take $O(n^2)$ time, where n is the # of vertices in the graph.
  - Better algorithms in $O(n^2)$ time, where e is the # of edges.
- If the precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph. This is a linear order consistent with the partial order of the graph
  - e.g., a serializability order for the schedule in the previous example would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.
- Topological sorting example



Test for View Serializability

- The precedence graph test for conflict serializability must be modified to be applicable for testing view serializability.
- The problem of checking if a schedule is view serializable falls in the class of NP-complete problems.
  - Thus existence of an efficient algorithm is unlikely.
  - However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability **after** it has executed is a little too late!
- **Goal:** Develop concurrency control protocols that will assure serializability.
- **Concurrency control protocols** will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline (i.e., a set of rules) that avoids non-seralizable schedules.
- Tests for serializability help understand why a concurrency control protocol is correct.
- Will be discussed in chapter 7

Recoverability

- Need to address the effect of transaction failures on concurrently running transactions.
- **Recoverable schedule:** For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i must appear before the commit operation of T_j.
- DBMS must ensure that schedules are **recoverable**.
- **Example**: The following schedule is not recoverable if T_9 commits immediately after the read operation.
  - If T_8 aborts, T_9 would have read an inconsistent DB state.

T_8	T_9
$read(A)$	
$write(A)$	
	$read(A)$
$read(B)$	

Recoverability . . .

- **Cascading rollback:** A single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)
  - If T_{10} fails, T_{11} and T_{12} must also be rolled back.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- **Main problem:** Can lead to the undoing of a significant amount of work.

Recoverability ...

- **Cascadeless schedules:** For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

- A cascadeless schedule avoids cascading rollbacks.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control and Isolation

- If only one transaction can execute at a time we get serial schedules, which provide a poor throughput.
  - Transaction acquires a lock on the entire DB before it starts and releases the lock after it has committed.
- Concurrency-control schemes is a tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
  - Some schemes allow only conflict-serializable schedules, while others allow view-serializable schedules that are not conflictserializable.
- **Desirable properties** of a schedule to guarantee DB consistency:
  - conflict/view serializable and recoverable
  - and preferably cascadeless