MDA104 Introduction to Databases

6. Transactions

Vlastislav Dohnal

Transactions

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. 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*)
  7. **commit**
- Main issues to deal with:
  - Transaction interruption due failures of various kinds
    - such as hardware failures and system crashes
  - Concurrent execution of multiple transactions
  - Termination of transaction using **abort** command

Example of Fund Transfer

- 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*)
  7. **commit**
- **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement**
  - once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- 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*)
  7. **commit**
- **Consistency requirement**
  - E.g. the sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - E.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- Transaction to transfer \$50 from account A to account B:

Transaction T1

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)
7. **commit**

Transaction T2

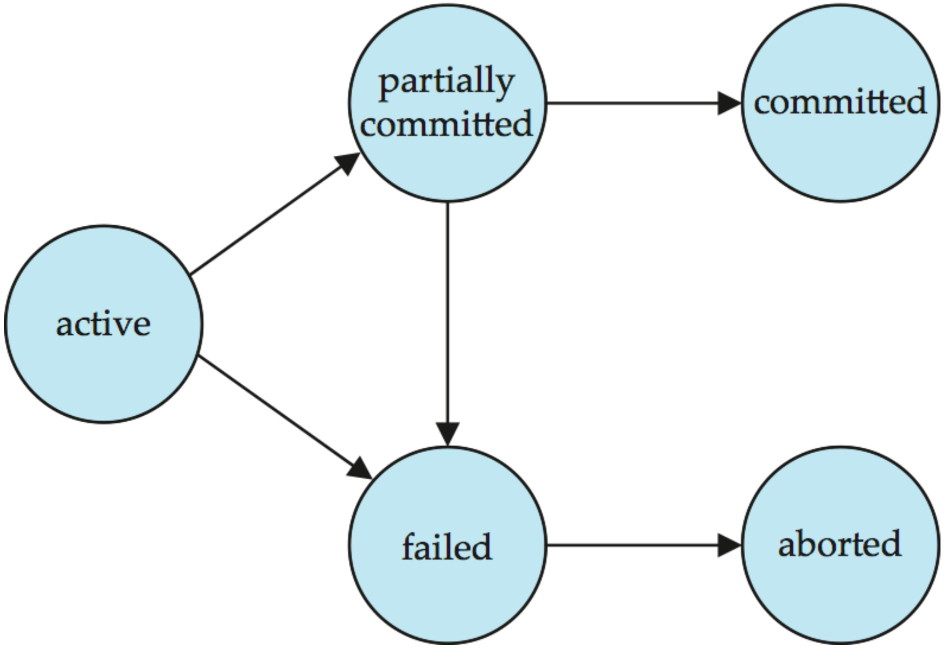
1. **read**(*A*)
2. **read**(*B*)
3. **print**(*A* + *B*)
4. **commit**

- **Isolation requirement** – if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database
  - The sum *A* + *B* will be less than it should be.
- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
  - It is a **sequence** of operations that form a desired outcome (the unit of program).
- To preserve the integrity of data the database system must ensure:
  - **Atomicity.**
    - Either all operations of the transaction are properly reflected in the database or none are.
  - **Consistency.**
    - Execution of a transaction in isolation preserves the consistency of the database.
  - **Isolation.**
    - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
    - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
  - **Durability.**
    - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State



- **Active**
  - the initial state
  - the transaction stays in this state while it is executing
- **Partially committed**
  - after the final statement has been executed.
- **Committed**
  - after successful completion.
- **Failed**
  - after the discovery that normal execution can no longer proceed.
- **Aborted**
  - after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - restart the transaction
      - can be done only if no internal logical error
    - kill the transaction

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions
    - E.g. short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Analysis of conflicting operations
    - Locking – of records, tables

Schedules

- **Schedule** – a sequence of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction

- A transaction that successfully completes its execution will have a **commit** instruction as the last statement
  - by default, transaction assumed to execute commit instruction as its last step
- A transaction that fails to complete its execution will have an **abort** instruction as the last statement (*rollback* command)

Schedule 1

- Let T_1 transfer \$50 from A to B, and
  T_2 transfer 10% of the balance from A to B.
- A serial schedule in which
  T_1 is followed by T_2:

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

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Schedule 3

- Let T_1 and T_2 be the transactions defined previously.
- The following schedule is not a serial schedule
  - but it is *equivalent* to Schedule 1 (serial schedule).

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) commit	
	read (B) $B := B + temp$ write (B) commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

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)	
commit	
	$B := B + temp$
	write (B)
	commit

These changes to A will be discarded by **write**(A) in T1

Serializability

- **Basic Assumption:** each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.

- A (possibly concurrent) **schedule is serializable** if it is equivalent to a serial schedule.
  - Different forms of schedule equivalence give rise to the notions of:
    1. **conflict serializability**
    2. **view serializability**

Simplified view of transactions

- 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.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

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

Schedule 3

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

Schedule 1

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

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

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Recoverable Schedules

- Need to address the effect of transaction failures on concurrently running transactions.
- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	
abort	

- Should T_8 abort, T_9 would have read (and possibly shown to the user) an inconsistent database state!
  - Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **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)

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

If T_{10} fails, T_{11} and T_{12} must also be rolled back

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur if
  - 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.
- Every cascadeless schedule is also recoverable

- It is desirable to restrict the **schedules** to those that are **cascadeless**

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view <u>serializable</u>, and
  - are <u>recoverable</u> and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable and recoverable
  - E.g.
    - a read-only transaction that wants to get an approximate total balance of all accounts
    - database statistics computed for query optimization can be approximate
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

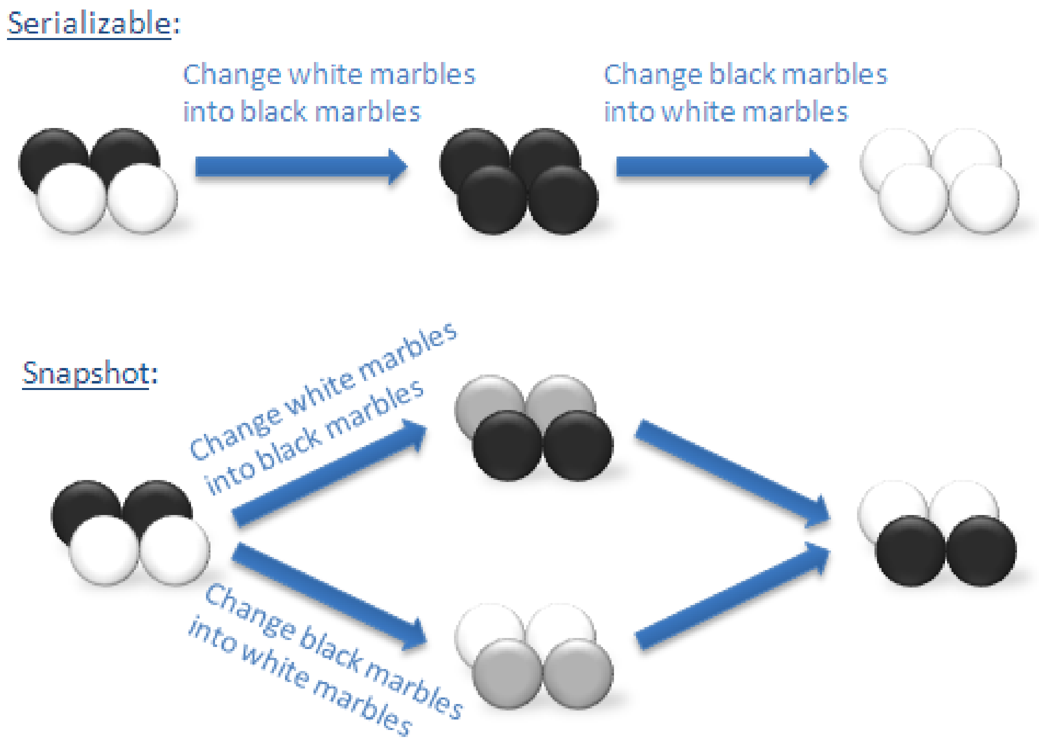
Levels of Consistency in SQL-92

- Consistency levels (from highest to lowest):
  - **Serializable** — default
  - **Snapshot isolation** — (not part of SQL-92) only committed records to be read, reads must return the value present at the beginning of transaction; better performance while retaining most of serializability.
  - **Repeatable read** — only committed records to be read, repeated reads of same record must return same value.
    - However, a transaction may not be serializable:
      it may find some new records inserted by a committed transaction.
  - **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
  - **Read uncommitted** — even uncommitted records may be read.

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default

Levels of Consistency

- Snapshot isolation does not mean serializable!
- Example:
  - One transaction turns each of the white marbles into black marbles.
  - The second transaction turns each of the black marbles into white marbles.



Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
  - A transaction begins implicitly.
    - Some systems may use **begin** to start a new transaction
  - A transaction ends by:
    - **Commit:** commits current transaction and begins a new one.
    - **Rollback:** causes current transaction to abort.

- Often, SQL statement also commits implicitly if it executes successfully
  - Mainly when libraries are used to access database.
  - Implicit commit can be turned off
    - E.g. in JDBC, connection.setAutoCommit(false);

Summary – Takeaways

- Definition of transaction
- ACID properties
- Simultaneous execution of transactions
  - schedule
  - serializability of schedules
  - levels of transaction isolation