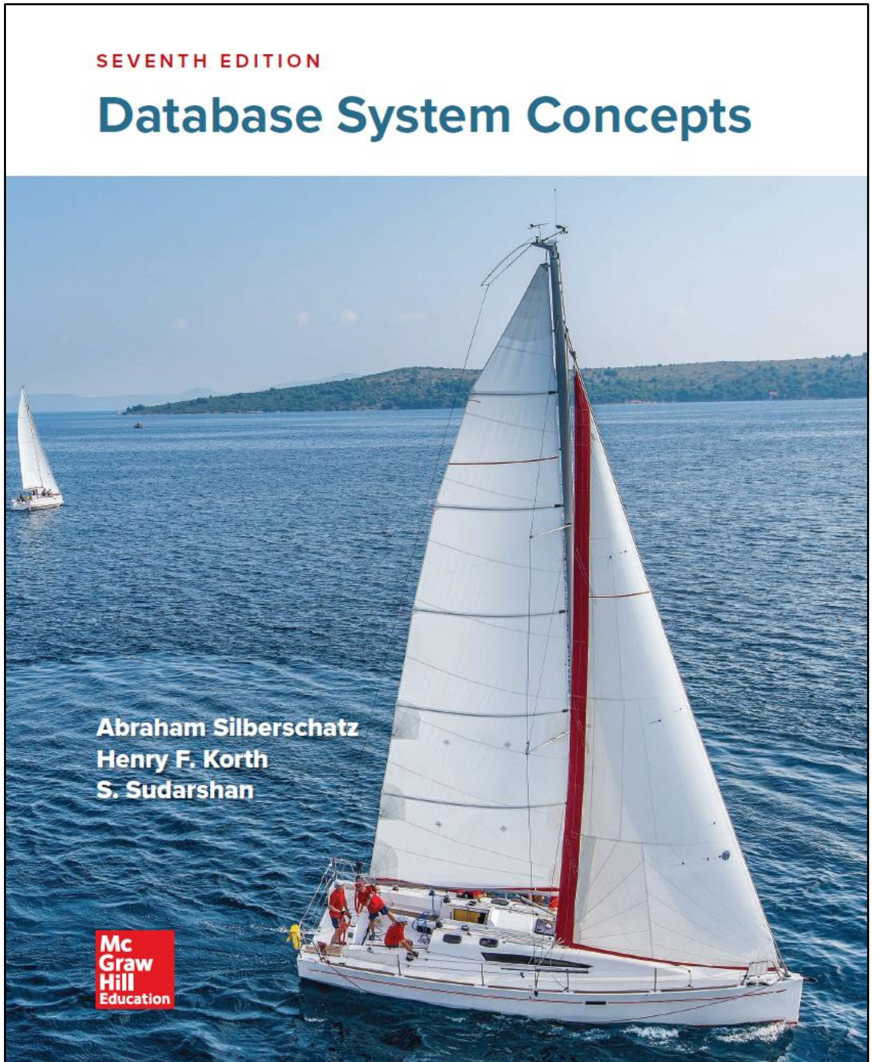
Data Administration in Information Systems

Transactions

Transactions and concurrency



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*)

  update *account* **set** *balance* = *balance* – 50 **where** *account_number* = ...

  update *account* **set** *balance* = *balance* + 50 **where** *account_number* = ...

- Two main issues to deal with:
    - Concurrent execution of multiple transactions
    - Failures of various kinds, such as hardware failures and system crashes

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)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - 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.)

- **Consistency requirement** in above example:
  - 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.)

- **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).

T1	**T2**
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), **read**(B), **print**(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

- Isolation can be ensured trivially by running transactions **serially**
  - i.e. one after the other
- However, executing multiple transactions concurrently has significant benefits

ACID Properties

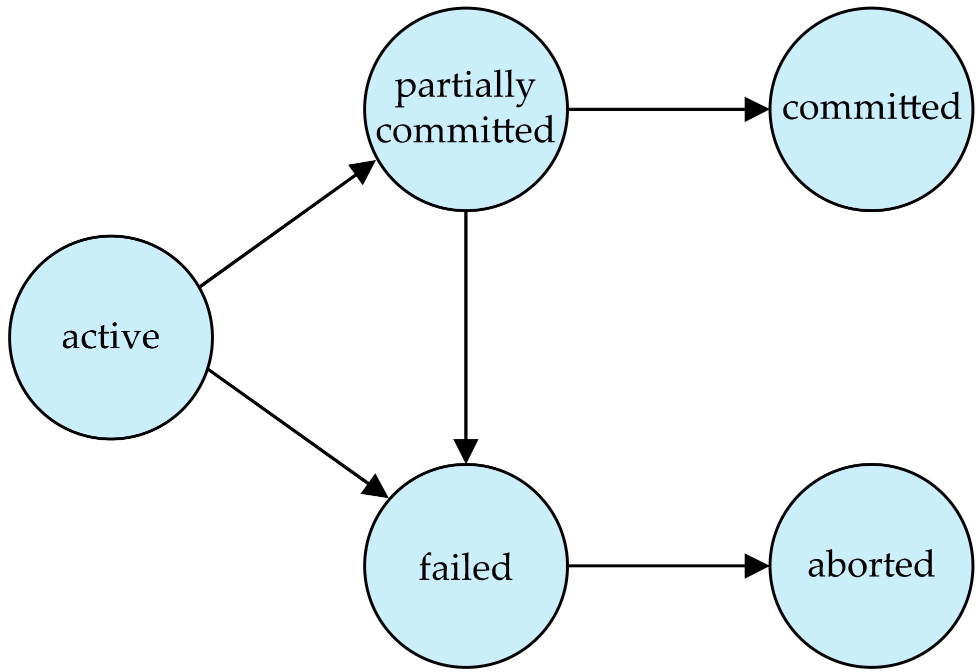
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **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.
- **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
- **Committed** – after successful completion.

Transaction State (Cont.)



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: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - i.e. to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- **Schedule** – a sequences 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, a transaction is assumed to execute a commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

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

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

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

- 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

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A concurrent schedule is serializable if it is equivalent to a serial schedule.
- We focus on a particular form of schedule equivalence called **conflict serializability**

Conflicting Instructions

- There is a **conflict** between transactions T_i and T_j if and only if there exists some item Q accessed by both transactions, and at least one of them writes Q.
  1. T_i : **read**(Q) T_j : **read**(Q) No conflict
  2. T_i : **read**(Q) T_j : **write**(Q) Conflict
  3. T_i : **write**(Q) T_j : **read**(Q) Conflict
  4. T_i : **write**(Q) T_j : **write**(Q) Conflict
- Intuitively, a conflict between T_i and T_j forces a (logical) temporal order between them.
- If the instructions of T_i and T_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 6, a serial schedule where T_2 follows T_1, by 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 6

Conflict Serializability (Cont.)

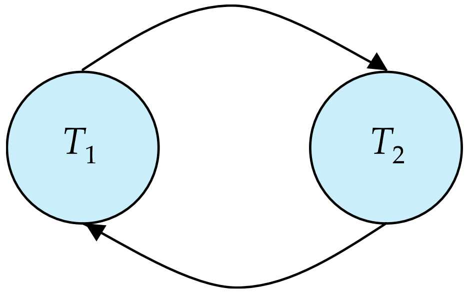
- Example of a schedule that is not conflict serializable:

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 >.

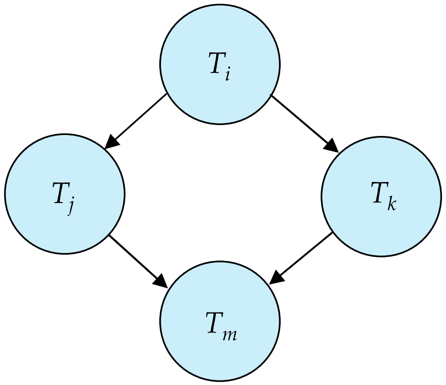
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph

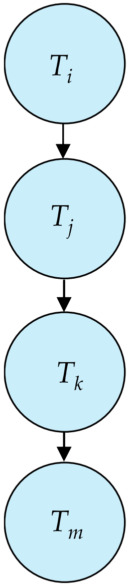


Test for Conflict Serializability

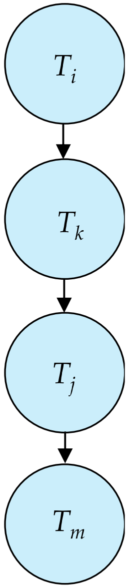
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- If precedence graph is acyclic, the serializability order can be obtained by a *linear sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for schedule (a) could be (b) or (c).



(a)



(b)



(c)

Test for Conflict Serializability: Examples

- The precedence graph for this schedule does not contain cycles
  - It is conflict serializable

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

Test for Conflict Serializability: Examples

- The precedence graph for this schedule contains a cycle
  - It is not conflict serializable

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

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.

Other Notions of Serializability

- The schedule below produces same outcome as the serial schedule < T_1 , T_5 >, yet is not conflict serializable.

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 analysis of operations other than **read** and **write**.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if transaction T_j reads a data item previously written by a transaction T_i , then the **commit** of T_j must appear after the **commit** of T_i
- The following schedule is not recoverable:

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

  – If T_8 rolls back, T_9 has read an inconsistent database state.
  – 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.
  – This can lead to the undoing of a significant amount of work.

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur.
  - If transaction T_j reads a data item previously written by a transaction T_i , then the **read** of T_j must appear after the **commit** of T_i.
- Every **cascadeless schedule** is also **recoverable**
  - Because if the **read** of T_j appears after the **commit** of T_i, then the **commit** of T_j will also appear after the **commit** of T_i.
- 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
  - **serializable**, and
  - **recoverable**, preferably **cascadeless**
- If only one transaction executes at a time, this generates serial schedules, but provides a poor degree of concurrency
  - Concurrency-control schemes allow concurrency while trying to comply with the requirements above.
- Testing a schedule for serializability after it has been executed is too late!
- **Goal** – develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency control protocols allow concurrent schedules, but ensure that the schedules are serializable, recoverable, and preferably cascadeless.
- Concurrency control protocols do not have access to the precedence graph until the transactions are finished.
  - Therefore, a protocol imposes a discipline that avoids non-serializable schedules (more about this later).
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

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

Levels of Consistency in SQL

- **Serializable** — ensures serializable execution.
- **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 records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of a record may return different (committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency in SQL (Cont.)

Isolation level	Dirty reads	Non-repeatable reads	Phantom reads
Serializable	no	no	no
Repeatable read	no	no	yes
Read committed	no	yes	yes
Read uncommitted	yes	yes	yes

- **Dirty reads**: the transaction can see the changes being done by other running transactions which have not commited yet.
- **Non-repeatable reads**: the data in a record may appear to change due to other transactions that have committed in the meantime.
- **Phantom reads**: the number of records in a table may appear to change due to other transactions that have committed in the meantime.

Levels of Consistency in SQL (Cont.)

- Lower degrees of consistency useful for gathering approximate information about the database
- Some systems do not ensure serializable schedules by default
  - Default isolation level is typically **read committed** or **repeatable read**
- Some systems have additional isolation levels
  - **Snapshot isolation** (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly
  - By default, each statement is a transaction that commits upon successful execution.
  - "Auto-commit" can be turned off, if desired.
- Explicit transactions start with **begin transaction** and end with **commit** or **rollback**
  - In most systems, the transaction is rolled back automatically upon error.
- The isolation level can be changed before the start of a new transaction
  - With the command **set transaction isolation level ...**