Introduction to DBMS
Transaction Processing

Prof. Dr. Agnès Voisard

Institute of Computer Science Databases and Information Systems Group and Fraunhofer FOKUS

Summer term 2025

v2

Contents

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

Read/Write operations

- Level of data items and disk blocks. Data item: field of a DB record, record, whole block
- DB operations
  - **Read-item (X)** or **read (X).** Execution:
    - Find the address of the disk block that contains X
    - Copy the disk block into a buffer in main memory
    - Copy item X from the buffer to the program variable
  - **Write-item (X)** or **write (X)**. Execution:
    - Find the address of the disk block that contains X
    - Copy the disk block into a buffer
    - Copy item X from the program variable X into its correct place in the buffer
    - Store the updated block from the buffer back to disk (update of DB on disk)

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

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
    - 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'd)

- 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*)
- **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 the 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'd)

- **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)	**Dirty read**
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

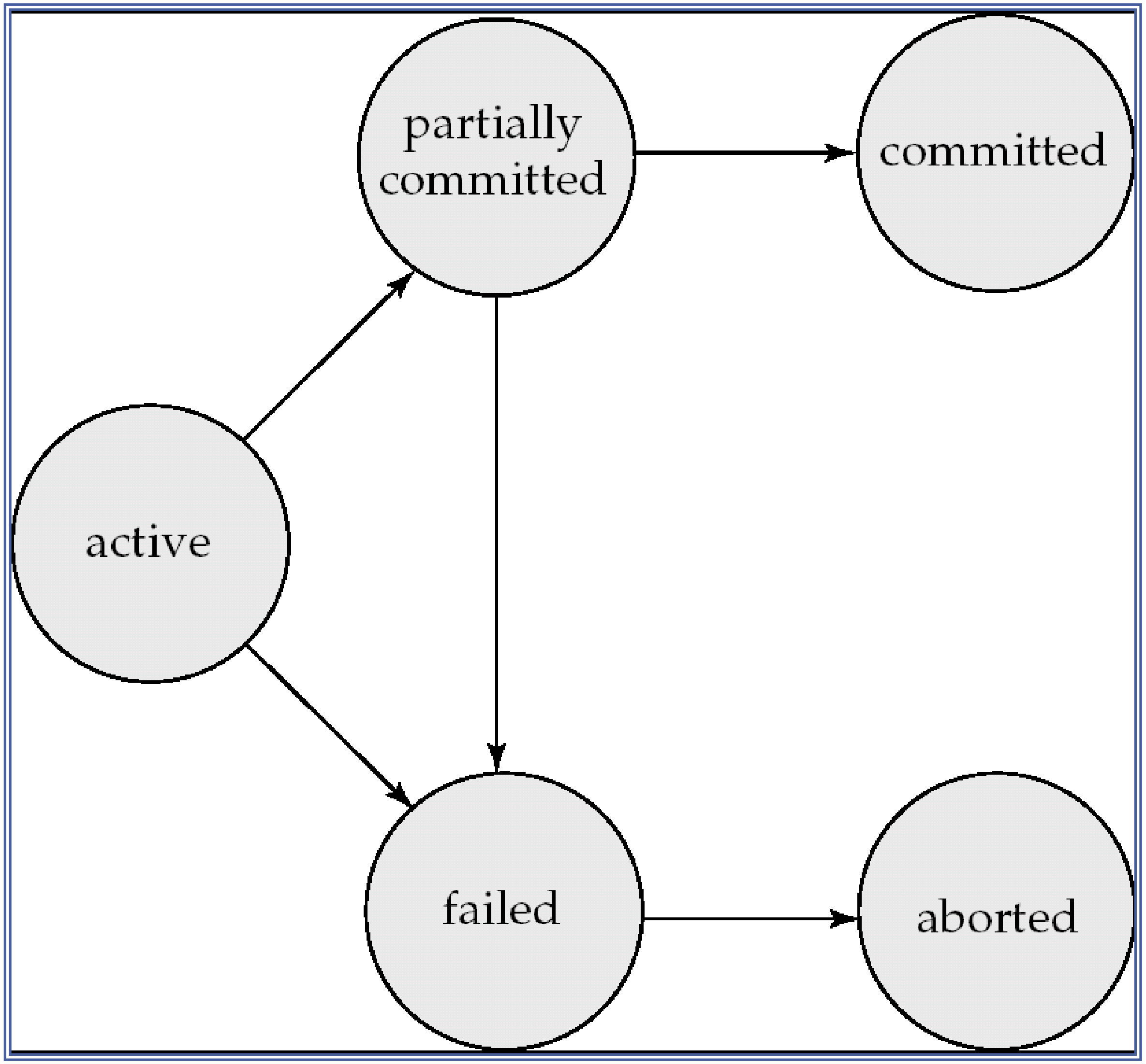
A **transaction** is a unit of program execution that accesses and possibly updates various data items.To preserve the integrity of data the database system must ensure:

- **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
- **Committed** – After successful completion
- **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
  - Error kill the transaction

Transaction State (cont'd)



Introduction to recovery

- Transaction can fail in the middle of execution

- Types of failure:
  - System crash
  - Transaction or system error (e.g., division by 0)
  - Errors detected during the transaction (e.g., data not found)
  - Concurrency control enforcement (transaction has to be aborted bc of deadlocks, see later)
  - Disk failure (blocks lose their data. May happen during a read/write operation)
  - Physical problems and catastrophes: power failure, sabotage, etc.

Introduction to recovery (cont'd)

Recovery manager keeps track of when the transaction starts, terminates, commits, or abort:

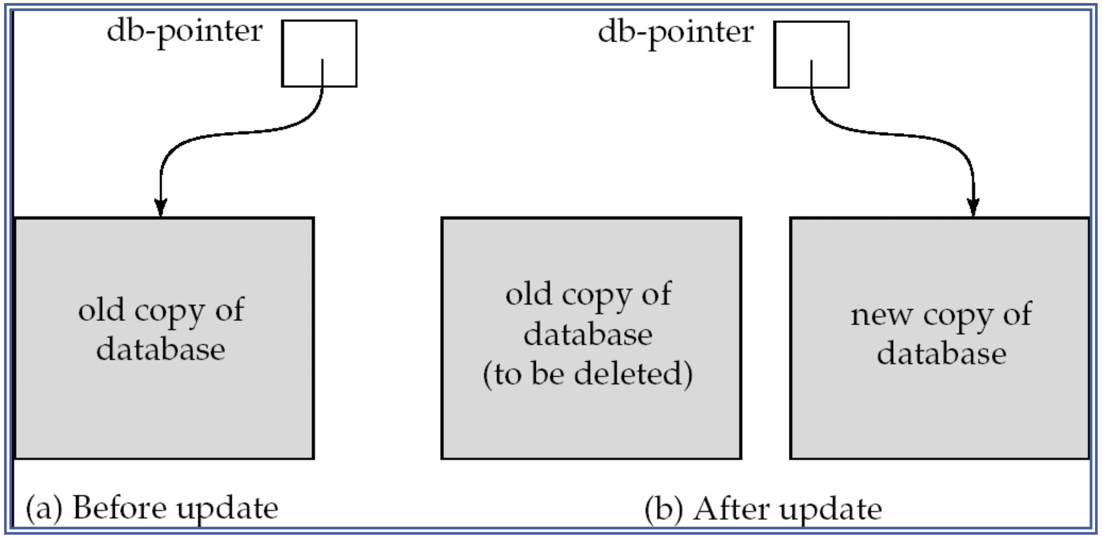
- BEGIN TRANSACTION
- READ or WRITE DB item
- COMMIT-TRANSACTION
  - Signals a successful end of the transaction
- ROLL-BACK (or ABORT)
  - Transaction has ended unsuccessfully
  - Any change done by the transaction must be undone
- END-TRANSACTION: READ and WRITE have ended
  - End of the transaction BUT it must be checked:
    - Whether changes can be permanently applied to the DB (COMMIT) or
    - Whether the transaction has to be aborted

+ UNDO/REDO, see later

- This is written is a journal (log)

Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability
- E.g., the ***shadow-database*** scheme:
  - All updates are made on a *shadow copy* of the database
    - **db_pointer** is made to point to the updated shadow copy after
      - The transaction reaches partial commit and
      - All updated pages have been flushed to disk.



Implementation of Atomicity and Durability (cont'd)

- db_pointer always points to the current consistent copy of the database
  - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted
- The shadow-database scheme:
  - Assumes that only one transaction is active at a time
  - Assumes disks do not fail
  - Useful for text editors, but
    - Extremely inefficient for large databases
    - Variant called shadow paging reduces copying of data, but is still not practical for large databases
  - Does not handle concurrent transactions

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 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
  - A schedule 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 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)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

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

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

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

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_x and T_y respectively, **conflict** if and only if there exists some item Q accessed by both l_x and l_y, and at least one of these instructions wrote Q.

  1. l_x = **read**(Q), l_y = **read**(Q). l_x and l_y don't conflict
  2. l_x = **read**(Q), l_y = **write**(Q). They conflict
  3. l_x = **write**(Q), l_y = **read**(Q). They conflict
  4. l_x = **write**(Q), l_y = **write**(Q). They conflict

- Intuitively, a conflict between l_x and l_y forces a (logical) temporal order between them
  - If l_x and l_y 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'd)

- 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'd)

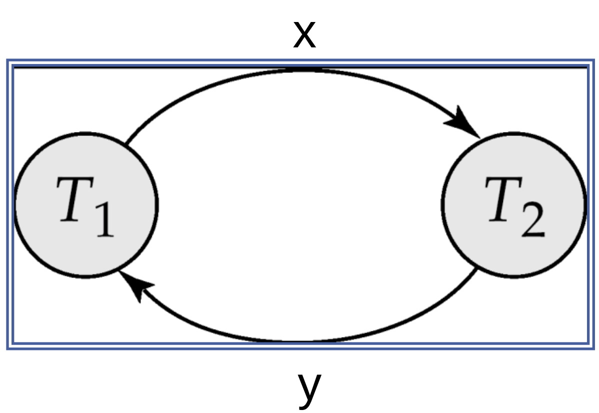
- Example of a schedule that is not conflict serializable:

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 $T_1, T_2, \ldots, T_n$
- Precedence graph: a direct graph where the vertices are the transactions (names)
- We draw an arc from T_x to T_y if the two transaction conflict, and T_x accessed the data item on which the conflict arose earlier
- We may label the arc by the item that was accessed.
- **Example 1**



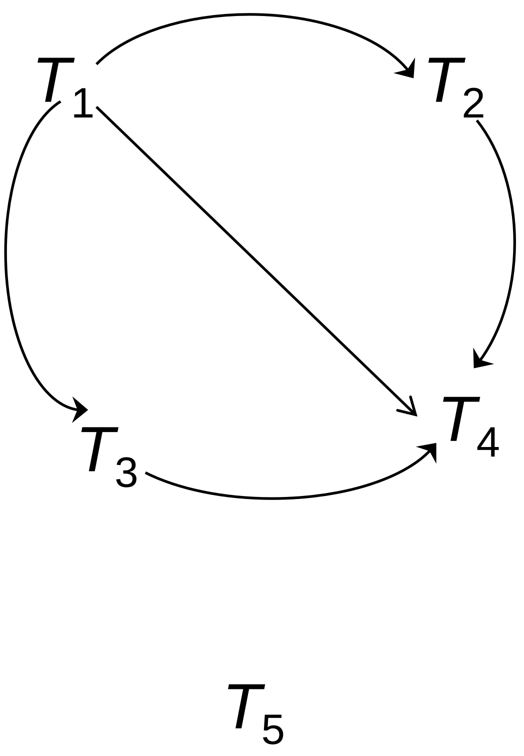
Precedence graph - constructionAlgorithm

- Given: schedule S.

1. For each T_x in S create a node
2. For each case in S where T_y does a read (X) after a write (X) by T_x
   - T_x: `write(X); ` T_y: `read(X);`
   - Create (T_x -> T_y)
3. For cases where T_y does a write (X) after T_x does a read (X)
   - T_x: `read (X); ` T_y: `write (X);`
   - Create (T_x -> T_y)
4. For cases where T_y does a write (X) after T_x does a write (X)
   - T_x: `write (X); ` T_y: `write (X);`
   - Create (T_x -> T_y)
5. S is serializable if the precedence graph has no cycle

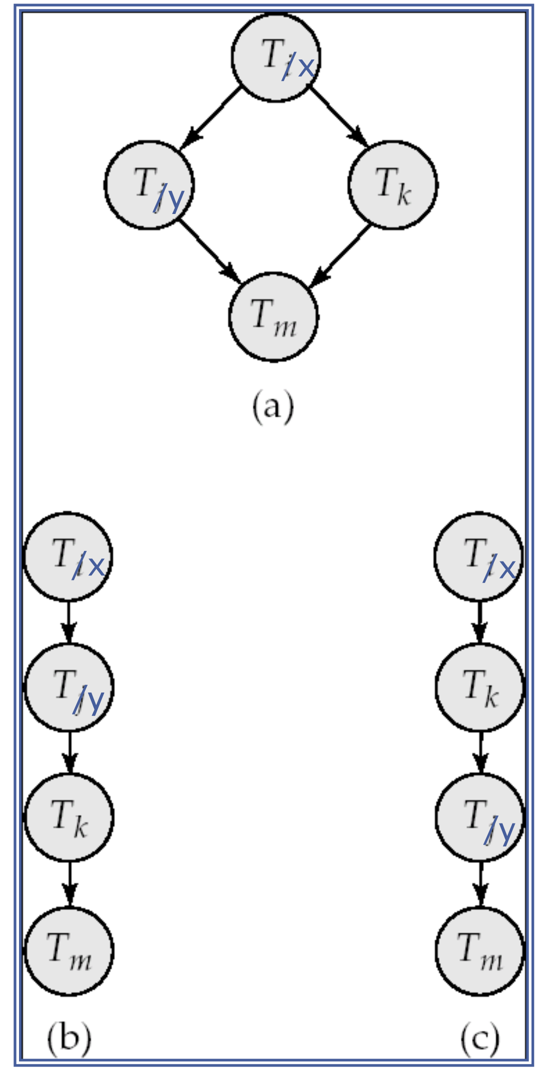
Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	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)				
write(U)				



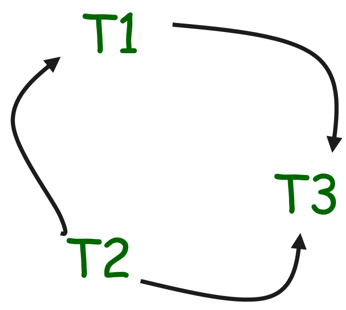
Test for Conflict Serializability

- A schedule is **conflict** serializable if and only if its precedence graph is acyclic
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph
  Better algorithms take order $n + e$ where e is the number of edges
- If 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
  - For example, a serializability order for Schedule (a) would be
    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$



Simple notation

- S, transactions T1, T2, T3, items u, x, y
- S: r1(y), r3(u), r2(y), w1(y), w2(x), w1(x), w2(z), w3(x)
- Precedence graph



Serializable!

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - Either **conflict** or **view serializable**, and
  - Are recoverable and preferably cascadeless (see later)

- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency

- Testing a schedule for serializability *after* it has executed is a little too late!

- **Goal**: To 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 conflict/view serializable, and are recoverable and cascadeless
- Concurrency control protocols generally **do not examine the precedence graph** as it is being created
  - Instead, a protocol imposes a discipline that **avoids non-seralizable schedules**
- 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 **accuracy** for **performance**

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one
  - **Rollback work** causes current transaction to abort
- In almost all database systems, by default, every SQL statement also **commits implicitly if it executes successfully**
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC, connection.setAutoCommit(false);

Summary

- Transaction concept
- ACID properties
- Introduction to recovery
- Serial schedules
- Conflict (precedence) graph

Levels of Consistency in SQL-92

- **Serializable** — default
- **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, 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
  - E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)