

Databases 2

Transactions

Nikolaus Augsten

`nikolaus.augsten@plus.ac.at`

FB Informatik

Universität Salzburg



database
research group

<https://dbresearch.uni-salzburg.at>

WS 2024/25

Version January 15, 2025

Outline

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability
- 4 Recoverability
- 5 Concurrency Protocols
- 6 Deadlocks
- 7 Implementation of Isolation / SQL

Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability
- 4 Recoverability
- 5 Concurrency Protocols
- 6 Deadlocks
- 7 Implementation of Isolation / SQL

What is a Transaction?

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- **Example:** transfer \$50 from account A to account B
 1. $R(A)$
 2. $A \leftarrow A - 50$
 3. $W(A)$
 4. $R(B)$
 5. $B \leftarrow B + 50$
 6. $W(B)$
- Two **main issues**:
 1. concurrent execution of multiple transactions
 2. failures of various kind (e.g., hardware failure, system crash)

ACID Properties

- Database system must guarantee **ACID for transactions**:
 - **Atomicity**: either all operations of the transaction are executed or none
 - **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 the other concurrent transactions.
 - **Durability**: After a transaction completes successfully, changes to the database persist even in case of system failure.

Atomicity

- **Example:** transfer \$50 from account A to account B
 1. $R(A)$
 2. $A \leftarrow A - 50$
 3. $W(A)$
 4. $R(B)$
 5. $B \leftarrow B + 50$
 6. $W(B)$
- What if **failure** (hardware or software) after step 3?
 - money is lost
 - database is inconsistent
- **Atomicity:**
 - either all operations or none
 - updates of partially executed transactions not reflected in database

Consistency

- **Example:** transfer \$50 from account A to account B
 1. $R(A)$
 2. $A \leftarrow A - 50$
 3. $W(A)$
 4. $R(B)$
 5. $B \leftarrow B + 50$
 6. $W(B)$
- **Consistency in example:** sum $A + B$ must be unchanged
- **Consistency in general:**
 - explicit integrity constraints (e.g., foreign key)
 - implicit integrity constraints (e.g., sum of all account balances of a bank branch must be equal to branch balance)
- **Transaction:**
 - must see consistent database
 - during transaction inconsistent state allowed
 - after completion database must be consistent again

Isolation – Motivating Example

- **Example:** transfer \$50 from account A to account B
 1. $R(A)$
 2. $A \leftarrow A - 50$
 3. $W(A)$
 4. $R(B)$
 5. $B \leftarrow B + 50$
 6. $W(B)$
- Imagine second transaction T_2 :
 - $T_2 : R(A), R(B), \text{print}(A + B)$
 - T_2 is executed between steps 3 and 4
 - T_2 sees an inconsistent database and gives wrong result

Isolation

- **Trivial isolation**: run transactions serially
- **Isolation** for concurrent transactions: For every pair of transactions T_i and T_j , it appears to T_i as if either T_j finished execution before T_i started or T_j started execution after T_i finished.
- **Schedule**:
 - specifies the **chronological order** of a sequence of instructions from various transactions
 - **equivalent schedules** result in identical databases if they start with identical databases
- **Serializable** schedule:
 - equivalent to some serial schedule
 - serializable schedule of T_1 and T_2 is either equivalent to T_1, T_2 or T_2, T_1

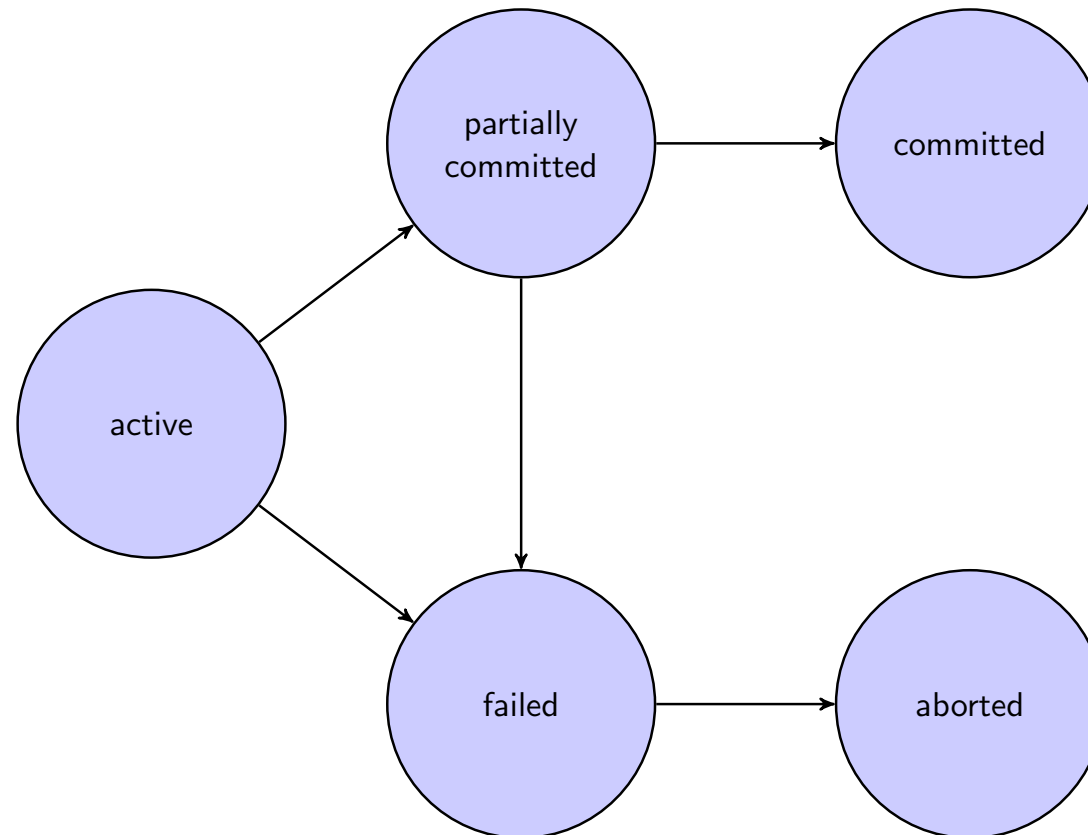
Durability

- When a transaction is done it **commits**.
- **Example**: transaction commits too early
 - transaction writes *A*, then commits
 - *A* is written to the disk buffer
 - then system crashes
 - value of *A* is lost
- **Durability**: After a transaction has committed, the changes to the database persist even in case of system failure.
- **Commit** only after all changes are permanent:
 - either written to log file or directly to database files
 - database must recover in case of a crash

Transaction State/1

- **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/2



Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions**
- 3 Serializability
- 4 Recoverability
- 5 Concurrency Protocols
- 6 Deadlocks
- 7 Implementation of Isolation / SQL

Concurrent Executions

- Multiple transactions are allowed to run **concurrently** in the system.
- **Advantages** of concurrent transactions:
 - **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.
- Concurrent transactions require **concurrency control protocol**:
 - mechanisms to **achieve isolation**
 - **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:
 - must **consist of all instructions** of the concurrent 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.
- 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 .
- An example of a **serial** schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

T_1	T_2
	<i>read(A)</i>
	<i>temp := A * 0.1</i>
	<i>A := A - temp</i>
	<i>write(A)</i>
	<i>read(B)</i>
	<i>B := B + temp</i>
	<i>write(B)</i>
	<i>commit</i>
<i>read(A)</i>	
<i>A := A - 50</i>	
<i>write(A)</i>	
<i>read(B)</i>	
<i>B := B + 50</i>	
<i>write(B)</i>	
<i>commit</i>	

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
<i>read(A)</i>	
$A := A - 50$	
<i>write(A)</i>	
	<i>read(A)</i>
	$temp := A * 0.1$
	$A := A - temp$
	<i>write(A)</i>
<i>read(B)</i>	
$B := B + 50$	
<i>write(B)</i>	
<i>commit</i>	
	<i>read(B)</i>
	$B := B + temp$
	<i>write(B)</i>
	<i>commit</i>

Note — In schedules 1, 2 and 3, the sum “ $A + B$ ” is preserved.

Schedule 4

- The following concurrent schedule does not preserve the sum of " $A + B$ "

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

Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability**
- 4 Recoverability
- 5 Concurrency Protocols
- 6 Deadlocks
- 7 Implementation of Isolation / SQL

Concurrent Executions

- **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:
 - **conflict serializability**
 - **view serializability**

Simplified model 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

- Conflicts of read and write instructions:

$T_i \downarrow \quad T_j \rightarrow$	$I_j = \text{read}$	$I_j = \text{write}$
$I_i = \text{read}$	no conflict	conflict
$I_i = \text{write}$	conflict	conflict

- Intuitively, a conflict between two instructions I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_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/1

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, then S and S' are **conflict equivalent**.
- A schedule S is conflict serializable if it is **conflict equivalent** to a serial schedule.

Conflict Serializability/2

- Schedule 3 and (serial) Schedule 6 are conflict equivalent, therefore Schedule 3 is serializable.

T_1	T_2
<i>read(A)</i>	
<i>write(A)</i>	
	<i>read(A)</i>
	<i>write(A)</i>
<i>read(B)</i>	
<i>write(B)</i>	
	<i>read(B)</i>
	<i>write(B)</i>

Table: Schedule 3

T_1	T_2
<i>read(A)</i>	
<i>write(A)</i>	
<i>read(B)</i>	
<i>write(B)</i>	
	<i>read(A)</i>
	<i>write(A)</i>
	<i>read(B)</i>
	<i>write(B)</i>

Table: Schedule 6

Conflict Serializability/3

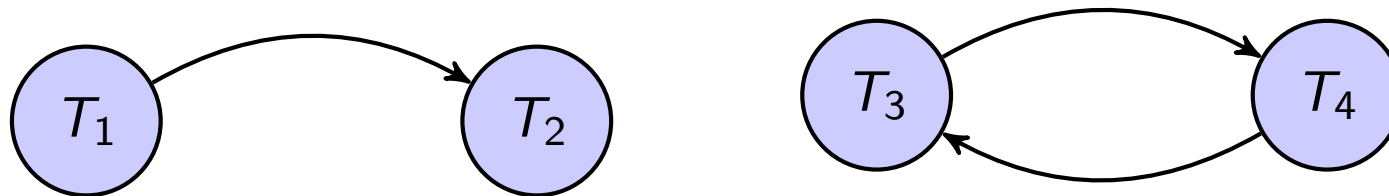
- Example of a schedule that is **not conflict serializable**:

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

- We are **unable to swap instructions** in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.

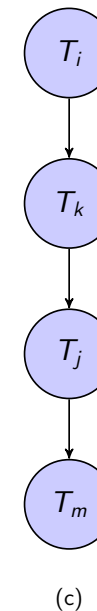
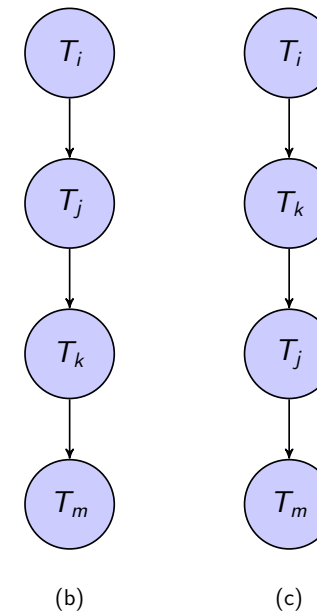
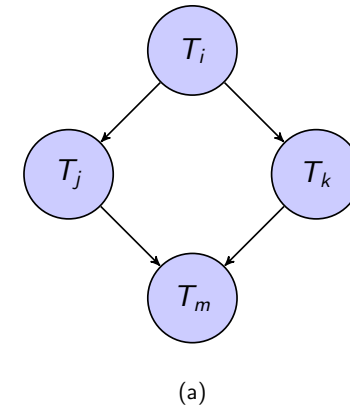
Precedence Graph

- Consider some **schedule** of a set of transactions T_1, T_2, \dots, 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 transaction 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**



Testing 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 the precedence graph is acyclic, the serializability order can be obtained by a **topological sorting** of the graph.
 - That is, a **linear order consistent with the partial order** of the graph.
 - For example, a serializability order for the schedule (a) would be one of either (b) or (c)



Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability
- 4 Recoverability**
- 5 Concurrency Protocols
- 6 Deadlocks
- 7 Implementation of Isolation / SQL

Recoverable Schedules

- **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 **must** appear before the commit operation of T_j .
- The **following schedule is not recoverable**: T_9 reads A written by T_8 but commits before T_8 .

T_8	T_9
$read(A)$	
$write(A)$	
	$read(A)$
	$C \leftarrow A$
	$write(C)$
	$commit$
$read(B)$	

- If T_8 aborts, T_9 has read and copied 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}
<i>read(A)</i>		
<i>read(B)</i>		
<i>write(A)</i>		
	<i>read(A)</i>	
	<i>write(A)</i>	
		<i>read(A)</i>
<i>abort</i>		

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** — 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**.
- Example of a schedule that is NOT cascadeless:

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

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

Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability
- 4 Recoverability
- 5 Concurrency Protocols**
- 6 Deadlocks
- 7 Implementation of Isolation / SQL

Concurrency Protocols

- A database must provide a mechanism that will ensure that all possible schedules are both:
 - serializable
 - recoverable and preferably cascadeless
- A concurrency protocol is a policy to guarantees serializable schedules.
- Serial schedule: A policy in which only one transaction can execute at a time provides a poor degree of concurrency.
- Various protocols allow concurrent schedules that are serializable:
 - lock-based protocols
 - timestamp ordering protocols
 - validation-based protocols
 - multi-version concurrency control

Lock-Based Protocols/1

- A **lock** is a mechanism to **control concurrent access** to a data item.
- Data items can be locked in **two modes**:
 1. **exclusive (X)** mode. Data item can be both **read as well as written**. X-lock is requested using **lock-X** instruction.
 2. **shared (S)** mode. Data item can **only be read**. S-lock is requested using **lock-S** instruction.
- Lock requests are made to the **concurrency-control manager** by the programmer. Transaction can proceed only after request is granted.

Lock-Based Protocols/2

- A lock on an item is granted to a transaction if the requested lock is **compatible with locks already held** on the item by other transactions.
- Lock-compatibility matrix:

	<i>S</i>	<i>X</i>
<i>S</i>	<i>true</i>	<i>false</i>
<i>X</i>	<i>false</i>	<i>false</i>

- Any number of transactions can hold a **shared lock** on an item.
- If any transaction holds an **exclusive lock** on the item, **no other transaction** may hold any lock on the item.
- If a **lock cannot be granted**, the requesting transaction is made to **wait** till all incompatible locks held by other transactions have been released. The lock is then granted.

Lock-Based Protocols/3

- Example of a transaction performing locking:

T_2 : **lock-S**(A)
 read(A)
 unlock(A)
 lock-S(B)
 read(B)
 unlock(B)
 display($A + B$)

- Locking is **not sufficient to guarantee serializability**: if A gets updated in-between the read of A and B , the displayed sum is wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol/1

- In the **Two-Phase Locking (2PL)** protocol, each transaction must go through two phases that restrict the order in which locks can be granted and released.
- Phase 1: **Growing Phase**
 - transaction may obtain locks
 - transaction may not release locks
- Phase 2: **Shrinking Phase**
 - transaction may release locks
 - transaction may not obtain locks

The Two-Phase Locking Protocol/2

- The 2PL protocol guarantees conflict serializability.
- The transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).
- The set of 2PL schedules is a subset of conflict serializable schedules, i.e., there can be conflict serializable schedules that cannot be obtained with 2PL.
- 2PL is necessary: In the absence of extra information (e.g., ordering of access to data) a locking protocol that does not follow 2PL cannot guarantee conflict serializability.

Timestamp Ordering Protocols

- Each transaction gets a timestamp when it enters the system.
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- Each data item Q gets two timestamp values:
 - Write timestamp: timestamp of youngest transaction that wrote Q .
 - Read timestamp: timestamp of youngest transaction that read Q .
- The timestamp ordering protocol ensures that any conflicting operations are executed in timestamp order.

Validation-Based Protocols

- **Optimistic** approach: Execute transaction first and check for serializability problems at the end.
- Execution of transaction T_i is done in **three phases**:
 1. **Read and execution phase**: Transaction T_i writes only to temporary local variables.
 2. **Validation phase**: Transaction T_i performs a **validation test** to determine if local variables can be written without violating serializability.
 3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.

Multiversion Concurrency Control (MVCC)

- MVCC schemes **keep old versions of data** item to increase concurrency.
- **Each successful write** results in the creation of a **new version** of the written data item.
- **Readers are never blocked**: an appropriate version of the data item is returned based on the timestamp of the reading transaction.
- **Snapshot Isolation**: MVCC scheme implemented e.g. in PostgreSQL.
 - each transaction gets a snapshot (conceptually a copy) of the database at its start
 - transaction operates on its snapshot and does not see updates of other transactions
 - conflicting updates are dealt with at time of update (first updater wins) or commit (first committer wins)

Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability
- 4 Recoverability
- 5 Concurrency Protocols
- 6 Deadlocks**
- 7 Implementation of Isolation / SQL

Deadlocks/1

- Consider the partial schedule

T_3	T_4
$lock-x(B)$	
$read(B)$	
$B := B - 50$	
$write(B)$	
	$lock-s(A)$
	$read(A)$
	$lock-s(B)$
$lock-x(A)$	

- Neither T_3 nor T_4 **can make progress** — executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B , while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A .
- Such a situation is called a **deadlock**.
- To handle the deadlock, one of T_3 or T_4 must be **aborted** and its **locks released**.

Deadlocks/2

- Two-phase locking **does not ensure freedom from deadlocks**.
- In addition to deadlocks, there is a possibility of starvation.
- **Starvation** occurs if the concurrency control manager is badly designed. For example:
 - The same transaction is **repeatedly rolled back** due to deadlocks.
 - A transaction **waits for an X-lock** on an item, while a sequence of other transactions request and are **granted an S-lock** on the same item.
- Concurrency control manager **can be designed to prevent** starvation.

Deadlocks/3

- The **potential for deadlock exists in most** locking protocols. Deadlocks are a necessary evil.
- When a deadlock occurs there is a possibility of **cascading rollbacks**.
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking** — a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter. Here, all locks are held till commit/abort. In this protocol, transactions can be serialized in the order in which they commit.

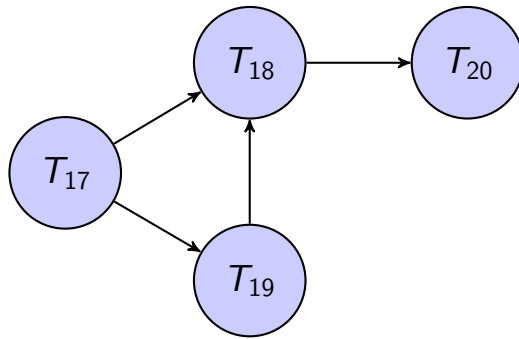
Deadlock Handling

- A system is **deadlocked** if there is a set of transactions such that every transaction in the set is **waiting for another** transaction in the set.
- How to deal with deadlocks?
 1. **Detection & Recovery:** allow deadlocks to happen and recover from the deadlock state.
 2. **Prevention:** ensure that the system will never enter into a deadlock state.

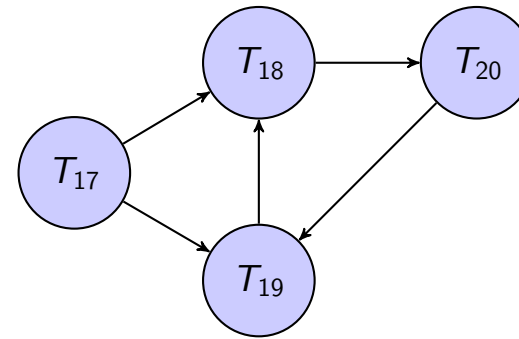
Deadlock Detection/1

- Deadlocks can be described as a **wait-for graph**, which consists of a pair $G = (V, E)$,
 - V is a set of **vertices** (all the **transactions** in the system)
 - E is a set of **edges**; each element is an **ordered pair** $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E , then there is a **directed edge** from T_i to T_j , implying that T_i is **waiting for** T_j to release a data item.
- When T_i **requests** a data item currently being held by T_j , then the edge $T_i \rightarrow T_j$ is **inserted** in the wait-for graph. This edge is **removed** only when T_j is **no longer holding** a data item needed by T_i .
- The system is in a **deadlock state** if and only if the **wait-for graph has a cycle**. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection/2



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- To **recover from a deadlock** state, some transaction must be aborted.
- How to **pick a victim** (transaction to be aborted)?
 - Select a transaction as victim that will **incur minimum cost**.
 - **Starvation** happens if same transaction is always chosen as victim.
 - Include the **number of rollbacks** into the cost factor to avoid starvation.
- How far to **roll back** victim transaction?
 - **total rollback**: abort the transaction and then restart it
 - more efficient to roll back transaction only **as far as necessary** to break deadlock

Deadlock Prevention Strategies/1

1. **Predeclaration:** Require that each transaction **locks all its data items** before it begins execution.
 - Problem: need to know data items to be locked upfront.
2. **Lock Order:** Impose order on all data items. Transaction can **lock only in the specified order**.
 - Easy to implement on top of existing 2PL implementation.
 - Problem: need to know data items to be locked upfront.
3. **Timeout-Based schemes:**
 - A transaction **waits** for a lock only for a **specified amount of time**.
 - **Roll back and restart** transaction if lock cannot be granted within timeout interval.
 - Problem: difficult to determine good value of the timeout interval.

Deadlock Prevention Strategies/2

4. Preemptive and non-preemptive scheme based on timestamps:

- Transactions have a **timestamps**: Older transactions (smaller timestamp) have precedence over younger transactions.
- **Preemptive**: Younger transaction is aborted if it holds a lock required by an older one (called wound-wait scheme).
- **Non-preemptive**: Younger transaction is aborted if it request a lock held by and older one (called wait-die scheme)
- A rolled back transactions is **restarted with its original timestamp**.

Inhalt

- 1 Transaction Concept
- 2 Concurrent Executions
- 3 Serializability
- 4 Recoverability
- 5 Concurrency Protocols
- 6 Deadlocks
- 7 Implementation of Isolation / SQL**

Weak Levels of Consistency

- Concurrency control protocols make a **trade-off** between the amount of concurrency they allow and the amount of overhead they impose.
- **Trade off** accuracy for performance: Some applications are willing to live with weak levels of consistency, **allowing schedules that are not serializable**.
- **SQL** defines three undesired phenomena of concurrent transactions and isolation levels to avoid them.

Undesirable Phenomena of Concurrent Transactions

- Dirty read

- transaction reads data written by concurrent uncommitted transaction
- problem: read may return a value that was never in the database because the writing transaction aborted

- Non-repeatable read

- different reads on the same item within a single transaction give different results (caused by other transactions)

- Example: Execute concurrently:

$T_1: x = R(A), y = R(A), z = y - x$

$T_2: W(A = 2 * A), \text{COMMIT}$

Then z can be either zero or the initial value of A (should be zero!).

- Phantom read

- repeating the same query later in the transaction gives a different set of result tuples
- other transactions can insert new tuples during a scan
- e.g., “Q: get accounts with *balance* > 1000” gives two tuples the first time, then a new account with *balance* > 1000 is inserted by an other transaction; the second time Q gives three tuples

Isolation Guarantees (SQL Standard)

- **Read uncommitted**: dirty, non-repeatable, phantom
 - reads may access uncommitted data
 - writes do not overwrite uncommitted data
- **Read committed**: non-repeatable, phantom
 - reads can access only committed data
 - **cursor stability**: in addition, read is repeatable within single SELECT
- **Repeatable read**: phantom
 - phantom reads possible
- **Serializable**:
 - none of the undesired phenomenas can happen

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.
 - **BEGIN [TRANSACTION ISOLATION LEVEL ...]**
 - **Isolation levels**: read committed, read uncommitted, repeatable read, serializable
- A transaction in SQL ends by:
 - **COMMIT** commits current transaction and begins a new one.
 - **ROLLBACK** causes current transaction to abort.
- Typically, an SQL statement **commits implicitly** if it executes successfully
 - Implicit commit can be turned off by a database directive, e.g. in JDBC, `connection.setAutoCommit(false)`;