Transactions: Definition

- Transaction (TA)
  Unit of work consisting of a sequence of operations

- Transaction principles (ACID):
  - Atomicity: the sequence of operations is executed completely or has no effect on the database
  - Consistency: if the database was in a consistent state before transaction execution it will be after
  - Isolation: concurrently executed transactions do not interfere
  - Durability: (persistency) all effects of a TA are permanent

Transactions: Read/Write Model

- Atomic DB-operations
  - READ[x]: TA reads object x
  - WRITE[y]: TA writes object x

- Transaction: sequence of reads and writes

- Transaction control
  - Begin of transaction (BOT) - often implicitly
  - Commit: successful end of transaction
  - Abort: unsuccessful end of transaction

- Example:
  - Insert a new tape for the movie "Psycho" (Hitchcock)
  - Read movie-id, read and write sequence, insert tape-tuple
  - TA_i = r[m], r[s], w[s], w[t], c

Transactions: Management in DBS

- Concurrency
  - Situation: Multiuser access to database
  - Challenge: Find a correct execution sequence of the atomic operations of multiple transactions
  - Ensures isolation

- Recovery
  - Situation: Failing transactions
  - Challenge: Recover a consistent database state
  - Ensures atomicity
  - Ensures persistency
Concurrent transactions: Problems

- Potential problems during interleaved execution
  - Lost update: same object concurrently updated by two transactions, one update lost
    \[ r_1[x], r_1[x], w_1[x], w_1[x], c_2, c_1 \]
  - Dirty read: reading of object value changed by (later) aborted transaction
    \[ r_1[x], w_1[x], r_1[x], w_1[x], c_2, a_1 \]
  - Non-repeatable read: different result when reading the same object more than once in a transaction
    \[ r_1[x], w_1[x], r_1[x], w_1[x], r_1[x], w_1[x], c_2, c_1 \]
  - Phantoms: non-repeatable read caused by insertions or deletions

Concurrent transactions: Schedule

- Schedule (History)
  - Informally: interleaved sequence of atomic actions of two or more transactions

A schedule \( S \) of a (finite) set of transactions \( T \) is a sequence of atomic actions \( a_i \) with partial ordering \( \leq_S \) if:
- Each atomic action of each \( TA \in T \) occurs exactly once in \( S \)
- No other action occurs in \( S \)
- If \( a_i \leq_S a_j \) in some \( TA \), then \( a_i \leq_S a_j \) in \( S \)

Example:
- \( TA 1 = r1[x], r1[y], w_1[y], r_1[z], w_1[x], c_1 \)
- \( TA 2 = r2[y], r_2[z], w_2[y], r_2[x], w_2[x], r_2[s], c_2 \)
- Schedule:
  \[ r1[x], r2[y], r2[z], w_2[y], r_2[x], w_2[x], r_1[y], w_1[x], c_2, w_1[x], c_1 \]
- No schedule:
  \[ r1[x], r2[y], r_1[y], w_2[y], w_1[y], r_1[z], r_2[z], r_2[x], w_2[x], c_2, w_1[x], c_1 \]

Concurrent transactions: Serializability theory

- Serial schedule
  - The execution of the \( TA \)s one after the other, in an arbitrary order, is called a serial execution

Example:
- \( TA 1 = r1[x], r1[y], w_1[y], r_1[z], w_1[x], c_1 \)
- \( TA 2 = r2[y], r_2[z], w_2[y], r_2[x], w_2[x], r_2[s], c_2 \)
- \( T1 \) then \( T2: r1[x], r1[y], w_1[y], r_1[z], w_1[x], c_1, r2[y], r_2[z], w_2[y], r_2[x], w_2[x], r_2[s], c_2 \)
- \( T2 \) then \( T1: r2[y], r_2[z], w_2[y], r_2[x], w_2[x], r_2[s], c_2, r_1[x], r_1[y], w_1[y], r_1[z], w_1[x], c_1 \)
Concurrent transactions: Serializability theory

- Correctness criterion for schedules: Serializability
  
  A schedule S of a transaction set T is called serializable, if it is equivalent to a serial execution of T.
  
  Serializable schedules are defined as correct.
  
- Equivalence of schedules
  
  1. Result-equivalence: Equivalent if same result on database
     ⇒ not really useful, used in some books
  
  2. Conflict-equivalence: Equivalent if same same order of conflict operations
     ⇒ typically applied

- Conflicts:
  
  - Lost update, dirty read, non-repeatable read, phantoms
  - Only if different transactions operate on the same object and at least one is a write operation

- Possible conflicts between two TAs
  
  - No conflict: r1[x], r2[x]
  - Possible: r1[x], w2[x]
  - Possible: w1[x], r2[x]
  - Possible: w1[x], w2[x]

- Conflict pairs:
  
  - Pairs of conflicting operations in different TAs on the same data object
  - Example: r1[x], r2[y], w1[x], w1[y], w2[y], c1, c2

Concurrent transactions: Serializability theory

- Conflicting operations
  
  op, [x] and op'[y] are in conflict, if
  
  - x = y
  - i ≠ j
  - op = w or op' = w

- Conflict relation
  
  Ordered conflict relation C of a schedule S:
  
  C(S) = \{(op, op') | op and op' conflicting operations, op <, op' in S\}
  
  - Example: C(S) = \{(r1[y], w1[y]), (w1[y], w2[y])\}

Concurrent transactions: Serializability theory

- Conflict-serializable schedules
  
  - Same conflict pairs as some serial schedule
  - Same order of conflict pairs as in some serial schedule

  A Schedule S of a transaction set T is conflict-serializable if it has the same conflict relation as some serial execution SER of T: C(S) = C(SER)

  - Example:
    
    S: r1[x], r2[x], r1[y], r2[z], w2[y], w2[x], c2, c1
    
    C(S): \{(r1[x], w2[x]), (r1[y], w2[y])\}
    
    SER: r1[x], r1[y], c1, r2[x], r2[z], w2[y], w2[x], c2, c1

12.11

Concurrent transactions: Serializability theory

12.12
**Concurrent transactions: Serializability theory**

- **Example**
  
  \[
  S: \quad r_1[x], r_2[x], r_1[y], r_2[z], w_2[y], w_2[x], w_1[y], r_1[z], c_2, w_1[x], c_1
  \]

  \[
  C(S) = \{ (r_1[x], w_2[x]), (r_2[x], w_1[x]), (r_1[y], w_2[y]), (w_2[y], w_1[y]), (w_2[x], w_1[x]) \}
  \]

  Requires T1 before T2 in a serial schedule

  Requires T2 before T1 in a serial schedule

  **Not conflict-serializable**

**Serializability Theorem:**

A schedule \( S \) is conflict serializable, if and only if its conflict graph does not contain a cycle.

- **Example:**

  \[
  S: \quad r_1[y], r_3[u], r_2[u], w_1[y], w_2[x], w_1[x], w_2[z], w_3[x], c_1, c_2, c_3
  \]

  \[
  C(S): \quad \{(w_2[x], w_1[x]), (w_2[x], w_3[x]), (w_1[x], w_3[x])\}
  \]

  **SER:** \( r_2[u], w_2[x], w_2[z], c_2, r_1[y], w_1[y], w_1[x], c_1, r_3[u], w_3[x], c_3 \)

  Serializable

**Conflict Graph (Precedence or dependency graph)**

- Directed graph
- Shows conflicts between transactions
- Nodes: transactions of \( S \)
- Arc \( T_A_i \rightarrow T_A_j \): conflicting pair \((op_i[x], op_j[x])\)

- **Example:**

  \[
  C(S) = \{ (r_1[x], w_2[x]), (r_2[x], w_1[x]), (r_1[y], w_2[y]), (w_2[y], w_1[y]), (w_2[x], w_1[x]) \}
  \]

  Intuitive correctness idea:
  
  - Determine "conflict-equivalent" serial schedule from graph
  - Exchange operations in \( S \) without switching conflict pairs

  **Proof:**
  
  - "⇒:" The nodes of a connected directed graph without cycles can be sorted topologically: \( a < b \) iff there is a path from \( a \) to \( b \) in the graph. Results in a serial schedule if non-conflicting TAs are added arbitrarily.
  
  - "⇒:" Suppose there is a cycle \( T_A_i \rightarrow T_A_j \) in the graph. Then there are conflicting pairs \((p,q),(q',p')\). No serial schedule will contain both \((p,q)\) and \((q',p')\). Induction over length of cycle proves the "only if").
Concurrent transactions: Serializability theory

- Conflict serializability sometimes too restrictive

Example:

$S: w1[y], w2[x], r2[y], w2[y], w1[x], w3[x], c1, c2, c3$

$C(S1) = \{(w1[y], r2[y]), (w1[y], w2[y]), (w2[x], w1[x]), (w2[x], w3[x]), (w1[x], w3[x])\}$

But effect of $S$ is the same as from the serial Schedule: $T1, T2, T3$

Concurrent transactions: Scheduling

- Concurrency control in DBS methods which schedule the operations of different TAs in a way which guarantees serializability

- Principle concurrency control methods
  - Locking (most important)
  - Optimistic concurrency control
  - Time stamps

- Transactions not known in advance

Concurrent transactions: Scheduling

- Serializability
  - Theoretical model which defines correctness of executions
  - Often not implemented
  - No real-time scheduling control

- Goal: effective real-time scheduling of operations with guaranteed serializability of resulting execution sequence

Concurrency control: Locking

- Locking:
  - Access on data only if TA "has lock" on data
  - Transactions request and release locks
  - Scheduler allows or defers operations based on lock table

- Implicit locking by scheduler

- Explicit locking in application programs
  - error prone
  - additionally allowed for in most DBS

- Not considered here: transaction independent locking, e.g. page lock while writing
Concurrency control: Locking

- Locking is pessimistic:
  - Assumption: during operation operation \(\text{op}[x]\) of TA1 a (potentially) conflicting operation \(\text{op}'[x]\) of TA2 will access object \(x\)
  - Avoid by locking \(x\) before accessing \(x\)

- Lock protocols
  - Simplistic
  - Preclaiming
  - Two Phase Locking (2PL)
  - Strict 2PL

Concurrency control: Lock Protocols

- Simplistic
  - Lock each object before writing
  - Unlock afterwards
  - \(\Rightarrow\) schedule will not be serializable

- Example:
  \[
  \begin{align*}
  \text{T1: } & r1[x], w1[x], c1 \\
  \text{T2: } & r2[x], w2[x], c2 \\
  \text{lock1(x), r1[x], unlock1(x), lock2(x), r2[x], unlock2(x),} \\
  \text{lock1(x), w1[x], unlock1(x), lock2(x), w2[x], unlock2(x)}
  \end{align*}
  \]

Concurrency control: Lock Protocols

- Preclaiming
  - Acquire all locks TA needs before performing an operation
  - Release locks if TAs does not get all - try again
  - Execute transaction
  - Release locks

- Problems:
  - Racing situation
  - Objects to access may not be known in advance
  - Not used in DBS

Concurrency control: Lock Protocols – 2PL

- Two phase locking (2PL)
  - Lock each object accessed by TA before usage
  - Respect existing locks of other TAs
  - No TA requests lock which it already holds: if T holds a read lock for \(x\), it will not request a write lock for \(x\)
  - No lock after unlock: No lock is requested by a TA, after a lock has been released by same TA
  - Locks are released at least at commit time
Concurrency control: Lock Protocols – 2PL

- Locked objects accessible in lock acquisition phase
  ⇒ Rule 3 is essential (see "select.... for update")
  Example:
  \[
  \begin{align*}
  &\text{rlock}(x), \ \text{r1}[x], \ x=x*10, \\
  &\text{rlock}(x), \ \text{r2}[x], \ x=x+1, \\
  &\text{wlock}(x), \ \text{w1}[x], \ \text{unlock}(x), \\
  &\text{wlock}(x), \ \text{w2}[x], \ \text{unlock}(2x)
  \end{align*}
  \]

- Lock after unlock may result in a lost update
  ⇒ Rule 4 is essential
  Example:
  \[
  \begin{align*}
  &\text{lock}(x), \ \text{r1}[x], \ x=x*10, \ \text{unlock}(x), \\
  &\text{lock}(x), \ \text{r2}[x], \ x=x+1, \ \text{w2}[x], \ \text{unlock}(2x), \\
  &\text{lock}(x), \ \text{w1}[x], \ \text{unlock}(1x)
  \end{align*}
  \]

2-Phase locking theorem

If all transactions follow the 2-phase locking protocol, the resulting schedule is serializable.

Proof sketch (continued)

- Suppose a resulting schedule is not serializable after 2PL
- ⇒ conflict graph contains a cycle
- ⇒ ∃TA1 and TA2 with conflict pairs (p1,q2) and (q2', p1')
  p1,q2 access the same object x, and q2', p1' an object y
  (assuming a cycle of length 1, induction for the general case)
- Let e.g. (p1,q2) = (r1[x], w2[x]),
  (q2', p1') = (w2[y], w1[y])
- Analyze all possible sequences of execution

Serializability does not imply 2PL

- ∃ serializable schedule not resulting from a 2PL scheduler

Cascading abort:

- Possible access of TA2 to object x which was unlocked by TA1 in release phase
- if TA1 aborts, TA2 used wrong state of x
- TA2 to be aborted by the system

May happen recursively: cascading abort
Concurrency control: Lock Protocols – S2PL

- **Strict 2-phase locking**
  - Prevents cascading abort
  - Release all locks at commit point

\[
\begin{array}{c}
\text{# locks} \\
\downarrow \\
\text{Begin TA} \\
\uparrow \\
\text{End TA} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Lock acquisition phase} \\
\downarrow \\
\text{Release all locks at commit} \\
\uparrow \\
\text{time} \\
\end{array}
\]

Example:

\[
S: r1[x], r2[y], w1[x], w1[y], w2[y], c1, c2
\]

- Lock(x)
- Lock(y)
- Lock(y)?
- Unlock(x)
- Unlock(y)

Concurrency control: Lock Protocol

- **Deadlock situation**
  - Transactions each waiting for locks the other one holds

Example:

\[
\begin{align*}
\text{TA1} & \\
\text{Lock(x)} & w[x] \\
\text{Lock(y)} & r[y] \\
\text{Lock(y)?} & & \text{wait} \\
\text{Lock(x)?} & & \text{wait} \\
\text{Unlock(x)} & c2 & \text{Unlock(y)}
\end{align*}
\]

- Inherent to lock-based scheduling

Concurrency control: Lock Modes

- **Shared and exclusive locks**
  - For each object \( x \) either one exclusive lock or many shared locks
  - Read lock: shared lock
  - Write lock: exclusive lock

\[
\begin{array}{c|ccc}
\text{Requester} & \text{Ex} & \text{Sh} & \text{H} \\
\hline
S & + & - & - \\
X & - & - & -
\end{array}
\]

- Read locks allow for read locks
- Exclusive lock prevents additional locks
Concurrency control: Lock Modes

Example:

\[ S: w1[x], r2[y], r1[y], r2[x], c1, c2 \]

- \( XLock(x) \)
- \( r[x] \)
- \( c1 \)
- \( Unlock(x) \)

\[ SLock(y) \]
\[ r[y] \]
\[ SLock(x)? \]
\[ wait \]
\[ Unlock(y) \]
\[ Unlock(x) \]

TA1

TA2

Shared and exclusive locks may prevent deadlocks

12.33

Concurrency control: Lock Modes

Update locks

- Update lock: shared lock for reading, allows for upgrading
- Read lock: shared lock
- Write lock: exclusive lock

Upgradeable locks

- Often first read of object then write

Example:

\[ S: r1[x], r2[x], w1[x], w2[x], c1, c2 \]

- \( XLock(x) \)
- \( r[x] \)
- \( c1 \)
- \( Unlock(x) \)

- \( SLock(x) \)
- \( r[x] \)
- \( Unlock(y) \)
- \( Unlock(x) \)

TA1

TA2

Upgrade not allowed: abort transaction

If upgrade allowed: deadlock

12.34

Update locks may prevent abort or deadlock \( \rightarrow \) Operations not always known in advance

12.35

TA1

TA2

Unlock(x)

Unlock(x)

Unlock(x)

Unlock(x)

12.36
Concurrency control: Lock Modes

- **Increment locks**
  - Useful for TAs incrementing or decrementing stored values
  - Examples: Money transfer, ticket selling...
  - Non-commutative to read or write

- **Equivalent schedules:**
  - Commutative increment actions
  - Example schedules:
    - \( S: r_1[x], r_2[x], inc_2[y], inc_1[y], c_1, c_2 \)
    - \( S: r_1[x], inc_1[y], r_2[x], inc_2[y], c_1, c_2 \)
    - \( S: r_1[x], r_2[x], inc_1[y], inc_2[y], c_1, c_2 \)

Concurrency control: Hierarchical locking

- **Object hierarchy:**
  - Relation \( R_1 \)
  - Block \( B_1 \) to \( B_2 \)
  - Record

- **Intention locks**
  - Intention to request lock
  - For each lock mode: e.g., IS and IX

- **Locking:**
  - Start at root of hierarchy
  - Before right element place intention locks
  - At the right element in hierarchy request lock

Concurrency control: Locking granularity

- **Single lock granularity insufficient**
  - Overhead: many record locks vs. one table lock
  - Phantoms: not prevented by record locks

- **Granularities:** Tuple, page, block, relation, ...

- **At least two lock granularities in most DBS**

- **Problem:**
  - TA requests lock on table \( R \)
  - Some rows of \( R \) locked by different TAs
  - Lock conflict: TA waiting for release of all record locks
  - Danger of starvation of TA

Concurrency control: Hierarchical locking

- **Lock situation:**
  - IS, IX signal locks deeper in hierarchy
  - Element locks lock all subordinate elements in hierarchy

- **Example:**
  - Goal: Lock on block \( B_1 \)
  - Warning on upper elements (Intention lock on relation \( R \))
  - Locks on block and records in \( B_1 \)
Concurrency control: Hierarchical locking

- **Advantage:**
  - one lookup to check if lock on higher level can be granted

- **Compatibility of locks:**
  - Exclusive lock not compatible
  - Intended exclusive only compatible to intention locks

- **Lock escalation**
  - Convert many locks on level \( i \) into one lock on level \( i-1 \)
  - Only for locks by single TA
  - Efficiency reasons (large locktables)

---

Concurrency control: Deadlocks

- **Lock conflict**
  - TA waits for lock held by other TA
  - Conflict solved when TA finished and lock released

- **Deadlocks inherent to 2PL**
  - Deadlock: at least two pairing conflicts
  - Releasing lock would break Rule 4
  - Keeping lock would lead to deadlock

- **Resolving deadlocks**
  - Cycle check in Waits-for-graph
  - Timeout

---

Example:

- **S:** \( w1[x], r2[y], r1[y], r2[x], c1, c2 \)

```
TA1
Lock(x)
wa
TA2
Lock(y)
r
Lock(y)
wait
Lock(x)
abort
continues
```

Waits-for-graph:

- Directed graph
- Shows TAs waiting for locks held by other TAs
- Nodes: transactions of S
- Arc \( TA_i \rightarrow TA_j \): TA \( i \) waits for lock held by TA \( j \)

- Cycles indicate deadlocks
- Deadlock prevention:
  - Refuse actions that cause cycles
  - Rollback of the requesting TA
  - Expensive method (maintain graph at all times)
Concurrency control: Deadlock resolving

- **Deadlock resolving with waits-for-graph**
  - Less expensive
  - One of the waiting transactions rolled back

- **TA selection for rollback**
  - Minimize costs: rollback of youngest TA, rollback TA with minimal #locks
  - Maximize free resources: rollback TA with maximal #locks
  - Avoid starvation: avoid rollback of same TA more than x times (employ counter)
  - Minimize cycles: rollback TA involved in maximal #cycles

Concurrency control: Deadlock resolving

- **Wait/Die method**
  - TA1 requests lock held by TA2
    - If TA1 older than TA2 \( ts(TA1) < ts(TA2) \)
      - TA1 waits
    - If TA1 younger than TA2 \( ts(TA1) > ts(TA2) \)
      - TA1 dies (TA1 aborted)

  Simple and efficient technique
  May roll back TAs without deadlock

  TA selection for rollback
  - Wait/Die method
  - Wound/Wait method

  Initial timestamp \( ts() \) for each TA
Concurrency control: Optimistic

- Optimistic concurrency control
  - Locks are expensive, Conflict solving hopefully is cheaper
  - Several types (backward, forward, …)

- Backward oriented concurrency control (BOCC)
  - Idea: TAs work on copies, check for conflicts at the end, commit (or abort)
  - BOT EOT
  - Read phase:
    - Copy data to private workspace
    - Execute operations on copy
  - Commit phase:
    - Write data into DB
  - Validation phase:
    - Resolve conflicts

BOCC Validation for TA:
- for all transactions (TA') which committed after BOT(T):
  - if $R(TA) \cap W(TA') = \emptyset$
    - then T.commit
  - else T.abort

Example:
- $x, y \in R(T2)$
- $z \in W(T1)$
- $x, y \in W(T3)$

Could result in unnecessary aborts
- Fast and atomic commit processing
  - Fast validation
  - Only one TA can validate at a time
- Useful in situation with few expected conflicts
Concurrency control: Timestamps

- Optimistic concurrency control based on timestamps
  - Each TA timestamp assigned
  - Idea: record timestamps of TA last access on element, compare order of timestamps with transaction schedule

- For each database element x
  - RT(x): timestamp of youngest TA that read x
  - WT(x): timestamp of youngest TA that wrote x
  - C(x): commit bit, true if last-writing TA has committed

- Goal: synchronize TAs such that resulting schedule is equivalent to serial schedule according to timestamp order

Concurrency control: Timestamps

- Scheduler actions
  - Grant request
  - Abort TA, restart TA with new timestamp
  - Delay TA

- Scheduling TA for reading:
  - If \( t_s(TA) < WT(x) \): abort TA
  - If \( t_s(TA) \geq WT(x) \): read and \( RT(x) := \max(t_s(TA), RT(x)) \)

- Scheduling TA for writing:
  - If \( t_s(TA) < RT(x) \): abort TA
  - If \( t_s(TA) < WT(x) \): abort TA
  - Else: write and \( WT(x) := t_s(TA) \) and \( RT(x) := t_s(TA) \)

Concurrency control: Multi-version timestamps

- Allows reads that would otherwise cause aborts
- Hold multiple versions of an object
  - The latest consistent version (in DB)
  - The updated, but not committed version (as copy)

- Example:

  ![Diagram](access-TA2-TA1-TA3)

  - TA2 would have to abort since \( t_s(TA2) < WT(x) = t_s(TA3) \)
  - Multi-version: give access to copy of older state
Concurrency control: Deadlock resolving

- Oracle:
  - Variant of Multi-versioning
  - Table and row level locks
  - Deadlock detection: waits-for-graph in central systems
  - Deadlock detection: Timeout in distributed environment (cycles hard to detect)
  - User influence: lock modes

- MySQL:
  - Multi-versioning and two-phase locking
  - Table and row level locks
  - Deadlock detection: waits-for-graph
  - Space-efficient lock table: lock escalation not needed
  - User influence: set priority of statement types

Transactions and Concurrency: Short summary

- ACID properties of transactions
- Schedules: Serial, (conflict) serializable
- Correctness of schedules: Serializability theorem
- Locking: Preclaiming, 2PL, strict 2PL, lock modes
- Optimistic concurrency control
- Timestamp-methods
- Serializability
- Cascading rollback
- Deadlocks
  - Detection (Waits-for-graph)
  - Avoidance (Waits-for-graph, Wait/Die, Wound/Wait)