SE/CS 351

- ACID properties

The ACID properties

- requirements that a transaction manager must meet for the transactions:
  - **Atomicity**: either all of the operations of a transaction are made durable or none of them are.
  - **Consistency**: after the transaction, the database is in a consistent state.
  - **Isolation**: operations in a transaction appear isolated from all other operations. Transactions have a virtual serial view on the system.
  - **Durability**: once the user has been notified of success, the transaction will persist, and not be undone.

ACID atomicity

- Either all of the operations of a transaction are made durable or none of them are.
- In a transfer transaction from account 123 to account 321:

  i.) withdraw $100 from account 123

  ii.) put $100 on account 321

  - It must not happen that the transaction stops after i.) and makes i.) durable.
  - Why not? This would violate an application level consistency constraint (balances must be kept).
- Commit must be requested by client.
- Database takes care of the rollback in case of abort.

ACID consistency

- Is referring to additional high-level features of the DB (and is not part of the basic transaction model we will use)
- Declarative integrity constraints, such as referential integrity, must hold between transactions.
- During the transaction, certain integrity constraints might be violated. On commit, integrity constraints must be fulfilled
  - by explicit operations in the transaction,
  - by automatic mechanisms (ON DELETE CASCADE),
  - by user-defined triggers.
- Any transaction still violating integrity constraints will be aborted.

ACID isolation

- Database operations in a transaction appear isolated from database operations of all other transactions
  - does not apply to non-database operations of client programs
- Operations for our example withdrawal:

  i.) check availability of funds:

  balance > $100 in account 123 at time 11:23:34

  ii.) withdraw $100 from account 123 at time 11:23:34+ε
- If someone withdraws funds between i. and ii. , then a problem can arise.
- Transactions must have a virtual serial view on the system.
- Isolation is expensive, and can be relaxed: *isolation levels*.

ACID durability

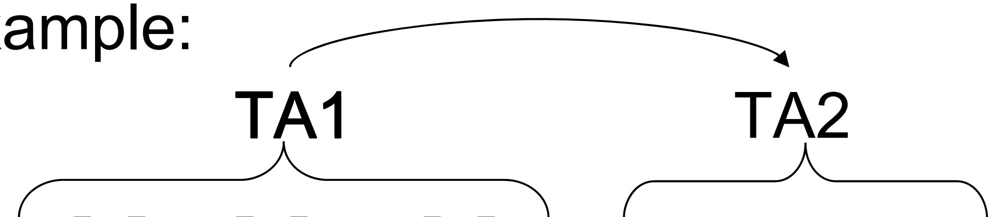
- Once the user has been notified of success of transaction t:
- The database system does not abort t any more.
- The effect of t is kept in a crash resistant way.
    - minimum requirement: transaction is written to persistent storage: resistant against
        - OS crash
        - system outage
    - preferred: protection against loss of persistent memory:
        - redundancy and geographic distribution
        - resistance against catastrophes

ACID isolation

- schedules that fulfill isolation
- data-disjoint and write-disjoint transactions
- the simple scheduler fulfils ACID isolation
- the simple scheduler is pessimistic
- scheduling as an online problem

Serial schedules

- A *serial* schedule is a schedule s, where one transaction starts only after all previous transactions have finished.
- Example:



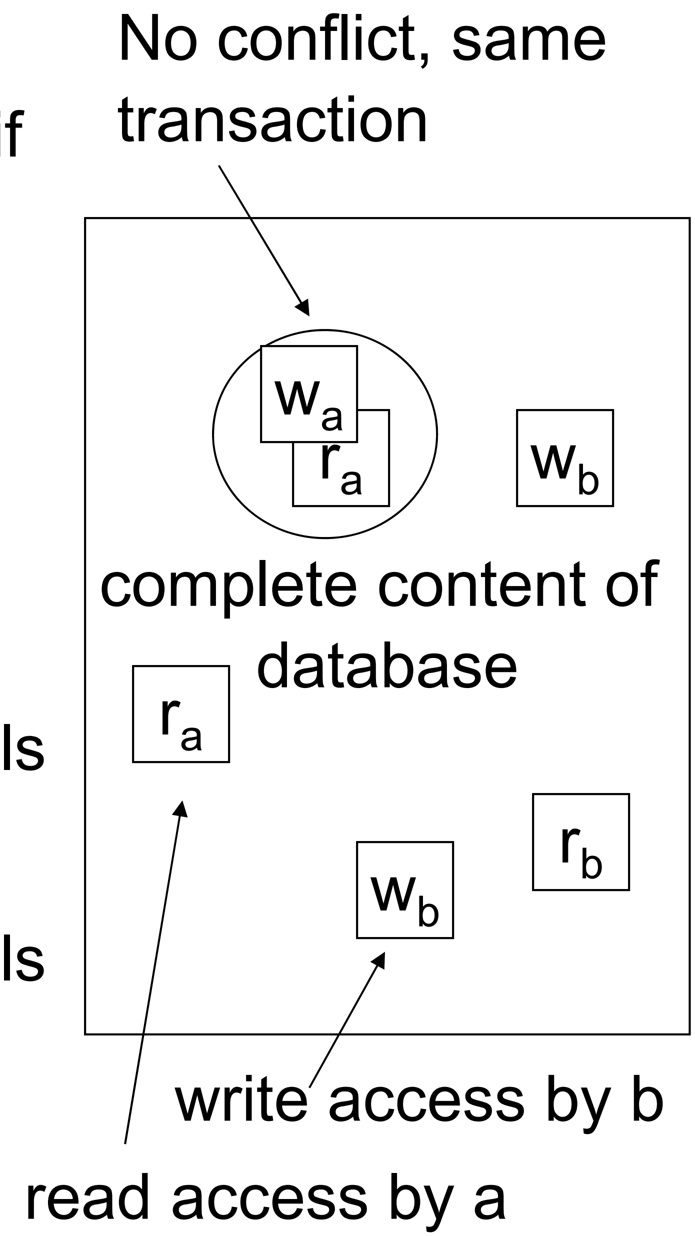
  - $\mathbf{r_1[x], r_1[y], w_1[x], c_1}$, $r_2[x], w_2[x], c_2$ is serial.
  - $\mathbf{r_1[x], r_1[y],}$ $r_2[x]$, $\mathbf{w_1[x], c_1,}$ $w_2[x], c_2$ is not serial.
  - $r_2[x], w_2[x], c_2$, $\mathbf{r_1[x], r_1[y], w_1[x], c_1}$ is serial.
- Serial schedules would be the result of mutually exclusive access by different transactions to the database: database uses a single lock!
- Serial schedules fulfil ACID-isolation!

Definition of serializable schedules

- For each numbering r of a set T of transactions, there is one serial schedule ser(r, T): execute the transactions in that order.
- A schedule s of a set of transactions T is *serializable* iff:
- There exists one numbering r of T so that s has the same effect as ser(r,T) on the database **and** clients, that means: for each data object x:
  - the value after s is the same as the value would be after ser(r,T) (remark: this is a contrafactual condition), and
  - all clients got the same read results for x.
- In summary: A schedule is serializable, if it is equivalent to a serial schedule in its effect on the database and the clients.
- ACID isolation means: only serializable schedules are allowed.

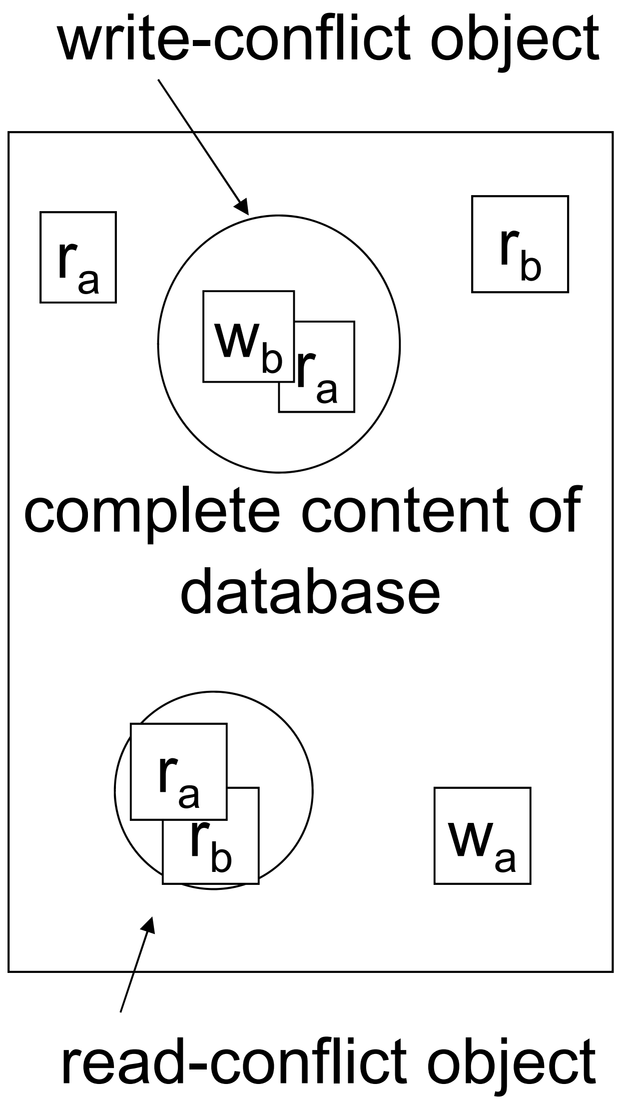
Data-disjoint transactions

- A set of transactions is *data-disjoint*, if no data object is accessed by more than one transaction from the set.
- For data-disjoint transactions, every schedule is serializable.
- Example:
  $r_1[x], r_1[y], w_1[x], c_1, r_2[z], w_2[z], c_2$ fulfils isolation.
- $r_1[x], r_1[y], r_2[z], w_1[x], c_1, w_2[z], c_2$ fulfils isolation, too.
- No scheduling necessary



Conflict objects

- data objects are called *conflict objects* for a set T of transactions if they are accessed by at least two transactions in T.
- At least one write access: *write-conflict* object
- No write access: *read-conflict object*.
- A set of transaction is data-disjoint, if it has no conflict objects.



Lock-based scheduling for isolation

- We have seen two different unproblematic cases: Transactions disjoint in time or disjoint on data.
- So the simple scheduler does the following:
- If two transactions are data disjoint, do nothing.
- If they have a conflict object: Order their execution in time:
  - Define an order in time in which they should be applied logically
  - The scheduler ensures that the schedule that is performed is equivalent to the serial schedule that would execute them in that order.
- Hence the produced schedules are *serializable.*

The simple scheduler fulfils full ACID isolation

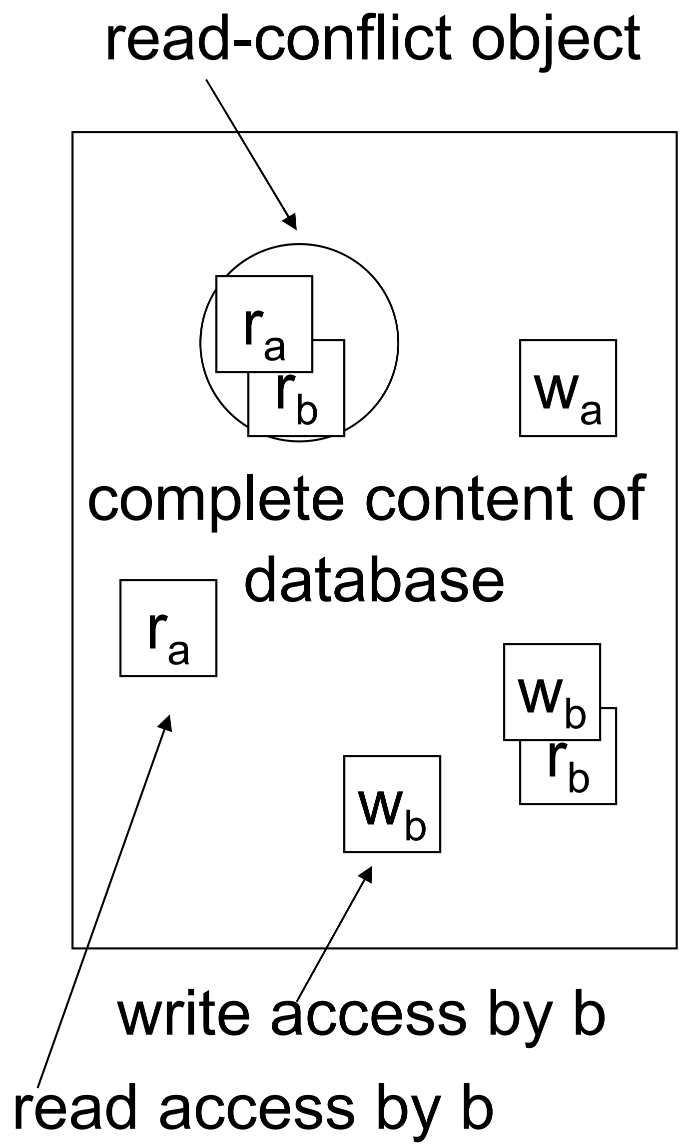
- All non-conflict objects can be ignored.
- For two transactions that have a conflict, the scheduler decides on an order in time. One transaction will do all operations on conflict objects before the other transaction.
- A transaction sees only results of transactions that come earlier in this order. It has only influence on transactions that come later in this order.
- the scheduler prevents a transaction TA1 from seeing effects that would go against this view:
    - Either TA1 is delayed
    - Or other transactions are delayed.

The simple scheduler fulfils full ACID isolation

- All non-conflict objects can be ignored.
- For two transactions that have a conflict, the scheduler decides on an order in time. One transaction will do all operations on conflict objects before the other transaction.
- A transaction sees only results of transactions that come earlier in this order. It has only influence on transactions that come later in this order.
- the scheduler prevents a transaction TA1 from seeing effects that would go against this view:
    - Either TA1 is delayed
    - Or other transactions are delayed.

Write-disjoint transactions

- A set of transactions is *write-disjoint*, if it has no write-conflict objects.
- Still, for write-disjoint transactions, ACID-isolation holds.
- Example:
  $r_1[x]$, $r_1[y]$, $r_2[y]$, $w_1[x]$, c_1, $w_2[z]$, c_2 is write-disjoint.
- Still no scheduling necessary. But what happens, if a transaction decides to write a read-conflict object?
  - Complex strategies necessary.



Scheduling as an online problem.

- The scheduler is an **online algorithm.**
- The scheduler has to make scheduling decisions based on incomplete local schedules.
- The scheduler cannot know what command comes later in each transaction.
- With prior knowledge of all local schedules, better solutions would be sometimes possible.
- For the simple scheduler: write-disjoint transactions have to wait for each other, although this is not necessary.

The simple scheduler is pessimistic

- The simple scheduler expects for every read, that it might be followed by a write:
- s: $r_1[x]$, $r_1[y]$, $w_1[x]$, c_1, $r_2[x]$, $w_2[x]$, c_2, $r_3[y]$, $w_3[z]$, c_3
- TA1: $r_1[x]$, $r_1[y]$, $w_1[x]$, c_1
- TA2: $r_2[x]$ ________________, $w_2[x]$, c_2
- TA3: $r_3[y]$ ________________________, $w_3[z]$, c_3

transactions TA2 and TA3 waiting, although no write conflict yet.

Oh, good that TA2 was waiting

Transaction TA3 waited, although it was not necessary!

- Treats all types of conflicts the same way.

Example schedulers and ACID isolation

- We will consider two schedulers that ensure ACID isolation.
- The **simple scheduler**:
  - only one type of lock, an exclusive lock.
  - Might delay write-disjoint transactions: therefore not really practical.
- The **common scheduler** (coming up next) is akin to schedulers used in reality;
  - it has a more complex locking mechanisms with non-exclusive and exclusive locks
  - will not delay write-disjoint transactions