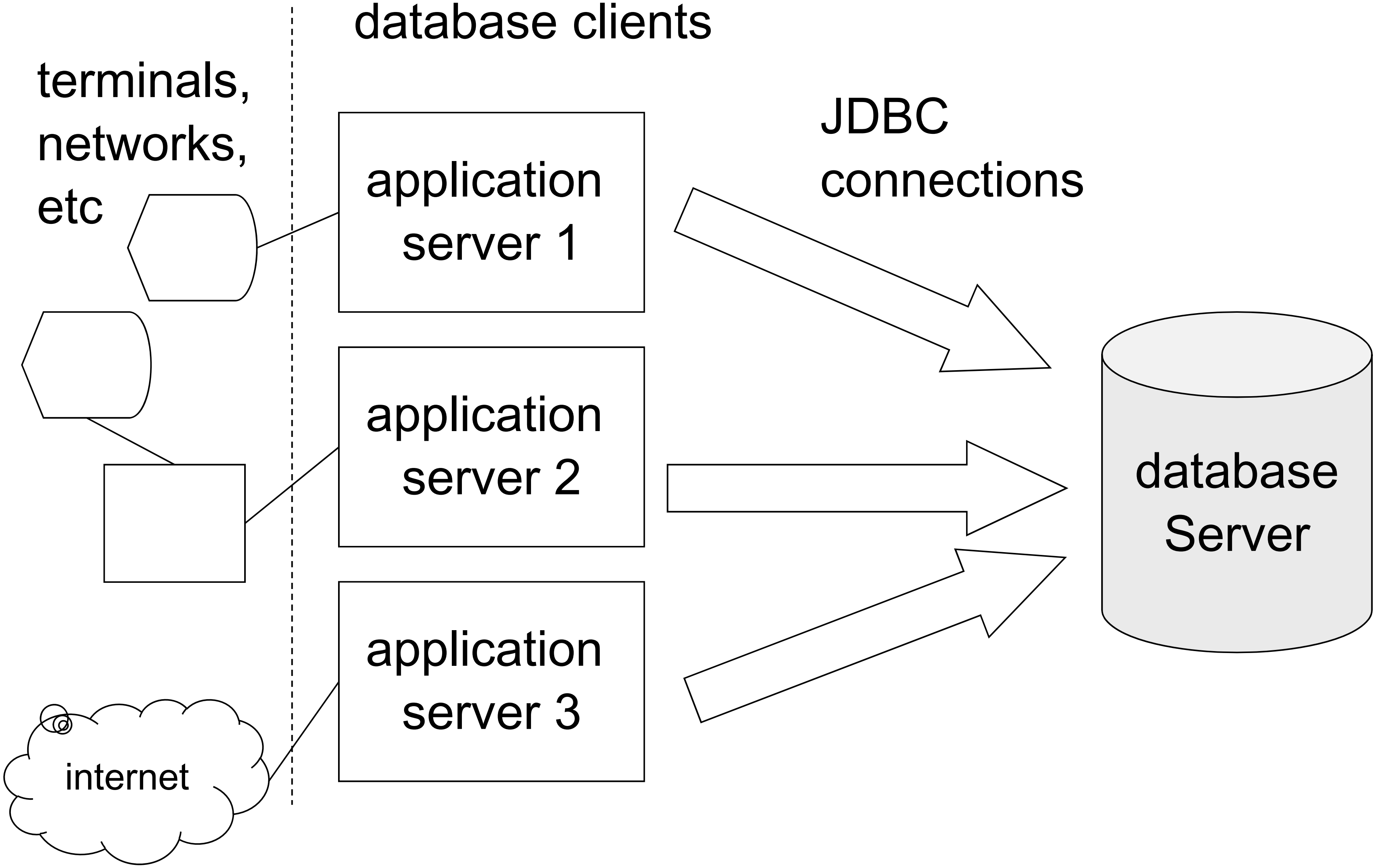
SE/CS 351

Introduction: transaction management

- motivation: concurrent access to the DB
- the notion of transactions
- the basic transaction model
- idea of lock-based schedulers
- the simple scheduler

Typical architecture for use of databases



database clients

terminals, networks, etc

application server 1

application server 2

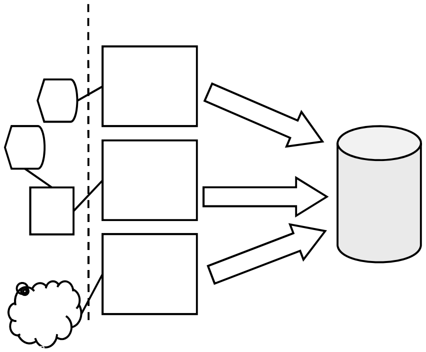
application server 3

JDBC connections

database Server

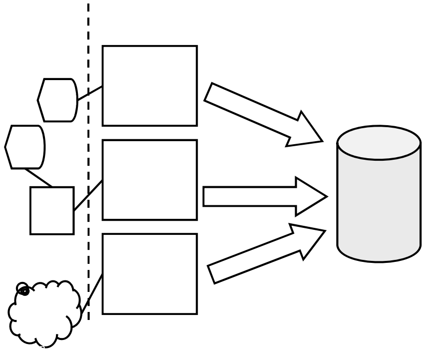
internet

Aspects of database access



- typical scenario:
  - the database is accessed by programs
  - they may be application servers
  - they act as client with regard to the database
- The database clients connect to the database through a remote interface: ODBC, JDBC.
- They issue SQL commands to the database.
  - The programs use basically the same interface as the interactive database client
  - unusual situation: programs use a dialogue interface

Problem: concurrent access to database



- Several connections to the database are open at the same time.
    - typical number of connections: in the order of 100. Why not much more? Answer:
    - Every connection has a large footprint in the database: consumes a lot of DB memory.
- Every connection issues operations.
- Concurrent operations may affect the same data.
    - may lead to inconsistencies.
- well known problem from semantics of programming languages, important for example in operation systems.

Transactions

- transaction: a sequence of operations that form a logical unit.
  - consider withdrawing money, after checking credit line.
  - Program: withdraw(account, amount)
- A single transaction is
  - a sequence of actually performed operations.

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

  - Issued by calling withdraw(account, amount) with actual parameters: withdraw(123, 100)

Scenario of a conflict in concurrent access

- Withdrawal after check is a very important pattern: flight booking, warehouse management, banking.
- Assume balance of account 123 is $120.
- Consider two withdrawals arriving concurrently:
- withdraw(123, 100): TA_1, withdraw(123, 110): TA_2
- The sequence of operations as they arrive at the DB:
  - TA_1: check availability of funds: balance of account 123 > $100 ?
  - TA_2: check availability of funds: balance of account 123 > $110 ?
  - TA_1: withdraw $100 from account 123
  - TA_2: withdraw $110 from account 123
- What is the problem here?

Distinguishing program and performed operations

- In our model, the transaction is only the sequence of commands that is received by the database, not the program creating the sequence. Motivation for the definition:
- The database does not see the program that creates the commands, cannot guess what command comes next, only the sequence of issued commands is visible.
- A frequent case will be that a single subprogram issues all the commands of a transaction.
- Sometimes the program that issues the commands is called a transaction, but this is informal speech.

DB access as read/write operations

- our main model: read/write operations on databases
  - example: making a transfer between two accounts.
    - i.) withdraw $100 from account 123
    - ii.) put $100 on account 321
- Operations for the transfer operation performed by the database client.
  - read a123 into local variable d,
  - write d-100 to a123 .
  - read a321 into local variable b
  - write b+100 to a321.

The basic transaction model

- allows precise reasoning on transaction scheduling.
- elementary operations in transaction s:
  - read on a data object: $r_s[x]$ - client gets the content
  - write on a data object: $w_s[x]$ - client provides new content
- transaction demarcation:.
  - Begin of Transaction (BOT_s) -
    - often implicit: first operation starts transaction
  - Commit: c_s - successful end of transaction
  - Abort: a_s - unsuccessful end of transaction

Example in the basic transaction model

- The operations in the transfer example:
  1. read a123 into local variable d,
  2. write d-100 to a123 .
  3. read a321 into local variable b
  4. write b+100 to a321.
  5. commit

i. withdraw \$100 from account 123

ii. put \$100 on account 321

- in the basic transaction model:

	1.	2.	3.	4.	5.
TA_1:	$r_1[x]$,	$w_1[x]$,	$r_1[y]$,	$w_1[y]$,	c_1

basic transaction model and SQL / JDBC

- SQL transactions can be translated into the basic transaction model for reasoning about concurrency.
- SELECT is one or many read operations
- UPDATE is
  - either one or many pure write operations
  - or read and write (one or many).
- INSERT creates only minor problems, the so called phantom phenomenon that will be discussed later. In the basic transaction model, inserts are not discussed.
- DELETE also creates only minor problems. In the basic transaction model, deletes are not discussed.

Transaction demarcation and SQL / JDBC

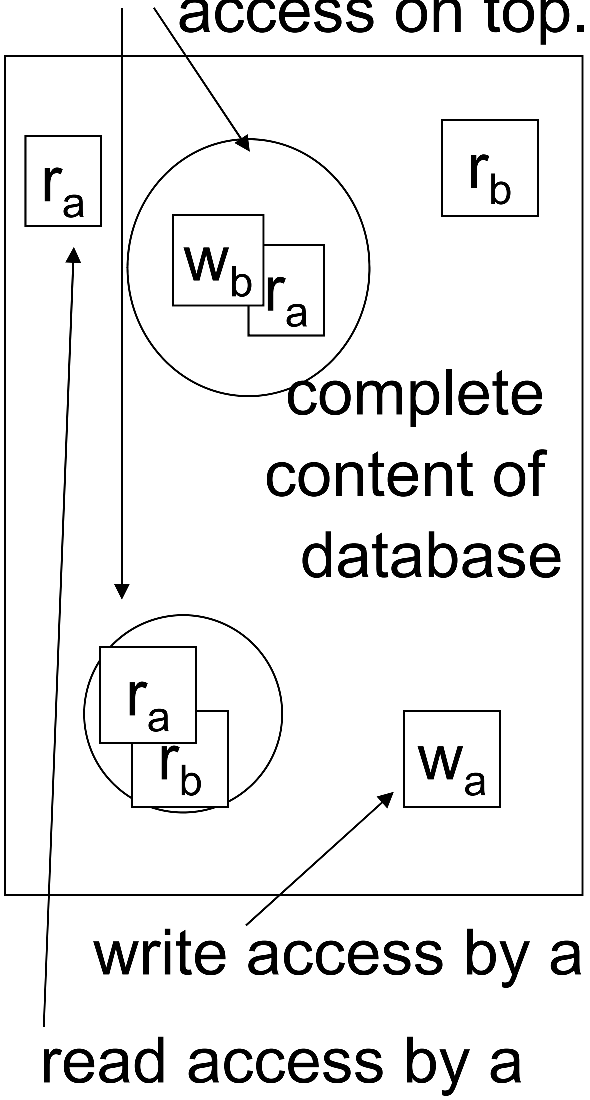
- `COMMIT` in SQL, `Connection.commit()` in JDBC.
- Warning: Autocommit must be turned off!
  - mysql> `set autocommit=0;`
  - JDBC: `myConnection.setAutoCommit(false);`
- abort by the user, a.k.a. rollback:
- `ABORT` in SQL, `Connection.rollback()` in JDBC.
  - All changed of the transaction are undone: see later under ACID Atomicity, hence the name rollback.
  - is always granted: interesting programming feature.
- Begin of Transaction (BOT_s) -
  - no separate command necessary in SQL, JDBC

Transaction demarcation and SQL / JDBC

- Database can abort transaction at any time before it has confirmed “COMMIT” to the client.
- Abort by the database and abort requested by the client have the same effect.
- client gets notification, e.g. as a response to a command:
- Even commit is only a request by the client
  - database can still respond with abort.
  - BUT: If database confirms commit, then
  - transaction is finished
  - no further abort of this transaction: transaction is durable, will survive system crashes.

Concurrent transactions accessing data

- Thought experiment:
- If concurrent transactions **read and write** random data in a huge database,
- conflicting access to the same data is not so likely, but happens.
- Basic idea of **lock-based** transaction management
  - tagging data that is accessed by a transaction t
  - remove tags after t has finished.
  - We can now detect potential conflicts


possible conflicts, later access on top.

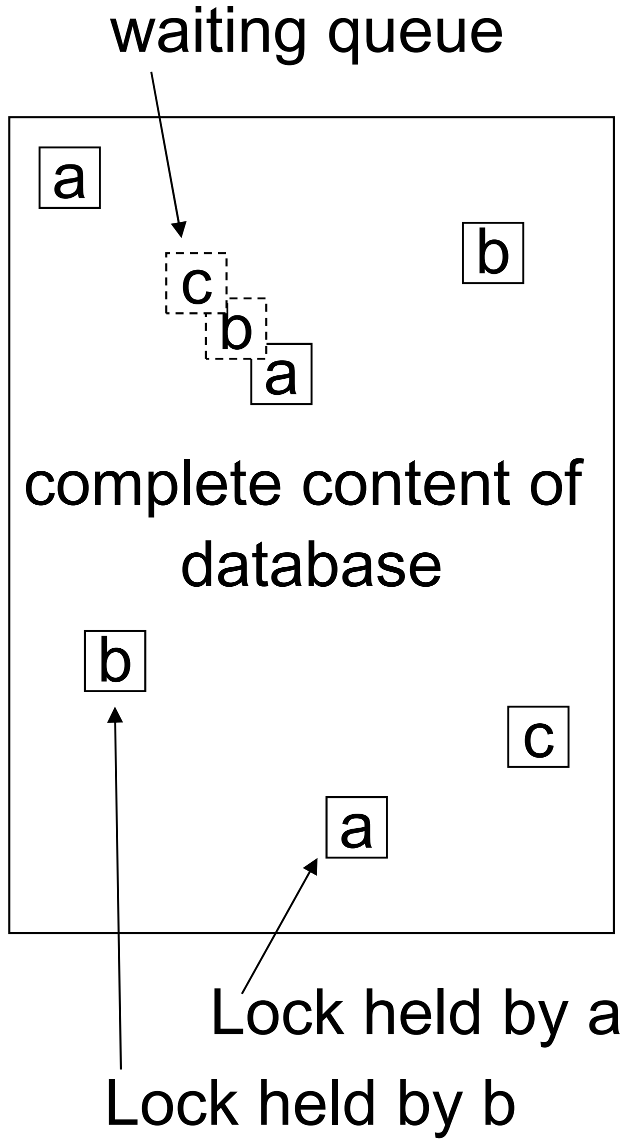
complete content of database

write access by a

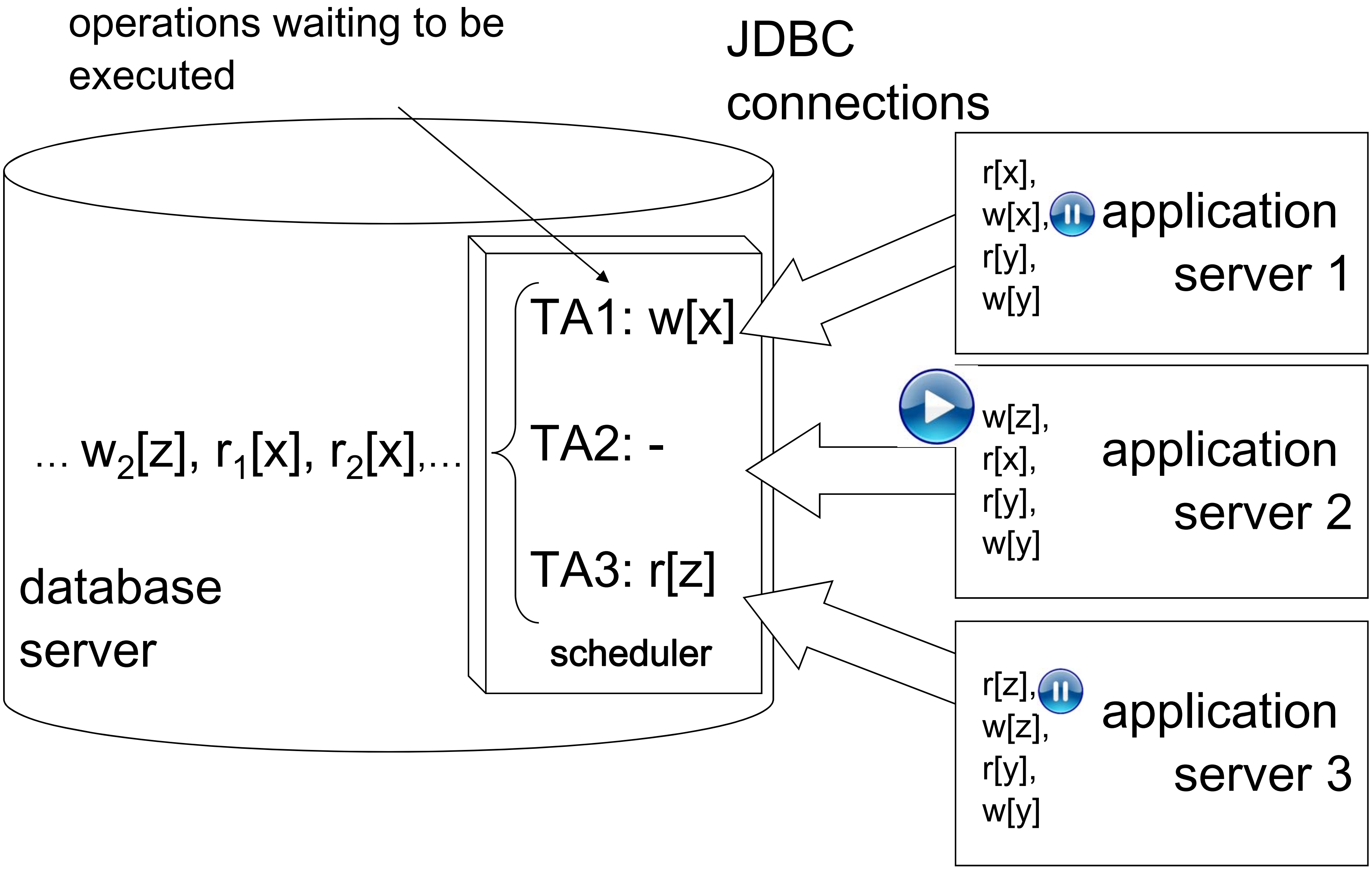
read access by a

Idea of lock-based transaction scheduling

- We tag the data that is accessed by a transaction. The tag is called a **lock**.
- in case of conflict: latecomer must wait in a queue for a particular data item.
- A lock is released immediately after its owner has committed **or** aborted.
- We can now detect **and solve** potential conflicts: scheduling.
- This solution makes use of transactions as units of work for releasing locks.



The principle of a scheduler



operations waiting to be executed

JDBC connections

... $w_2[z]$, $r_1[x]$, $r_2[x]$,...

database server

TA1: w[x]

TA2: -

TA3: r[z]

scheduler

r[x], w[x], r[y], w[y]

application server 1

w[z], r[x], r[y], w[y]

application server 2

r[z], w[z], r[y], w[y]

application server 3

A contract view of the transaction service

- Transactional databases have the following contract with their clients:
- The service offered by the database to the client: Database offers virtual view of exclusive access: client does not encounter concurrent actions of other transactions during any of its own transactions. Will be later precisely stated as ACID properties of transactions.
- The price the client has to pay:
  - Database client (application server) must provide transaction demarcation.
  - Database client accepts that a transaction t may be aborted by the database any time during the course of t.

Example (bad) *phenomenon*: dirty read

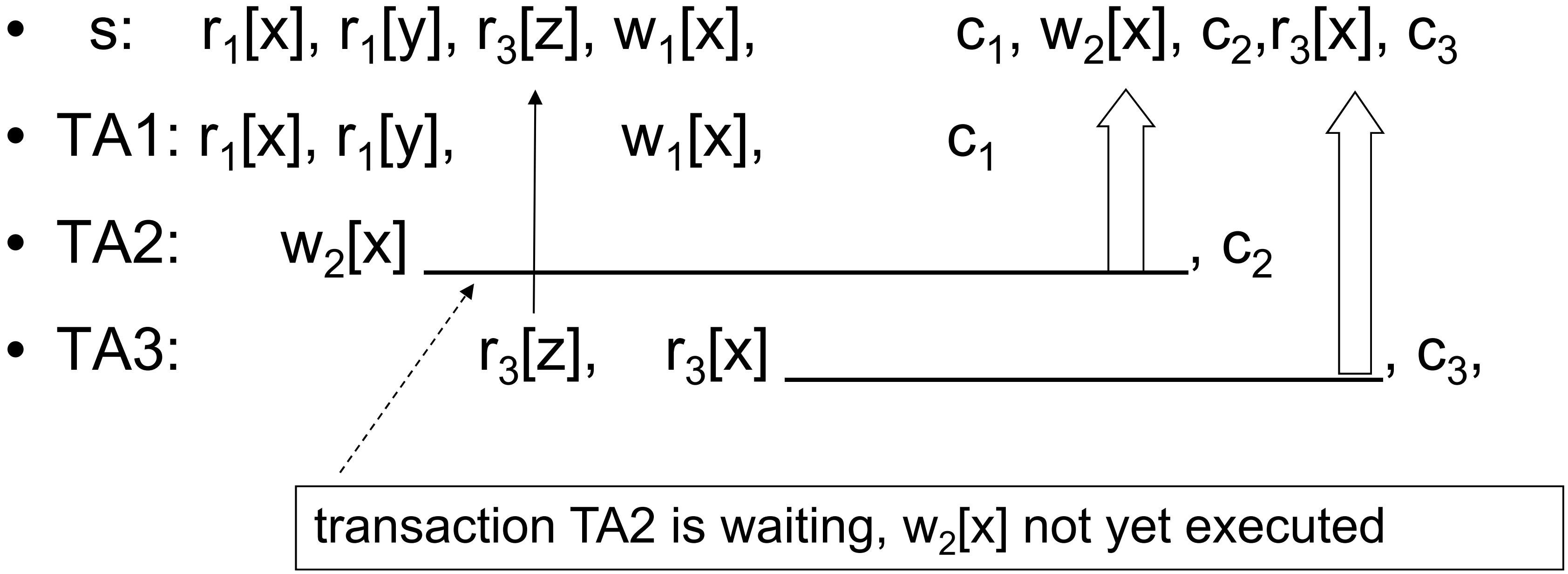
- Transaction TA2 performs a dirty read if it reads an uncommitted write result of TA1
  - scheduling example: .... w1[x], r2[x],

dirty read

- Dirty reads might be no problem for:
  - transactions that gather overview data
  - transactions that investigate options for later transactions
- But they are dangerous for other transactions
  - TA1 might be aborted and its changes might be undone.
  - In case TA1 is aborted, if TA2 has worked with the dirty value of x, then the result might be inconsistent.

Schedulers produce schedules

- Since schedulers cannot execute operations before they are issued by the clients, schedulers delay operations.



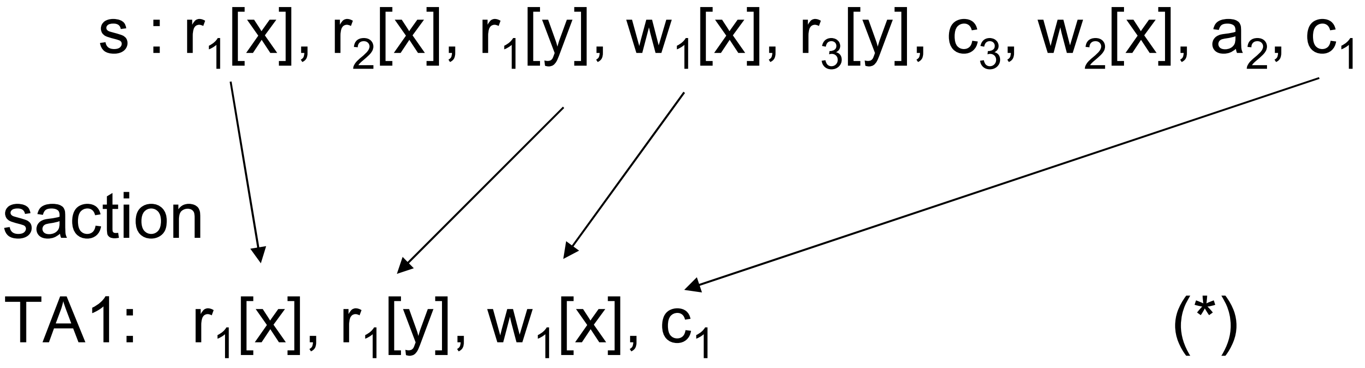
- s: $r_1[x]$, $r_1[y]$, $r_3[z]$, $w_1[x]$, c_1, $w_2[x]$, c_2,$r_3[x]$, c_3
- TA1: $r_1[x]$, $r_1[y]$, $w_1[x]$, c_1
- TA2: $w_2[x]$ ________, c_2
- TA3: $r_3[z]$, $r_3[x]$ ________, c_3,

transaction TA2 is waiting, $w_2[x]$ not yet executed

- The scheduling happens by waiting (best of all possible worlds, solve problem by doing nothing ☺)
- Local transactions are merged into a single *schedule*

Schedules in the basic transaction model

- Purpose: A schedule orders operations of a **set** of transactions in time.
- Example:
  - Schedule



$s : r_1[x], r_2[x], r_1[y], w_1[x], r_3[y], c_3, w_2[x], a_2, c_1$

  - Transaction

TA1: $r_1[x], r_1[y], w_1[x], c_1$ (*)

  - Lines do not cross: the schedule respects the local order in the transaction. (*) is called the local schedule of TA1.

Example schedulers

- We will discuss mainly two different schedulers, we call them the simple scheduler and the common scheduler.
- The **simple scheduler**:
  - ensures isolation: lock-based scheduling works!
  - Has a very simple locking protocol with only one type of lock, an exclusive lock.
  - Will delay transactions very often (is pessimistic) and is therefore not really practical.
- The **common scheduler** is akin to schedulers used in reality; it has a more complex locking mechanisms with non-exclusive and exclusive locks and will delay fewer transactions.

The simple scheduler [e.g. Ullmann 1980]

- The simple scheduler uses only one type of locks:
- Exclusive lock: At each point in time there can be only one lock per data object.
- has an *owner* transaction (that has *acquired* the lock)
- Only the owner can access the data object; a transaction has to acquire an exclusive lock in order to make **any** access (read or write) to a data object.
- Each data object has one queue for transactions waiting for the lock.
- A transaction waiting for a lock is blocked: It cannot execute any other operation. Accordingly, one transaction can be only in one queue.

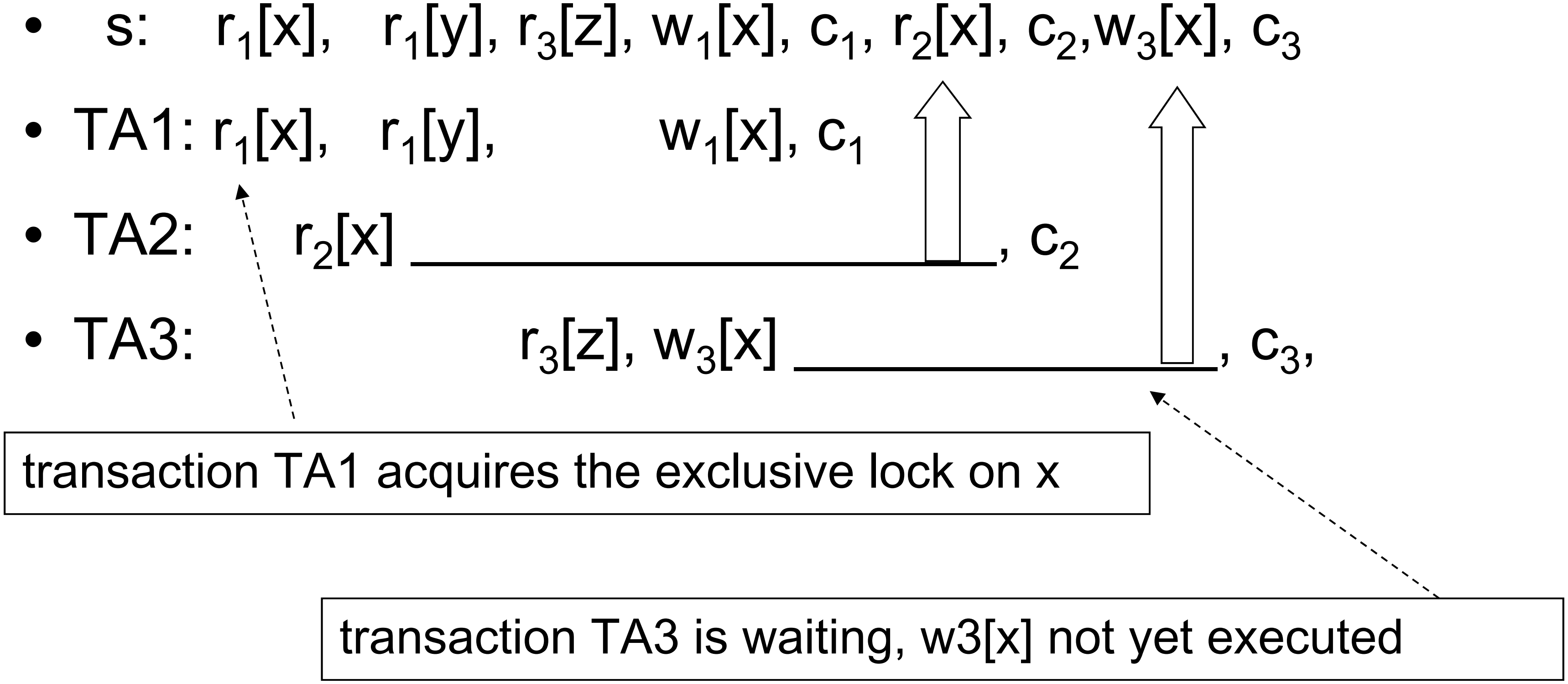
Transactions wait for exclusive locks

For the simple scheduler:

- If a transaction makes any access to an unlocked data object x, then the transaction acquires the lock.
    - Executes its operation on x
- If a transaction TA1 makes any access to a locked object x, the scheduler puts TA1 into the *waiting queue* for x.
- The waiting queue is managed first-in-first-out (FIFO).
- If a lock is released, and transactions are waiting, the scheduler takes the first of the waiting transactions out of the queue and grants the lock to this transaction.

Example schedule diagram

- Schedule delivered by the simple scheduler, and the times when the transactions issued the command:



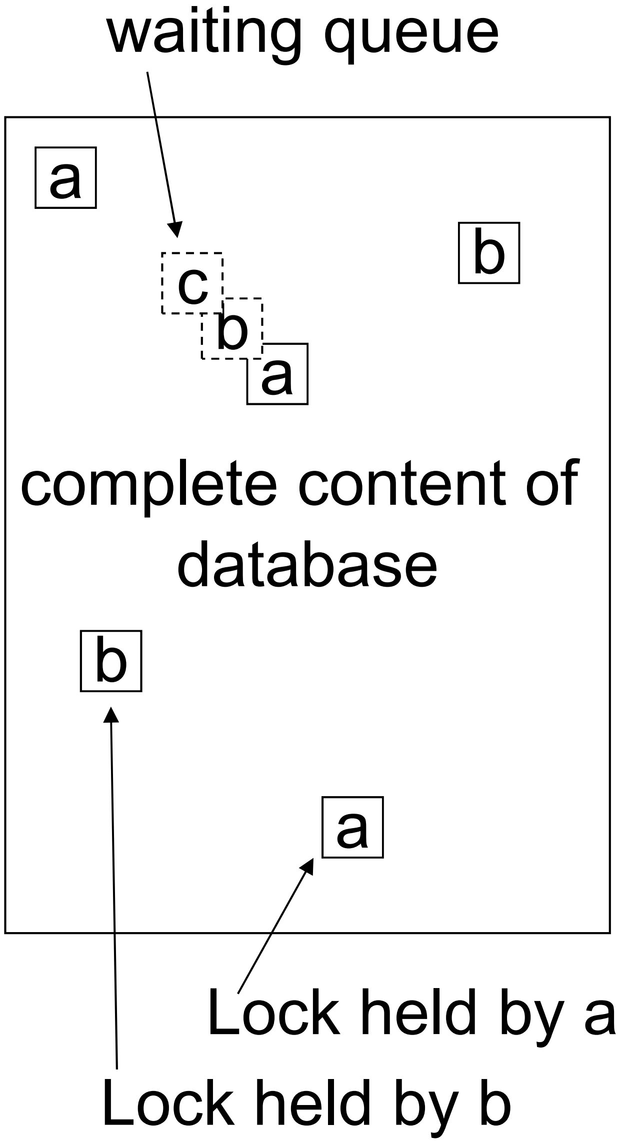
- s: $r_1[x]$, $r_1[y]$, $r_3[z]$, $w_1[x]$, c_1, $r_2[x]$, c_2,$w_3[x]$, c_3
- TA1: $r_1[x]$, $r_1[y]$, $w_1[x]$, c_1
- TA2: $r_2[x]$ __________________________, c_2
- TA3: $r_3[z]$, $w_3[x]$ ___________________, c_3,

transaction TA1 acquires the exclusive lock on x

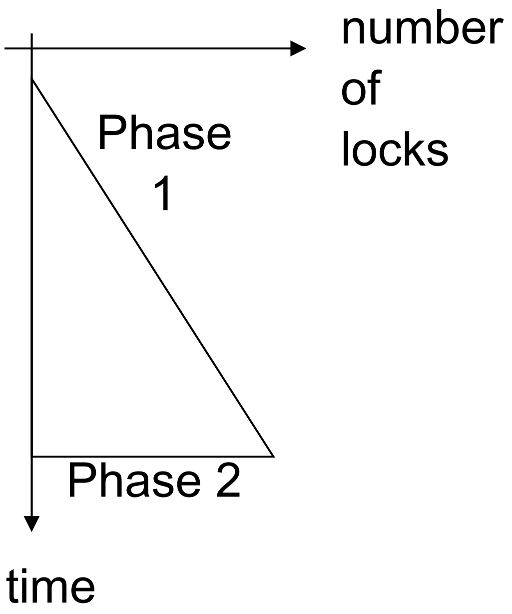
transaction TA3 is waiting, w3[x] not yet executed

The scheduler and the locks are one unit

- The locks and the queues for latecomers are **datastructures** of the scheduler.
- The scheduler is the **algorithm** for managing and using the locks.
- We lock single data items in the read-write model.
- Access to the data object has to be given by the scheduler according to the specification of the locking protocol.



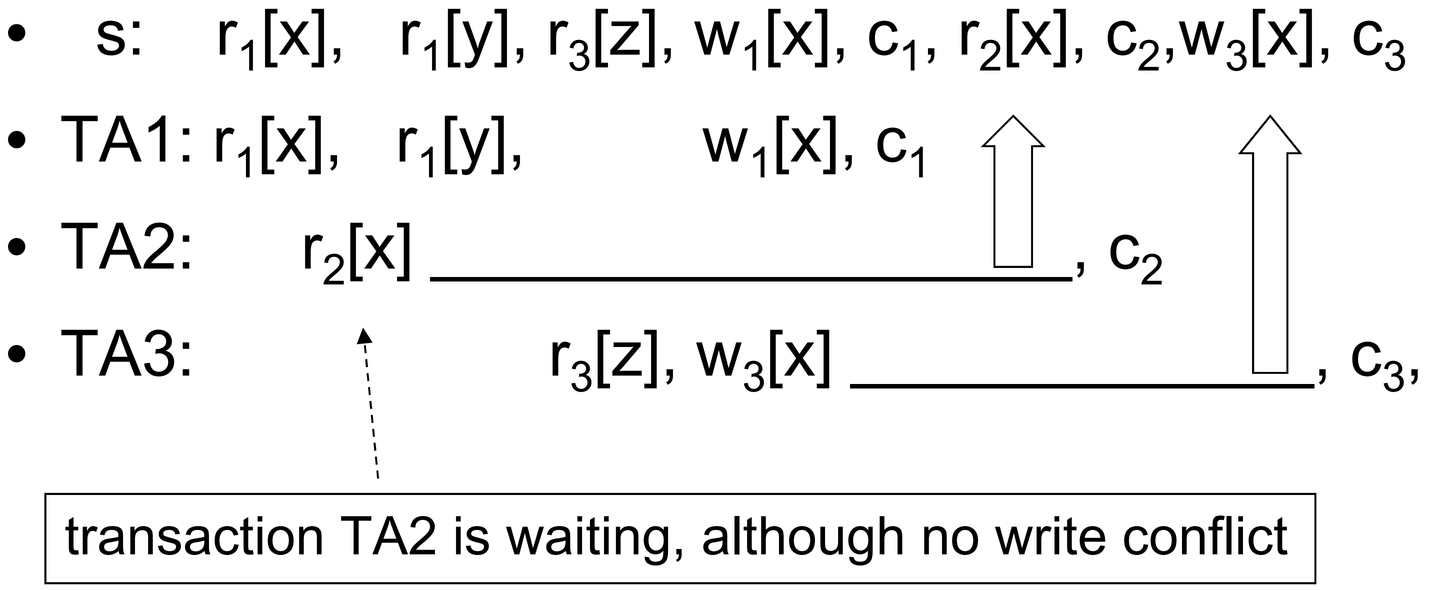
Strict two phase locking (S2PL)



- The simple scheduler uses strict two phase locking (S2PL)
- S2PL is used by many schedulers.
- S2PL has many favourable theoretical properties.
- transactions acquire locks during their lifetime (first phase).
- They release all locks immediately after commit/abort, but not earlier (strict second phase).
- No explicit lock commands necessary: comfortable, no break of abstraction.

The simple scheduler ignores read/write difference

- for the purpose of scheduling, read and write operations are not distinguished by the simple scheduler:



- s: $r_1[x]$, $r_1[y]$, $r_3[z]$, $w_1[x]$, c_1, $r_2[x]$, c_2,$w_3[x]$, c_3
- TA1: $r_1[x]$, $r_1[y]$, $w_1[x]$, c_1
- TA2: $r_2[x]$ ______________________, c_2
- TA3: $r_3[z]$, $w_3[x]$ ________________, c_3,

transaction TA2 is waiting, although no write conflict

- Treats all types of conflicts the same way.
- Nevertheless we denote the operations still as read and write operations.

Summary

- The notion of transactions: transactions are a sequence of actual commands, not a program.
- Transactions group operations into a logical unit.
- Concurrent access to data can lead to conflicts; conflicts can lead to inconsistencies.
- Locking with the S2PL protocol can detect all conflicts, without need for explicit, user-level locking commands.
- The simple scheduler can avoid inconsistencies by delaying transactions.