11 Modelling Transaction correctness

11.1 Why transactions?

11.2 Modeling transactions: histories and schedules

- Correctness criteria
  - Serial execution
    - History

11.3 Serializability

- Conflict graph
  - Serializability theorem

Kemper / Eickler chap 11.1-11.5, Elmasri/Navathe chap. 19

11.1 Why transactions?

Freie Universität Berlin

Remember...

Transaction: **a unit of work** which consists of a sequence of **steps** (operations on the Database)

Transactional program:

```
BEGIN
 op1; op2; .....;opn; //internal op or SELECT, UPDATE,
COMMIT                //INSERT, DELETE on database
```

– System must **guarantee "correct execution"**
– DBS has to be a "**dependable** (**fault tolerant, reliable**) system"

ACID

Freie Universität Berlin

Why is there a problem at all??

- Concurrent execution of multiple transactions (TA): Execution of ops belonging to different TAs may be interleaved. (Why?)
- TA may be aborted
- Systems may crash

Important: **ACID paradigm**.

A Database System should.....

Transaction semantics

Freie Universität Berlin

... guarantee certain execution l properties

"All or nothing" semantics

ATOMICITY

All effects are made permanent at COMMIT, not before .

TA has no effect after ROLLBACK

"Now and forever"

DURABILITY

DBS guarantees the effects after COMMIT has been processed successfully

"Solve concurrency conflicts"

Focus → ISOLATION

Conflict resolution of concurrent operations on DB

"Keep consistent DB consistent"

Preservation of integrity

CONSISTENCY

COMMIT processing

Freie Universität Berlin

- The COMMIT command is issued by the application

```
try {
 stmt.executeUpdate(sql1);
 stmt.executeUpdate(sql2);
 // Wenn keine Fehler aufgetreten sind,
 // Änderungen festschreiben
 con.commit();
} catch(SQLException e) { ...}
```

- The database server will either return control to the caller after successful processing the commit or throw an exception, if the TA cannot be committed for some reason
- If committed, the effects of TA can only be reversed by a **compensating transaction**.

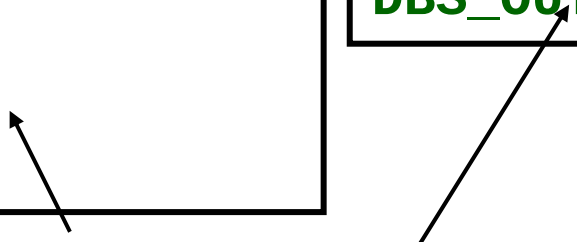
Example

Freie Universität Berlin

```
SELECT balance INTO :myVar
FROM account
WHERE acc# = :myAcc;
If myVar + dispo – amount >=0
 UPDATE account SET
   balance = myVar – amount
 WHERE acc# = :myAcc;
Call ATM_pay_out;
ENDIF;
COMMIT;
```

```
…
SELECT SUM(balance),owner

FROM account
GROUP BY owner;
COMMIT;
DBS_OUTPUT.PutLine(…);
```



concurrent execution in **independent DB sessions**

Conflict? Not a big deal in this case,
but may be SUM is incorrect.

Focus: Isolation

Freie Universität Berlin

Worst case: lost update

t

T1:

```
2 progVar ← read(x);

4 progVar++;
5 write (x ← progVar)
```

T2:

```
1 progVar ← read(x);

3 progVar++;

6 write (x ← progVar)
```

Concurrent Execution

Read of T1 and T2: x==7; Increment by T1: x== 8,
Increment by T2: x==8

Potential threats

Freie Universität Berlin

Lost update: Independent updaters change the same object.

One of the updates has no effect.

Every serious DBS has technical means to prevent a lost update.

Isolation levels

Freie Universität Berlin

How much isolation does a TA need?

Application dependent:

Is it acceptable that browsing in an online shop does not show the correct price of a few products?

Isolation level: Defines the degree of data corruption a program is willing to accept.

The more isolation the less parallelism

Isolation levels

Freie Universität Berlin

REPEATABLE READ

- all read / write conflicts prevented, reads repeatable

SERIALIZABLE

- repeatable read and **no phantoms**

```
TA2 : r(a), x= a.................  r(a);r(b),x:=x+b,...
TA1 :          Insert(z); Commit;
        -- TA2: SUM of some attribut of relation S,
        -- TA1: inserts a row into S
```

Isolation levels

Freie Universität Berlin

- **Read uncommitted** dangerous: may cause inconsistencies
- **Read committed** is the default in most systems (e.g. Oracle)
- **Serializable** important for high frequent short transactions with many potential conflicts.
- **AUTOCOMMIT-mode**: implicit COMMIT after each SQL-statement

TA abort

Freie Universität Berlin

ABORT

Caused by system, kills transaction

- system failure ⇨ user session is aborted ⇨ system recovery
- transaction rollback caused by internal state (e.g. deadlock)

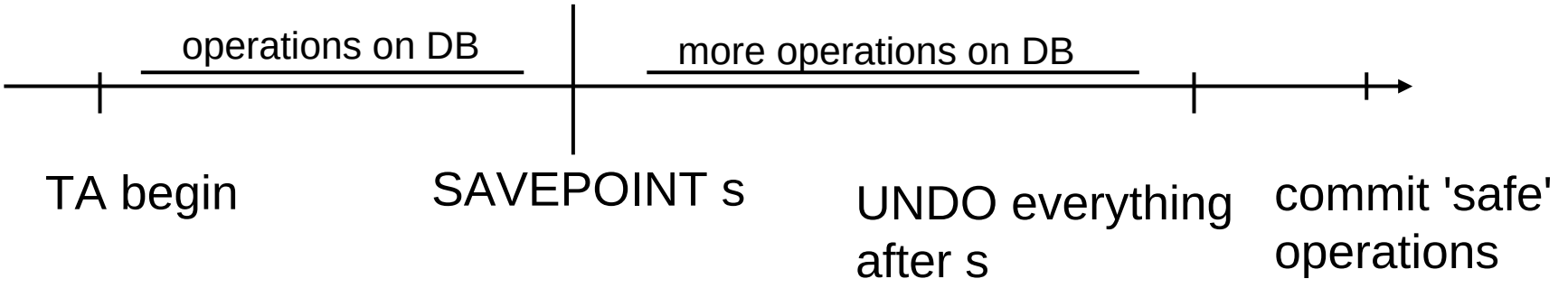
  Recovery of TA by system, of application process control flow by programmer.
  Important: handling of **DB exceptions**

SAVEPOINTS

Freie Universität Berlin

Rollback can be expensive in long TAs

Use SAVEPOINTs to limit work to be redone




```
UPDATE Movie SET title = 'bla'
 where year < where year < to_date (2000,'YYYY')
savepoint first;
UPDATE DVD SET .....
IF ... ROLLBACK  TO SAVEPOINT first
```

11.2 Correctness criteria for synchronization

The issue

- Transaction steps on a database are executed concurrently $op_i,...,op_j, ..,op_k$ (i.e. SQL calls)
- **No way to forecast** which step comes next (process scheduling).
- But **certain sequences are forbidden** because they violate the intended isolation level

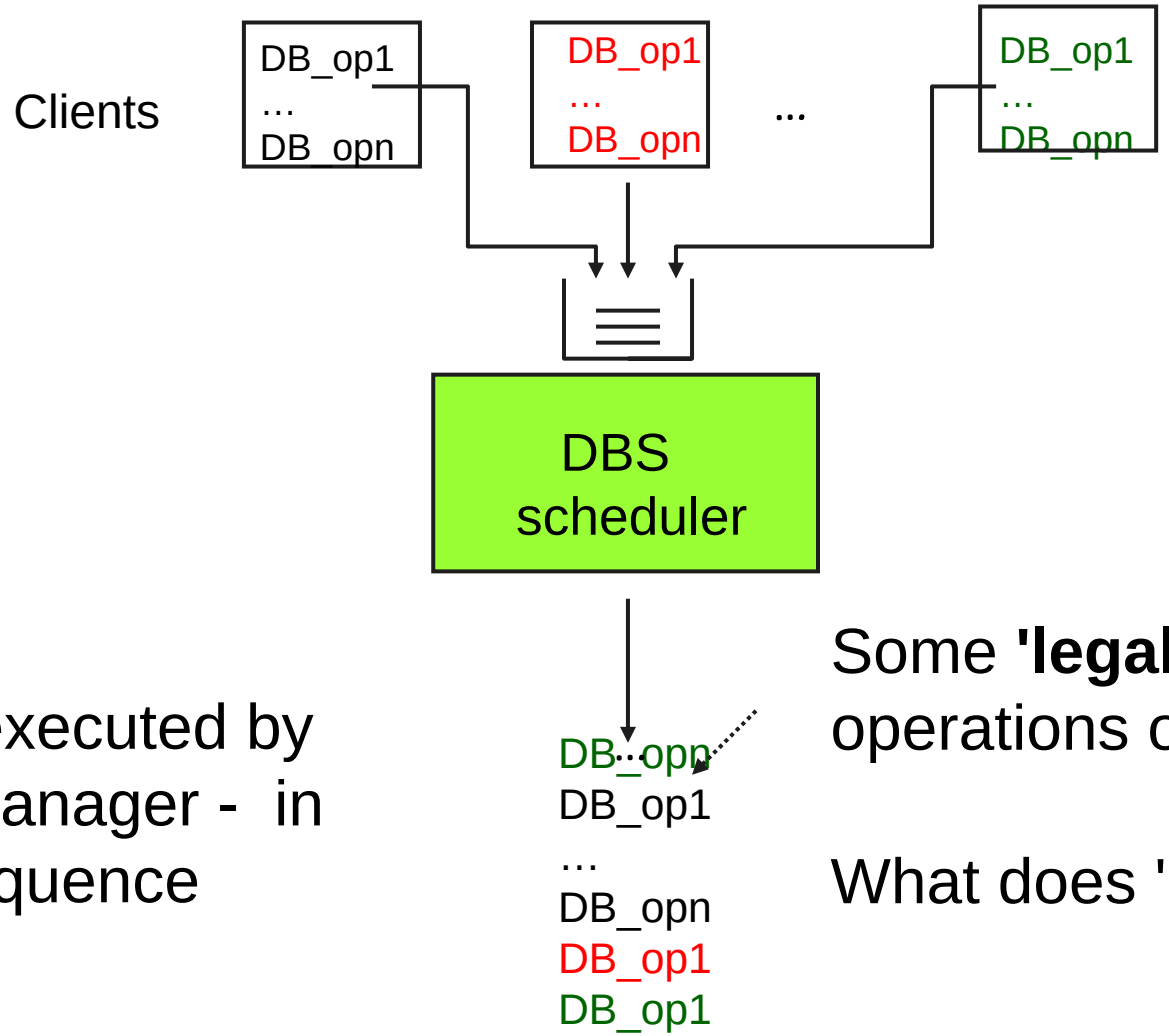
The goal

- A scheduling method which prevents operation sequences which potentially violate isolation

DBS Scheduler

Freie Universität Berlin

System point of view



To be executed by data manager - in this sequence

Some **'legal' sequence** of operations on DB

What does 'legal' mean?

Scheduling and Concurrency control

Basic questions:

(i) When is a sequence correct of steps of different transactions correct in the that it does not violate an isolation level (**correctness criteria**)

(ii) Which mechanisms do we have in order to enforce a (correct or legal) sequence?

Modeling TAs

Freie Universität Berlin

Read/Write model:

Transaction: sequence of following **atomic DB-operations**

READ i [x]	- TA i reads Object x: $r_i[x]$
WRITE i [y]	- TA i writes Object x: $w_i[x]$,
	⇨ DB state change
Commit i	- TA i terminates successfully: c_i

- Operations of different TAs interleaved
  ⇒ Sequence of r / w steps of different transactions.
- Assumption for now: **no abort**, since aborted TA do not leave any effect in DB

The Model

A transaction is modelled as a **sequence of reads and writes:**

$TA_j = r_j(x), r_j(y), w_j(y), r_j(z), w_j(x), w_j(s) , w_j(z) , c_j$

c_j : "*successful* commit ",

Consistency conventions (only for model):

TA do not read or write the same item twice

Scheduler produces a sequence of steps for many competing transactions...

Histories and schedules

Freie Universität Berlin

Def.: A **history** S of a (finite) set of transactions T is a sequence <**op**> of atomic actions **op** if the following conditions hold:

(1) An atomic action of a TA $\in$ T occurs exactly once in S

(2) No other action occurs in S

(3) If **op** < **op'** in some TA, then **op** < **op'** in S

"<" is the canonical ordering induced by the sequence of operations in TA and S rsp. (*)

e.g. r1[x], r2[y], r2[z], w2[y], r2[x], r1[y], w2[x], w1[y], r1[z], r2[s], c2, w1[x], c1

Def.: A **schedule** is a prefix of S.

(*) Partial order of steps would be ok, but formally more involved

Correctness of transaction execution

Informal correctness criterion

Execution of a set of TA is **intuitively correct**, if they are executed **one after the other** – in an arbitrary order.

Def.: Such an order is called a **serial execution**.

e.g. r1[x], r2[y], r2[z], w2[y], r2[x], r1[y], w2[x], w1[y], r1[z], r2[s], c2, w1[x], c1

r2[y], r2[z], w2[y], r2[x],, w2[x], r2[s], c2 r1[x], r1[y],w1[y], r1[z],, w1[x], c1
r1[x], r1[y]w1[y], r1[z],, w1[x], c1, r2[y], r2[z], w2[y], r2[x],, w2[x], r2[s], c2

i.e. T1, T2 or T2,T1

Correctness of histories

Freie Universität Berlin

Informal correctness criterion makes sense:

- no isolation conflicts

- order of TAs is determined by applications

Task:

Characterize the interleaved histories;
correct or not correct?

11.3 Serializability

Freie Universität Berlin

Def.: Given a history (schedule) H of transactions TA = {t1,...tn}.
If an execution of H produces the **same** database state **as *some* serial execution** of T, **H is called serializable**

- **more than one** possible serialization
- needed: a simple criterion based on steps of transactions
- Conflicting operations between transactions?

Informal serializability

Freie Universität Berlin

History H

r1[x=1], r2[y=5], w2[y=y+2], r1[z=3], w1[x=x+z], r2[z], c2, r1[y=7],w1[y=2*y], c1

x==1, y==5, z ==3

x==4, **y==14**, z as before

T1,T2

r1[x=1], r1[z=3], w1[x=x+z], r1[y=5],w1[y=2*y], c1, r2[y=14], w2[y=y+2], r2[z], c2,

x==1, y==5, z ==3

x==4, **y==12**, z as before

T2,T1

r2[y=5], w2[y=y+2], r2[z], c2, r1[x=1], r1[z=3], w1[x=x+z],, r1[y=7],w1[y=2*y], c1

x==1, y==5, z ==3

x==4, **y==14**, z as before

H serializable!

Serial execution

Freie Universität Berlin

History H':

r1[x=1],r2[x=1],w2[x++], w1[x++], c2, c1

x==1 x==2

T2, T1 $\Rightarrow$ x=3

T1, T2 $\Rightarrow$ x=3

$\Rightarrow$ H' **not** serializable!

Wanted: a less cumbersome criterion for serializability

Conflict operations

Freie Universität Berlin

Conflict serializability

Def.: Conflict operations:
$op_i(x)$ and $op_j(y)$ conflict
$\Leftrightarrow i \neq j$ and $x = y$ and $op_i = w$ or $op_j = w$

i.e. - no conflicts between reads,
- conflict if writes on the *same object s* by different transactions

Example

H: r1[x], r2[y], w1[x], w1[y], w2[y]

Conflict pairs

Serializability: intuitive idea

Freie Universität Berlin

Interchange operations in a schedule in order to achieve an equivalent serial schedule.

Non-conflicting operations of different TAs may be inter-changed

no interchange of **conflicting operations**

..... r2[x], r1[x], w2[x] ... → r1[x], r2[x], w2[x] ...

But: .... r1[x], r2[x], w2[x], w1[x], ... → ..... r1[x], w1[x], r2[x], w2[x],.....

not allowed, TA_2 reads stale data.

Conflicting operations

Freie Universität Berlin

.... r1[y], r2[x], w2[x], r1[x], ... →.... r2[x], w2[x], r1[y], r1[x], ...

But: .. r1[y], r2[x], w2[x], r1[x], ... →.... r1[y], r2[x], r1[x] w2[x], ...

not allowed.: ... w1[z=f(x,y)] : different effects

Semantic difference if x is read before or after it has been changed by a different transaction !

Conflict relation

Def.: Conflict relation of a schedule (history) S:

C(S) = {(op,op') | op and op' are conflicting and op < op' in S}

Example:

TA 1 = r1[x], r1[y], w1[y], r1[t], w1[x], c1

TA 2 = r2[y], r2[z], w2[z], r2[x], w2[x], r2[s], c2

r2[y], r1[x], r1[y], w1[y], r2[z,] w2[z], r2[x], r1[t], w1[x], c1,w2[x], r2[s], c2

C(S) = {(r2[y], w1[y]), (r1[x],w2[x]) (w1[x], w2[x]), (r2[x], w1[x])}

Equivalent schedules

Freie Universität Berlin

Conflict equivalence

Def.: Two histories H, H' are **conflict equivalent**

$\Leftrightarrow$ C(H) = C(H'),
i.e. they have the same conflict relation.

H = r2[y], r1[x], r1[y], w1[y], r2[z,] w2[z], r2[x], r1[t], w1[x], c1,w2[x], r2[s], c2

H' = r1[x], r1[y], r2[y], w1[y] r2[z,] w2[z] , r1[t] r2[x], w1[x], c1,w2[x], r2[s], c2

C(H) = {(r2[y], w1[y]), (r1[x],w2[x]) (w1[x], w2[x]), (r2[x], w1[x])} = CS(H')

Conflict Serializable

Freie Universität Berlin

Def.: A history S of a transaction set T is **conflict serializable** (CS) (*),

if it has the **conflict equivalent to some serial execution** SER of T: C(S) = C(SER)

Note: if S is CS then **not every serial execution** has the **same effects** on the data, but **there exists one** which leaves the database in the same state, i.e. has the same effects !

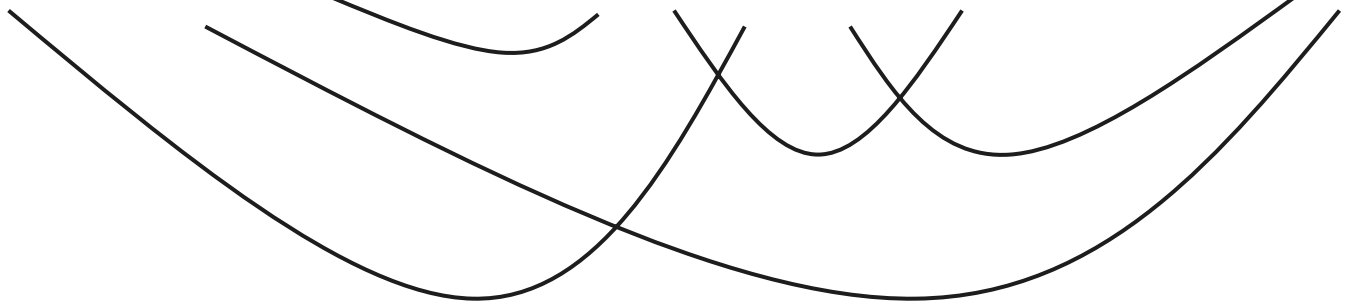
(*) Sometimes we say just: serializable, although there is the less restrictive notion of "view serializable"

Serializability

Freie Universität Berlin

Example

S: r1[x], r2[x], r1[y], r2[z], w2[y], w2[x], w1[y], r1[z], c2, w1[x], c_1



C(S) =

{ (r1[x], w2[x]), (r2[x], w1[x]), (r1[y], w2[y]) (w2[y], w1[y]), (w2[x], w1[x]) }

T1 must occur before T2 (r1[x], w2[x]) in a serial schedule

...and T2 must occur before T1: (r2[x], w1[x])

NOT conflict serializable

Serializability

Freie Universität Berlin

Conflict Graph (Precedence | dependency graph)

Def.: Conflict graph:

(a) Nodes: Transactions {T1, ….Tn}

(b) Directed Edges E :

(Tj,Tk) $\in$ E :$\Leftrightarrow$

exists a conflicting pair (opj [x] , op'k [x])

Represents the conflict relation of the transactions.

What does a cycle in this graph mean?

Conflict graph and serializability

Freie Universität Berlin

Conflict graph CG(S)

.

How does **the conflict graph of a serializable schedule** look like?

Serializability theorem:

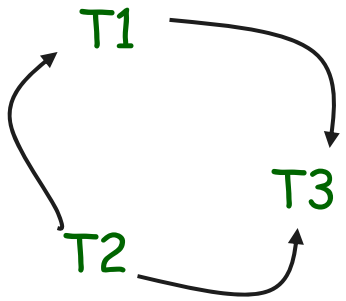
A **history S is conflict serializable**, if and only if its **conflict graph does not contain a cycle**

Serializability

Freie Universität Berlin

Example:

S: r1[y], r3[u], r2[y], w1[y], w2[x], w1[x], w2[z], w3[x]



Serializable!

Correctness of serializability theorem?

Serializability

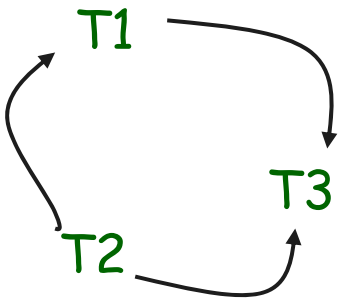
Freie Universität Berlin

Proof of Serializability Theorem:

" $\Leftarrow$" "

show: no cycle $\Rightarrow$ serializable"

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 TA_i, ...TA_k .



Not unambigous in general.

Serializability

Freie Universität Berlin

" $\Rightarrow$ " "Serializable $\Rightarrow$ no cycle"

Suppose there is a cycle TA_i -> TA_j of length 2 in CG(S) .
Then there are conflicting pairs (p,q) and (q',p'), p,p' from TA_i, q,q' from TA_j. No serial schedule will contain both (p,q) and (q',p').

Induction over length of cycle proves the "only if"

Induction: cycle of length n, then 2 TA may be "joined".

$\Rightarrow$ cycle of length n

A word of caution ...

Freie Universität Berlin

- Serializability formal is a **correctness criterion, not a method** which **produces conflict serializable schedules.**

- We **never see a history explicitly** – it would be too late anyway to check for cycles in the corresponding conflict graph at the end of the day...

- We are looking for methods (**synchronization methods**) which enforce the scheduler to **produce only conflict serializable schedules**.

- This has to be proven according to the correctness criterion ("No cycles in the Conflict Graph).

Summary of the TA model

Freie Universität Berlin

- **Serial executions** of a fixed set of transactions T trivially **have isolation properties**
- Schedules of T with the **same effects as** an (arbitrary) **serial execution** are intuitively **correct**
- If all **conflicting pairs** of atomic operations are executed **in the same order** in some schedule S' as in the schedule S, the **effects** of S and S' would be **the same**
- Conflict graph is a simple criterion to check conflict serializability
- **Conflict serializability** is more **restrictive** than necessary (see view serializability -> literature)
- Serializability is a **theoretical model** which defines correctness of executions.