KU LEUVEN

Inleiding tot databanken

7. Transacties & Herstel

Prof. dr. Paolo Pilozzi

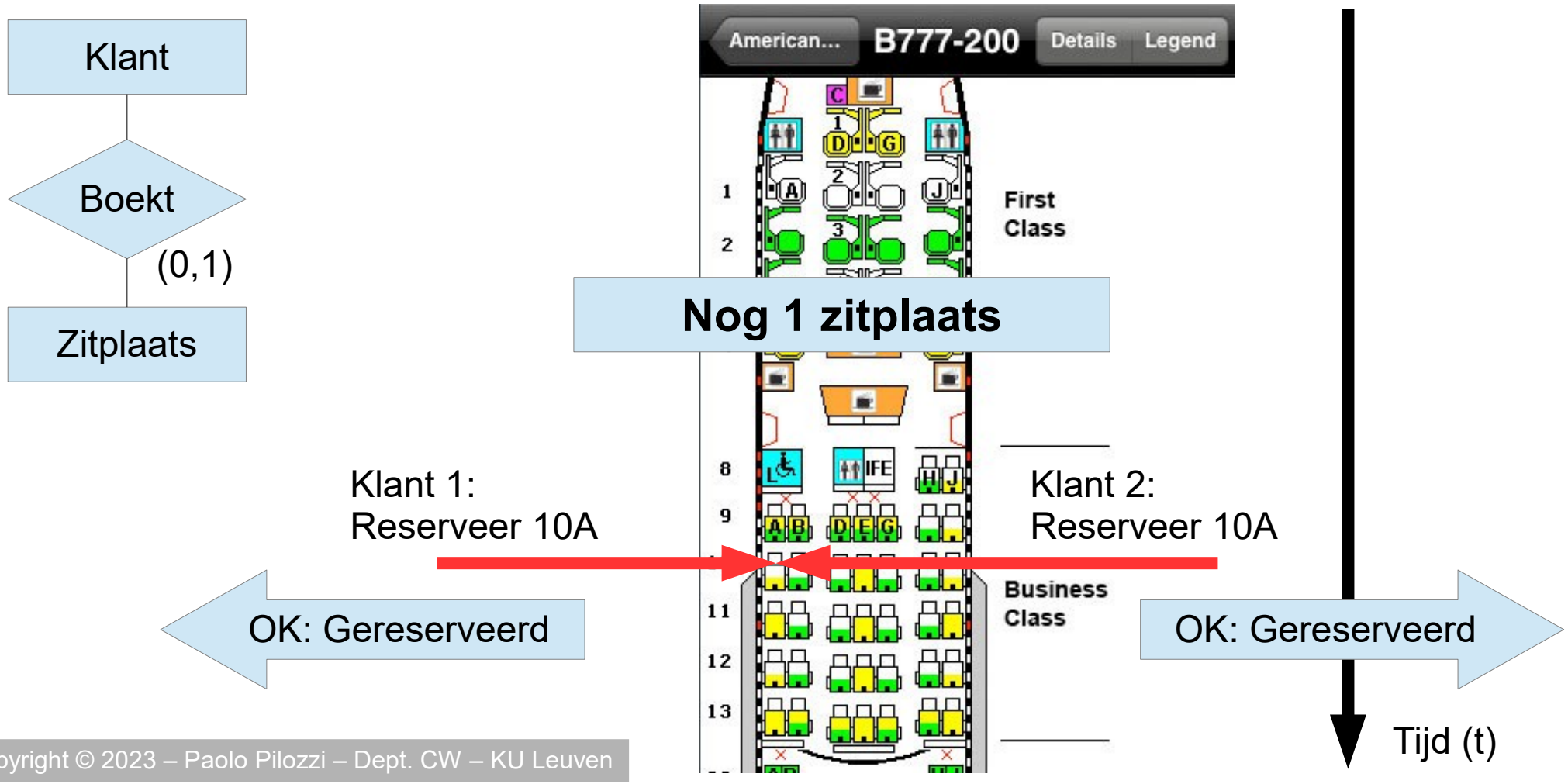
Overzicht

Transacties zijn programma's uitgevoerd op een databank, met concurrentie (gelijktijdig uitvoeren van transacties op data) en herstel (in geval van fouten tijdens transacties) als belangrijke onderdelen.
Hoofdstuk 20, 22.1-3 – Oefenzitting 6

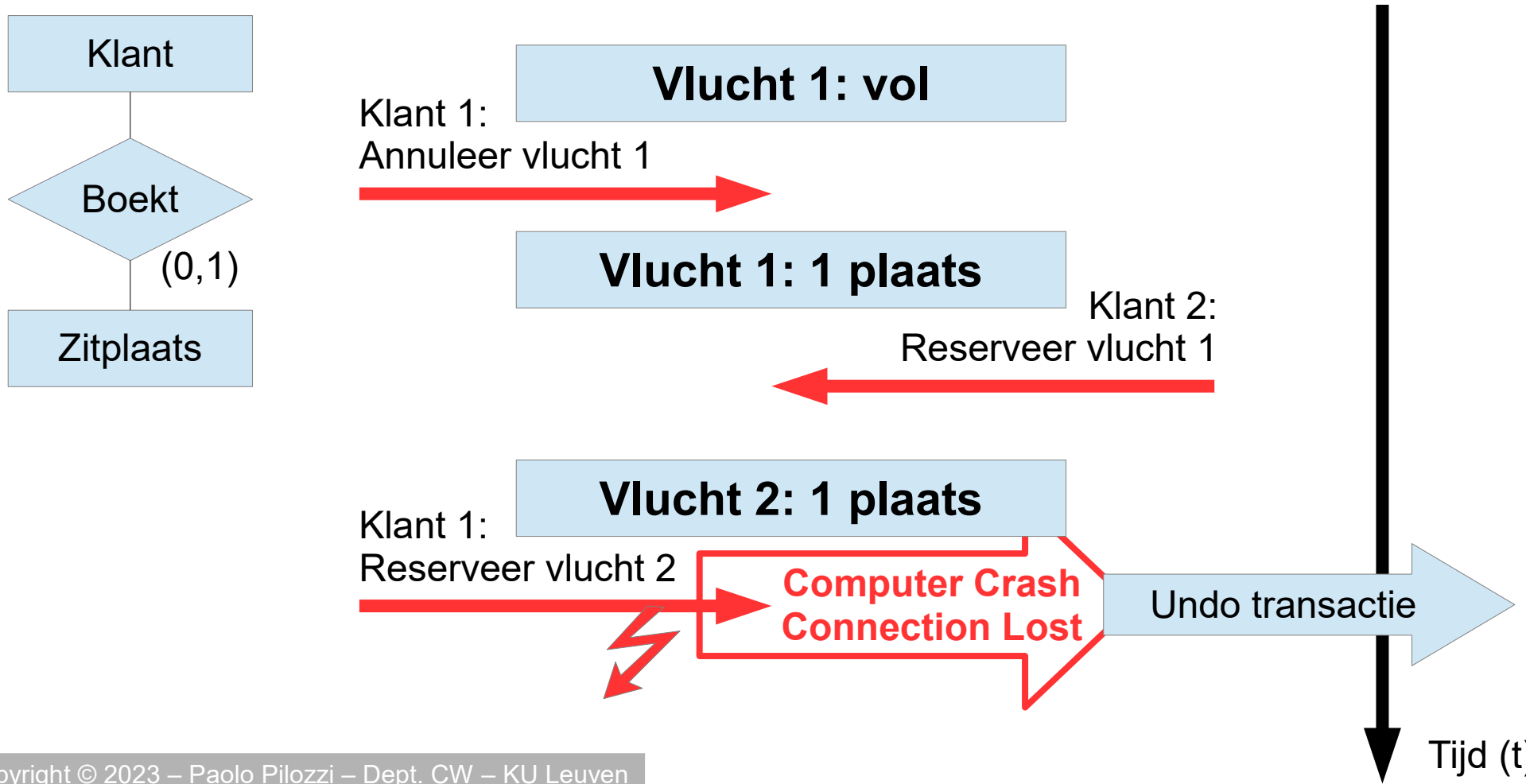
HC10: Transacties & Herstel

* **Inleiding**
* Transacties
* Concurrentie
* Herstel

Voorbeeld 1 - Concurrentie



Voorbeeld 2 - Herstel



Overzicht

Transacties zijn programma's uitgevoerd op een databank, met concurrentie (gelijktijdig uitvoeren van transacties op data) en herstel (in geval van fouten tijdens transacties) als belangrijke onderdelen.
Hoofdstuk 20, 22.1-3 – Oefenzitting 6

HC10: Transacties & Herstel

* Inleiding
* **Transacties**
* Concurrentie
* Herstel

Transacties

= Uitvoering van programma's op data in databank

* Read-only transactie:
  - Alleen gegevens ophalen (=> read_item(X))
* Update transactie:
  - Ook met aanpassing van gegevens (=> write_item(X))

Een transactie is een geheel

- Kan uit meerdere databank operaties bestaan
- Wordt helemaal of niet uitgevoerd

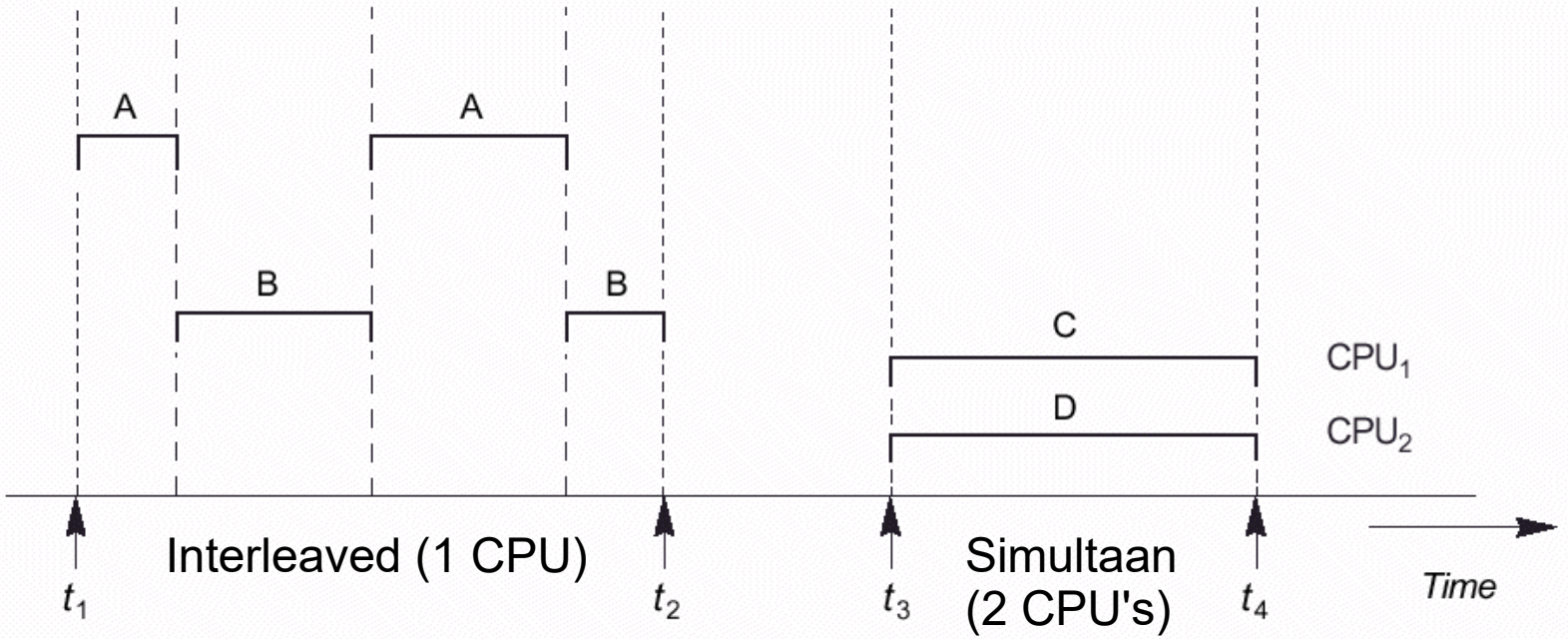
Gelijktijdige verwerking (cfr. concurrentie) is wenselijk

- We veronderstellen immers meerdere gelijktijdige gebruikers

Transacties

= Uitvoering van een programma op databank

* Interleaved
* Gelijktijdig (simultaan/in parallel)



Transacties

=> Gewenste eigenschappen ("ACID properties"):

* Atomicity (Ondeelbaarheid) → Herstel
  - Transactie wordt volledige uitgevoerd of niet.
* Consistency preservation → Programmeur & DBMS (constraints)
  - Consistente databank moet na transactie consistent blijven.
* Isolation (Geïsoleerdheid) → Concurrentiecontrole
  - Effect van transactie: alsof het de enige transactie is.
    => Geen interferentie met andere transacties!
* Durability (Duurzaamheid) → Herstel
  - Effect van transactie moet persistent zijn
    => Effect mag niet verloren gaan!

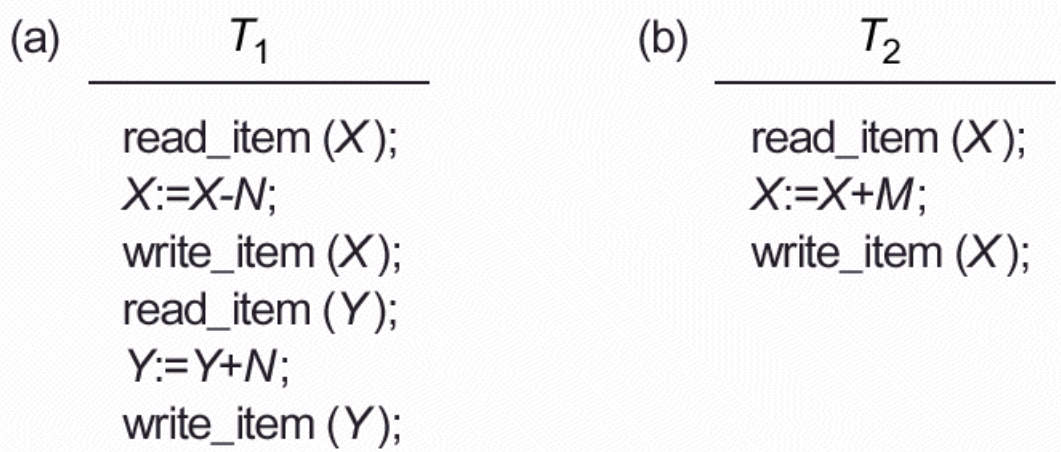
Transacties - Concurrentie

=> Gelijktijdige behandeling transacties is problematisch

Voorbeeld: Vliegtuigreservatiesysteem
* Transactie T1: annuleer reservatie N plaatsen op vlucht V1 en
reserveer N plaatsen op vlucht V2
* Transactie T2: reserveer M plaatsen op vlucht V1

* Mogelijke problemen:
  - Verloren aanpassing
  - Tijdelijke aanpassing
  - Foutieve sommering

=> Vermijden d.m.v. concurrentiecontrole



Transacties - Concurrentie

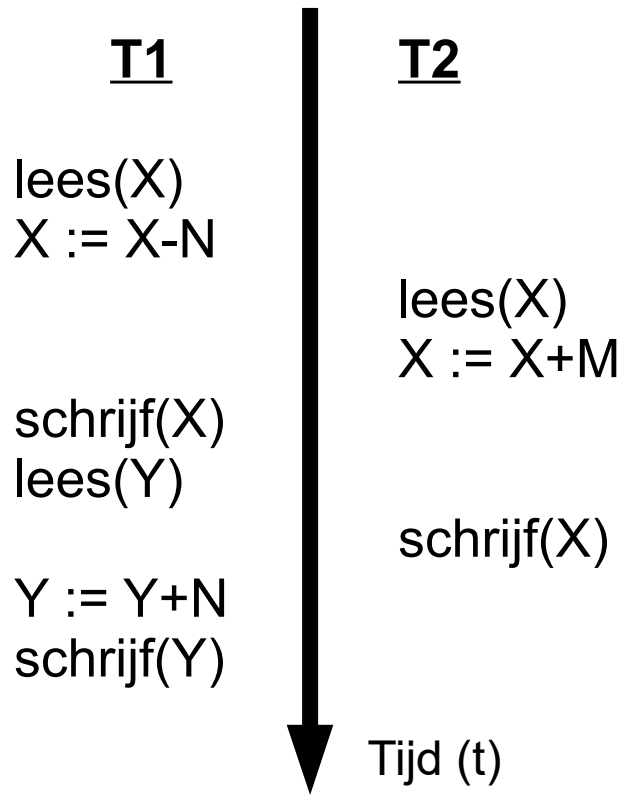
=> Gelijktijdige behandeling transacties is problematisch

Probleem 1: Verloren aanpassing

* X reservaties op V1
* Y reservaties op V2
* Wijziging van T1 door T2 teniet gedaan

Voorbeeld:
X = 84
N = 5
M = 4

Resultaat:
X = 88 i.p.v. 83



Transacties - Concurrentie

=> Gelijktijdige behandeling transacties is problematisch

Probleem 2: Tijdelijke aanpassing (Dirty read)

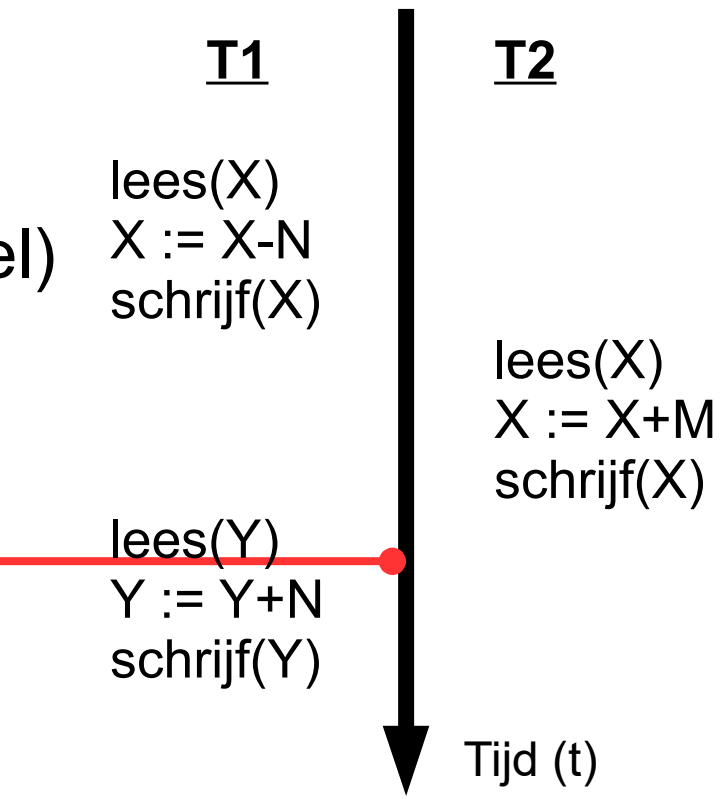
* T1 wordt afgebroken door crash (falen)
* Gewijzigde waarden worden hersteld
* T2 ongeldige waarde gebruikt (voor herstel)

Voorbeeld: lets gaat fout bij lezen van Y

=> T1 afgebroken

=> X hersteld naar waarde voor T1

=> T2 heeft foutieve waarde gebruikt

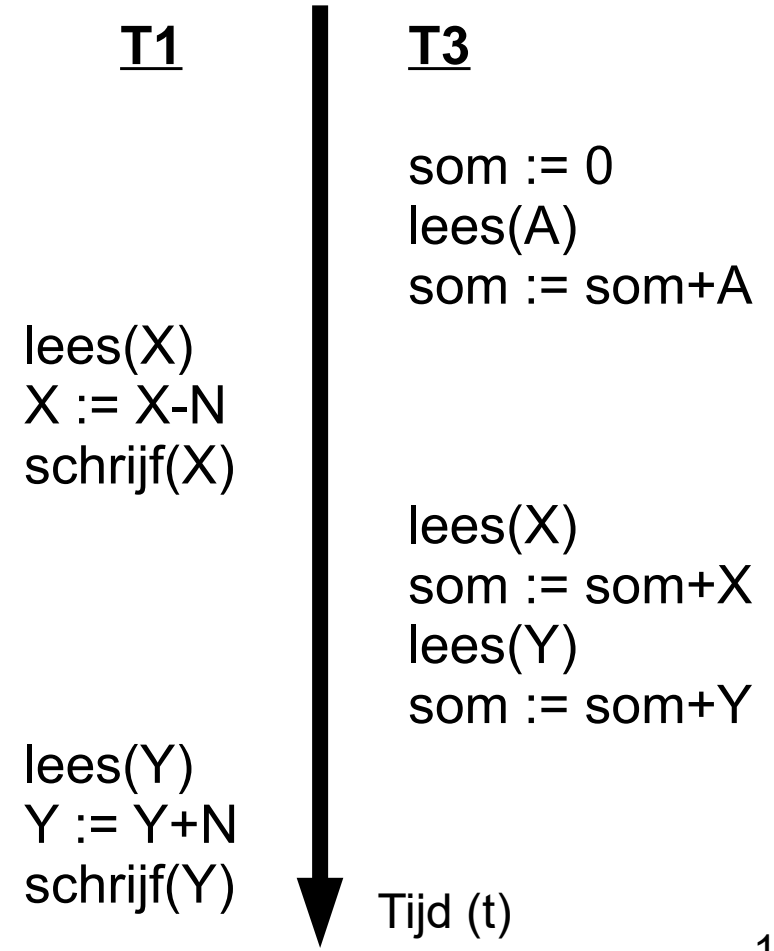


Transacties - Concurrentie

=> Gelijktijdige behandeling transacties is problematisch

Probleem 3: Foutieve sommering
* Aggregaatfunctie gebruikt foutieve waarde
  => Deels voor en na wijzigingen

Voorbeeld:
* T3 berekent totaal aantal reservaties
  op vluchten terwijl T1 wordt uitgevoerd



Transacties - Concurrentie

=> Gelijktijdige behandeling transacties is problematisch

Probleem 4: Niet-herhaalbare lezing (= non-repeatable read)

* Gerelateerd aan: Foutieve sommering

=> Lees zelfde waarde kort na elkaar:
Waarde ondertussen gewijzigd door andere transactie

Voorbeeld:

* Reservatie vliegtuigtickets
  - Controleer op vrije plaatsen
  - Zoja: reserveer plaatsen
  - Mislukt: plaatsen niet meer vrij door gelijktijdige reservatie

Transacties

Mogelijke fouten tijdens transacties:

1. Computer-crash: Inhoud van geheugen kan verloren gaan
2. Transactie/Systeemfout: Fouten bij gebruik en uitvoering
   - Verkeerde parameter, overflow, deling door 0,
   logische programmeerfout, etc.
3. Uitzonderingscondities: Bestand kan niet gelezen worden, etc.
4. Door concurrentiecontrole: Transactie afgebroken door deadlock

=> *Oorspronkelijke toestand herstellen!*

5. Schijffout: Beschadigd spoor op schijf
6. Fysieke problemen (catastrofes): Brand, stroomonderbreking, etc.

=> Gebruik van backups

Overzicht

Transacties zijn programma's uitgevoerd op een databank, met concurrentie (gelijktijdig uitvoeren van transacties op data) en herstel (in geval van fouten tijdens transacties) als belangrijke onderdelen.
Hoofdstuk 20, 22.1-3 – Oefenzitting 6

HC10: Transacties & Herstel

* Inleiding
* Transacties
* **Concurrentie**
* Herstel

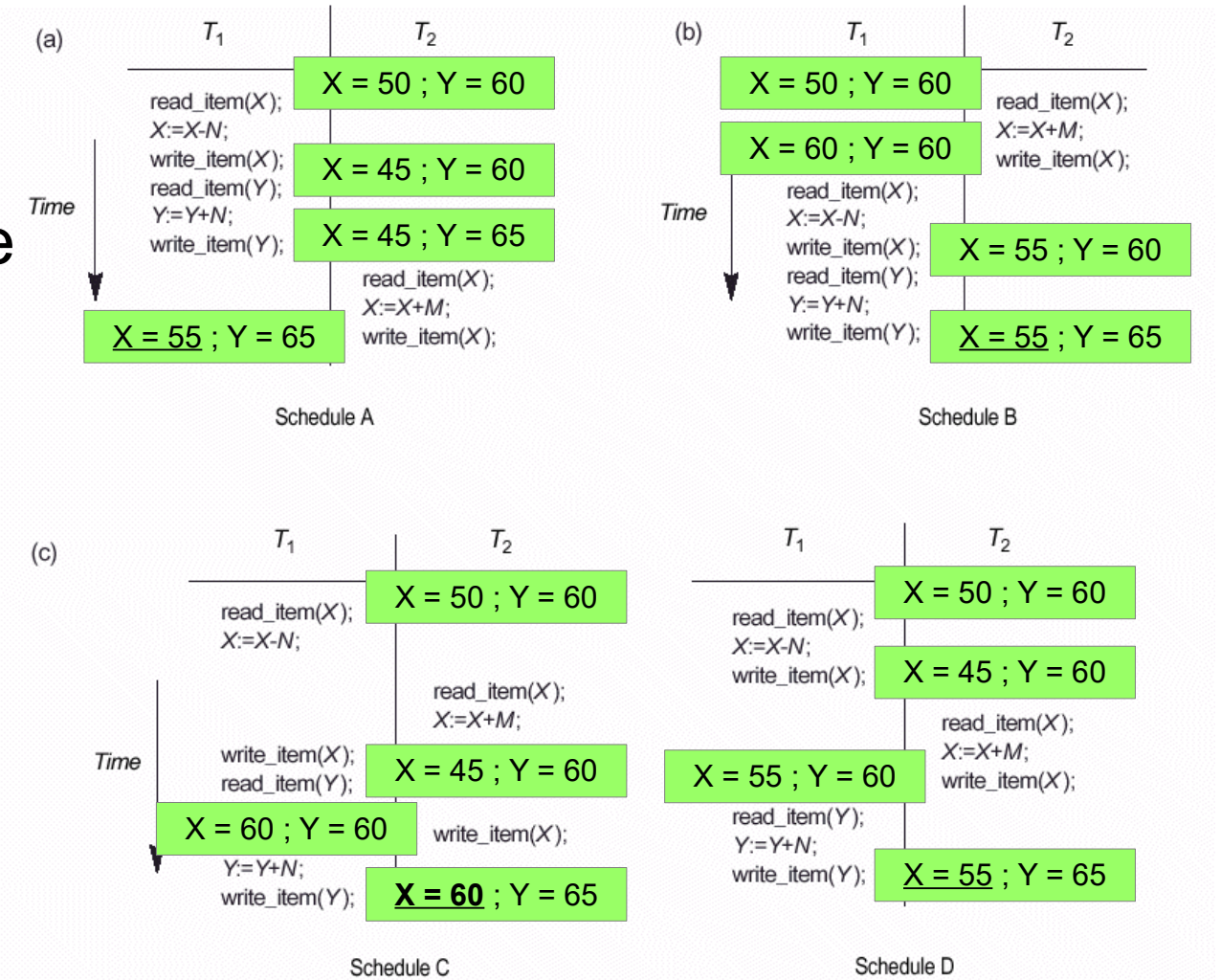
Concurrentiecontrole

Transactieroosters

= Operaties van meerdere transacties in chronologische volgorde neergeschreven

Voorbeeld:

* N = 5
* M = 10
* X = 50
* Y = 60



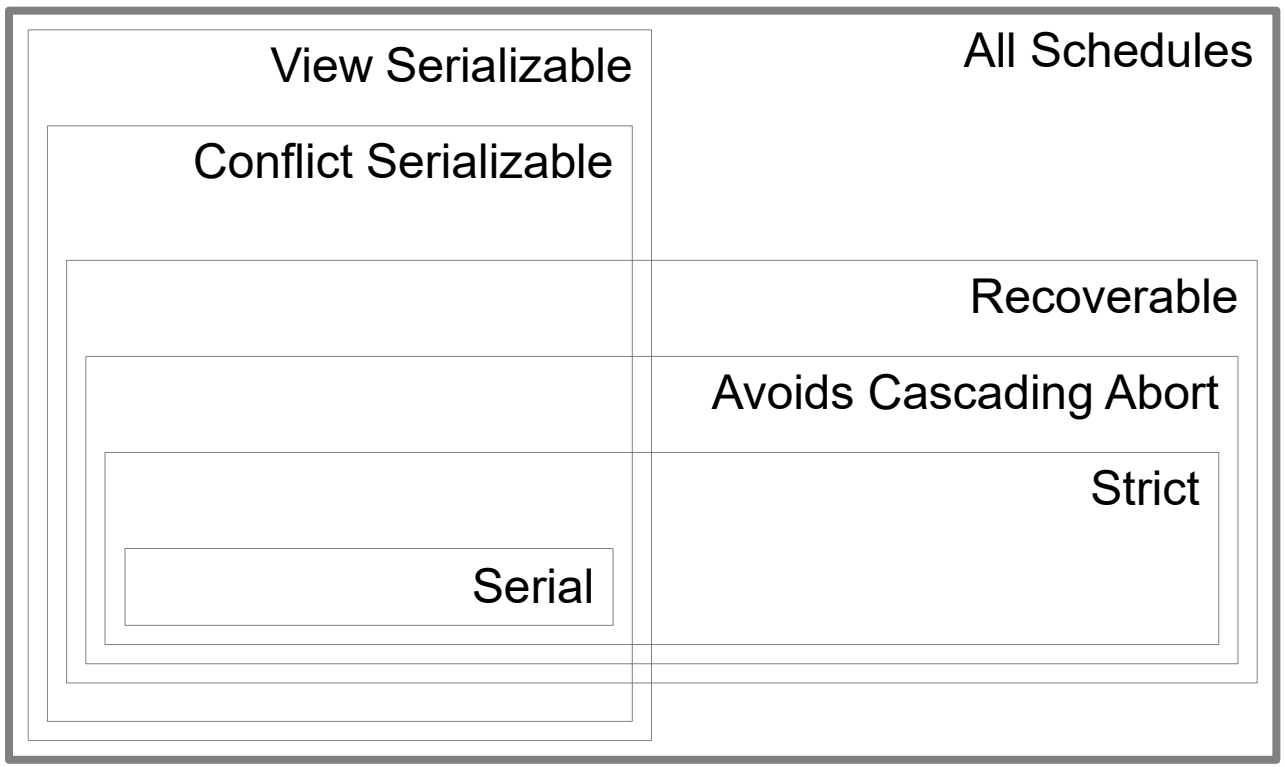
Concurrentiecontrole

Eigenschappen Transactieroosters:

* Operaties conflicteren indien
  - van verschillende transacties,
  - ze hetzelfde dataelement gebruiken, en
  - minstens één ervan een write_item is
* Een rooster S voor n transacties Ti is volledig indien
  - S alle operaties van alle Ti bevat (inclusief abort-operaties als laatste operatie transacties) en geen andere,
  - operaties van Ti in dezelfde volgorde voorkomen in S, en
  - voor elke paar conflicterende operaties geldt: volgorde in S ligt eenduidig vast

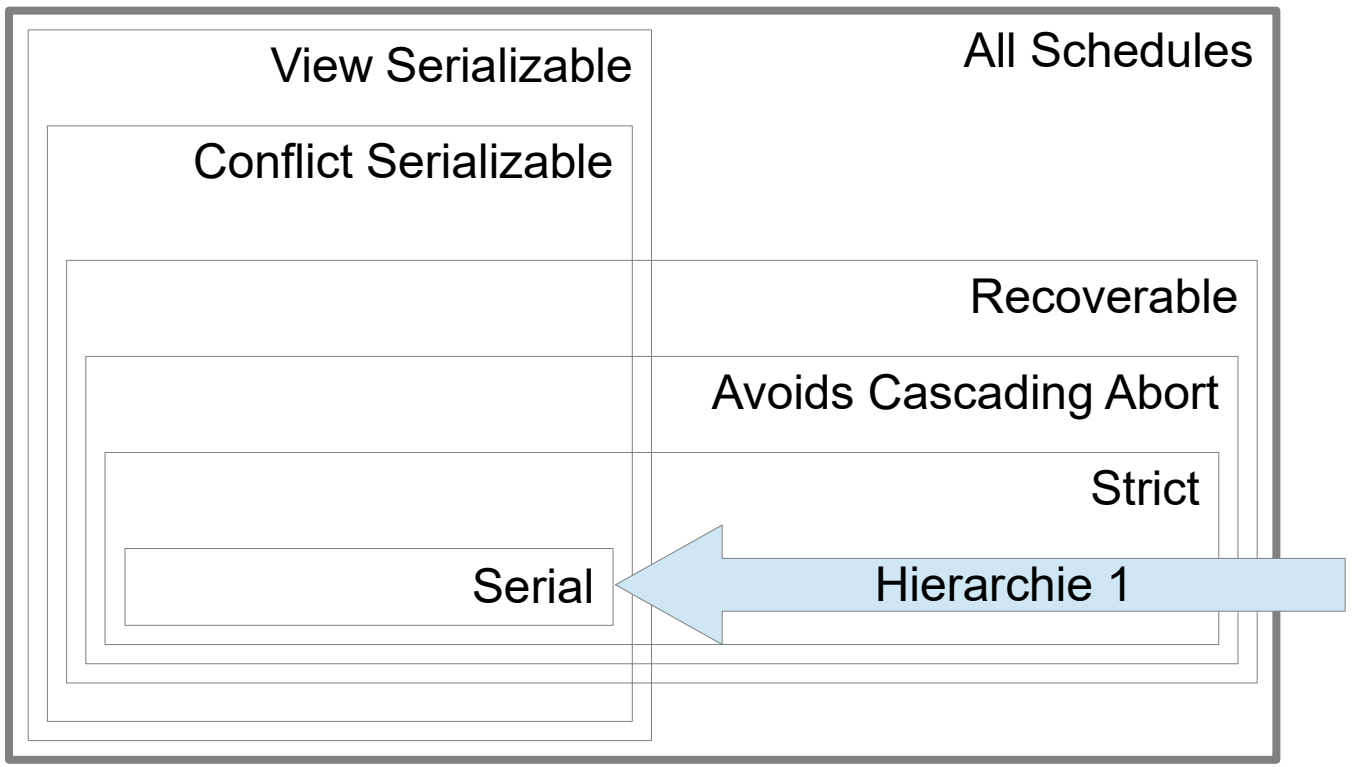
Concurrentiecontrole

Twee hierarhieën van eigenschappen v. roosters:



Concurrentiecontrole

Twee hierarhieën van eigenschappen v. roosters:



Concurrentiecontrole

Transactieroosters – Hierarchie 1: Herstelbaarheid

=> Recoverable rooster

Een rooster is herstelbaar a.s.a. een transactie die gecommit (vastgelegd) werd niet meer ongedaan gemaakt kan worden.

=> Voldoende voorwaarde: T commit enkel na commit van elke transactie die een waarde schrijft die T leest

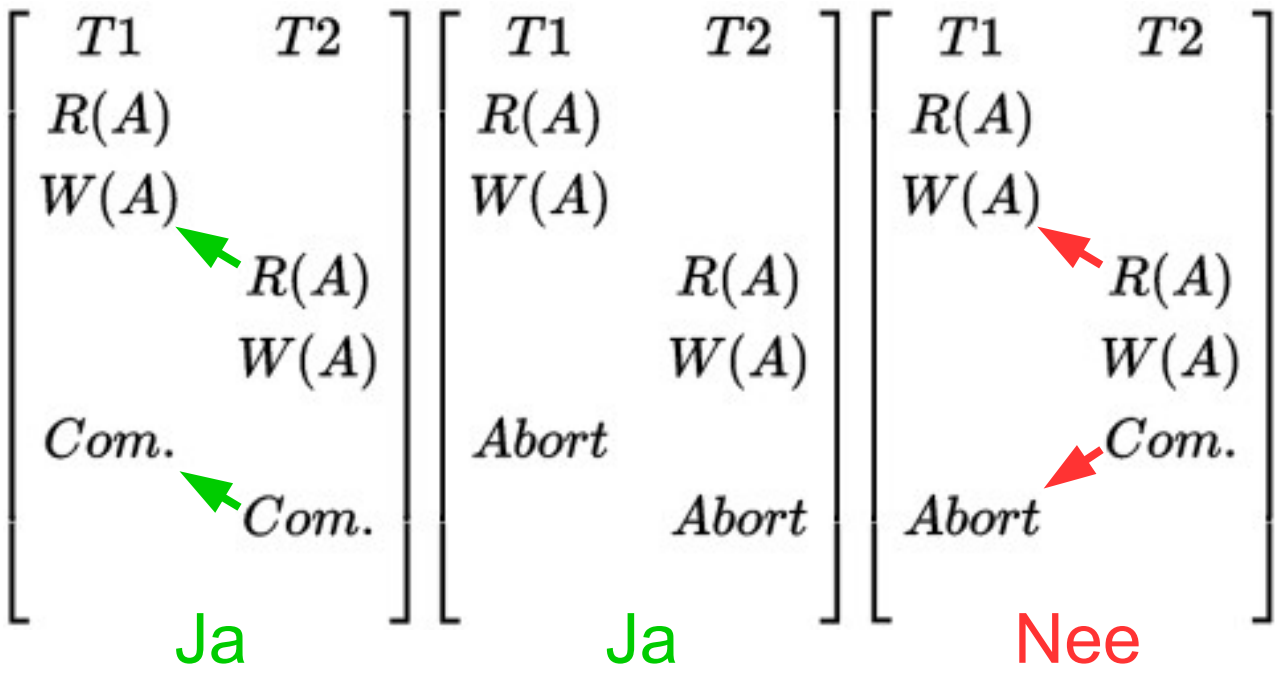
Herstelbaar impliceert niet "eenvoudig herstelbaar":

* Mogelijks "cascading rollback" (in cascade terugdraaien)
* Eén transactie, T, terugdraaien kan andere transacties ook laten terugdraaien (Zij die lezen wat door T geschreven werd).
* Cascading rollback is tijdsintensief.

Concurrentiecontrole

Transactieroosters – Hierarchie 1: Herstelbaarheid

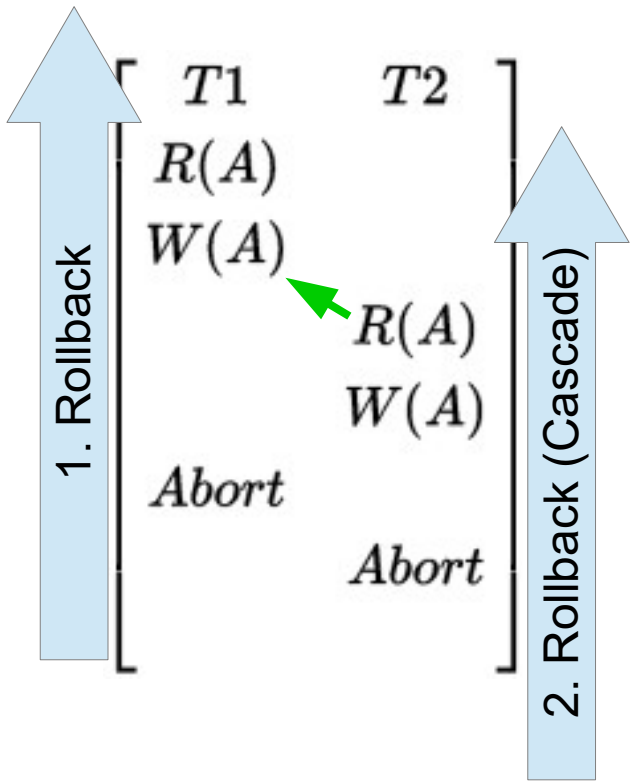
Voorbeeld:
Legende:
- R: Lees
- W: Schrijf
- Com.: Commit
- Abort

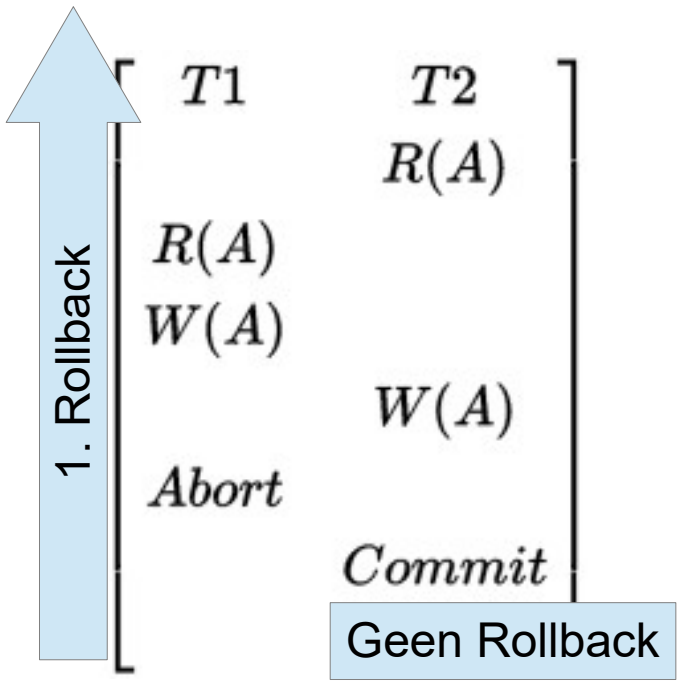


Concurrentiecontrole

Transactieroosters – Hierarchie 1: Herstelbaarheid

Voorbeeld:
Legende:
- R: Lees
- W: Schrijf
- Com.: Commit
- Abort





Concurrentiecontrole

Transactieroosters – Hierarchie 1: Cascadeloze Roosters

=> Avoids cascading abort

Elke transactie T leest enkel items geschreven door transacties die al gecommit werden.

=> Garandeert dat geen cascading rollbacks nodig zijn

Is een extra beperking!

=> Minder mogelijkheden om transacties gelijktijdig uit te voeren.

Concurrentiecontrole

Transactieroosters – Hierarchie 1: Strikte Roosters

=> Strict

Elke transactie T leest en schrijft enkel items na commit (abort) van de laatste transactie die dat item geschreven heeft.

UNDO write_item: gewoon oorspronkelijke waarde terugzetten

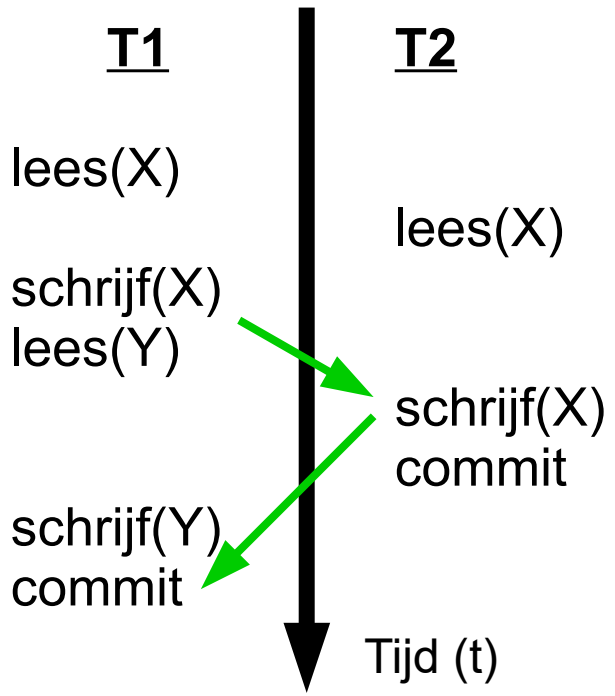
Meest restrictieve!

=> Relatief weinig roosters mogelijk
=> Eenvoudigst om te herstellen
=> In de praktijk vaak gebruikt

Concurrentiecontrole

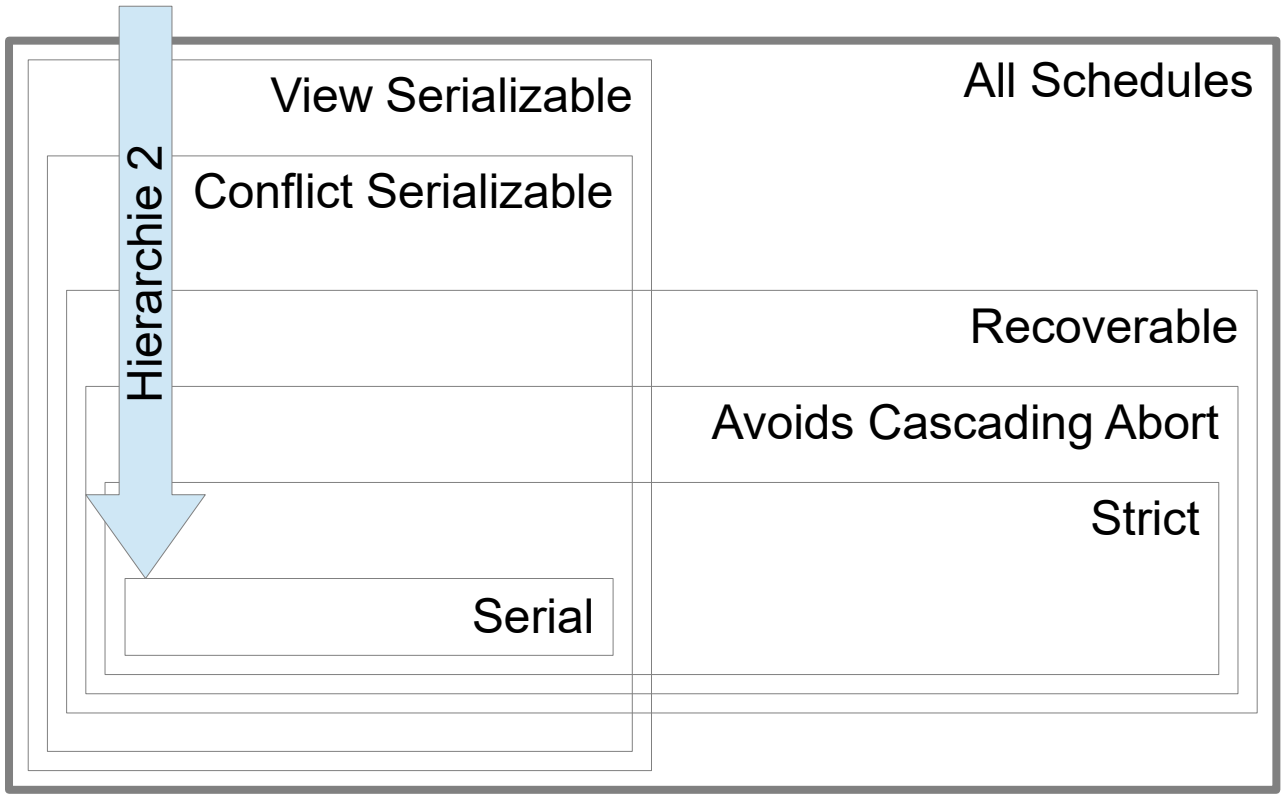
Transactieroosters – Hierarchie 1
Voorbeeld: Herstelbaar, maar met verloren aanpassing

Herstelbaar?	Ja
Cascadeloos?	Nee
Strikt?	Nee



Concurrentiecontrole

Twee hierarchieën van eigenschappen v. roosters:



Concurrentiecontrole

Transactieroosters – Hierarchie 2: Roosters serializeren

=> Serial

Serieel rooster:

* Tussen eerste en laatste opdracht van een transactie T worden geen opdrachten van eender welke andere transactie uitgevoerd

=> Transacties worden na elkaar uitgevoerd

=> Er kan geen interferentie zijn

=> Als transacties onafhankelijk: elk serieel rooster correct

Nadeel: Sterke beperking op concurrentie

Concurrentiecontrole

Transactieroosters – Hierarchie 2: Serializeerbaarheid

Een rooster S van n transacties is serializeerbaar a.s.a. het equivalent is met een serieel rooster met dezelfde n transacties.

Er zijn meerdere soorten equivalenties definieerbaar:

* resultaat-equivalentie => voor beginvoorwaarden, zelfde resultaat
  - Te zwak: voor andere beginvoorwaarden mogelijks niet zelfde resultaat
    - Voorbeeld:

X = 100 of niet

S1	S2
read_item(X)	read_item(X)
X := X+10	X := X*1.1
write_item(X)	write_item(X)

* conflict-equivalentie is beter

Concurrentiecontrole

Transactieroosters – Hierarchie 2: Conflict-serializeerbaar

=> Conflict serializable

Twee roosters S1 en S2 zijn conflict-equivalent a.s.a. de volgorde van 2 conflicterende operaties steeds dezelfde is in beide roosters.

Rooster S is conflict-serializeerbaar a.s.a. S conflict-equivalent is met een serieel rooster.

Testen van conflict-serializeerbaarheid op basis van "precedence graph".

Gerichte (directed) grafen (graphs) => zie oefenzitting

Knopen (nodes): transacties

Bogen (edges): volgorde transacties met conflicterende operaties

Concurrentiecontrole

Transactieroosters – Hierarchie 2: View-serializeerbaar

=> View serializable

Roosters S1 en S2 zijn view-equivalent a.s.a.

* voor elke read_item(X) in Ti in S1 geldt:
  - laatste write_item(X) voor read_item(X) moet in beide roosters zelfde write_item van zelfde transactie Tj zijn
* voor elke X waarvoor een write_item(X) voorkomt:
  - laatste write_item(X) moet zelfde write_item van zelfde transactie Tk zijn in beide roosters

=> overeenkomende leesopdrachten in S1 en S2 lezen/zien dezelfde waarde

=> laatst geschreven waarde voor item is zelfde in beide roosters

Een rooster is view-serialiseerbaar a.s.a. het view-equivalent is met een serieel rooster.

Concurrentiecontrole

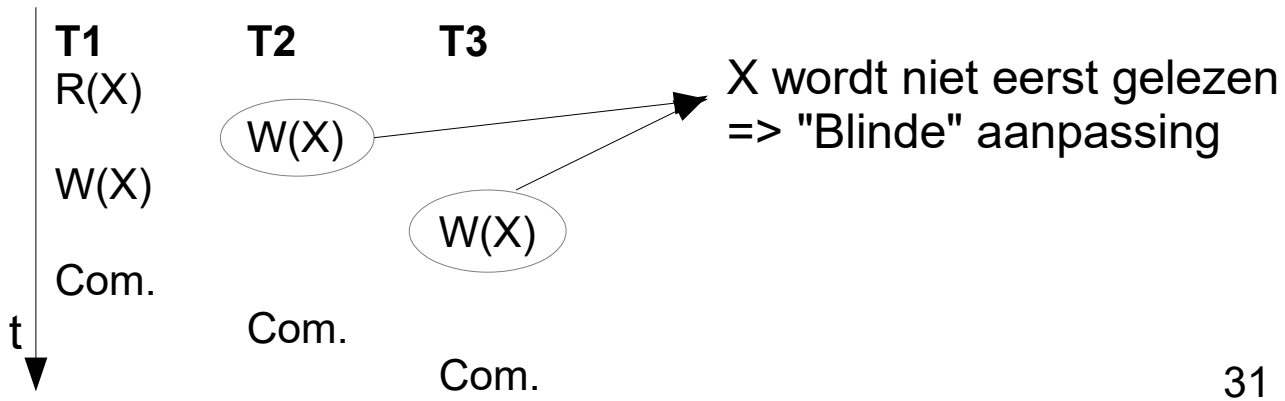
Transactieroosters – Hierarchie 2

View-serializeerbaar vs. Conflict-serializeerbaar

* Hetzelfde gegeven de "constrained write" eigenschap
  = aan elk write_item(X) gaat een read_item(X) vooraf, en de geschreven waarde hangt enkel af van de gelezen waarde
* Anders view-equivalentie minder restrictief dan conflict-equivalentie
* Maar: Testen van view-equivalentie is NP-compleet!

Voorbeeld:

Dit rooster is view-serializeerbaar
Want view-equivalent met:
T1 → T2 → T3
Maar niet conflict-serializeerbaar



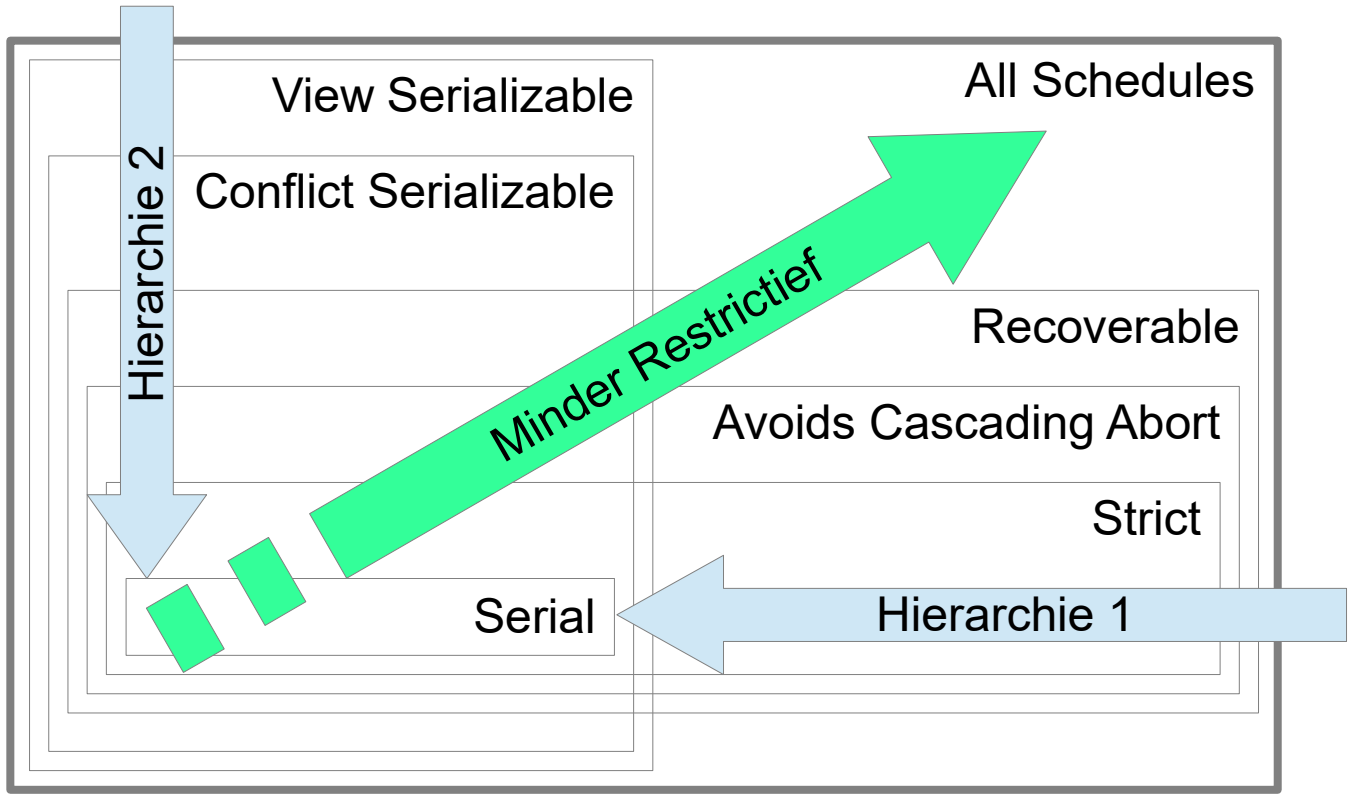
Concurrentiecontrole

Twee hierarchieën van eigenschappen v. roosters:

Minder restrictief

<=> Flexibelere/meer mogelijkheden tot Concurrentie

<=> Verificatie van Serializeerbaarheid wordt moeilijker



Concurrentiecontrole

Transactieroosters – Hierarchie 2

=> Testen/Verzekeren van serializeerbaarheid

* Volgende problemen:
  - Interleaving v. operaties bepaald door OS => niet te voorspellen
  - Transacties worden continu aangeboden
    => Begin en einde van roosters moeilijk te voorspellen
  - Indien rooster niet serializeerbaar blijkt: Herstel nodig => Duur!
* Vermijden van deze problemen: Concurrentiecontrole
  => Gebruikt bij opstellen van transactieregels (protocols) om serializeerbaarheid te garanderen
  Voorbeeld: Lees/schrijf grendels (locks), tijdstempels (timestamps), ...

Overzicht

Transacties zijn programma's uitgevoerd op een databank, met concurrentie (gelijktijdig uitvoeren van transacties op data) en herstel (in geval van fouten tijdens transacties) als belangrijke onderdelen.
Hoofdstuk 20, 22.1-3 – Oefenzitting 6

HC10: Transacties & Herstel

* Inleiding
* Transacties
* Concurrentie
* **Herstel**

Herstel

Mogelijke fouten tijdens transacties:

1. Computer-crash: Inhoud van geheugen kan verloren gaan
2. Transactie/Systeemfout: Fouten bij gebruik en uitvoering
   - Verkeerde parameter, overflow, deling door 0,
   logische programmeerfout, etc.
3. Uitzonderingscondities: Bestand kan niet gelezen worden, etc.
4. Door concurrentiecontrole: Transactie afgebroken door deadlock

=> *Oorspronkelijke toestand herstellen!*

5. Schijffout: Beschadigd spoor op schijf
6. Fysieke problemen (catastrofes): Brand, stroomonderbreking, etc.

=> Gebruik van backups (+ REDO van volledige transacties)

Herstel

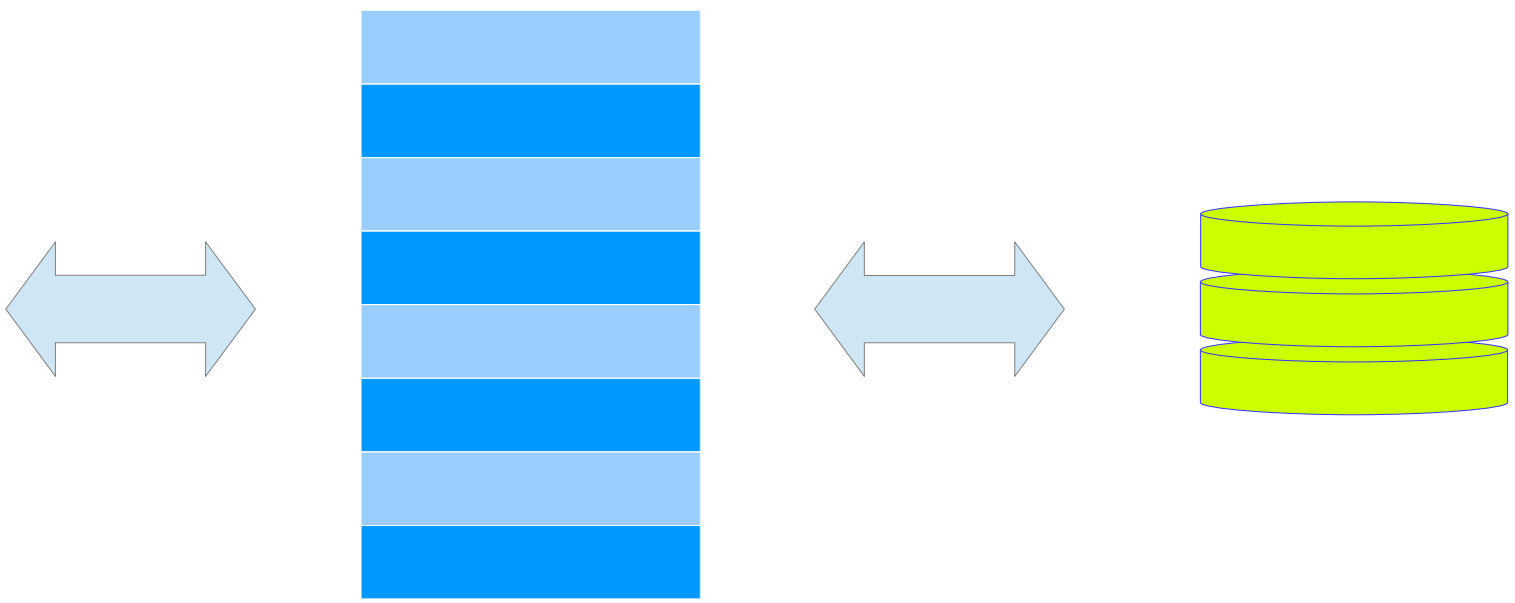
=> Gewenste eigenschappen ("ACID properties"):

* Atomicity (Ondeelbaarheid) → Herstel
  - Transactie wordt volledige uitgevoerd of niet.
* Consistency preservation
  - Consistente databank moet na transactie consistent blijven.
* Isolation (Geïsoleerdheid)
  - Effect van transactie: alsof het de enige transactie is.
    => Geen interferentie met andere transacties!
* Durability (Duurzaamheid) → Herstel
  - Effect van transactie moet persistent zijn
    => Effect mag niet verloren gaan!

Herstel

Wanneer schrijven we (welke) informatie naar schijf?

Transacties		Geheugen (Cache)	Schijf
T1	**T2**		
R(Y)			
	R(X)		
	R(Y)		
R(Z)			
W(Z)			
Com.			
	W(X)		



Herstel

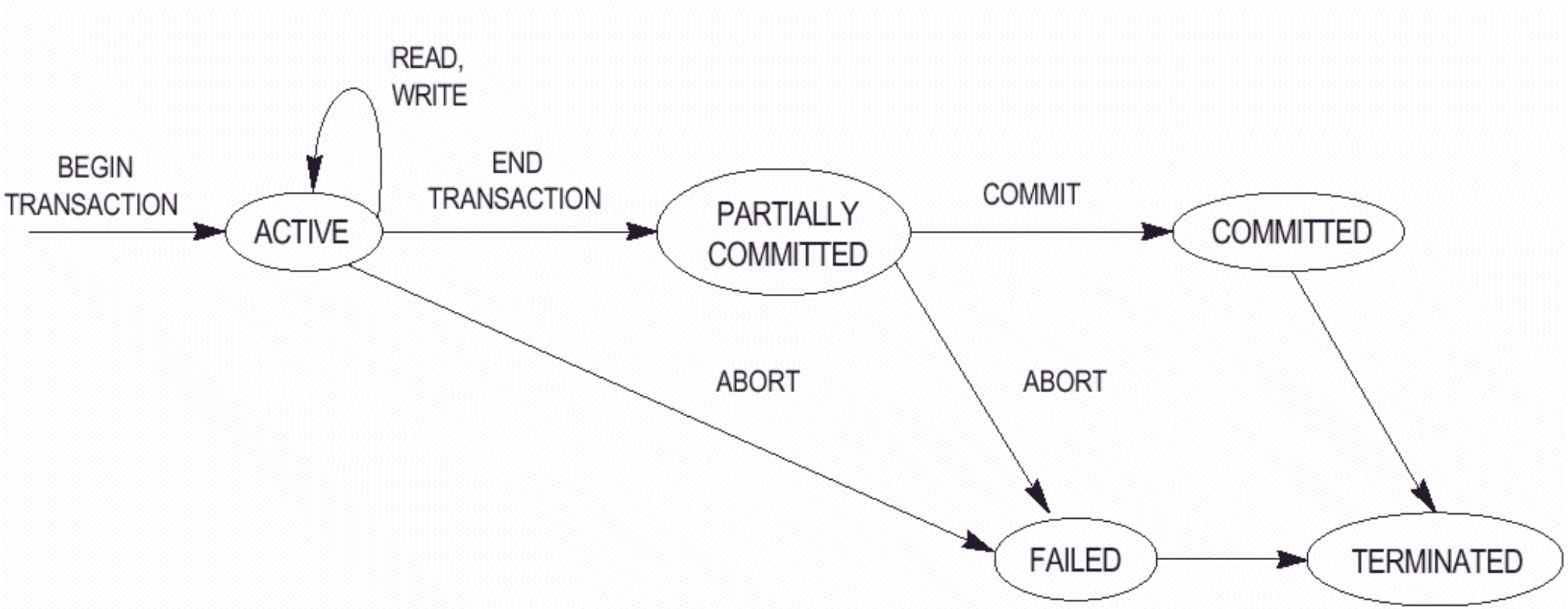
Lezen en Schrijven:

* read_item(X):
    * Vind adres van blok dat X bevat
    * Kopieer dat blok in een buffer
    * kopieer X in programmavariabele X

* write_item(X):
    * Vind adres van blok dat X bevat
    * Kopieer dat blok in een buffer
    * Kopieer de programmavariabele X op de juiste plaats
    * Bewaar aangepast blok terug op schijf (nu of later)

Herstel – Status v/e transactie

=> DBMS moet dit bijhouden voor herstel



Herstel – Status v/e transactie

* BEGIN_TRANSACTION: Geeft begin van transactie aan
* READ/WRITE: Alle lees -en schrijfoperaties
* END_TRANSACTION: Geeft einde van transactie aan
  => Kunnen wijzigingen veroorzaakt door de transactie definitief doorgevoerd worden op de databank (=committed)?
  => Moet de transactie ongedaan gemaakt moet worden als gevolg van de concurrentiecontrole?
* COMMIT_TRANSACTION: Succesvol einde transactie
  => Wijzigingen definitief
* ROLLBACK (of ABORT): Gefaald einde transactie
  => Wijzigingen terugdraaien
* UNDO/REDO (zie volgende slides)

Herstel – Logging

DBMS houdt status bij in Log bestand met info transacties:

* [start_transaction,T]
* [write_item,T,X,oude_waarde,nieuwe_waarde]
* [read_item,T,X]
* [commit,T]
* [abort,T]
* [checkpoint,<lijst van actieve transacties>]

=> Laatste stuk is in Log buffer, volledig Log bestand is op schijf

* Voor toevoegen van [commit,T] aan Log:
  eerst Log buffer naar schijf schrijven (= force-writing)

Herstel – Checkpoints

Checkpoint:

* Pauzeer actieve transacties
* Schrijf alle aangepaste buffers naar schijf
* Schrijf checkpoint record
* Schrijf log naar schijf
* Zet actieve transacties voort

=> Bij herstel zijn REDOs van transacties die gecommit werden voor het laatste checkpoint niet nodig.
=> DBMS herstel manager bepaalt frequentie van checkpoints
=> "Fuzzy checkpointing" vermijdt pauzes

Herstel

Assumptie: Strikte roosters

=> Tussen [write_item,T,X,…] en
latere [read_item,T',X] of [write_item,T',X,…]
altijd [commit, T]

=> T' mag X pas lezen/schrijven nadat de door T geschreven waarde definitief is

Voor een transactie T, met [start_transaction, T] in log:

* Als [commit, T] voor laatste checkpoint in log
  => Wijzingen T zeker op schijf
* Als [commit, T] na laatste checkpoint
  => Wijzingen T mogelijks niet op schijf: REDO
* Als Geen [commit, T]
  => Wijzingen T mogelijks al op schijf: UNDO

Herstel – Uitgestelde aanpassing

Kenmerken uitgestelde aanpassing:

* Voor commit point: Veranderingen in log en buffers, niet op schijf
* Bij commit point: schrijf log naar schijf, maak verandering permanent

=> Nooit UNDO nodig
- Nog niet definitieve wijzigingen nooit op schijf doorgevoerd
- Enkel doorgevoerd in cache

=> No-UNDO/ REDO herstel-algoritme.

Herstel – Uitgestelde aanpassing

No-UNDO/REDO herstel-algoritme

1. Twee lijsten:
   * L_c: alle transacties T met [commit,T] na laatste checkpoint
   * L_a: alle transacties T met [start_transaction,T], en zonder [commit,T], in log
2. REDO:
   A. Doorloop log vanaf vroegste [start_transaction,T], met T in L_c
      - REDO elke [write_item,T',X,oud,nieuw] met T' in L_c
   B. Herstart alle transacties in L_a

Herstel – Uitgestelde aanpassing

Voorbeeld:

[start_transaction, T1]
[read, T1, A, 17]
[start_transaction, T2]
[read, T2, C, 23]
[write, T2, C, 23, 45]
[commit, T2]
[start_transaction, T3] <<< [2.A]
[checkpoint, [T1,T3]]
[write, T3, B, 89, 20] >>> REDO
[start_transaction, T4]
[read, T4, D, 2]
[read, T1, C, 45]
[read, T4, E, 9]
[write, T3, F, 9, 24] >>> REDO
[write, T1, C, 45, 65]
[write, T4, D, 2, 11]
[write, T4, E, 9, 7]
[commit, T3]
[read, T1, B, 20]
[write, T1, A, 17, 41]
CRASH

[1] Lc = {T3} en La = {T1,T4}
[2.B] Herstart T1 en T4

Herstel – Onmiddelijke aanpass.

Kenmerken onmiddelijke aanpassing:

* Aanpassingen kunnen naar schijf geschreven worden voor commit
* Wel steeds log-inschrijving voor aanpassing
  - Write-ahead log protocol
  => Log (op schijf) moet zekere voor onzekere nemen:
  zelfs als aanpassing nog niet doorgevoerd (CRASH tussenin)

=> Effect van aanpassing moet soms ongedaan gemaakt worden

=> UNDO/REDO herstel algoritme

Herstel – Onmiddelijke aanpass.

UNDO/REDO herstel-algoritme

1. Twee lijsten:
   * L_c: alle transacties T met [commit,T] na laatste checkpoint
   * L_a: alle transacties T met [start_transaction,T], en zonder [commit,T], in log
2. UNDO:
   * Doorloop log van einde tot vroegste [start_transaction,T], met T in L_a
     - UNDO elke [write_item,T',X,oud,nieuw] met T' in L_a
3. REDO:
   A. Doorloop log vanaf vroegste [start_transaction,T], met T in L_c
     - REDO elke [write_item,T',X,oud,nieuw] met T' in L_c
   B. Herstart alle transacties in L_a

Herstel – Onmiddelijke aanpass.

Voorbeeld:

[start_transaction, T1] <<< [2]
[read, T1, A, 17]
[start_transaction, T2]
[read, T2, C, 23]
[write, T2, C, 23, 45]
[commit, T2]
[start_transaction, T3] <<< [3.A]
[checkpoint, [T1,T3]]
[write, T3, B, 89, 20] >>> REDO
[start_transaction, T4]
[read, T4, D, 2]
[read, T1, C, 45]
[read, T4, E, 9]
[write, T3, F, 9, 24] >>> REDO
[write, T1, C, 45, 65] >>> UNDO
[write, T4, D, 2, 11] >>> UNDO
[write, T4, E, 9, 7] >>> UNDO
[commit, T3]
[read, T1, B, 20]
[write, T1, A, 17, 41] >>> UNDO
CRASH

[1] Lc = {T3} en La = {T1,T4}
[3.B] Herstart T1 en T4