Einige Synchronisationsprobleme

- Welche Synchronisationsprobleme werden durch *CSR* verhindert?
  - Lost Update
    - $s = r_1(x)\ r_2(x)\ w_1(x)\ c_1\ w_2(x)\ c_2 \notin CSR$
  - Dirty Read
    - $s = r_1(x)\ w_1(x)\ r_2(x)\ \boldsymbol{a_1}\ w_2(x)\ \boldsymbol{c_2} \in \boldsymbol{CSR}$
  - Non Repeatable Read
    - $s = r_1(x) r_2(x) w_2(x) c_2 r_1(x) c_1 \notin CS$

Ist Konfliktserialisierbarkeit ausreichend?

- Konfliktserialisierbarkeit ist für praktische Anwendungen wichtig
  - Effizient überprüfbar:
    - Konfliktgraph berechenbar mit linearem Aufwand in der Länge des Schedules
    - Test auf Azyklizität mit höchstens quadratischem Aufwand in der Anzahl der Knoten
  - Konfliktbeziehungen sind unabhängig vom Abbruch einer Transaktion
- Konfliktserialisierbarkeit allein ist nicht ausreichend, vgl. folgenden Schedule:

$$s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2 \in CSR$$

- Warum?

Ist Konfliktserialisierbarkeit ausreichend?

- Konfliktserialisierbarkeit ist für praktische Anwendungen wichtig
  - Effizient überprüfbar:
    - Konfliktgraph berechenbar mit linearem Aufwand in der Länge des Schedules
    - Test auf Azyklizität mit höchstens quadratischem Aufwand in der Anzahl der Knoten
  - Konfliktbeziehungen sind unabhängig vom Abbruch einer Transaktion
- Konfliktserialisierbarkeit allein ist nicht ausreichend, vgl. folgenden Schedule:

$$s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)\ c_2 \in CSR$$

- Warum?

  - Was passiert in dem Schedule: t_1 liest x, schreibt dann x dann. t_2 liest dann das modifizierte x. Dann bricht t_1 aber ab, und t_2 schreibt und committet einen Wert für x in die Datenbank, der nicht da sein sollte.
  - Also: **Dirty Read**
  - Datenbank muss dafür sorgen, dass $w_1(x)$ nicht durchgeführt wird und t_2 nicht von t_1 liest

Zusätzlich: Rücksetzbarkeit (Recovery)

Anforderung an Zulässige Schedule:

- Transaktionen sollte zu jedem Zeitpunkt vor der Ausführung eines commit zurückgesetzt werden können.
- Das Zurücksetzen einer Transaktion sollte keine anderen Transaktionen beeinflussen

Formalisierung ist notwendig:

„liest-von“-Notation

- Sei s ein Schedule, dann bezeichnet $p <_s q$ das Auftreten von Aktion p vor q
- Seien $t_i, t_j \in trans(s)$:
  1) **t_i liest x von t_j in s**, falls alle drei Bedingungen gelten:
     a) $w_j(x) <_s r_i(x)$
        - Eine Aktion $r_i(x)$ liest x von $w_j(x)$
     b) $a_j \not<_s r_i(x)$
        - t_j ist zum Zeitpunkt des Lesens nicht abgebrochen
     c) $w_j(x) <_s w_k(x) <_s r_i(x) \Rightarrow a_k <_s r_i(x)$
        - $w_j(x)$ ist der letzte echte Schreiber auf x vor $r_i(x)$
  2) **t_i liest von t_j in s**, falls t_i irgendein x von t_j in s liest

Schritt	t_1	t_2	t_3
1	$w_1(x)$		
2	$w_1(y)$		
3		$r_2(u)$	
4		$w_2(x)$	
5		$r_2(y)$	
6			$r_3(x)$
7		$w_2(z)$	
8		a_2	
9	$r_1(z)$		
10	c_1		
11			c_3

Beispiel:

- Sei s ein Schedule mit Transaktionen $t_1, t_2, t_3 \in trans(s)$:
  $s = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ r_3(x)\ w_2(z)\ a_2\ r_1(z)\ c_1\ c_3$

Dann gilt:

t_3 liest x von t_2 aber nicht von t_1
t_2 liest y von t_1
t_1 liest z nicht von t_2

Rücksetzbarkeit

- Ein Schedule s heißt **rücksetzbar**, falls für alle Transaktionen in s gilt, dass sie erst dann committed wird, wenn alle Transaktionen, von denen sie gelesen hat, auch committed wurden.

Formal:

Für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest von } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit RC (recoverable).

Rücksetzbarkeit

- Ein Schedule s heißt **rücksetzbar**, falls für alle Transaktionen in s gilt, dass sie erst dann committed wird, wenn alle Transaktionen, von denen sie gelesen hat, auch committed wurden.

Formal:

Für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest von } t_j \text{ in } s \wedge c_i \in s \Rightarrow c_j <_s c_i$$

- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit RC (recoverable).

Rücksetzbarkeit: Beispiel

Seien s_I, s_{II} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_I) \cup trans(s_{II})$:

$s_I = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_1(z)\ c_1$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7		c_2
8	$w_1(z)$	
9	c_1	

$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

- Dann gilt:
  - t_2 liest y von t_1 in s_I und $c_2 \in s_I$, aber $c_1 \not\prec_{s_I} c_2$. Hieraus folgt $s_I \notin RC$.
  - $s_{II} \in RC$, da die Commit-Operation von t_2 hinter der von t_1 steht

Weiteres Problem:

Schauen wir uns s_{II} nochmal genauer an:

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

- Was passiert wenn t_1 direkt nach $r_2(y)$ abbricht?

Weiteres Problem:

Schauen wir uns s_{II} nochmal genauer an:

$$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

- Was passiert wenn t_1 direkt nach $r_2(y)$ abbricht?
  - Dirty Read-Situation: Abbruch von t_1 sieht Abbruch von t_2 nach sich.
  - **kaskadierender Abort**

Vermeidung kaskadierender Aborts

- Ein Schedule s **vermeidet kaskadierende Aborts**, falls für alle Transaktionen $t_i, t_j \in trans(s)$ mit $i \neq j$ gilt:

$$t_i \text{ liest } x \text{ von } t_j \text{ in } s \Rightarrow c_j <_s r_i(x)$$

- Eine Transaktion darf nur Werte von bereits erfolgreich abgeschlossenen Transaktionen lesen.
- Die **Klasse aller Schedules, welche kaskadierende Aborts vermeiden**, bezeichnen wir mit ACA (avoids cascading aborts).

Vermeidung kaskadierender Aborts: Beispiel

Seien s_{II}, s_{III} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{II}) \cup trans(s_{III})$:

$s_{II} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_1(z)\ c_1\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5		$r_2(y)$
6		$w_2(y)$
7	$w_1(z)$	
8	c_1	
9		c_2

$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

- Dann gilt:
  - $s_{II} \notin ACA$
  - $s_{III} \in ACA$

Schauen wir uns s_{III} nochmal genauer an

Schauen wir uns s_{III} nochmal genauer an.

$$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

- Weiteres Problem tritt auf in s_{III}, falls t_1 direkt nach $w_2(x)$ abstürzt
  - t_1 hat zwar Objekt x geschreiben, t_2 braucht aber nicht abgebrochen zu werden
  - Bei Abort von t_1 muss t_2 analysiert werden um den richtigen Wert von x zu bestimmen.
  - Gibt es Schedules bei denen das nicht nötig ist?

Striktheit

- Ein Schedule s heißt **strikt**, falls für alle Transaktionen $t_i, \in trans(s)$ und für alle Aktionen $p_i(x) \in op(t_i)$ gilt:

$$w_j(x) <_s p_i(x), i \neq j \Rightarrow a_j <_s p_i(x) \vee c_j <_s p_i(x)$$

- Kein Objekt wird gelesen oder überschrieben, bis die Transaktion, welche es zuletzt geschrieben hat, (erfolgreich oder erfolglos) beendet ist.
- Die **Klasse aller strikten Schedules** bezeichnen wir mit ST.

Striktheit: Beispiel

Seien s_{III}, s_{IV} zwei Schedules mit Transaktionen $t_1, t_2 \in trans(s_{III}) \cup trans(s_{IV})$:

$s_{III} = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4		$w_2(x)$
5	$w_1(z)$	
6	c_1	
7		$r_2(y)$
8		$w_2(y)$
9		c_2

$s_{IV} = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$

Schritt	t_1	t_2
1	$w_1(x)$	
2	$w_1(y)$	
3		$r_2(u)$
4	$w_1(z)$	
5	c_1	
6		$w_2(x)$
7		$r_2(y)$
8		$w_2(y)$
9		c_2

- Dann gilt:
  - $s_{III} \notin ST$
  - $s_{IV} \in ST$

Korrektheit von Schedules

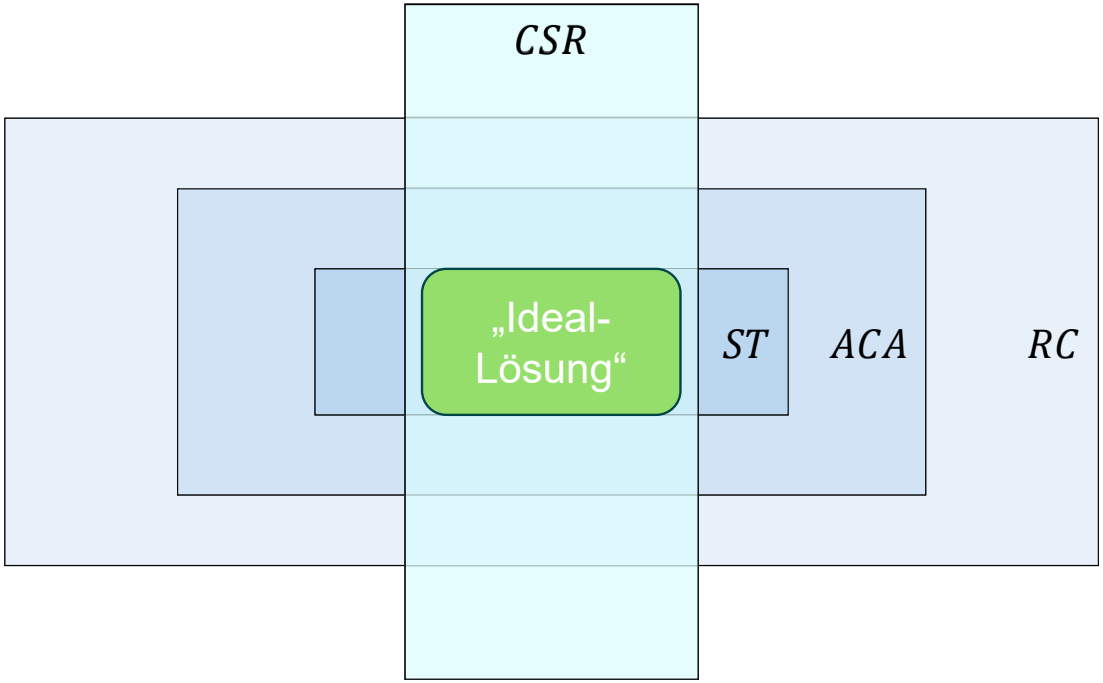
- Fehlersicherheit (ST, ACA, RC) und Konfliktserialisierbarkeit (CSR) sind „orthogonale“ Anforderungen an Schedules

- Ein Schedule s heißt **korrekt**, falls er sowohl konfliktserialisierbar als auch fehlersicher ist, d.h. falls er in der Klasse CSR und in einer der Klassen RC, ACA oder ST liegt.

Zusammenhang zwischen den Klassen RC, ACA, ST, und CSR

- Es gilt der folgende Zusammenhang: $ST \subset ACA \subset RC$

CSR : Konfliktserialisierbarkeit
RC : Recoverable
ACA : Avoids cascading aborts
ST : Strikt



Transaktionsverwaltung

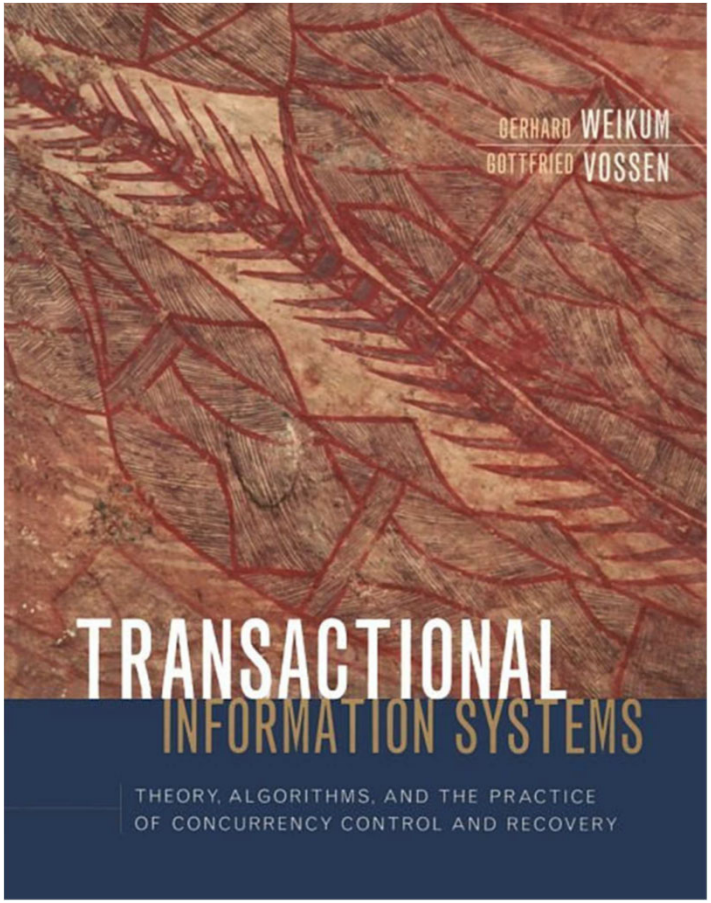
1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. **Protokolle zur Synchronisation (Scheduler)**
4. Fehlertoleranz (Recovery)

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

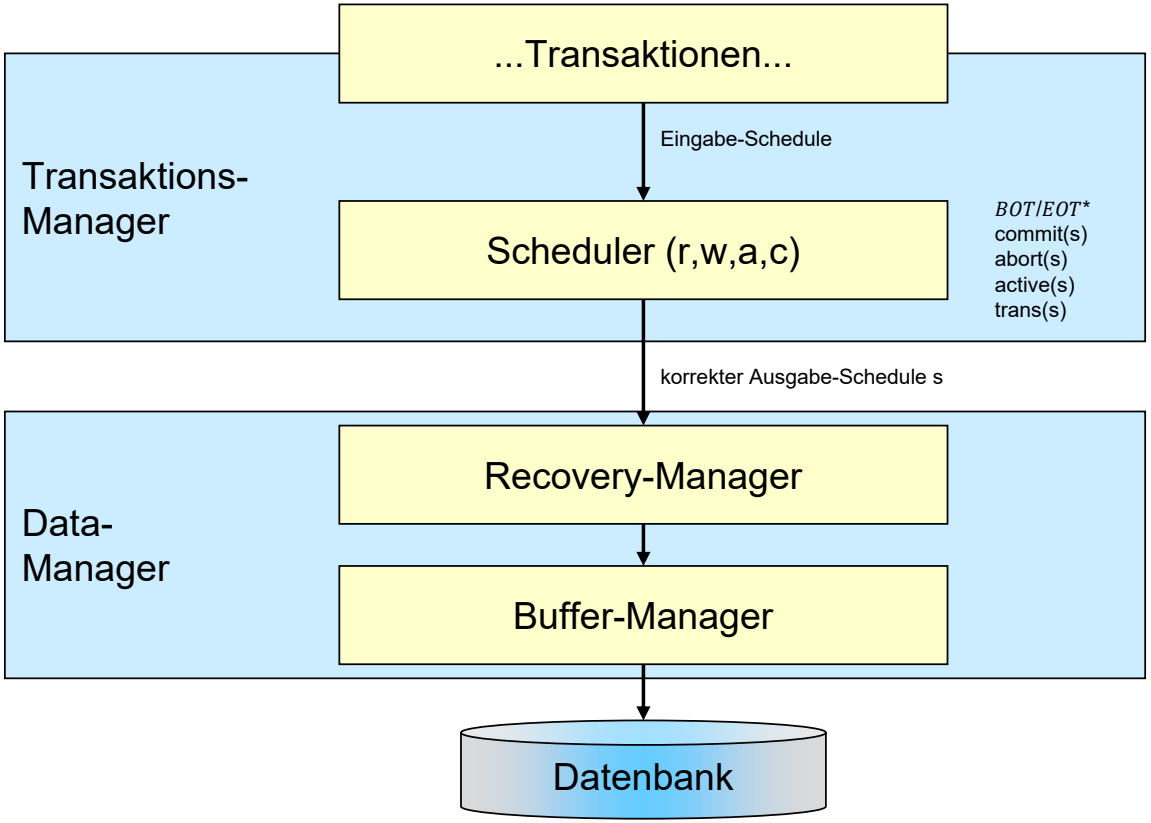
Literatur

Gerhard Weikum and Gottfried Vossen
Transactional Information Systems:
Theory, Algorithms, and
The Practice of Concurrency Control and Recovery

ISBN 1-55860-508-8



Transaktionenverarbeitende Schichten eines DBMS



* BOT – begin of transaction
EOT – end of transaction

Transaktions-Manager

- Der **Transaktions-Manager** nimmt die auszuführenden Transaktionen entgegen und verwaltet
  - die Mengen $trans$, $commit$, $abort$ und $active$
  - für jede Transaktion eine Warteschlange von ausführbereiten Aktionen
- Einzelne Aktionen werden dem **Scheduler** übergeben
- Der Transaktions-Manager umschließt jede Transaktion mit den folgenden Aktionen:
  - BOT (begin of transaction)
    - Kennzeichnet den Beginn einer Transaktion
  - EOT (end of transaction)
    - Kennzeichnet des Ende einer Transaktion
- Wird $EOT(t_i)$ für eine Transaktion t_i erkannt, so wird diese Aktion durch c_i ersetzt.
- Tritt ein Fehler in einer Transaktion t_i auf, so wird dieser durch die Aktion a_i gekennzeichnet.

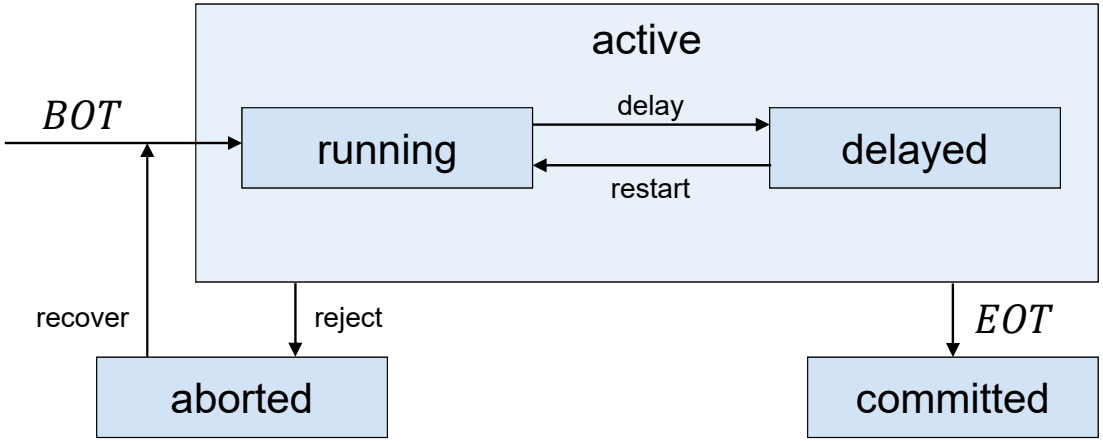
Data-Manager

- Der **Data-Manager** führt die Aktionen des (korrekten) Ausgabe-Schedules der Reihe nach aus:
  - für Aktion r liest er ein Datenobjekt aus der Datenbank in den Puffer
  - für Aktion w schreibt er ein Datenobjekt in die Datenbank oder in den Puffer
  - für Aktion c macht er das Ergebnis der Transaktion „permanent“
  - für Aktion a macht er die Transaktion „ungeschehen“
- Der Data-Manager besteht aus zwei Komponenten:
  - Der **Recovery-Manager** ist dafür verantwortlich, dass in der Datenbank nur die Effekte von freigegebenen und keine Effekte von abgebrochenen Transaktionen erscheinen.
  - Der **Buffer-Manager** stellt die Schnittstelle zur Datenbank dar und verwaltet den Puffer.

Scheduler

- Der **Scheduler** erhält als Eingabe-Schedule Aktionen vom Typ r, w, a, c und muss daraus einen korrekten Ausgabe-Schedule erzeugen
- Dabei kann der Scheduler die Aktion
  a) ausführen (execute)
     - Die Aktion wird sofort ausgegeben, d.h. an den Ausgabe-Schedule angehängt
     - Für alle Aktionen r, w, a, c
  b) zurückweisen (reject)
     - Die Aktion wird nicht ausgeführt, Abbruch der entsprechenden Transaktion t_i mit a_i
     - Nur für Datenoperationen r, w
  c) verzögern (delay)
     - Die Aktion wird weder ausgeführt noch abgelehnt, sondern an den Transaktions-Manager zurückgegeben
     - Nur für Datenoperationen r, w

Zustände einer Transaktion



- Nach dem BOT wird eine Transaktion aktiv
- Die Transaktion kann dann entweder im laufenden oder verzögerten Zustand sein
- Nach dem EOT wird die Transaktion geschrieben, falls sie voher nicht abgebrochen wird

Sicherheit/Ausdrucksstärke eines Schedulers

- Bezeichne $\varepsilon(S)$ das **Erzeugnis** eines Schedulers S, d.h. die Menge aller Schedules, welche S als Ausgabe erzeugen kann.
- Ein Scheduler S ist
  - ***s-sicher***, falls $\varepsilon(S) \subseteq CSR$ gilt
  - ***f-sicher***, falls $\varepsilon(S) \subseteq RC$ (oder ACA oder ST) gilt
  - ***sicher***, falls $\varepsilon(S) \subseteq CSR \cap RC$ (oder ACA oder ST) gilt

- *S-sichere* Scheduler erzeugen konfliktserialisierbare Ausgabe-Schedules.
- *F-sichere* Scheduler erzeugen fehlersichere Ausgabe-Schedules.
- Die *Ausdrucksstärke* eines Schedulers bestimmt sich dadurch, wie gut er die Gesamtmenge sicherer Schedules abdeckt (möglichst viele der konfliktserialisierbaren Schedules bei möglichst effizienter f-Sicherheit).

Sicherheit/Ausdrucksstärke eines Schedulers

- Bezeichne $\varepsilon(S)$ das **Erzeugnis** eines Schedulers S, d.h. die Menge aller Schedules, welche S als Ausgabe erzeugen kann.
- Ein Scheduler S ist
  - ***s-sicher***, falls $\varepsilon(S) \subseteq CSR$ gilt
  - ***f-sicher***, falls $\varepsilon(S) \subseteq RC$ (oder ACA oder ST) gilt
  - ***sicher***, falls $\varepsilon(S) \subseteq CSR \cap RC$ (oder ACA oder ST) gilt

- *S-sichere* Scheduler erzeugen konfliktserialisierbare Ausgabe-Schedules.
- *F-sichere* Scheduler erzeugen fehlersichere Ausgabe-Schedules.
- Die *Ausdrucksstärke* eines Schedulers bestimmt sich dadurch, wie gut er die Gesamtmenge sicherer Schedules abdeckt (möglichst viele der konfliktserialisierbaren Schedules bei möglichst effizienter f-Sicherheit).

Wie kann ein Scheduler konfliktserialisierbare Schedules garantieren?

- Konfliktgraph ist Methode wie ein Scheduler Transaktionen auf Serialisierbarkeit testen kann
- Was kann ein Scheduler machen wenn Transaktionen nicht serialisierbar sind?

Sperrende und Nicht-sperrende Scheduler

- **Sperrende Scheduler**
  - Pessimistische Ablaufsteuerung durch Sperrverfahren, Locking
  - Lese- und Schreibsperren verhindern, dass Änderungen nebenläufige Transaktionen beeinflussen
  - *Nachteil*: Schreibende und nur-lesende Transaktionen müssen ggf. warten, bis andere Transaktionen abgeschlossen sind.
  - *Vorteil*: In der Regel nur wenige Rücksetzungen aufgrund von Synchronisationsproblemen nötig.
  - ➢ Standardverfahren in kommerziellen DBMS
- **Nicht-sperrende Scheduler**
  - Optimistische Ablaufsteuerung durch Zeitstempelverfahren
  - Transaktionen dürfen bis zum COMMIT ungehindert arbeiten.
  - Bei COMMIT wird geprüft, ob ein Konflikt aufgetreten ist (Validierung). Die Transaktion wird ggf. zurückgesetzt.
  - Die Validierung wird anhand von Zeitstempeln durchgeführt (“Gab es seit Beginn der Transaktion ein Commit einer anderen Transaktion, das dieselben Daten betrifft?”).
  - ➢ Nur geeignet, falls Konflikte zwischen Schreibern sehr selten auftreten.

Sperren (Locking)

- **Sperren** (Locks) dienen zur Synchronisation von Zugriffen auf gemeinsam genutzte Datenobjekte
- Diese Sperren werden vom Scheduler für Transaktionen gesetzt und freigegeben
- Gesetzte Sperre signalisiert Unverfügbarkeit des Datenobjektes
  - Vor Zugriff wird Verfügbarkeit geprüft und Sperre gesetzt.
  - Datenobjekt ist für spezifische Transaktion gesperrt
  - Nach Zugriff wird Sperre aufgehoben.
- Für jedes Datenobjekt x gibt es zwei Arten von Sperren:
  - Lese-Sperre: $rl(x)$ (read lock) und $ru(x)$ (read unlock)
  - Schreib-Sperre: $wl(x)$ (write lock) und $wu(x)$ (write unlock)

- Sperren in Konflikt bedeutet gleichzeitig Operationen in Konflikt
- Ein sperrender Scheduler fügt jeder Transaktion *rl/wl/ru/wu*-Aktionen hinzu

Drei Regeln zur Anwendung von Sperren

- Für jede Transaktion t_i, die vollständig in einem Schedule s enthalten ist, gilt:
  - **(L1) Jede Lese-/Schreiboperation wird in der richtigen Reihenfolge ge- und entsperrt:**
    - Falls t_i eine Aktion $r_i(x)$ bzw. $w_i(x)$ enthält, so steht irgendwo davor $rl_i(x)$ bzw. $wl_i(x)$ und irgendwo dahinter $ru_i(x)$ bzw. $wu_i(x)$ in s
  - **(L2) Jedes von der Transaktion verwendete Datenobjekt wird gesperrt:**
    - Für jedes x das von t_i verwendet wird, existiert genau ein $rl_i(x)$ bzw. $wl_i(x)$ in s
  - **(L3) Kein Entsperren ist überflüssig:**
    - Kein ru_i/wu_i ist redundant

- Die von einem sperrenden Scheduler produzierten Schedules enthalten also neben den Daten- sowie Terminierungsoperationen der betreffenden Transaktionen auch Sperr- sowie Entsperroperationen.
- Ist s ein Schedule, so bezeichnet $DT(s)$ die **Projektion** von s auf sämtliche Aktionen des Typs r, w, a, c.

- Die Regeln sind für das einzuführende Sperrprotokoll relevant!

Sperren: Beispiel

- Seien $t_1 = r_1(x)\ w_1(y)$ und $t_2 = w_2(x)\ w_2(y)$ zwei Transaktionen und s_I, s_{II} die folgenden zwei Schedules:

s_I

#	t_1	t_2
1	$rl_1(x)$	
2	$r_1(x)$	
3	$ru_1(x)$	
4		$wl_2(x)$
5		$w_2(x)$
6		$wl_2(y)$
7		$w_2(y)$
8		$wu_2(x)$
9		$wu_2(y)$
10		c_2
11	$wl_1(y)$	
12	$w_1(y)$	
13	$wu_1(y)$	
14	c_1	

Dann gilt:

Beide Schedules erfüllen die Regeln (L1)-(L3)

$DT(s_I) = r_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1 \notin CSR$

$DT(s_{II}) = r_1(x)\ w_1(y)\ c_1\ w_2(x)\ w_2(y)\ c_2 = t_1\ t_2$

s_{II}

#	t_1	t_2
1	$rl_1(x)$	
2	$r_1(x)$	
3	$wl_1(y)$	
4	$w_1(y)$	
5	$ru_1(x)$	
6	$wu_1(y)$	
7	c_1	
8		$wl_2(x)$
9		$w_2(x)$
10		$wl_2(y)$
11		$w_2(y)$
12		$wu_2(x)$
13		$wu_2(y)$
14		c_2

Sperrprotokoll

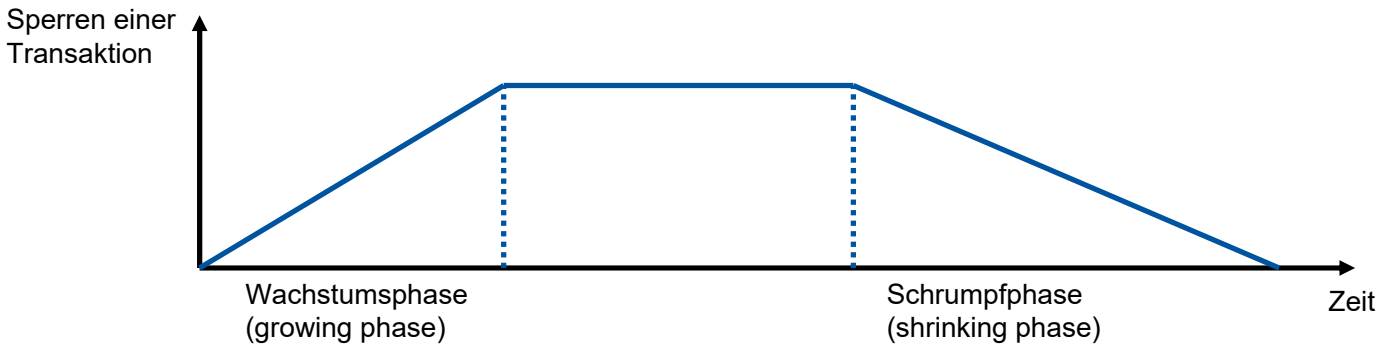
- Ein Scheduler arbeitet nach einem **Sperrprotokoll**, falls für jeden Ausgabe-Schedule s und jede Transaktion $t_i \in trans(s)$ gilt:
  - t_i erfüllt die Regeln (L1)-(L3), sowie
  - **(L4)** ist x durch t_i und t_j gesperrt, für $t_i, t_j \in trans(s)$ und $i \neq j$, so sind diese Sperren nicht in Konflikt, d.h. sie sind kompatibel gemäß der nachfolgend dargestellten Tabelle (d.h., mehrere Lesesperren sind zulässig):

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	-
$wl_j(x)$	-	-

  - ist $i = j$, so ist das Setzen von Schreib-/Lesesperren bei schon bestehenden Lese-/Schreibsperren derselben Transaktion t_i/t_j zulässig.

2-Phasen-Sperrprotokoll (*2PL*)

- Ein Sperrprotokoll ist **zweiphasig**, falls für jeden erzeugten Schedule s und jede Transaktion $t_i \in trans(s)$ gilt:
  - Nach der ersten ru_i/wu_i-Aktion folgt keine weitere rl_i/wl_i-Aktion
- In der ersten Phase einer Transaktion werden Sperren ausschließlich gesetzt, in der zweiten Phase werden Sperren ausschließlich freigegeben:



- Ein entsprechender Scheduler heißt $2PL$-Scheduler (2-Phase-Locking)

Zusammenfassung der Regeln für einen $2PL$-Scheduler:

- **(L1) Jede Lese-/Schreiboperation wird in der richtigen Reihenfolge ge- und entsperrt.**
- **(L2) Jedes von der Transaktion verwendete Datenobjekt wird gesperrt.**
- **(L3) Kein Entsperren ist überflüssig.**
- **(L4) Ist x in zwei Transaktionen gesperrt, sind so sind diese Sperren kompatibel.**
- **(2PL) In der ersten Phase einer Transaktion werden Sperren ausschließlich gesetzt, in der zweiten Phase werden Sperren ausschließlich freigegeben**

Korrektheitsbeweis: Prinzip

- Zu zeigen: $\varepsilon(2PL) \subset CSR$ (die vom Scheduler erzeugten Schedules sind konfliktserialisierbar)
- Beweis in zwei Schritten:

1. Formalisierung der Eigenschaften von Schedules $s \in \varepsilon(2PL)$
2. Zeigen, dass diese Eigenschaften Konfliktserialisierung zur Folge haben

Lemma 1: Eigenschaften von Schedules $s \in \varepsilon(2PL)$

Sei ein Schedule $s \in \varepsilon(2PL)$.
Dann gilt für jede Transaktion $t_i \in DT(s)$:

1. **Jede Lese-/Schreiboperation wird in korrekter Reihenfolge ge- und entsperrt:** Falls eine Aktion $p_i(x)$ in $CP\big(DT(s)\big)$ vorkommt, dann kommen auch $pl_i(x)$ und $pu_i(x)$ darin vor mit der folgenden Reihenfolge: $pl_i(x) < p_i(x) < pu_i(x)$.

2. **Sind zwei Aktionen in Konflikt, so sind die Sperren nicht gleichzeitig gesetzt:** Falls $t_j \in commit\big(DT(s)\big)$ mit $i \neq j$ und die Aktion $p_i(x)$ und $q_j(x)$ in $CP\big(DT(s)\big)$ in Konflikt, so gilt entweder $pu_i(x) <_s ql_j(x)$ oder $qu_j(x) <_s pl_i(x)$

3. **Alle Sperren werden zuerst gesetzt (1. Phase) und dann aufgehoben (2. Phase)**: Sind die Aktionen $p_i(x)$ und $q_i(y)$ in $CP\big(DT(s)\big)$ so gilt: $pl_i(x) <_s pu_i(y)$

4. **L2 und L3 gelten**

Beweis: Eigenschaften folgen aus den Definitionen: 1. folgt aus L1, 2. aus L4 3. folgt aus (2PL). 4. ist direkt die Definition

Korrektheitsbeweis: Schritt 1, Teil 2

Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

Sei ein Schedule $s \in \varepsilon(2PL)$ und G der Konfliktgraph von $CP(DT(s))$. Dann gilt:

1. Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.
2. Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.
3. G ist azyklisch.

Korrektheitsbeweis: Schritt 1, Teil 2

Beweis Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

1. **Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.**
2. Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.
3. G ist azyklisch.

Beweis:

1. Da (t_i, t_j) eine Kante in G, ist, gibt es eine Konfliktbeziehung $(p_i(x), q_i(x))$. Nach Lemma 1.1 gibt es Lock- und Unlock Operationen in $CP(s)$ mit $pl_i(x) <_s p_i(x) <_s pu_i(x)$.
2. Nach Lemma 1.2 (nicht Gleichzeitigkeit) folgt aus der Konfliktbeziehung $(p_i(x), q_j(x))$ entweder $pu_i(x) <_s ql_j(x)$ oder $qu_j(x) <_s pl_i(x)$.
   Aus $qu_j(x) <_s pl_i(x)$ folgt aber $q_j(x) <_s p_i(x)$, was im Widerspruch zu Lemma 1.1 und Definition Konfliktmenge steht.
   Also folgt $pu_i(x) <_s ql_j(x)$ und damit auch $p_i(x) <_s q_j(x)$ (Lemma 1.1)

Korrektheitsbeweis: Schritt 1, Teil 2

Beweis Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

1. Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.
2. **Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.**
3. G ist azyklisch.

Beweis durch Induktion über $k < n$:

1. $k = 2$: Siehe Beweis Lemma 2.1
2. $k > 2$: Wir nehmen an die Behauptung ist wahr für k mit $k < n$.
   Dann gibt es Aktionen und Datenobjekte $p_1(x), o_k(z) \in CP(DT(s))$ mit $pu_1(x) <_{CP(s)} ol_k(z)$ (B1).
   (t_k, t_{k+1}) ist eine Kante in G. Nach Lemma 2.1 gibt Aktionen und Datenobjekte $o'_k(x), q_{k+1}(y) \in CP(DT(s))$ mit $o'_k u(x) <_{CP(s)} ql_{k+1}(y)$ (B2)
   Nach Lemma 1.3 (2PL) gilt $ol_k(z) <_{CP(s)} o'u_k(y)$ (B3).
   Aus den drei Ungleichungen (B1), (B3) und (B2) folgt mittels Transitivität von "$<_{CP(s)}$": $\mathrm{pu}_1(\mathrm{x}) <_\mathrm{s} \mathrm{ql}_{\mathrm{k}+1}(\mathrm{y})$.
   qed.

Korrektheitsbeweis: Schritt 1, Teil 2

Beweis Lemma 2: Konfliktgraph von $s \in \varepsilon(2PL)$

1. Ist (t_i, t_j) eine Kante in G, so gibt es ein Datenobjekt x und zwei Aktionen $p_i(x)$, $q_j(x)$ mit $pu_i(x) <_s ql_j(x)$.
2. Ist $(t_1, t_2, \ldots, t_n)$ ein Pfad in G, so gilt $pu_1(x) <_s ql_n(y)$ für zwei Objekte x, y und Aktionen $p_1(x), q_n(y)$.
3. **G ist azyklisch.**

Beweis durch Widerspruch:

Angenommen G enthält einen Zyklus: $(t_1, t_2, \ldots, t_n, t_1)$ mit $n > 1$.
Dann gibt es zwei Aktionen $p_1(x)$, $q_1(x)$ mit $pu_1(x) <_s ql_1(x)$ (mit Lemma 2.2)
Das steht im Widerspruch zu Lemma 1.3 (2PL Regel)

Korrektheitsbeweis: Schritt 2,

Theorem 1: Sicherheit: $\boldsymbol{\varepsilon(2PL) \subset CSR}$

$\subseteq$ folgt direkt aus Lemma 2.3 ($\boldsymbol{G}$ ist azyklisch.)

Echte Teilmenge folgt aus folgenden Schedule:

$$s = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1$$

$s \in CSR$ (da $s \approx_s t_3t_1t_2$) aber $s \notin \epsilon(2PL)$ da aus

- $wu_1(x) <_s rl_2(x)$
- $ru_3(y) <_s wl_1(y)$
- $rl_2(x) <_s r_2(x)$
- $r_3(y) <_s ru_3(y)$
- $r_2(x) <_s r_3(y)$

folgt $wu_1(x) <_s wl_1(y)$, was der zwei-Phasen Eigenschaft widerspricht.

Konservatives/Statisches 2-Phasen-Sperrprotokoll (*C2PL*)

- **Wichtig:** Das C2PL Sperrprotokoll erweitert das 2PL Sperrprotokoll.
  - C2PL folgt allen Regeln des 2PL, und erweitert dieses um die Regeln auf dieser Folie.
- Alle Sperren einer Transaktion werden gesetzt, bevor die erste r/w-Aktion ausgeführt wird:

- Eigenschaften
  - Vorteil: Deadlocks vermieden, weil Transaktion wartet, bis alle Locks verfügbar sind - aber Risiko des „Verhungerns“, wenn viele Sperren gebraucht werden.
  - Nachteil: Alle r/w-Aktionen müssen im Vorhinein (BOT) bekannt sein - potentiell zu viele Sperren!

Strenge/Dynamische 2-Phasen-Sperrprotokoll (*S2PL*)

- **Wichtig:** Das S2PL Sperrprotokoll erweitert das 2PL Sperrprotokoll.
  - Das S2PL folgt allen Regeln des 2PL, und erweitert dieses um die Regeln auf dieser Folie.
- Alle Sperren einer Transaktion werden *sofort* nach der letzten Aktion aufgehoben:

- Es ist ausreichend, *Schreibsperren* bis nach der letzten $\mathrm{r/w}$-Operation zu halten.
- Ein $S2PL$-Scheduler ist *sicher*, d.h. es gilt sogar: $\boldsymbol{\varepsilon(S2PL) \subseteq CSR \cap ST}$.
- S2PL garantiert also Lösungen im „Idealbereich“ mit einem einfachen, effizienten und jederzeit anwendbaren Verfahren, das auch gut mit Fehlertoleranz (Recovery) kombinierbar ist, diese aber auch braucht (S2PL ist nicht Deadlock-frei).
- Werden **alle** Sperren bis nach der letzten r/w-Operation gehalten, spricht man von einem starken S2PL-Scheduler (SS2PL).

Ausblick: Verbesserungsmöglichkeiten für x2PL-Sperrprotokolle

- Falls die von x*2PL* gesperrten Objekte „groß“ sind, sind nur wenige Sperren zu verwalten, aber Konflikte zwischen Sperren sind häufiger.
- Falls die von x*2PL* gesperrten Objekte „klein“ sind, ist mehr Nebenläufigkeit möglich, aber die Verwaltungskosten für Sperren sind auch höher.
- Verallgemeinerung von x*2PL* zu hierarchischen Sperrobjekten:
  - Multiple Granularity Locking (MGL)
  - Tree Locking (TL)

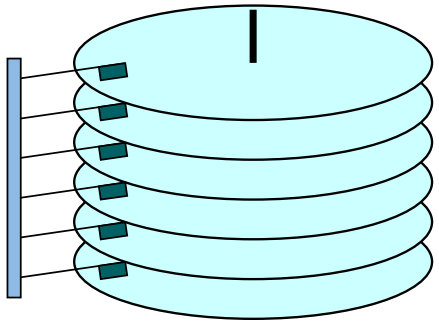
Transaktionsverwaltung

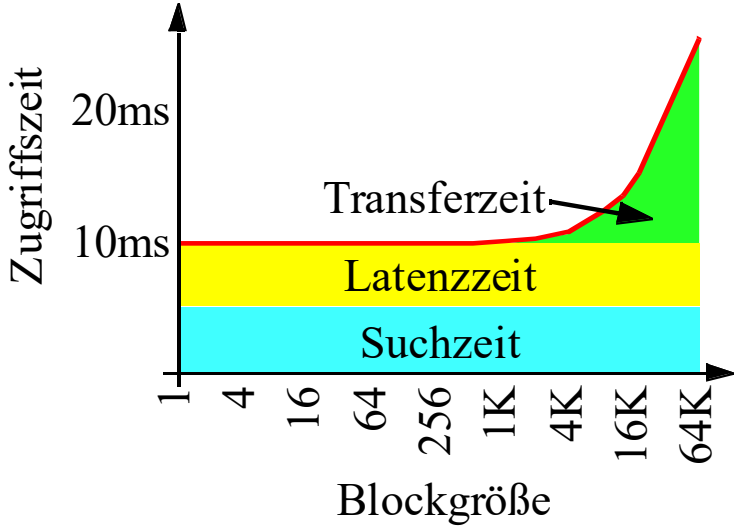
1. Das Transaktionskonzept
2. Synchronisation (Concurrency Control)
3. Protokolle zur Synchronisation (Scheduler)
4. **Fehlertoleranz (Recovery)**

Informatik 5 – Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Technischer Kontext Speicherhierarchie

- Hauptspeicherzugriff (< 50 ns)
- Zugriffszeit bei Festplatten
  - Armpositionierung: Suchzeit (□ 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (□ 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)
- Blockorientierter Zugriff
  - Größere Transfereinheiten (Blöcke, Seiten) sind günstiger als einzelne Bytes
  - Gebräuchliche Seitengrößen: 2kB oder 4kB
  - Kompatibel mit Paging-Mechanismus des Betriebssystems
  - Sequentielle Blockerarbeitung minimiert Armbewegung





Klassifikation von Fehlern

Ein DBMS soll die dauerhafte und konsistente Verfügbarkeit des Datenbestandes (Atomicity und Durability) sicherzustellen. Ein wichtiger Aspekt ist die Toleranz gegenüber Fehlern, die im laufenden Betrieb und auf den Datenträgern auftreten können.

- **Medienfehler**
  Verlust von permanenten Daten durch Zerstörung des Speichermediums, z.B. Plattencrash, Brand, Wasserschaden, etc.
- **Transaktionsfehler**
  Lokaler Fehler einer noch nicht abgeschlossenen Transaktion, z.B. aufgrund
  - Fehler und Abbruch des Anwendungsprogrammes (z.B. Division durch Null, …)
  - Verletzung von Integritätsbedingungen oder Zugriffsrechten
  - expliziter Abbruch der Transaktion (**rollback**) durch den Benutzer
  - Konflikte mit nebenläufigen Transaktionen
- **Systemfehler**
  Verlust von Hauptspeicherinformation z.B. wegen Stromausfall, Ausfall der CPU, Absturz des Betriebssystems, etc. Die permanenten Speicher (Platten) sind nicht betroffen.

Technische Grundlagen

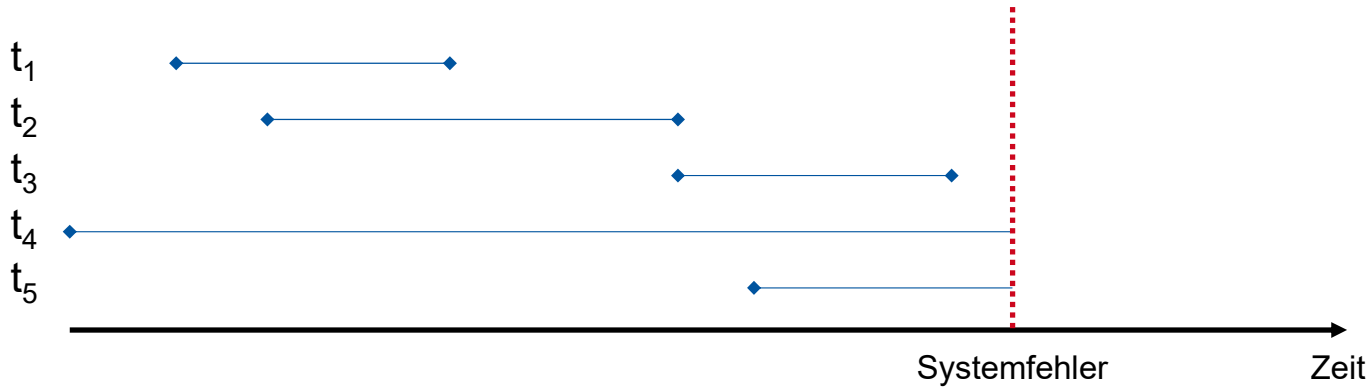
Es gibt unterschiedliche technische Maßnahmen zur Wiederherstellung:

- Für Medienfehler: Duplizierung (d.h. gezielte Redundanz) des Datenbankzustandes,
  - Bänder (oder andere Backup-Speicher): Kaltstart auf altem Datenbankzustand
  - Spiegelplatten erlauben Warmstart (Wiederanlauf bei laufendem Betrieb)
  - verteilte Rechenzentren an anderem Ort erhöht Unabhängigkeit (z.B. Erdbeben in Kalifornien, Katasterdaten Palästina)

- Bei Transaktions- und Systemfehlern
  - Mitprotokollierung der laufenden Schedules incl. BOT, COMMIT, ABORT) von Transaktionen in Logfiles
  - Bei allen Schreiboperationen müssen sowohl der alte als auch der neue Wert geloggt werden
  - Lese-/ Schreibobjekte sind meist physische Speicherblöcke (Paging)

- Logfiles
  - Logfiles entstehen sequentiell
  - eigenes Sekundärspeichermedium ermöglichen schnelles Lesen und Schreiben

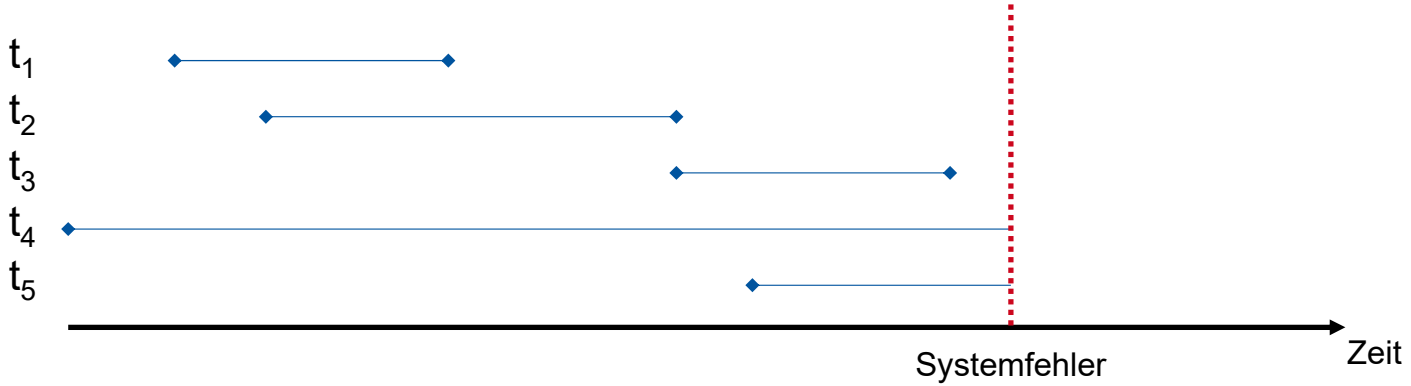
Szenario nach einem Fehler



Transaktionen werden in zwei Klassen eingestuft:

- Transaktionen, die vor dem Fehler bereits COMMITed waren. (t_1, t_2, t_3)
  Durability erfordert ein **REDO**, falls ihre Ergebnisse nicht im stabilen Speicher sind.
- Transaktionen, die zum Zeitpunkt des Fehlers noch aktiv waren (t_4, t_5).
  Atomarität erfordert ein **UNDO**, falls einige Ergebnisse bereits im stabilen Speicher sind.

Grundprinzipien eines Warmstart Recovery-Algorithmus (ARIES, C. Mohan, IBM)



- Ein korrekter Recovery-Algorithmus muss berücksichtigen, dass
  - auch für Logfiles das Zusammenspiel zwischen Hauptspeicher und stabilem Speicher gilt
  - jederzeit, sogar *während* der Recovery erneut Fehler auftreten können.
- **Write-Ahead Logging** bei Schreiboperationen:
  - Der alte Wert muss vor einer Schreiboperation geloggt werden (sicheres UNDO).
  - Der neue Wert muss ins stabile Log, bevor er in den stabilen Datenspeicher geschrieben wird, und spätestens vor dem COMMIT (sichere Durability).
  - Nach Absturz geht das **REDO** der abgeschlossenen Transaktionen, deren Ergebnisse noch nicht im stabilen Speicher stehen, dem **UNDO** der laufenden Transaktionen voraus.

Zusammenfassung Transaktionsmanagement

- Transaktionen sind Operationsfolgen auf Datenbanken
- Erfüllt ACID-Prinzips (trotz Mehrbenutzerbetrieb und Fehler)
- Korrekte Synchronisation und Fehlersicherheit werden formal mit Read-Write-Modells über zwei Korrektheitskriterien und Scheduler definiert:
  - Konfliktserialisierbarkeit aller aktiven und abgeschlossenen Transaktionen (Test: Zyklentest in Konfliktgraphen, Algorithmus z.B.: 2PL)
    - verhindert Lost Updates und Phantom-Problem
    - Fehlersicherheitskriterien wie Rücksetzbarkeit, Vermeidung kaskadierender Aborts und Striktheit (verhindern Dirty Read mit unterschiedlichen Trade-Offs zwischen Recoveryeffizienz und Einschränkungen der Verzahnung (z.B. S2PL)).

- Fehlertoleranz im laufenden Betrieb:
  - durch Redundanz in Form von Backups/Spiegelung (Medienfehler) und Operationslogs.
  - Mit **Write-Ahead**-Logging kombiniert mit **REDO/UNDO-Recovery** erlaubt Wiederanlauf auch wenn während des Recovery weitere Abstürze stattfinden.

Ausblick „Implementierung von Datenbanken“

Lange Transaktionen (auch **Workflows** genannt) erstrecken sich über Tage, Wochen oder Monate. Sie finden sich in vielen komplexen Designanwendungen. Dabei sollen auch „inkonsistente“ Zwischenzustände dem Recovery unterliegen:

- Definition eines *Sicherungspunktes*, auf den sich eine (noch aktive) Transaktion zurücksetzen lässt. Die Änderungen dürfen allerdings noch nicht endgültig festgeschrieben werden, da die Transaktion bis hin zur Bestätigung des COMMIT noch scheitern kann.
  SQL: **savepoint** <identifier>
- *Zurücksetzen* der aktiven Transaktion auf einen definierten Sicherungspunkt.
  SQL: **rollback to savepoint** <identifier> oder nur **rollback to** <identifier>

Weitere aktuelle Fragestellungen im heutigen Hochleistungsdatenmanagement:

- Vom ACID-Prinzip zum *CAP-Theorem für verteilte Datenbanken.*
- *Hauptspeicherdatenbanken* (SAP-HANA) verändern die Speicherhierarchie.
- *NoSQL*-Datenbanken unterstützen nicht-relationale Datenformate.
- *Blockchain*-Datenbanken betonen Nachvollziehbarkeit, *Data Spaces* Datensouveränität.