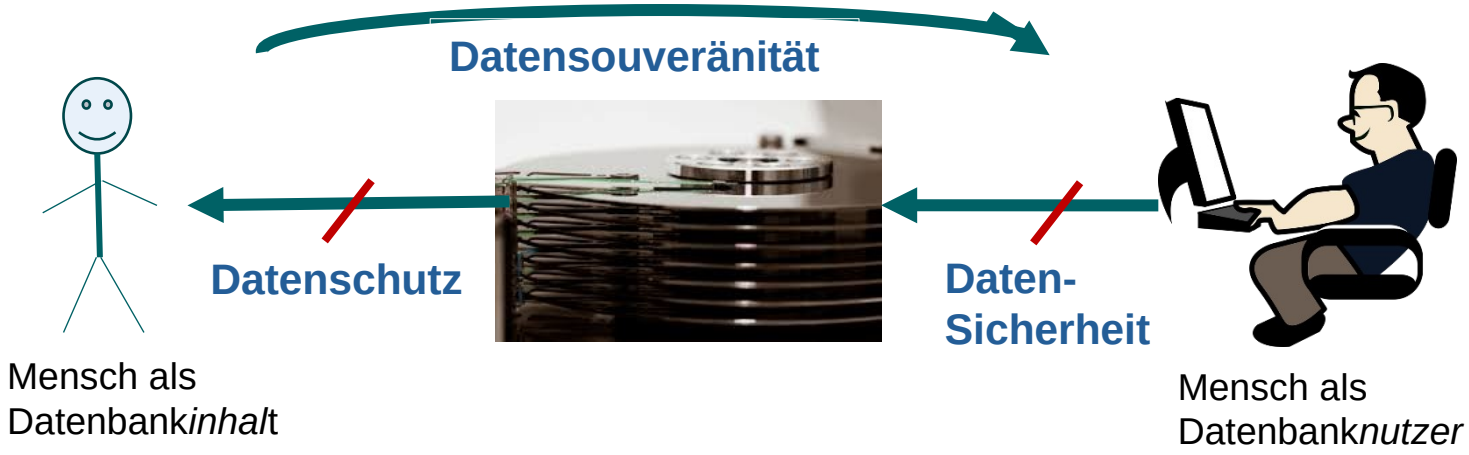
Transaktionsverwaltung

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

Transaktionsverwaltung

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

Begriffsabgrenzung



- **Datensicherheit (Data Security):** (technischer) Schutz *der Daten* vor Zugriffen und Manipulation durch unberechtigte Benutzer
- **Datenschutz (Data Privacy):** (gesetzlicher) Schutz *vor den Daten* einer Person gegen den Missbrauch ihrer Daten (z.B. europäische Datenschutz-Grundverordnung (DSGVO, englisch GDPR siehe https://de.wikipedia.org/wiki/Datenschutz-Grundverordnung)
- **Datensouveränität:** die Fähigkeit einer Person, einer Organisation oder eines Staats, die *Nutzung „ihrer Daten“* durch konkrete Benutzer(gruppen) selbst zu bestimmen und zu kontrollieren

DBS-spezifische Maßnahmen zur Datensicherheit

- Sichten (Views) als Schutzmechanismus (insbesondere “read-only” Sicht)
  - Es ergeben sich Probleme, wenn auch Änderungen erlaubt sein sollen
- Verwaltung von Rechten entsprechend einem Sicherheitsmodell
  - **Discretionary Access Control (Vorbild: Selbstbestimmung)**
  - **Mandatory Access Control (Vorbild: Staatsbürokratie)**

Das Sicherheitsmodell DAC (Privatbereich, Industrial Data Spaces)

- Für das **Discretionary Access Control** (DAC) werden die folgenden Informationen verwaltet:
  - Wer hat das Recht? (**Sicherheitssubjekte**)
    - Benutzeridentifikator für einzelne Benutzer oder Menge von Benutzern
  - Auf welche Teile der Datenbank bezieht sich das Recht? (**Sicherheitsobjekte**)
    - beschrieben durch Anfragesprache oder Datendefinitionssprache
  - Worin besteht das Recht bzw. die erlaubten Operationen?
    - Zugriffsrechte für Lesen und für Ändern (Einfügen, Entfernen, Modifizieren)
  - Darf das Recht weitergegeben werden?

- Grundprinzipien:
  - Jeder Benutzer hat Rechte bzgl. der von ihm erzeugten Objekte
  - Datenbankadministrator (DBA) hat alle Rechte.

Das Sicherheitsmodell MAC (staatliche Organisationen)

- Beim Modell **Mandatory Access Control** (MAC) werden Subjekte und Objekte mit einer Sicherheitseinstufung markiert (z.B. ‘streng geheim’, ‘geheim’, ‘vertraulich’, ‘unklassifiziert’):
  - **Vertrauenswürdigkeit** (‘Clearance’) eines Subjektes *s*: *clearance*(*s*)
  - **Sensitivität** (‘Classification’) eines Objektes o: *classification*(*o*)

- Für die Zugriffskontrolle werden die folgenden Regeln verwendet:
  - Ein Subjekt *s* darf ein Objekt *o* nur lesen, wenn das Objekt eine geringere oder gleiche Sicherheitseinstufung besitzt: *classification*(*o*) *clearance*(*s*)
  - Beim Ändern muss ein Objekt *o* mindestens die Einstufung des schreibenden Subjektes *s* erhalten: *clearance*(*s*) *classification*(*o*)

- Die zweite Regel kontrolliert den Informationsfluss, um den Missbrauch durch autorisierte Benutzer zu verhindern. Eine ‘streng geheime’ Information könnte sonst durch einen berechtigten Benutzer öffentlich zugänglich gemacht werden (*Write-Down*).

Bewertung von Sicherheitsmodellen

- Discretionary Access Control:
  - Einfaches und sehr gebräuchliches Modell
  - Erzeuger von Daten sind mit der Verantwortung für deren Sicherheit belastet
  - Abhängig von der Granularität der Objekte (Datenbank, Relationen, Tupel) kann die Verwaltung der Rechte sehr aufwändig werden

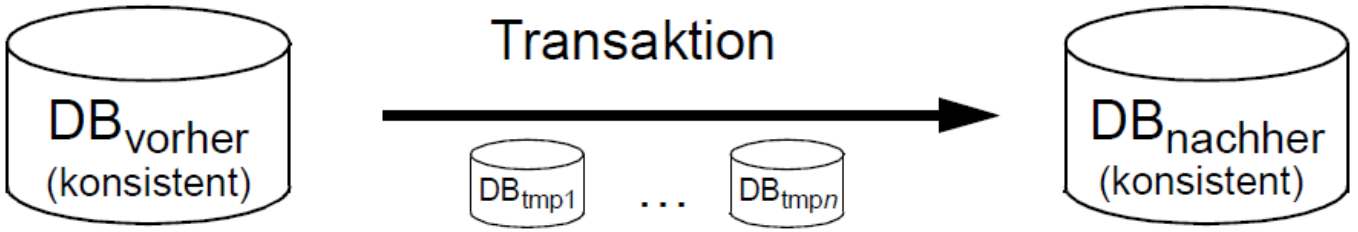
- Mandatory Access Control:
  - Potentiell größere Sicherheit durch abgestufte Sicherheitsklassen
  - Jedes Objekt wird einzeln eingestuft hoher Verwaltungsaufwand in der Datenbank
  - Benutzer mit unterschiedlicher Einstufung können nur schwer zusammenarbeiten, da von höher eingestuften Benutzern veränderte Daten von niedriger eingestuften Kollegen nicht mehr lesbar sind

Transaktionsverwaltung

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

Transaktionen

Transaktionen (TAs) sind die Einheiten *integritätserhaltender Zustandsänderungen* einer Datenbank:



- Eine Transaktion ist „eine Folge von Operationen […], welche eine gegebene Datenbank in ununterbrechbarer Weise von einem konsistenten Zustand in einen [anderen] (nicht notwendig verschiedenen) konsistenten Zustand überführt.“
  [Vossen, 2008; S. 638]

Transaktionen: Beispiel

Beispiel Bankwesen: Überweisung von Huber an Meier in Höhe von 200 EUR

- Möglicher Bearbeitungsplan:
  1. Erniedrige Stand von “Huber” um 200
  2. Erhöhe Stand von “Meier” um 200

- Möglicher Ablauf:

Konto	
Kunde	Stand
Meier	1.000,00
Huber	1.500,00

1.

Konto	
Kunde	Stand
Meier	1.000,00
Huber	1.300,00

Systemabsturz

2.

Wichtig: Inkonsistenter Datenbankzustand darf nicht entstehen bzw. nicht dauerhaft bestehen bleiben

ACID-Prinzip korrekter Transaktionsabwicklung

Eine grundlegende Charakterisierung von Transaktionsanforderungen ist durch das **ACID**-Prinzip (Härder/Reuter 1983) gegeben:

- **Atomicity** (Atomarität, “alles-oder-nichts”-Prinzip)
  Der Effekt einer Transaktion kommt entweder ganz oder gar nicht zu tragen
- **Consistency** (Konsistenz, Integritätserhaltung)
  Durch eine Transaktion wird ein konsistenter Datenbankzustand wieder in einen konsistenten Datenbankzustand überführt
- **Isolation** (Isoliertheit, logischer Einbenutzerbetrieb)
  Innerhalb einer Transaktion nimmt ein Benutzer Änderungen durch andere Benutzer nicht wahr
- **Durability** (Dauerhaftigkeit, Persistenz)
  Der Effekt einer abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten

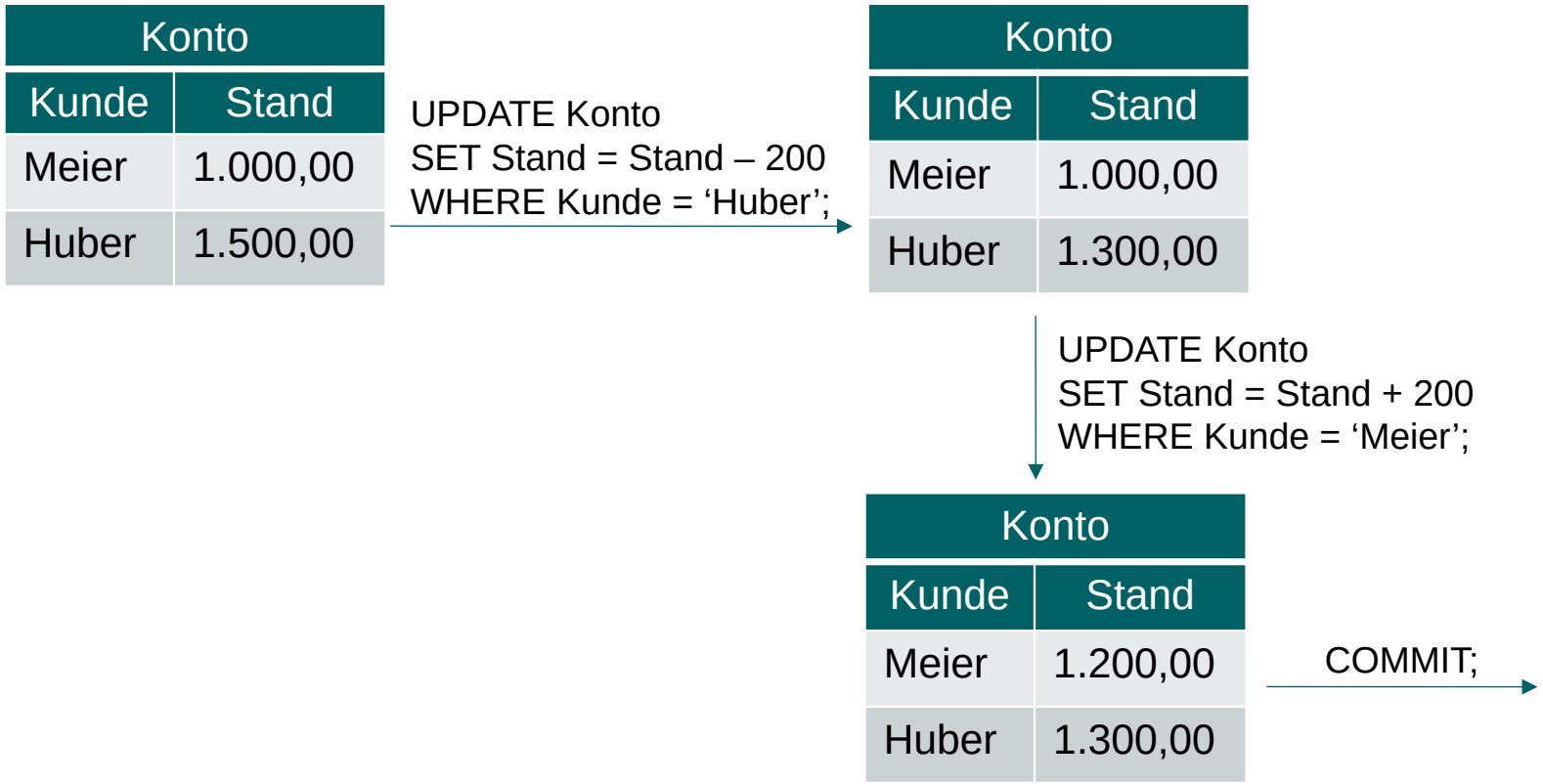
Jede Transaktion muss in endlicher Zeit ausgeführt werden, so dass die ACID Eigenschaften erhalten bleiben

Steuerung von Transaktionen

- **begin of transaction**
  - Beginn der Befehlsfolge einer neuen Transaktion
  - SQL: Transaktionen werden stets implizit begonnen, es gibt kein **begin work** o.ä.
- **end of transaction, commit**
  - Bestätigung der Transaktion durch den Benutzer
    - Die Änderungen seit Beginn der Transaktion werden endgültig bestätigt
    - Der jetzt erreichte Zustand soll dauerhaft gespeichert werden
    - Der Zustand wird durch den Benutzer für konsistent erklärt
  - SQL: **commit work** oder nur **commit**
- **abort transaction**
  - Abbruch der Transaktion durch das Programm bzw. den Benutzer (Rücksetzen, ABORT)
    - Die Änderungen der Transaktion werden zurückgesetzt
    - Der ursprüngliche Zustand vor der Transaktion wird wiederhergestellt
  - SQL: **rollback work** oder nur **rollback**

Steuerung von Transaktionen: Beispiel

Beispiel Bankwesen: Überweisung von Huber an Meier in Höhe von 200 EUR

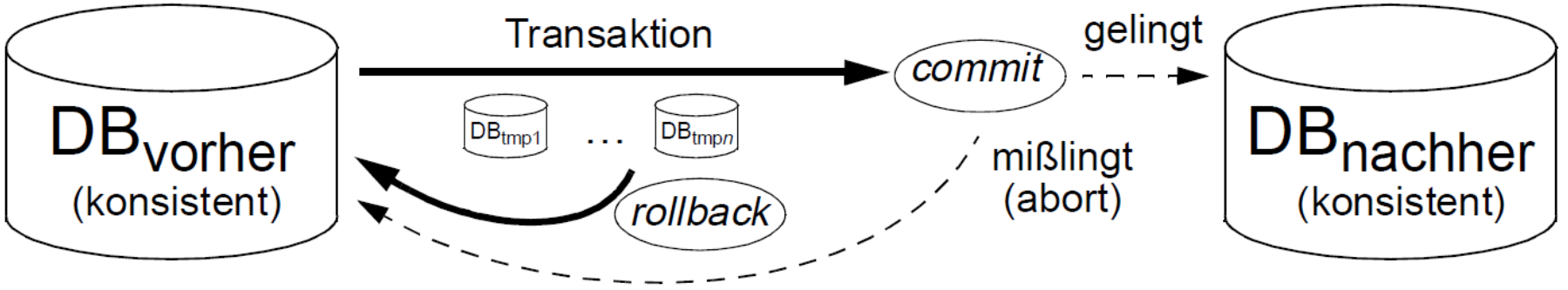


Ausführung einer Transaktion (commit)

Ein **COMMIT** kann …

- *gelingen*: Der neue Zustand wird dauerhaft gespeichert, oder
- *scheitern*: Der ursprüngliche Zustand wie zu Beginn der Transaktion bleibt erhalten (bzw. wird wiederhergestellt). Ein COMMIT kann u.a. scheitern, wenn die Verletzung von Integritätsbedingungen erkannt wird.

Schematischer Ablauf:



Transaktionsverwaltung

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

Transaktionsmodell: Abstraktion auf Lese- und Schreiboperationen

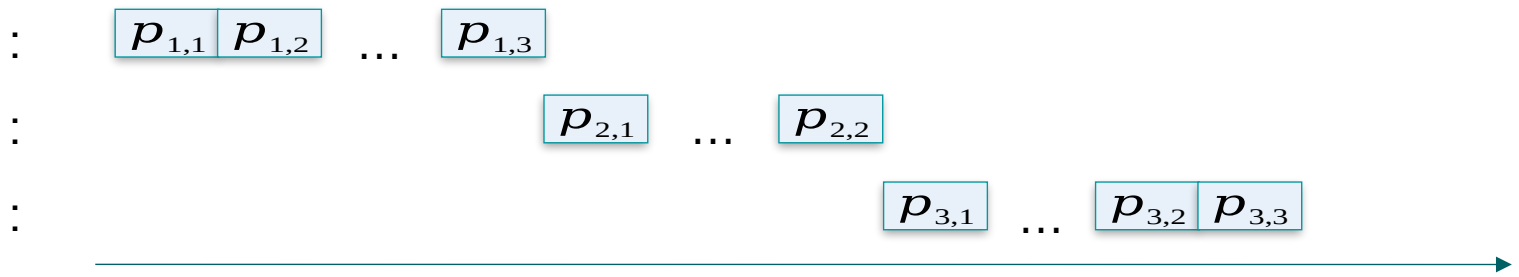
- Eine **Transaktion** (), d.h. ein DB-Programm während der Ausführung, wird vereinfacht betrachtet als ein Programm, das nur aus Lese- und Schreiboperationen auf einer Datenbank besteht
  - Leseoperation **read(x)** liest DB-Objekt x
  - Schreiboperation **write(x)** schreibt DB-Objekt x
- Die Beschränkung auf Lese- und Schreib-Operationen ist aus der Realität abstrahiert (z.B. Weiterverarbeitung von Daten) und dient zur vereinfachten Analyse
- Semantische Information über ein DB-Programm werden nicht mehr betrachtet
- Die Abstraktion ist hinreichend für viele praktische Anwendungen

Transaktionen: Definition

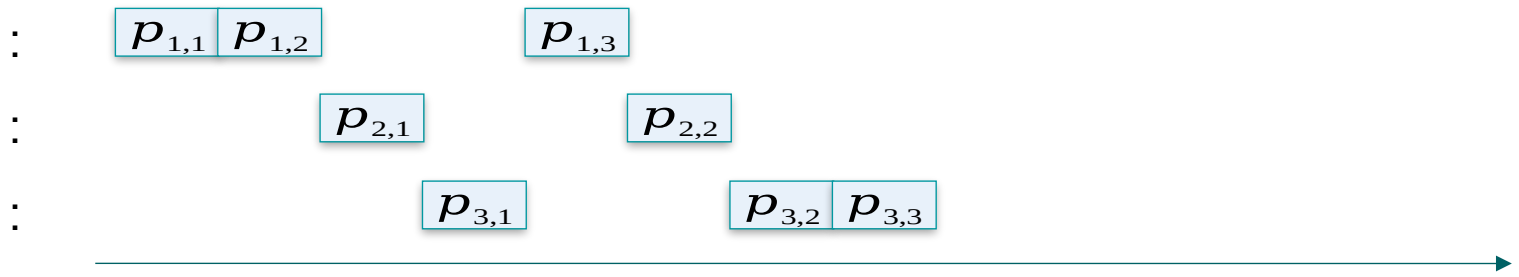
- **Annahme**: eine Datenbank ist eine Menge von Objekten auf die nur mittels Schreib- und Leseoperationen zugegriffen wird
- Eine **Transaktion** ist definiert als eine endliche Folge von Schreib- und Leseoperationen:
  - mit
  - für
- Indizes kennzeichnen verschiedene (nebenläufige) Transaktionen, z.B.:

Bearbeitung mehrerer Transaktionen

- Wie können mehrere nebenläufige Transaktionen effizient ausgeführt werden?
- Serieller Ablaufplan (Schedule):



- Verzahnter Schedule:



Probleme beim Ausführen von Transaktionen

Nach dem ACID-Prinzip “Isolation” sollen Transaktionen im logischen Einbenutzerbetrieb ablaufen, d.h. innerhalb einer Transaktion ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen.

Ist die Isolierung der Transaktionen in einem Datenbanksystem nicht sichergestellt, können verschiedene **Anomalien/Synchronisationsprobleme** auftreten:

- Verlorengegangene Änderungen (**Lost Updates**)
- Zugriff auf “schmutzige” Daten (**Dirty Read**, **Dirty Write**)
- Nicht-reproduzierbares Lesen
- Phantomproblem

Probleme beim Ausführung von Transaktionen

Die o.a. Anomalien werden unter anderem am Beispiel der folgenden Flugdatenbank erläutert:

Passagiere		
Flug#	Name	Platz
LH745	Müller	3A
LH745	Meier	6D
LH745	Huber	5C
BA932	Schmidt	9F
BA932	Huber	5C

Fluginfo	
Flug#	AnzPass
LH745	3
BA932	3

Verlorengegangene Änderungen (Lost Update)

Änderungen einer Transaktion können durch Änderungen anderer Transaktionen überschrieben werden und dadurch verloren gehen.

- Beispiel:
  Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus:
  UPDATE Fluginfo SET AnzPass = AnzPass + 1 WHERE Flug# = ‘BA932’;
- Möglicher Ablauf:

T1	T2
$A_{1,i}$ Read Fluginfo[AnzPass] —> x_1	
	$A_{2,j}$ Read Fluginfo[AnzPass] —> x_2
	$A_{2,j+1}$ $x_2 := x_2+1$
	$A_{2,j+2}$ Write x_2 —> Fluginfo[AnzPass]
$A_{1,i+1}$ $x_1 := x_1+1$	
$A_{1,i+2}$ Write x_1 —> Fluginfo[AnzPass]	

- Ergebnis:
  Beide Transaktionen haben die Anzahl der Passagiere (für denselben Flug) jeweils um eins erhöht.
  Obwohl zwei Erhöhungen stattgefunden haben, ist in der Datenbank nur die Erhöhung von T1 wirksam.
  Die Änderung von T2 ist verlorengegangen.

Zugriff auf “schmutzige” Daten (Dirty Read, Dirty Write)

Als “schmutzige” Daten bezeichnet man Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden.

- Beispiel:
  - T1 erhöht das Gehalt um 500 Euro, wird aber später abgebrochen
  - T2 erhöht das Gehalt um 5% und wird erfolgreich abgeschlossen
- Möglicher Ablauf:

	T1		T2
$A_{1.1}$	UPDATE … Gehalt := Gehalt + 500;		
		$A_{2.1}$	UPDATE Personal SET Gehalt := Gehalt * 1.05; COMMIT
	ROLLBACK		

- Ergebnis:
  - Der Abbruch der ändernden Transaktion T1 macht die geänderten Werte ungültig, sie werden zurückgesetzt. Die Transaktion T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*).
  - Verstoß gegen *ACID*: Dieser Ablauf verursacht einen dauerhaften fehlerhaften Datenbankzustand (*Consistency*), bzw. T2 muss nach COMMIT zurückgesetzt werden (*Durability*).

Nicht-reproduzierbares Lesen

Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts.

- Beispiel:
  - T1: Gib die Fluginfo für alle Flüge und die Anzahl der Passagiere für BA932 aus
  - T2: Buche den Platz 3F auf dem Flug BA932 für Passagier Meier
- Möglicher Ablauf:

	T1		T2
$A_{1.1}$	SELECT * FROM Fluginfo;		
		$A_{2.1}$	INSERT INTO Passagiere (*) VALUES ('BA932', 'Meier', '3F');
		$A_{2.2}$	UPDATE Fluginfo SET AnzPass = AnzPass + 1 WHERE Flug# = 'BA932';
		$A_{2.3}$	COMMIT;
$A_{1.2}$	SELECT AnzPass FROM Fluginfo WHERE Flug# = 'BA932';		

- Ergebnis:
  Die Anweisungen A1.1 und A1.2 liefern ein unterschiedliches Ergebnis für den Flug BA932, obwohl die Transaktion T1 den Datenbankzustand nicht geändert hat.

Phantomproblem

Das Phantomproblem ist ein nicht-reproduzierbares Lesen in Verbindung mit Aggregatfunktionen.

- Beispiel:
  - AnzPass werde jetzt durch COUNT(*) berechnet und nicht mehr in FlugInfo gespeichert
  - T1: Drucke die Passagierliste sowie die Fluginfo für den Flug LH745
  - T2: Buche den Platz 7D auf dem Flug LH745 für Phantomas
- Möglicher Ablauf:

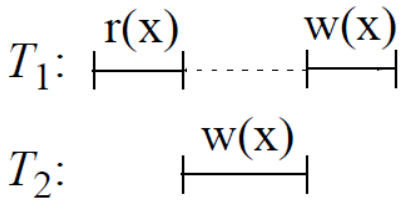
	T1		T2
$A_{1.1}$	SELECT * FROM Passagiere WHERE Flug# = 'LH745';		
		$A_{2.1}$	INSERT INTO Passagiere VALUES ('LH745', 'Phantomas', '7D');
		$A_{2.2}$	COMMIT;
$A_{1.2}$	SELECT AnzPass = COUNT(*) FROM Passagiere WHERE Flug# = 'LH745';		

- Ergebnis:
  Für die Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere schon berücksichtigt ist.

Beobachtung:
mögliche Problemursache Nicht-serialisierbare Schedules

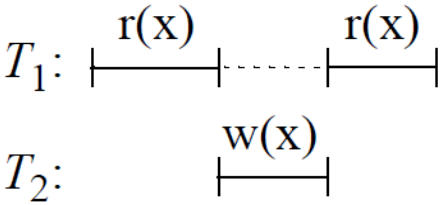
- Man hat früh beobachtet, dass viele Probleme offenbar dann auftreten, wenn Schedules „nicht serialisierbar“ sind, z.B.

$s = r_1(x) w_2(x) w_1(x)$



Lost Update

$s = r_1(x) w_2(x) r_1(x)$



Non-repeatable Read

- Erster Lösungsansatz zur Formalisierung des Synchronisationsproblems:

Nur „serialisierbare“ Schedules dürfen zugelassen werden.

Transaktionsverwaltung

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

Formaler Korrektheitsbegriff 1: Serialisierbarkeit von Schedules

- Ein **Schedule** für eine Menge von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.
- Ein **serieller Schedule** ist ein Schedule von , in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden. Serielle Schedules erfüllen offensichtlich die Isolationsbedingung des ACID-Prinzips.
- Ein Schedule von ist **serialisierbar**, wenn er „äquivalent“ ist zu einem (beliebigen) seriellen Schedule (Permutation von).

Um Serialisierbarkeit sicherzustellen, brauchen wir einen **Äquivalenzbegriff**, der

- möglichst viel Parallelität gestattet (Mehrbenutzerfähigkeit!),
- auch bei großen Nutzerzahlen effizient zu testen, möglichst automatisch zu garantieren
- zu jedem Zeitpunkt der Lebensdauer eines DBMS anwendbar ist.

- Das Konzept **Konfliktserialisierbarkeit** erfüllt alle diese Anforderungen!

Schedule: Definition

- Sei eine endliche Menge von Transaktionen.
- Das **Shuffle-Produkt** von bezeichnet die Menge aller Folgen von Aktionen, in welchen die gegebenen Transaktionen als Teilfolgen auftreten und keine weiteren Aktionen vorkommen.
- Ein **vollständiger Schedule** für ist eine Folge mit den zusätzlichen Pseudoaktionen (commit) und (abort) für jedes entsprechend den folgenden Regeln:
  - für alle
  - oder steht in hinter der letzten Aktion von für alle
- Die **Menge aller vollständigen Schedules** wird als bezeichnet.
- Ein vollständiger Schedule ist **seriell**, wenn es eine Permutation von gibt so dass gilt

Notationen für Transaktionszustands-Klassen

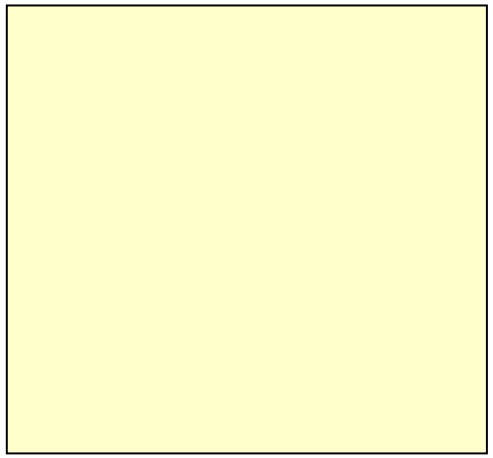
Sei ein Schedule. Dann definieren wir die folgenden Mengen für :

- Die Menge aller Aktionen:
- Die Menge aller Transaktionen:
- Die Menge aller bestätigten Transaktionen:
- Die Menge aller abgebrochenen Transaktionen:
- Die Menge aller aktiven Transaktionen:

Beispiel

- Seien die folgenden Transaktionen wie folgt gegeben:

- Dann gilt für die folgenden Schedules:
  - ist ein Schedule
  - ist ein Schedule
  - ist ein serieller Schedule



Konfliktmenge

- Sei ein Schedule und zwei Transaktionen mit .
- Zwei Datenoperationen und in stehen in **Konflikt**, falls sie auf demselben Objekt arbeiten und mindestens eine Datenoperation eine Schreiboperation ist.
- Die Menge der **Konfliktbeziehungen** von Schedule ist definiert als:

- Die Menge bezeichnet im Folgenden die Menge der Konfliktbeziehungen eines Schedules , *bereinigt von abgebrochenen Transaktionen*.
  - Randbemerkung: Wir betrachten hier also nur laufende und erfolgreich abgeschlossene Transaktionen. *Dirty Read-Probleme* können in diesem Modell NICHT formuliert werden, weil sie ja nur Abbruch von Transaktionen entstehen.

Konfliktmenge: Beispiel

- Sei der folgende Schedule gegeben:

- Dann sind die Konfliktmengen wie folgt gegeben:

Konfliktäquivalenz

- Zwei Schedules und heißen **konfliktäquivalent**, bezeichnet mit , falls folgendes gilt:
  - Die Menge der Aktionen sind gleich, d.h.:
  - Die Menge der bereinigten Konfliktbeziehungen sind gleich, d.h.:

- Konfliktäquivalenz kann wie folgt überprüft werden:
  - Für zwei gegebene Schedules (mit derselben Menge von Operationen), bestimme die Menge der bereinigten Konfliktbeziehungen und überprüfe sie auf Gleichheit.

Konfliktäquivalenz: Beispiel

- Seien die folgenden Schedules gegeben:

- Die beiden Schedules sind konfliktäquivalent, d.h. es gilt .

Konfliktserialisierbarkeit

- Ein vollständiger Schedule heißt **konfliktserialisierbar**, falls ein serieller Schedule existiert mit .
- Die **Klasse aller konfliktserialisierbaren Schedules** bezeichnen wir mit *CSR*.

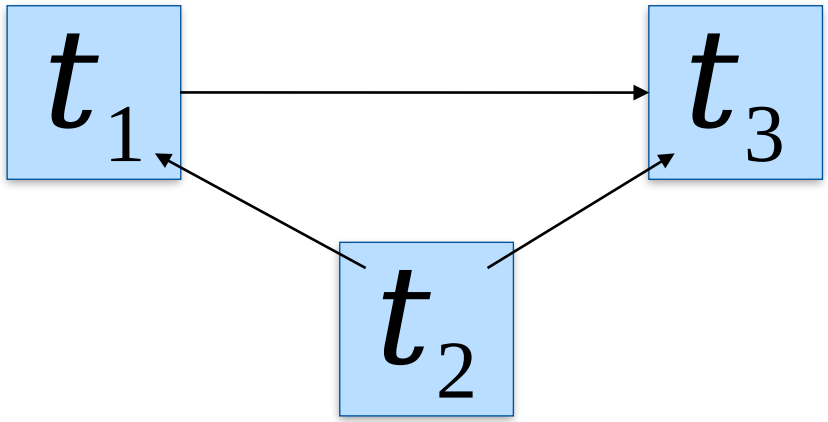
- Beispiele:

- Ob ein Schedule zur Klasse der konfliktserialisierbaren Schedules gehört kann einfach geprüft werden.

Konfliktgraph

- Sei ein Schedule. Der **Konfliktgraph** von ist wie folgt definiert:
  - Kontenmenge
  - Kantenmenge

- Beispiel:
  - Konfliktgraph für Schedule :



Serialisierbarkeitssatz

- Die Zugehörigkeit eines Schedules zur Klasse der konfliktserialisierbaren Schedules *CSR* kann durch die folgende Aussage bestimmt werden:

- Beweis wird mittels topologischem Sortieren geführt

- Korollar: Zugehörigkeit zu CSR kann in polynomieller Zeit geprüft werden

Einige Synchronisationsprobleme

- Welche Synchronisationsprobleme werden durch *CSR* verhindert?
  - Lost Update
  - Dirty Read
  - Phantom

Konfliktserialisierbarkeit und Fehlersicherheit

- 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:

  - aus Sicht der Fehlersicherheit nicht akzeptabel, da Objekt von liest und danach abbricht (Dirty Read)
  - Datenbank muss dafür sorgen, dass nicht durchgeführt wird und nicht von liest
- **Fehlersicherheit** eines Schedules ist die Eigenschaft, dass dieser Schedule sich bei Abbruch einer oder mehreren Transaktionen genauso verhält wie ein ähnlicher Schedule, der ausschließlich die nicht abgebrochenen Transaktionen enthält. Fehlersicherheit ist eine „orthogonale“ Eigenschaft, die erst zusammen mit Konfliktserialisierbarkeit zu korrekter Synchronisation führt

„liest-von“-Notation

- Sei ein Schedule, dann bezeichnet das Auftreten von Aktion p vor q
- Seien :

1) **liest von in** , falls alle drei Bedingungen gelten:

a)

- Eine Aktion liest von

b)

- ist zum Zeitpunkt des Lesens nicht abgebrochen

c)

- ist der letzte echte Schreiber auf vor

2) **liest von in** , falls irgendein von in s liest

„liest-von“-Notation: Beispiel

- Sei ein Schedule mit Transaktionen :

- Dann gilt:
  - liest von aber nicht von
  - liest von
  - liest nicht von

Rücksetzbarkeit

- Ein Schedule heißt **rücksetzbar**, falls für alle Transaktionen
  mit gilt:

liest von in

- Eine Transaktion wird freigegeben (committed), wenn alle anderen Transaktionen von denen sie gelesen hat, auch freigegeben wurden.
- Die **Klasse aller rücksetzbaren Schedules** bezeichnen wir mit (recoverable).

Rücksetzbarkeit: Beispiel

Seien zwei Schedules mit Transaktionen :

- Dann gilt:
  - liest von in und , aber . Hieraus folgt .
  - , da die Commit-Operation von hinter der von steht.

- Weiteres Problem tritt auf in , falls direkt nach abbrechen würde
  - Dirty Read-Situation würde Abbruch von nach sich ziehen

Vermeidung kaskadierender Aborts

- Ein Schedule **vermeidet kaskadierende Aborts**, falls für alle Transaktionen mit gilt:

liest von in

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

Vermeidung kaskadierender Aborts: Beispiel

Seien zwei Schedules mit Transaktionen :

- Dann gilt:

- Weiteres Problem tritt auf in , falls direkt nach abstürzt
  - braucht nicht abgebrochen zu werden
  - Objekt muss auf den richtigen Zustand zurückgesetzt werden

Striktheit

- Ein Schedule heißt **strikt**, falls für alle Transaktionen und für alle Aktionen gilt:

- 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 .

Striktheit: Beispiel

Seien zwei Schedules mit Transaktionen :

- Dann gilt:

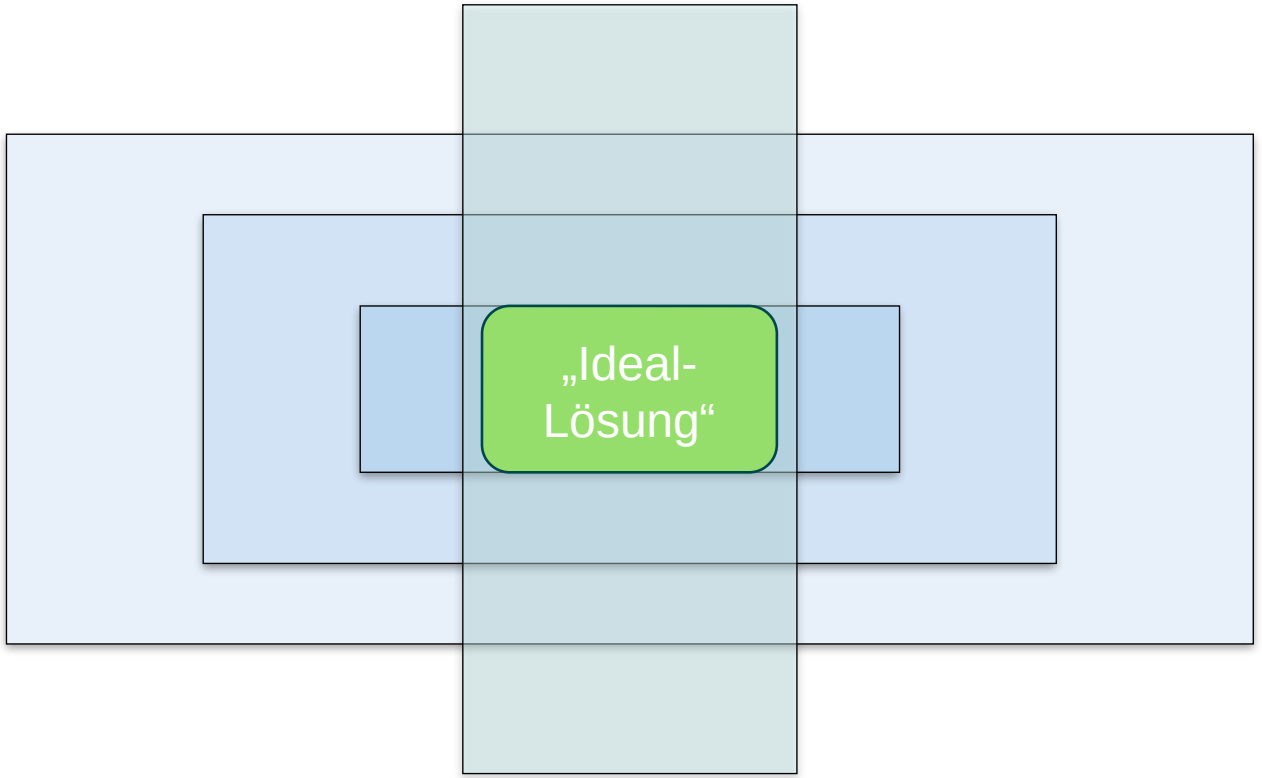
Korrektheit von Schedules

- Fehlersicherheit () und Konfliktserialisierbarkeit () sind „orthogonale“ Anforderungen an Schedules

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

Fehlersichere Schedules

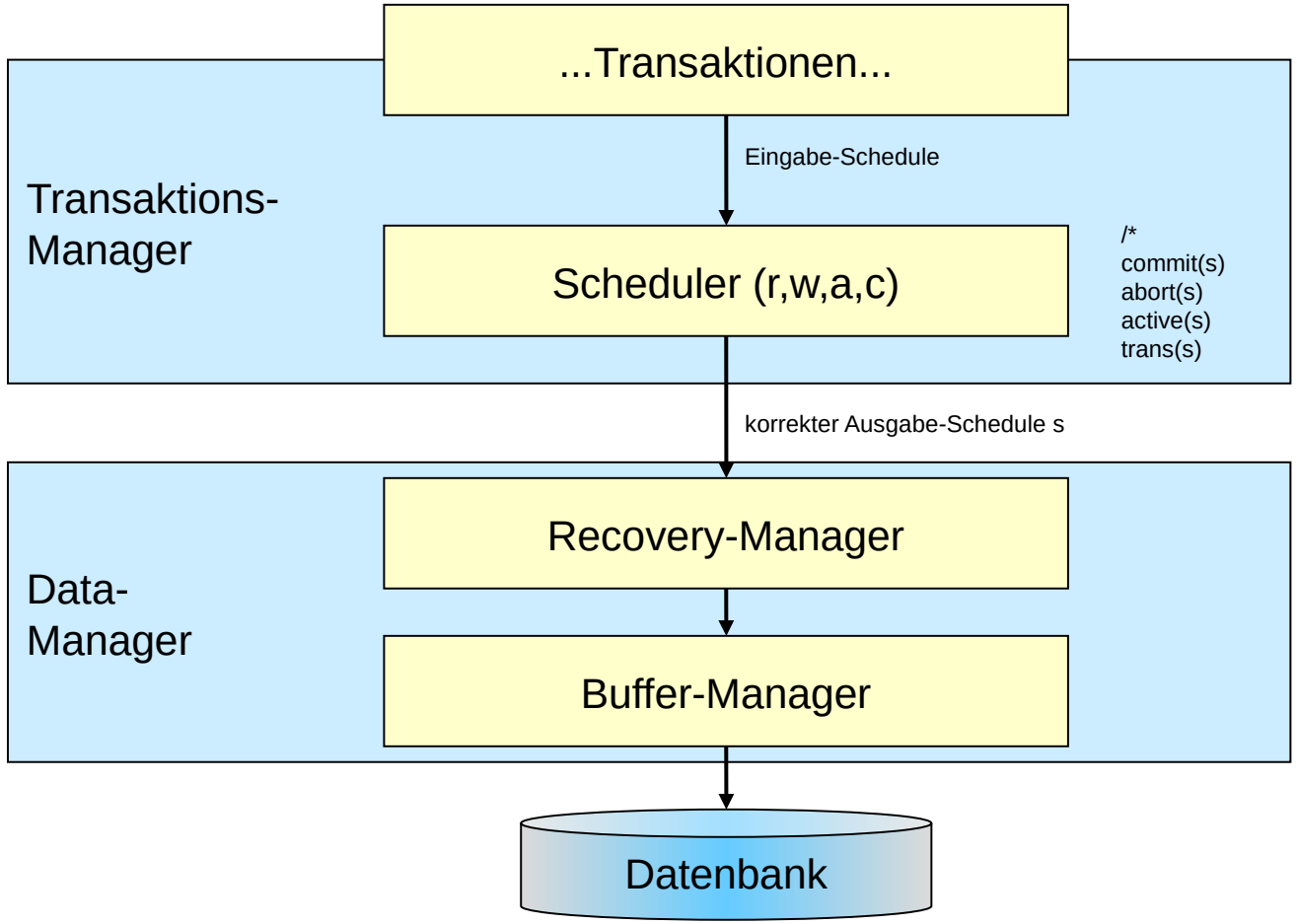
- Es gilt der folgende Zusammenhang:



Transaktionsverwaltung

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

Transaktionenverarbeitende Schichten eines DBMS



* – begin of transaction
– end of transaction

Transaktions-Manager

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

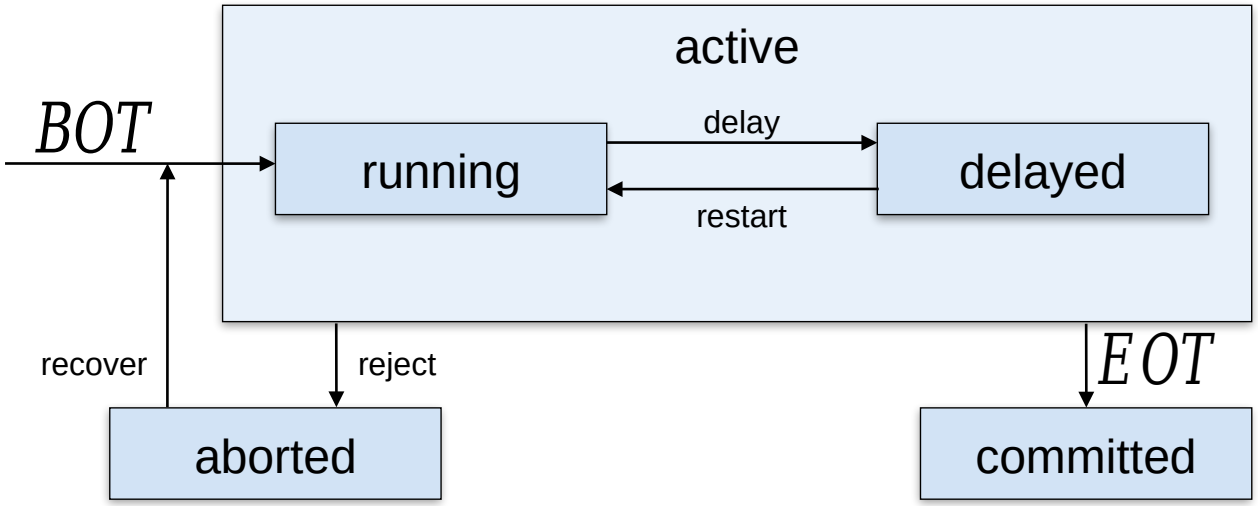
Data-Manager

- Der **Data-Manager** führt die Aktionen des (korrekten) Ausgabe-Schedules der Reihe nach aus:
  - für Aktion liest er ein Datenobjekt aus der Datenbank in den Puffer
  - für Aktion schreibt er ein Datenobjekt in die Datenbank oder in den Puffer
  - für Aktion macht er das Ergebnis der Transaktion „permanent“
  - für Aktion 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 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
  b) zurückweisen (reject)
     - Die Aktion wird nicht ausgeführt, Abbruch der entsprechenden Transaktion mit
     - Nur für Datenoperationen
  c) verzögern (delay)
     - Die Aktion wird weder ausgeführt noch abgelehnt, sondern an den Transaktions-Manager zurückgegeben
     - Nur für Datenoperationen

Zustände einer Transaktion



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

Sicherheit/Ausdrucksstärke eines Schedulers

- Bezeichne das **Erzeugnis** eines Schedulers , d.h. die Menge aller Schedules, welche als Ausgabe erzeugen kann.
- Ein Scheduler ist
  - ***s-sicher***, falls gilt
  - ***f-sicher***, falls (oder oder) gilt
  - ***sicher***, falls (oder oder) 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).

Sperrende und Nicht-sperrende Scheduler

- **Sperrende Scheduler**
  - Pessimistische Ablaufsteuerung durch Sperrverfahren, Locking
  - Durch Lese- und Schreibsperren wird verhindert, dass Änderungen sich auf nebenläufige Transaktionen auswirken können.
  - *Nachteil*: Schreibende, aber auch nur-lesende Transaktionen müssen ggf. warten, bis andere schreibende (oder auch lesende) 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 ungehindert bis zum COMMIT 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) werden zur Synchronisation von Zugriffen auf gemeinsam verwendete Datenobjekte verwendet
- Diese Sperren werden vom Scheduler für die Transaktionen gesetzt und wieder freigegeben
- Eine gesetzte Sperre bedeutet, dass das Datenobjekt nicht verfügbar ist
  - Vor einem Zugriff wird die Sperre abgefragt und gesetzt, falls das Datenobjekt verfügbar ist.
  - Das Datenobjekt ist für eine bestimmte Transaktion gesperrt.
  - Nach einem Zugriff wird die Sperre wieder aufgehoben.
- Für jedes Datenobjekt gibt es zwei Arten von Sperren:
  - Lese-Sperre: (read lock) und (read unlock)
  - Schreib-Sperre: (write lock) und (write unlock)

- Sperren in Konflikt Operationen in Konflikt
- Ein sperrender Scheduler erweitert jede Transaktion um *rl/wl/ru/wu*-Aktionen

Regeln zur Anwendung von Sperren

- Für jede Transaktion , die vollständig in einem Schedule enthalten ist, gilt:
  - **(L1)** Falls eine Aktion bzw. enthält, so steht irgendwo davor bzw. und irgendwo dahinter bzw. in
  - **(L2)** Für jedes das von verwendet wird, existiert genau ein bzw. in
  - **(L3)** Kein / ist redundant

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

Sperren: Beispiel

- Seien und zwei Transaktionen und die folgenden zwei Schedules:

- Dann gilt:
  - Beide Schedules erfüllen die Regeln (L1)-(L3)

Sperrprotokoll

- Ein Scheduler arbeitet nach einem **Sperrprotokoll**, falls für jeden Ausgabe-Schedule und jede Transaktion gilt:
  - erfüllt die Regeln (L1)-(L3)
  - ist durch und gesperrt, für und , so sind diese Sperren nicht in Konflikt, d.h. sie sind kompatibel gemäß der nachfolgend dargestellten Tabelle:

	+	-
	-	-

  - ist , so ist das Setzen von Schreib-/Lesesperren bei schon bestehenden Lese-/Schreibsperren derselben Transaktion zulässig.

2-Phasen-Sperrprotokoll (*2PL*)

- Ein Sperrprotokoll ist **zweiphasig**, falls für jeden erzeugten Schedule und jede Transaktion gilt:
  - Nach der ersten /-Aktion folgt keine weitere /-Aktion
- In der ersten Phase einer Transaktion werden Sperren ausschließlich gesetzt, in der zweiten Phase werden Sperren ausschließlich freigegeben:

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

Sicherheit von *2PL*

- Wir bezeichnen die Menge der bereits freigegebenen Operationen in einem Schedule mit (committed projection)
- Sei ein Schedule eines -Schedulers, dann gilt für jede Transaktion :
  - **Jede Lese-/Schreiboperation wird ge- und entsperrt**
    - Falls in vorkommt, dann kommen auch und darin vor mit der folgenden Reihenfolge:
      < < .
  - **Sind zwei Aktionen in Konflikt, so sind die Sperren nicht gleichzeitig gesetzt**
    - Falls mit und die Aktionen und in
      in Konflikt, so gilt entweder
      < oder <
  - **Alle Sperren werden zuerst gesetzt (1. Phase) und dann aufgehoben (2. Phase)**
    - Sind die Aktionen und in , so gilt:
      <

Konfliktserialisierbarkeit von -Schedules

- Sei ein Schedule eines -Schedulers und der Konfliktgraph von . Dann gilt:
  - Ist eine Kante in , so gilt < für ein Objekt und zwei Aktionen und in Konflikt.
  - Ist ein Weg in , so gilt < für zwei Objekte und und Aktionen und .
  - **ist azyklisch**.

➢ Ein -Scheduler ist *s-sicher*, d.h. es gilt: .

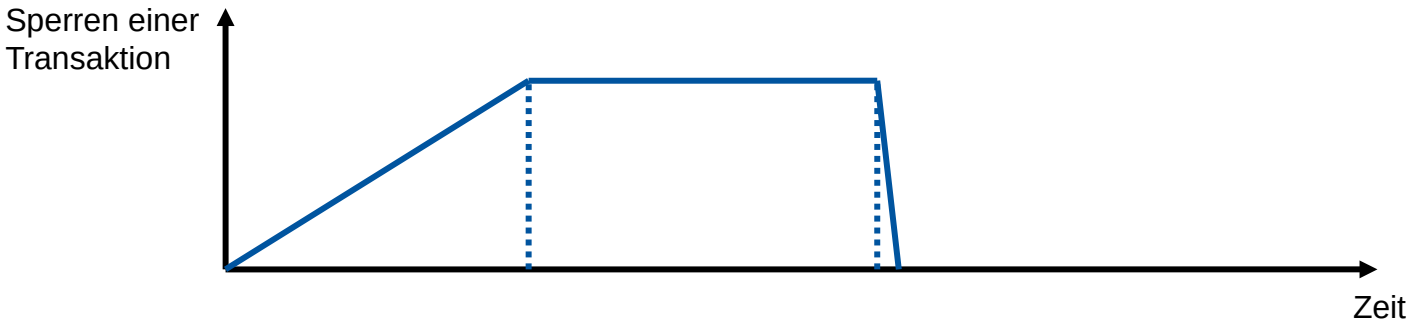
Konservatives/Statisches 2-Phasen-Sperrprotokoll (*C2PL*)

- **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 -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 -Aktionen müssen im Vorhinein () 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 r/w-Operation zu halten.
- Ein -Scheduler ist *sicher*, d.h. es gilt sogar: .
- 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

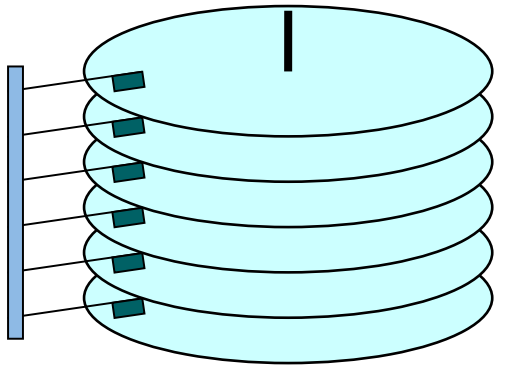
- Falls die von x*2PL* gesperrten Objekte „groß“ sind, sind nur wenige Sperren zu verwalten, aber Konflikte zwischen Sperren sind häufiger.
- Falls die von x*2PL* gesperrten Objekte „klein“ sind, ist mehr Nebenläufigkeit möglich, aber die Verwaltungskosten für Sperren sind auch höher.
- Verallgemeinerung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

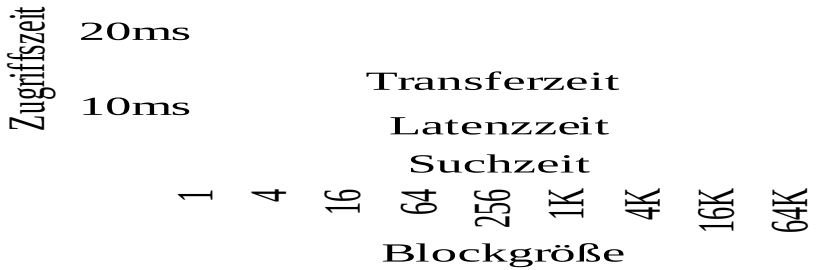
Transaktionsverwaltung

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

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



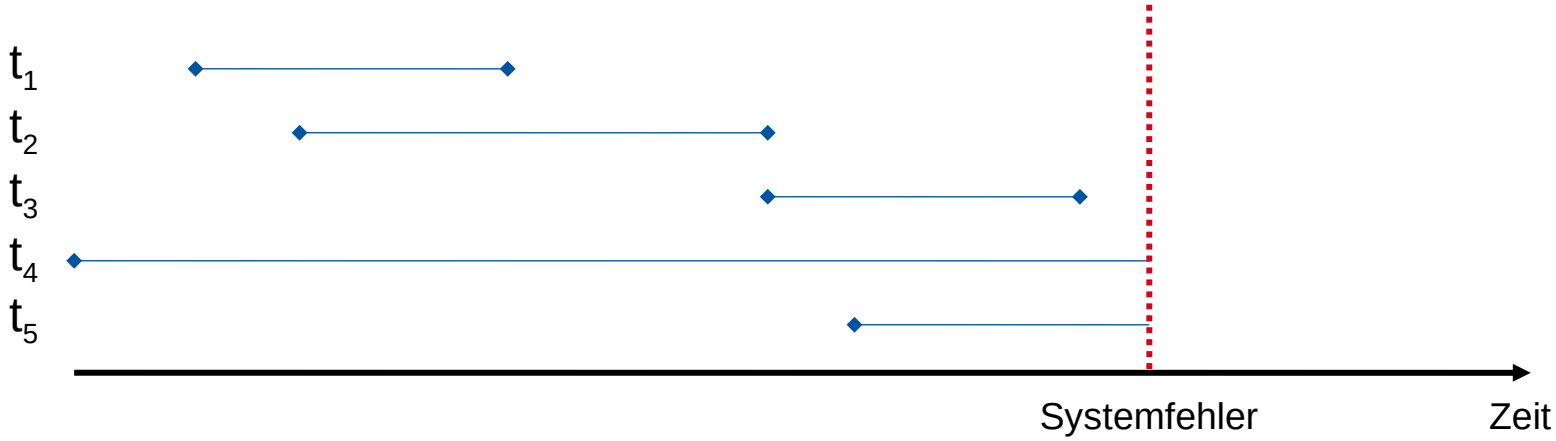


Klassifikation von Fehlern

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.

- **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.
- **Medienfehler**
  Verlust von permanenten Daten durch Zerstörung des Speichermediums, z.B. Plattencrash, Brand, Wasserschaden, etc.

Szenario nach einem Fehler



Transaktionen werden in zwei Klassen eingestuft:

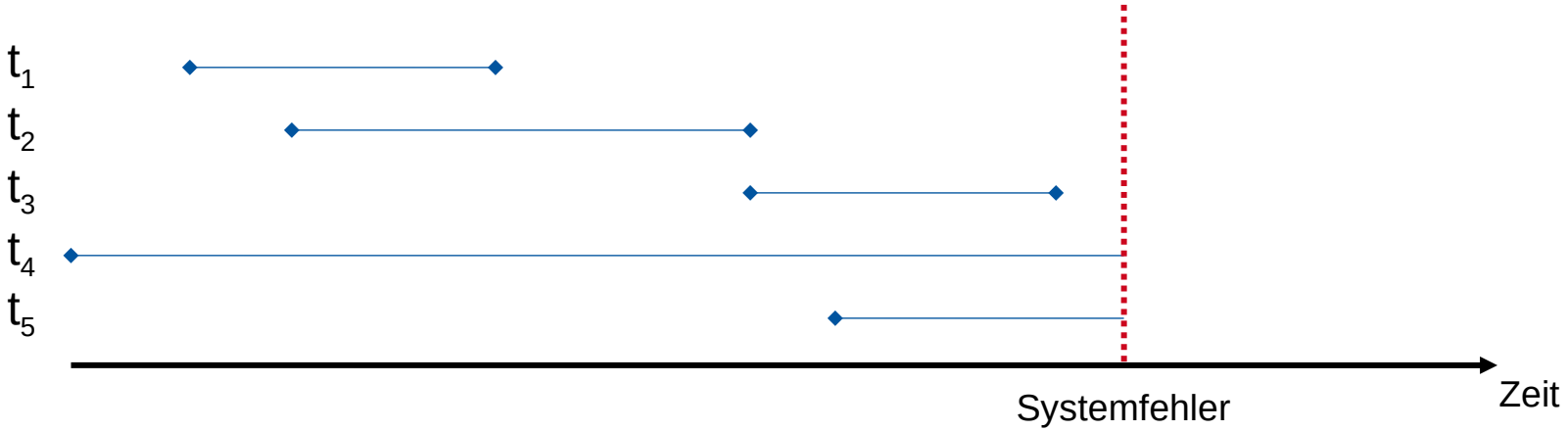
- Transaktionen, die vor dem Fehler bereits COMMITed waren.
  Durability erfordert ein **REDO**, falls ihre Ergebnisse nicht im stabilen Speicher sind.
- Transaktionen, die zum Zeitpunkt des Fehlers noch aktiv waren.
  Atomarität erfordert ein **UNDO**, falls einige Ergebnisse bereits im stabilen Speicher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

Technische Grundlagen

Es gibt unterschiedliche technische Maßnahmen zur Wiederherstellung:

- Für Medienfehler: Duplizierung (d.h. gezielte Redundanz) des Datenbankzustandes,
  - Bänder oder andere Backup-Speicher erfordern Kaltstart auf altem Datenbankzustand nach Medienfehlern
  - Spiegelplatten erlauben prinzipiell auch Warmstart (Wiederanlauf bei laufendem Betrieb)
  - zusätzliches Rechenzentrum an anderem Ort erhöht die statistische Unabhängigkeit gegenüber externen Großschäden (z.B. Erdbeben in Kalifornien, Katasterdaten Palästina)
  - etc.
- Bei Transaktions- und Systemfehlern: Mitprotokollierung der laufenden Schedules incl. BOT, COMMIT, ABORT) von Transaktionen in Logfiles
  - Da die Programmsemantik auf das Read-Write-Modell beschränkt ist, müssen bei allen Schreiboperationen sowohl der alte als auch der neue Wert geloggt werden. Die Lese-/ Schreibobjekte sind hier meist physische Speicherblöcke (Paging).
  - Logfiles entstehen sequentiell wenn auf eigenem Sekundärspeichermedium, schnelles Lesen und Schreiben möglich

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, deren Management trotz massivem Mehrbenutzerbetrieb und unter Annahme unterschiedlichster Fehler die Grundanforderungen des ACID-Prinzips formal korrekt und effizient erfüllen soll.

Korrekte Synchronisation und Fehlersicherheit werden formal unter Nutzung des 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 wird durch Redundanz in Form von Backups/Spiegelung (Medienfehler) und Operationslogs sichergestellt. Mit Write-Ahead-Logging kombiniert mit REDO/UNDO-Recovery erlaubt Wiederanlauf sogar dann, 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.