CAU
Christian-Albrechts-Universität zu Kiel

Christian Albrechts Universität zu Kiel
Institut für Informatik
AG Archäoinformatik (Data Science)

Skript zur Vorlesung

Informationssysteme

Wintersemester 2019/20

Kapitel 9.1: Transaktionssysteme
Datensicherheit (Recovery)

Skript © 2019 Matthias Renz, CAU Kiel
(basiert auf dem Skript der LMU München)

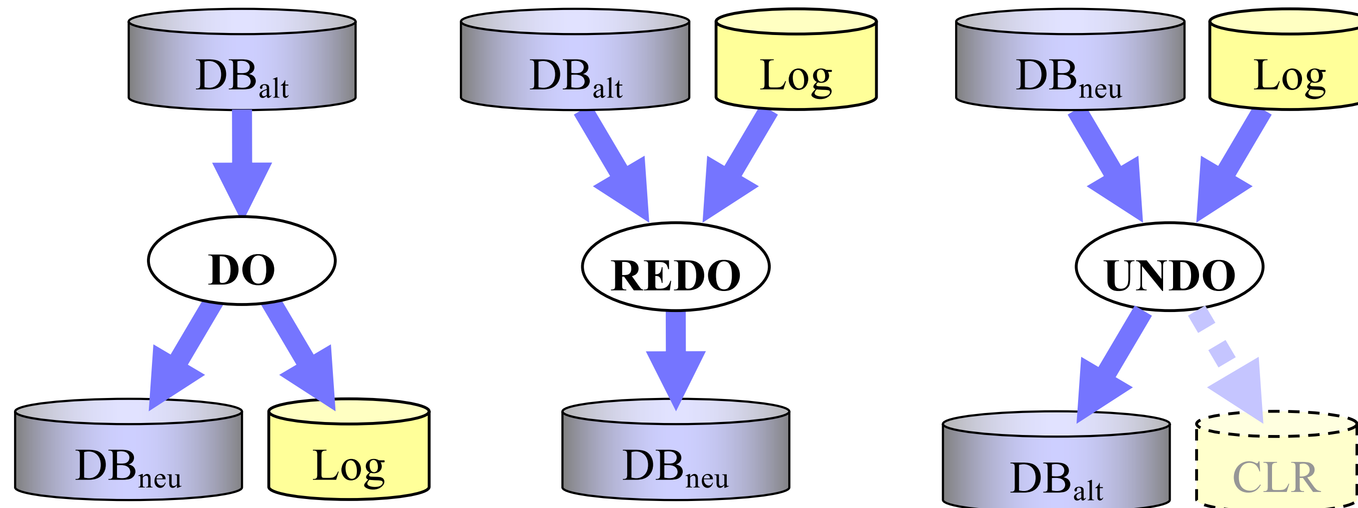
CAU Christian-Albrechts-Universität zu Kiel

Überblick: Fehler und Recovery

Fehler-Art:	Ursachen:	Behandlung:
Transaktionsfehler: Lokaler Fehler einer noch laufenden TA	Fehler im Anwendungsprogramm, Abbruch einer TA (rollback), Integritätsverletzung	Lokale Erfassung aller Aktionen (logging), Lokales UNDO, TA wird rückgesetzt
Systemfehler: Fehler mit Hauptspeicherverlust (Crash Recovery) (Warmstart)	Stromausfall, Absturz, Etc.	Verwaltung einer Log-Datei, Rücksetzen aller nicht abgeschlossenen TAs, Nachführen aller noch nicht festgeschriebenen TAs
Medienfehler: Fehler mit Hintergrundspeicher-verlust (Kaltstart)	Platten Crash, Wasserschaden	Datenbankzustände sichern (Archivkopie, Backup), Nachführen aller TAs nach letztem Backup-Zustand.

Protokolierung (Logging)

- Wiederherstellung (Recovery) des gültigen DB Zustands durch führen eines Protokolls (Logging).
- Für jede Änderungsoperation (**DO**) auf der DB wird ein Protokolleintrag (Log-Eintrag) erzeugt und gespeichert.
- Log-Einträge werden für das Rücksetzen (**UNDO**) von TAs, sowie für die Wiederholung (**REDO**) von (zurückgesetzten) TAs eingesetzt.
  - REDO: Information zum Nachvollziehen der Änderungen erfolgreicher TAs.
  - UNDO: Information zum Zurücknehmen der Änderungen unvollständiger TAs.

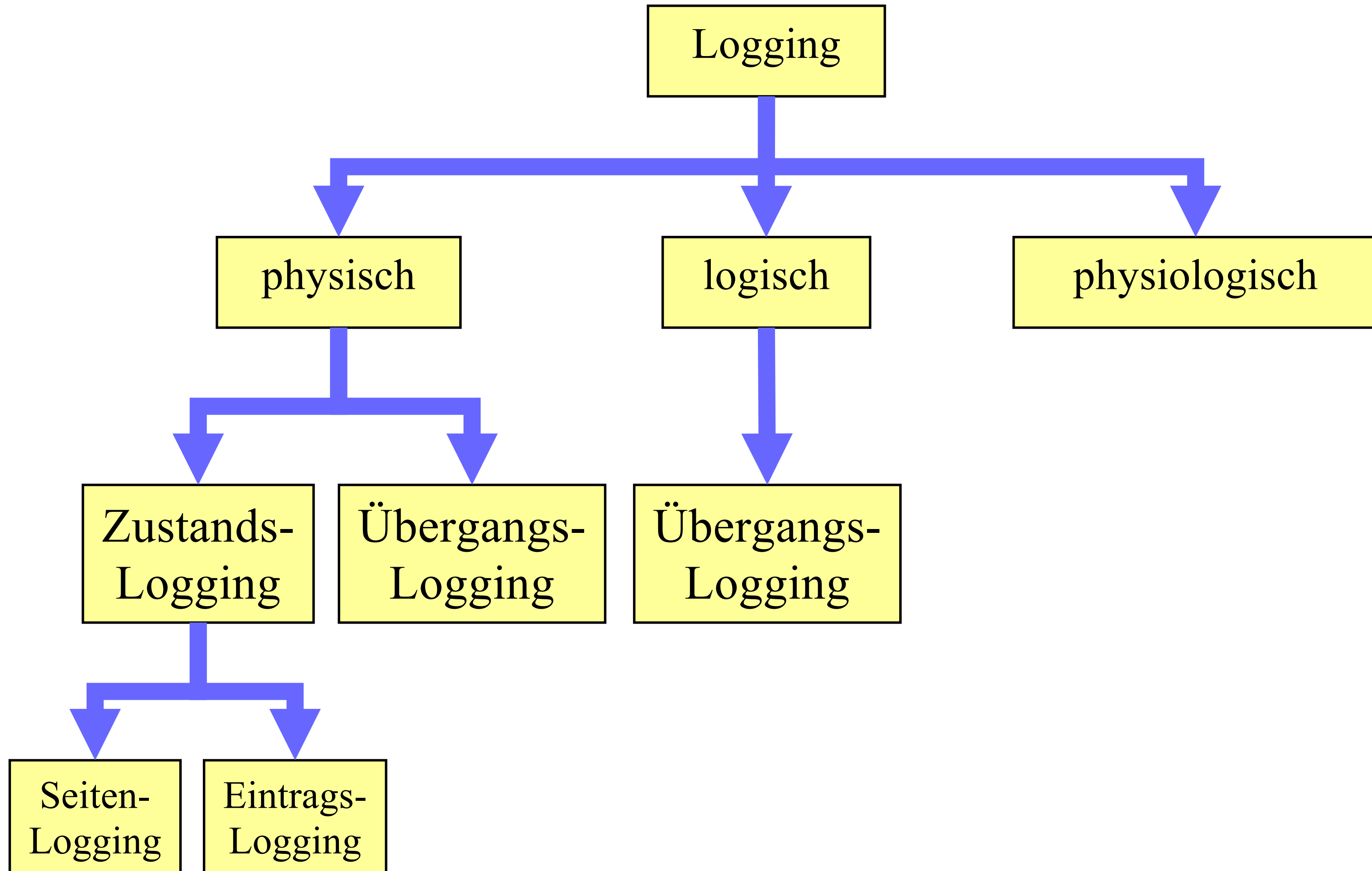


CLR = Compensation Log Record (zur Behandlung von Fehlern während der Recovery)

Christian-Albrechts-Universität zu Kiel

Protokolierung (Logging)

- Klassen von Logging-Verfahren:



Protokolierung (Logging)

Physisches Logging

- Protokoll auf der Ebene der physischen Objekte (Seiten, Datensätze, Indexeinträge)

- Zustandslogging
  - Protokollierung der Werte vor und nach jeder Änderung: Alte Zustände BFIM (Before-Images) und neue Zustände AFIM (After-Images) der geänderten Objekte werden in die Log-Datei geschrieben

- Übergangslogging
  - Protokollierung der Zustandsdifferenz zwischen BFIM und AFIM

Protokolierung (Logging)

Zustandslogging auf Seitenebene

- vollständige Kopien von Seiten werden protokolliert
- **Recovery sehr einfach und schnell** (Seiten einfach zurückkopieren)
- sehr **großer Logumfang** und hohe I/O-Kosten auch bei nur kleinen Änderungen
- Seitenlogging impliziert Seitensperren → hohe Konfliktrate bei Synchronisation

Protokolierung (Logging)

Zustandslogging auf Eintragsebene

- statt ganzer Seiten werden nur tatsächlich geänderte Einträge protokolliert
- kleinere Sperrgranulate als Seiten möglich
- **Protokollgröße reduziert sich** typischerweise um mind. 1 Größenordnung
- Log-Einträge werden in Puffer gesammelt → wesentlich weniger Plattenzugriffe
- **Recovery ist aufwändiger**: zu ändernde Datenbankseiten müssen vollständig in den Hauptspeicher geladen werden, um die Log-Einträge anwenden zu können

Christian-Albrechts-Universität zu Kiel

Protokolierung (Logging)

Übergangslogging

- Protokollierung der Zustandsdifferenz zwischen BFIM und AFIM
- Aus BFIM muss AFIM berechenbar sein
- Realisierbar durch XOR-Operation $\oplus$ (eXclusive-OR)[1]:

1) XOR:
$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

	Zustands-Logging	**Übergangs-Logging**
DO Änderung $A_{alt} \rightarrow A_{neu}$	Protokollierung von $BFIM = A_{alt}$, $AFIM = A_{neu}$	Protokollierung von $D = A_{alt} \oplus A_{neu}$
REDO (in DB liegt A_{alt})	Überschreibe A_{alt} mit $AFIM$	$A_{neu} = A_{alt} \oplus D$
UNDO (in DB liegt A_{neu})	Überschreibe A_{neu} mit $BFIM$	$A_{alt} = A_{neu} \oplus D$

Protokolierung (Logging)

Logisches Logging

- Spezielle Form des Übergangs-Logging: Protokollierung von Änderungsoperationen mit ihren aktuellen Parametern
- Protokoll auf hoher Abstraktionsebene ermöglicht **kurze Log-Einträge**
- Probleme für REDO: Änderungen umfassen typischerweise mehrere Seiten (Tabelle, Indexe)
  - Atomares Einbringen der Mehrfachänderungen schwierig
  - Logische Änderungen sind **aufwändiger durchzuführen** als physische Änderungen
- Probleme für UNDO: Mengenorientierte Änderungen können komplexe Recovery-Operationen verursachen:
  - Bsp.: DELETE FROM Products WHERE Group = 'G1'
    => UNDO erfordert viele Einfügungen, falls Produktgruppe G1 umfangreich ist
  - Bsp.: UPDATE Products SET Group = 'G2' WHERE Group = 'G1'
    => UNDO muss alte und neue Produkte der Gruppe G2 unterscheiden

Protokolierung (Logging)

Physiologisches Logging

- Kombination von physischem und logischem Logging: Protokollierung von elementaren Operationen innerhalb einer Seite
  - Physical-to-a-page
    - Protokollierungseinheiten sind geänderte Seiten
    - gut verträglich mit Pufferverwaltung und direktem (atomarem) Einbringen
  - Logical-within-a-page
    - logische Protokollierung der Änderungen auf einer Seite
- Bewertung
  - Log-Einträge beziehen sich nicht auf mehrere Seiten wie bei logischem Logging
  - Dadurch einfachere Recovery als bei logischem Logging
  - Log-Datei ist länger als bei logischem Logging aber kürzer als bei physischem Logging.

Protokolierung (Logging)

Log-Datei

- Art der Protokolleinträge
  - Beginn, Commit und Rollback von Transaktionen
  - Änderungen des DB-Zustandes durch Transaktionen
  - Sicherungspunkte (Checkpoints)
- Komponenten von Änderungseinträgen:

  (LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)
  - **LSN** (Log Sequence Number): eindeutige Kennung des Log-Eintrags in chronologischer Reihenfolge
  - **TA-Id**: eindeutige Kennung der TA, die die Änderung durchgeführt hat
  - **Page-Id**: Kennung der Seite auf der die Änderungsoperation vollzogen wurde (ein Eintrag pro geänderter Seite)
  - **REDO**: gibt an, wie die Änderung nachvollzogen werden kann
  - **UNDO**: beschreibt, wie die Änderung rückgängig gemacht werden kann
  - **PrevLSN**: Zeiger auf vorhergehenden Log-Eintrag der jeweiligen TA (Effizienzgründe)

CAU Christian-Albrechts-Universität zu Kiel

Protokolierung (Logging)

- Beispiel: Logisches Logging

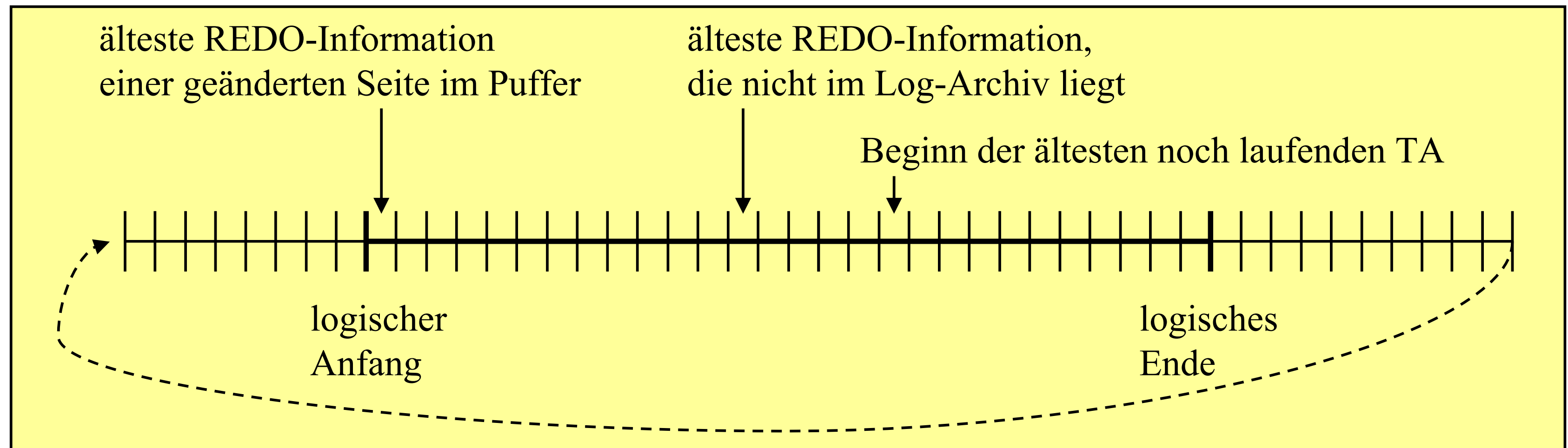
Ablauf T_1	**Ablauf T_2**	**Log-Eintrag** *(LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)*
begin		(#1, T_1, begin, 0)
read(A, a_1)		
	begin	(#2, T_2, begin, 0)
	read(C, c_2) //80	
$a_1 := a_1 - 50$		(#3, T_1, p_A, A-=50, A+=50, #1)
write(A, a_1)		
	$c_2 := 100$	(#4, T_2, p_C, C=100, C=80, #2)
	write(C, c_2)	
read(B, b_1) //70		
$b_1 := 50$		(#5, T_1, p_B, B=50, B=70, #3)
write(B, b_1)		
commit		(#6, T_1, commit, #5)
	read(A, a_2)	
	$a_2 := a_2 - 100$	(#7, T_2, p_A, A-=100, A+=100, #4)
	write(A, a_2)	
	commit	(#8, T_2, commit, #7)

Protokolierung (Logging)

- Die Log-Datei ist eine **sequentielle** Datei: Schreiben neuer Protokolldaten an das aktuelle Dateiende

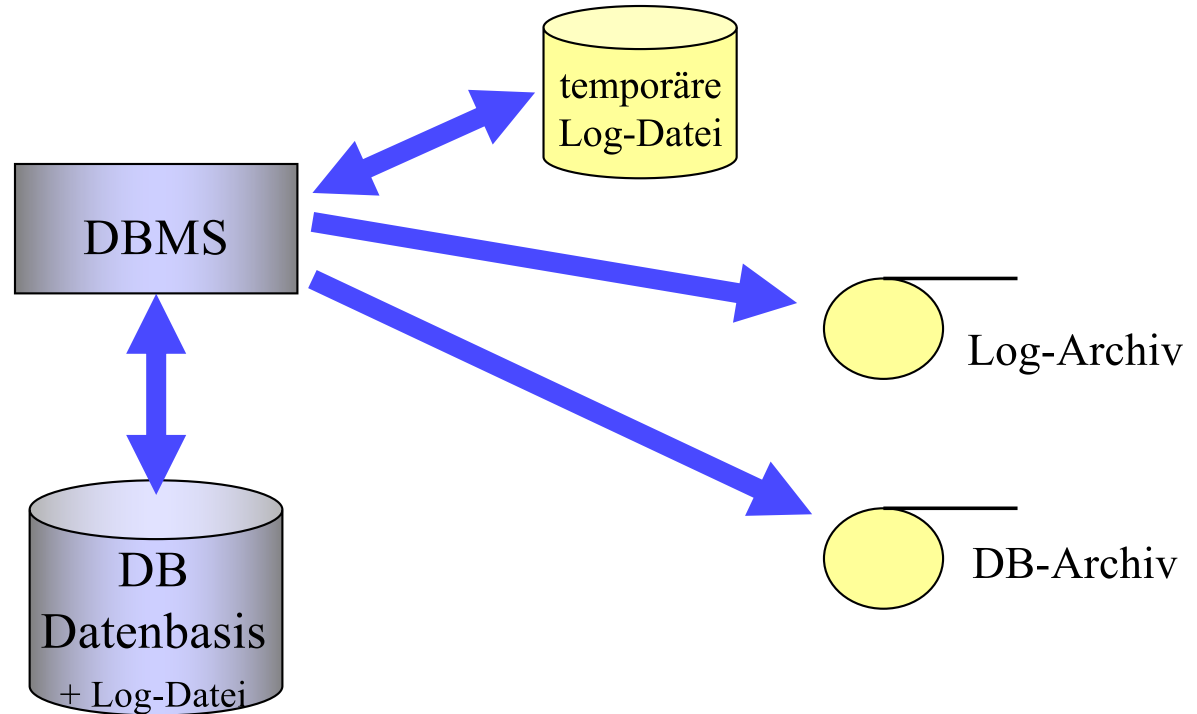
  => Ringpuffer-Organisation

  – Log-Daten sind für Crash-Recovery nur begrenzte Zeit relevant:
    - UNDO-Sätze für erfolgreich beendete TAs werden nicht mehr benötigt
    - Nach Einbringen der Seite in die DB wird REDO-Information nicht mehr benötigt



  – REDO-Information für Geräte-Recovery (Kaltstart bei Medienfehler) ist im Log-Archiv zu sammeln.

Systemkomponenten zur Datensicherung



- Behandlung von Transaktions- und Systemfehlern

  DB + temporäre Log-Datei → DB

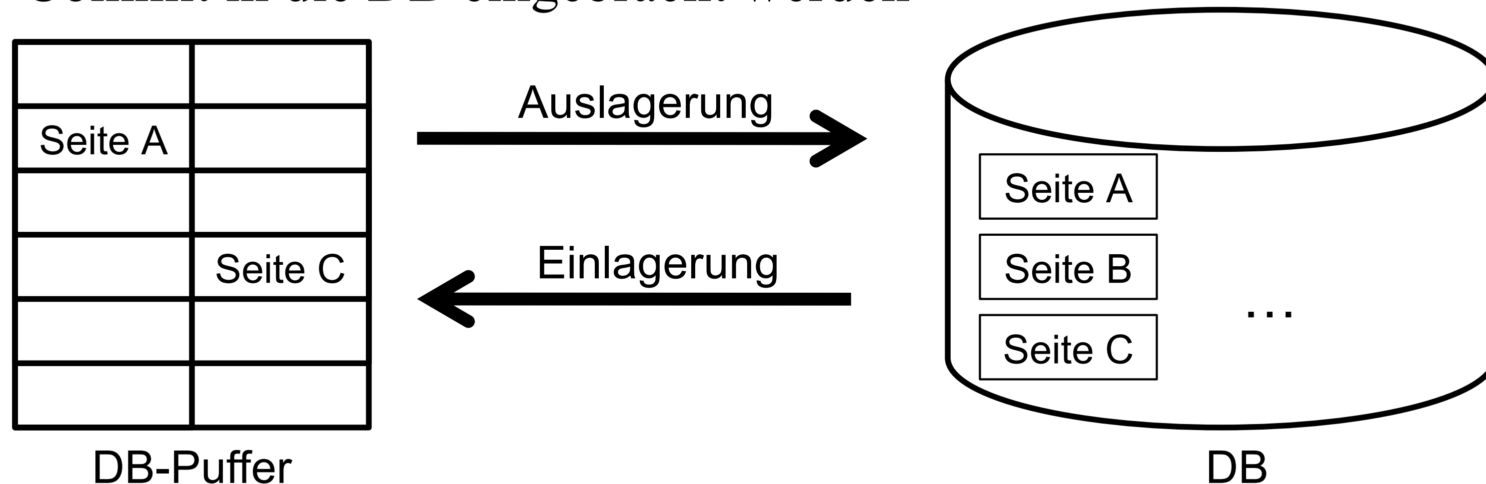
- Behandlung von Medienfehlern

  DB-Archiv + Log-Archiv → DB

Abhängigkeiten zu Systemkomponenten

Die Speicherhierarchie

- i.d.R. besteht die Speicherhierarchie bei DBMS aus zwei Ebenen
  - DBMS-Puffer (Hauptspeicher) [kurz: DB-Puffer]
  - DB (Hintegrundspeicher)
- Im laufenden Betrieb werden die Operationen der einzelnen TAs im DB-Puffer ausgeführt
- Die DB muss gemäß dem ACID-Prinzip transaktionskonsistent gehalten werden, d.h. Änderungen durch TAs müssen nach dem Commit in die DB eingebracht werden



Abhängigkeiten zu Systemkomponenten

Abhängigkeiten

- Aus der zweistufigen Speicherhierarchie ergeben sich insbesondere Abhängigkeiten der Recovery/des Loggings zur Speicherverwaltung
  - DB-Puffer ist begrenzt => was passiert, wenn Puffer voll?

    => Pufferverwaltung (Verdrängungsstrategien)
  - Wann schreibe ich Änderungen in die Datenbank?

    => Pufferverwaltung (Ausschreibestrategien)
  - Wie schreibe ich die Änderungen aus?

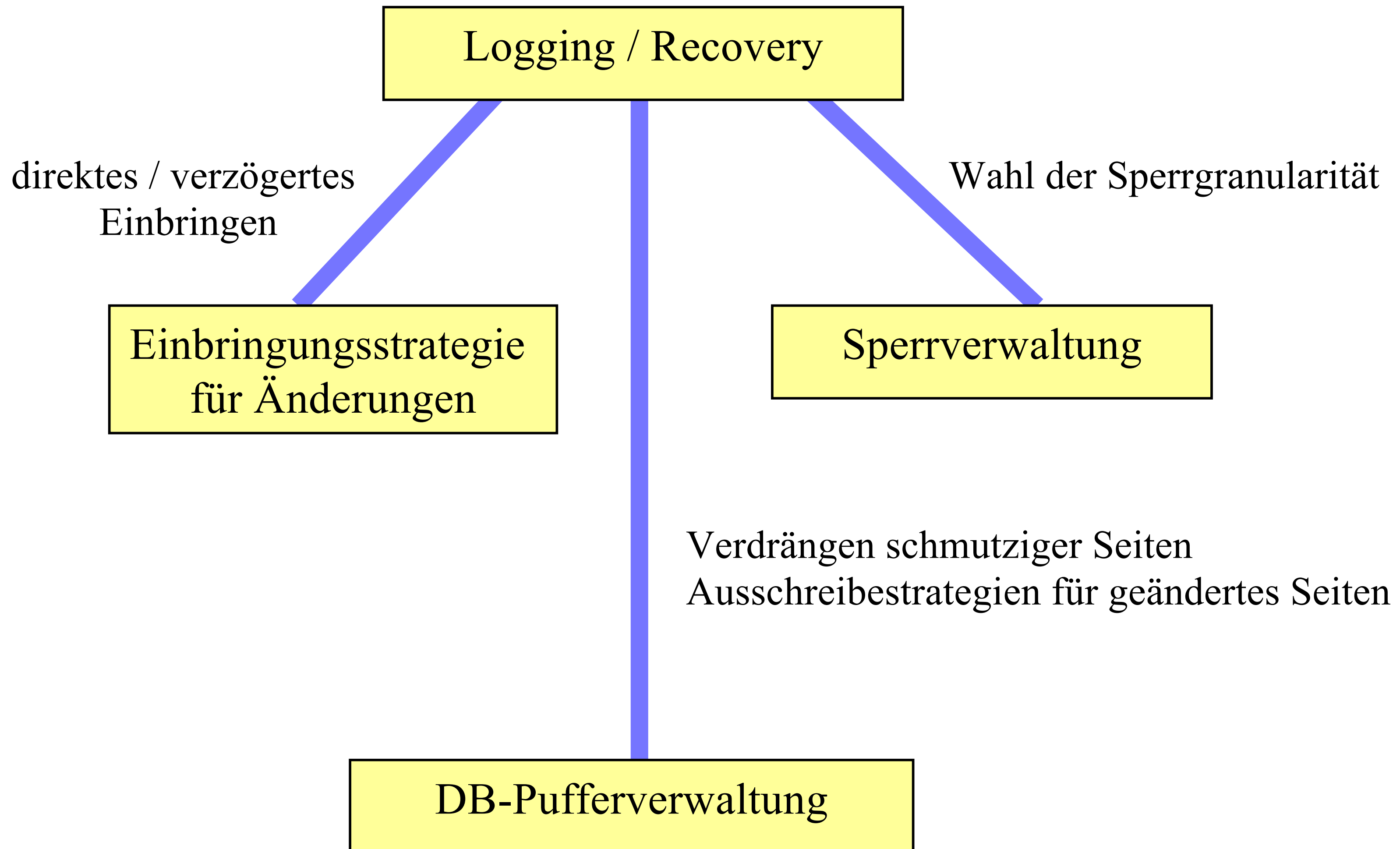
    => HGS-Verwaltung (Einbringungsstrategien)
    (HGS: Hintergrundspeicher)

  => Abhängig davon: wann schreibe ich Log-Datei auf Platte?

- Zudem besteht eine Abhängigkeit der Recovery/des Loggings zur Sperrverwaltung (bei pessimistischer Synchronisation)

Abhängigkeiten zu Systemkomponenten

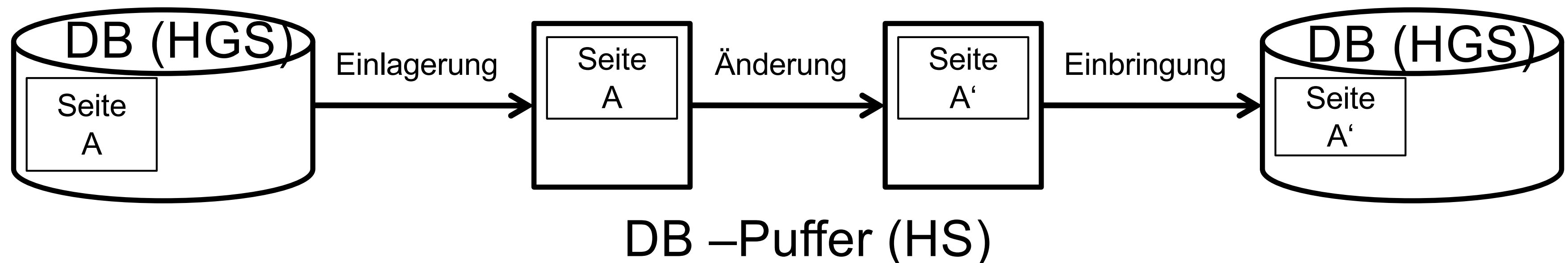
Schematischer Überblick der Abhängigkeiten



Abhängigkeiten zu Systemkomponenten

Einbringungsstrategie: Direktes Einbringen (NonAtomic, Update-in-Place)

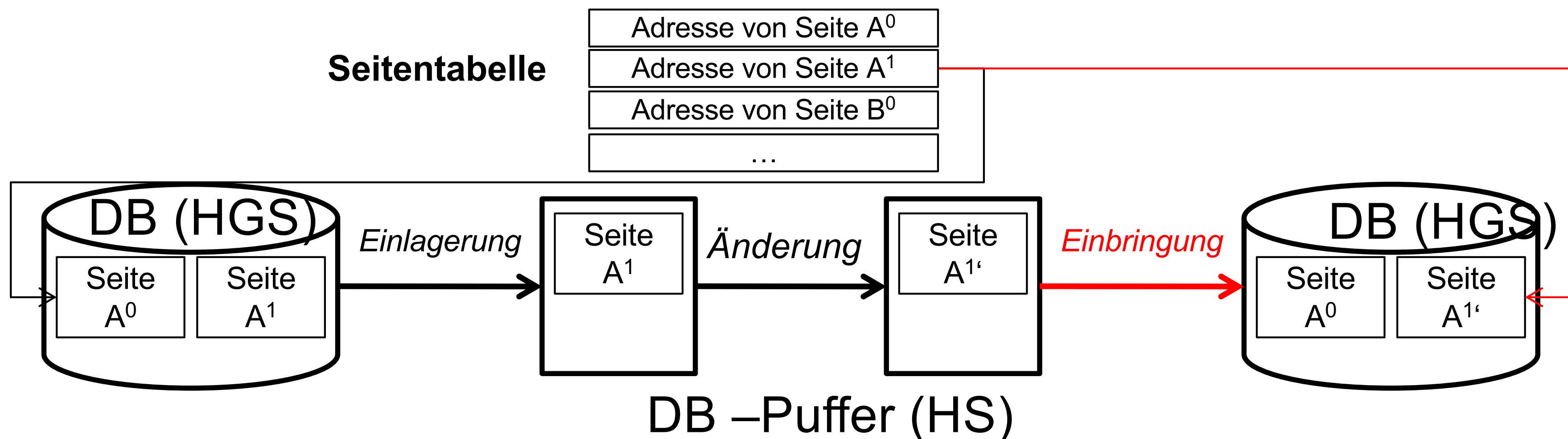
- Jede Seite (Block) hat eine (fixe) Speicheradresse auf der Platte.
- Geänderte Seiten werden immer auf ihren Block zurück geschrieben, d.h. der alte Inhalt der Seite in der DB wird dabei überschrieben.
- Ausschreiben einer Seite ist dadurch gleichzeitig Einbringen in die permanente DB (es gibt keinen Zwischenspeicher für Seiten).
- Es ist nicht möglich mehrere Seiten atomar einzubringen, d.h. Unterbrechungsfreiheit des Einbringens kann nicht garantiert werden (daher: NonAtomic).
- Dies ist die gängigste Methode in heutigen DBMS.
- UNDO-Informationen müssen explizit (Log-Datei) gespeichert werden



Abhängigkeiten zu Systemkomponenten

Einbringungsstrategie: Indirektes Einbringen (Atomic)

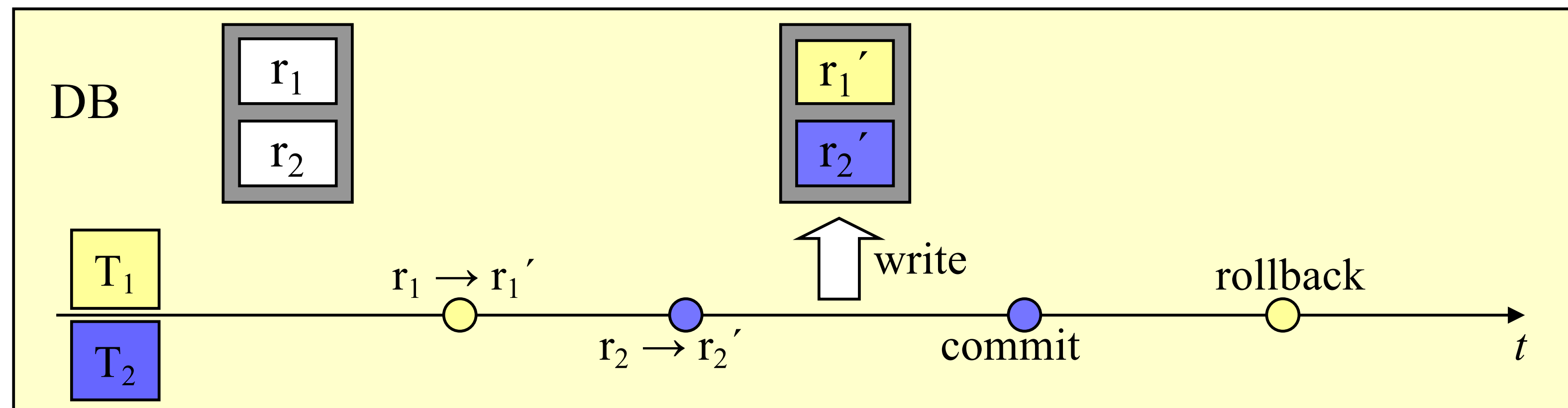
- Geänderte Seite wird in **separaten** Block auf Platte geschrieben.
  - Twin-Block-Verfahren: jede Seite hat zwei Blöcke auf der Platte.
  - Schattenspeichertechnik: nur modifizierte Seiten haben zwei Blöcke.
- Atomares Einbringen mehrerer geänderter Seiten ist durch **Umschalten** von Seitentabellen möglich (daher: **Atomic**).
- Alte Versionen der Objekte bleiben erhalten, d.h. es muss **keine UNDO-Information** explizit gespeichert werden.



Abhängigkeiten zu Systemkomponenten

Einfluss der Sperrgranularität

- Log-Granularität muss kleiner oder gleich der Sperrgranularität sein, sonst Lost Updates möglich
- D.h. Satzsperren erzwingen feine Log-Granulate
- Beispiel für Problem bei "Satzsperren mit Seitenlogging"



- T1, T2 ändern die Datensätze r1, r2, die auf derselben DB-Seite liegen
- Die Seite wird in die DB zurück geschrieben, T2 endet mit COMMIT
- Falls T1 zurückgesetzt wird, geht auch die Änderung r2 —> r2' verloren
- Lost Update, d.h. Verstoß gegen die Dauerhaftigkeit des COMMIT

Abhängigkeiten zu Systemkomponenten

Pufferverwaltung: Verdrängungsstrategien

- Ersetzung schmutziger Seiten im Puffer (Wohin werden Seiten verdrängt? Warum nur schmutzige Seiten?)
  - Seite ist schmutzig wenn: $\text{Seite}_{\text{Puffer}} \neq \text{Seite}_{\text{DB}}$
  - **No-Steal**
    - Schmutzige Seiten dürfen nicht aus dem Puffer entfernt werden, bzw. von einer anderen Seite ersetzt werden.
    - DB enthält keine Änderungen nicht-erfolgreicher TAs.
    - **UNDO**-Recovery der DB ist **nicht erforderlich**.
    - Probleme bei langen Änderungs-TAs, da große Teile des Puffers blockiert werden => Einschränkung der Parallelität.
  - **Steal**
    - Schmutzige Seiten dürfen jederzeit ersetzt und in die DB eingebracht werden.
    - DB kann unbestätigte Änderungen enthalten.
    - **UNDO**-Recovery der DB ist **erforderlich.**
    - effektivere Puffernutzung bei langen TAs mit vielen Änderungen.

Abhängigkeiten zu Systemkomponenten

Pufferverwaltung: Ausschreibestrategien (EOT-Behandlung)

- **Wann** werden Änderungen in die DB eingebracht?
  - **Force**
    - Alle geänderte Seiten werden spätestens bei EOT (vor COMMIT) in die DB geschrieben.
    - keine **REDO**-Recovery erforderlich bei Systemfehler.
    - hoher I/O-Aufwand, da Änderungen jeder TA einzeln geschrieben werden.
    - Vielzahl an Schreibvorgängen führt zu schlechteren Antwortzeiten, länger gehaltenen Sperren und damit zu mehr Sperrkonflikten.
    - Große DB-Puffer werden schlecht genutzt.
  - **No-Force**
    - Änderungen können auch erst nach dem COMMIT in die DB geschrieben werden.
    - Beim COMMIT werden lediglich REDO-Informationen in die Log-Datei geschrieben.
    - **REDO**-Recovery erforderlich bei Systemfehler.
    - Änderungen auf einer Seite von mehreren TAs können gesammelt werden.

Abhängigkeiten zu Systemkomponenten

Kombination:

	No-Steal	**Steal**
Force	kein *UNDO*, kein *REDO* (nicht für Update-in-Place)	*UNDO*, aber kein *REDO*
No-Force	kein *UNDO*, aber *REDO*	*UNDO* und *REDO*

- Bewertung **Steal / No-Force**
  - erfordert zwar **UNDO** als auch **REDO**, ist aber allgemeinste Lösung.
  - beste Leistungsmerkmale im Normalbetrieb.
- Bewertung **No-Steal / Force**
  - optimiert den Fehlerfall auf Kosten des Normalfalls (sehr teures COMMIT).
  - für Update-in-Place nicht durchführbar:
    - wegen No-Steal dürfen Änderungen erst nach COMMIT in die DB gelangen, was jedoch Force widerspricht (No-Steal → No-Force)
    - wegen Force müssten Änderungen vor dem COMMIT in der DB stehen, was bei Update-in-Place unterbrochen werden kann, UNDO wäre nötig (Force → Steal)

Abhängigkeiten zu Systemkomponenten

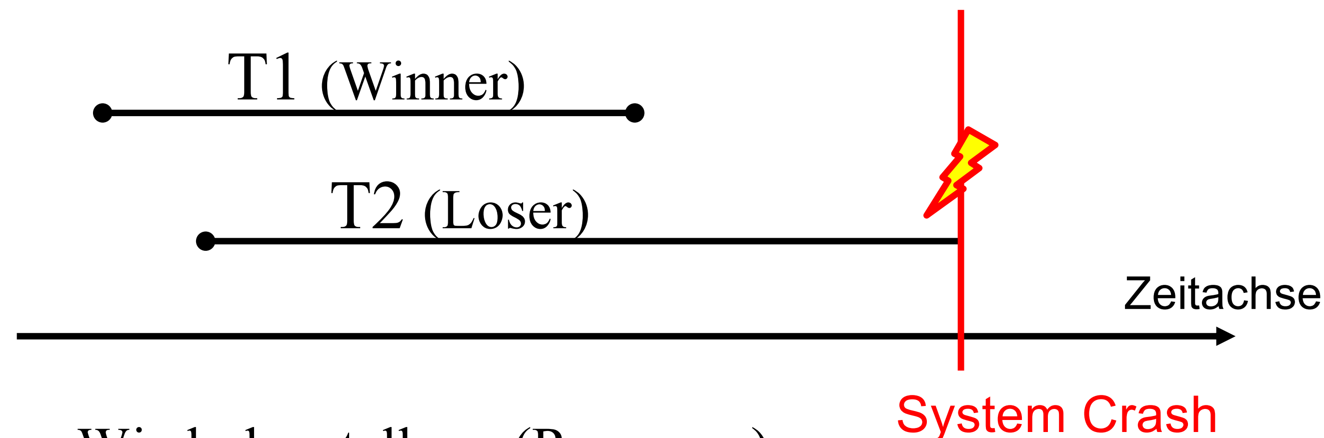
WAL-Prinzip und COMMIT-Regel

- WAL-Prinzip (Write-Ahead-Log)
  - UNDO-Information (z.B. BFIM) muss **vor Änderung** der DB im Protokoll stehen.
  - Wichtig, um schmutzige Änderungen rückgängig zu machen.
  - Nur relevant für Steal.
  - Wichtig bei direktem Einbringen.
- COMMIT-Regel (Force-Log-at-Commit)
  - REDO-Information (z.B. AFIM) muss **vor** dem **COMMIT** im Protokoll stehen.
  - Voraussetzung für Crash-Recovery bei No-Force.
  - Erforderlich für Geräte-Recovery (auch bei Force).
  - Gilt für direkte und indirekte Einbringstrategien gleichermaßen.

Christian-Albrechts-Universität zu Kiel

Recovery (Warmstart)

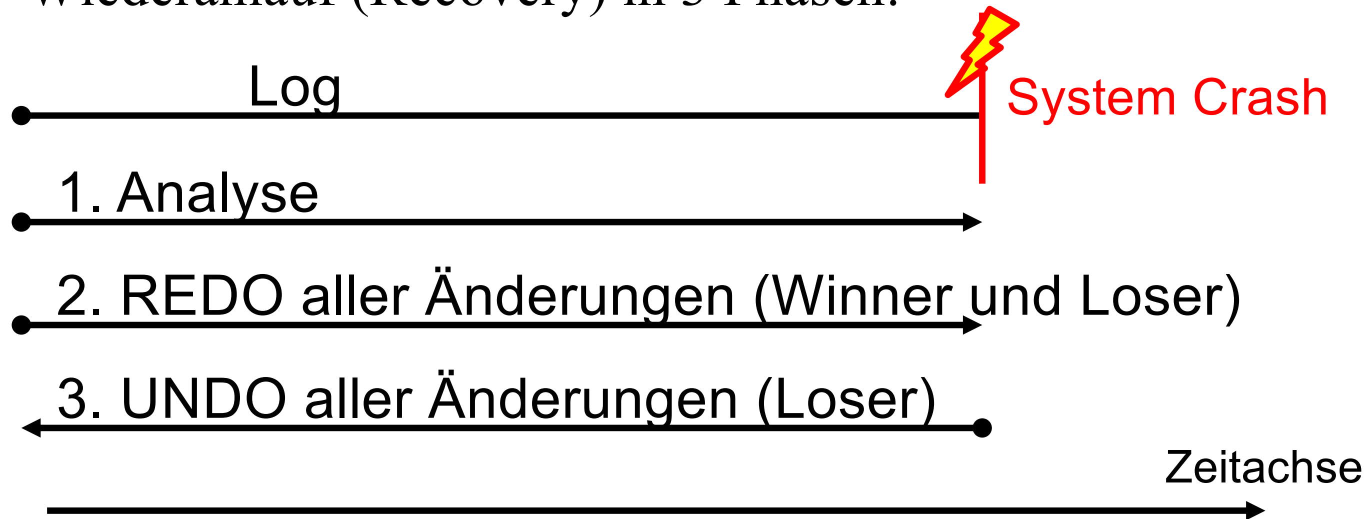
- Annahme: Systemfehler (Fehler mit Verlust des Hauptspeichers)
- Unterscheidung von zwei Transaktionstypen, die von dem Systemfehler betroffen sind:
  - T1: Abgeschlossene Transaktionen die noch nicht in der DB festgeschrieben sind.
  - T2: Zum Zeitpunkt des Absturzes noch nicht abgeschlossene Tas



  - Wiederherstellung (Recovery):
    - Transaktionen (sog. **Winner**) vom Typ T1 müssen vollständig nachvollzogen werden (REDO).
    - Transkationen (sog. **Loser**) vom Typ T2 müssen rückgängig gemacht werden (UNDO).

Recovery (Warmstart)

- Wiederanlauf (Recovery) in 3 Phasen:



Phasen:

1. Analyse: Temporäre Log-Datei wird analysiert
   → Indentifizierung von Winner und Loser.
2. Wiederholung der Historie: Alle protokollierten Änderungen werden nachgeführt.
3. UNDO der Loser: Die Änderungsoperationen der Loser-Transaktionen werden in umgekehrter Reihenfolge rückgängig gemacht.

Transaktionsunterstützung in SQL

- (Standardmäßig) keine explizite Anweisung für den Begin einer Transaktion (BOT).
- Explizites Ende muss spezifiziert sein, entweder mit *commit* oder *rollback*.
- Spezifikation von Transaktionen in SQL:
  - TAs können Merkmale zur Transaktionskontrolle zugewiesen werden mit

    SET TRANSACTION …

  - **Merkmale:**
    - **Zugriffsmodus** (read only, read write)
    - **Diagnosebereichsgröße**
    - **Isolationsstufe** (read uncommitted/committed, serializable)

Transaktionsunterstützung in SQL

- Zugriffsmodus (Access Mode):
  - Werte:
    - READ ONLY — Nur lesenden Datenzugriff
    - READ WRITE — Erlaubt update, insert, delete, und create
  - Defaultwert ist READ WRITE (außer bei Isolationsstufe READ UNCOMMITTED, s.u.)
  - Wichtig für Parallelisierung von Transaktionen.

- Diagnosebereichsgröße (Diagnostic Area Size):
  - Diagnose liefert dem Benutzer Feedback-Informationen für Fehler und/oder Ausnahmen der letzten SQL Anweisung.
  - Mit Operation DIAGNOSTICS SIZE *n* wird die Anzahl von Bedignungen im Diagnosebereich spezifiziert.

Transaktionsunterstützung in SQL

- Isolationsstufe (Isolation Level):
  - Kontrolle über den Grad der Isolation einzelner TAs.
  - Spezification über Operation ISOLATION LEVEL *stufe*
  - Stufen (aufsteigend):

Stufe	Beschreibung
• READ UNCOMMITTED	keine Lese-Einschränkung
• READ COMMITTED	Nur Lesen von bestätigten (committeten) Objekten.
• REPEATABLE READ	Nach dem Lesen werden Änderungen von Objekten durch andere TAs nicht zugelassen.
• SERIALIZABLE	Maximale Zugriffs-einschränkung.

  - Stufe SERIALIZABLE entspricht nicht (exakt) der Definition von serialisierbaren Transaktionen (s. Kap. 9a) !!!

Transaktionsunterstützung in SQL

- Mögliche Verletzungen der Isolationsstufen:

Isolationsstufe	Art der Verletzung		
	Dirty Read	**Nonrepeatable Read**	**Phantom-problem**
READ UNCOMMITTED	JA	JA	JA
READ COMMITTED	NEIN	JA	JA
REPEATABLE READ	NEIN	NEIN	JA
SERIALIZABLE	NEIN	NEIN	NEIN

  - JA: Verletzung möglich
  - NEIN: Verletzung nicht möglich

Transaktionsunterstützung in SQL

- Beispiel einer Transaktion in SQL:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
        READ WRITE
        DIAGNOSTICS SIZE 5
        ISOLATION LEVEL SERIALIZABLE;

EXEC SQL INSERT INTO
        ANGESTELLTER (VNAME, NNAME, SSN, ANR, GEHALT)
        VALUES (‘Robert‘, ‘Smith‘, ‘991004321‘, 2, 35000);

EXEC SQL UPDATE ANGESTELLTER
        SET GEHALT = GEHALT * 1.1 WHERE ANR = 2;

EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;

THE_END: …;
```

Im Fehlerfall wird die gesamte Transaktion rückgängig gemacht.