FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksysteme I
Datenbanken und Informationssysteme

Prof. Dr. Viktor Leis

WS 2019/2020

Professur für Datenbanken und Informationssysteme

Transaktionen

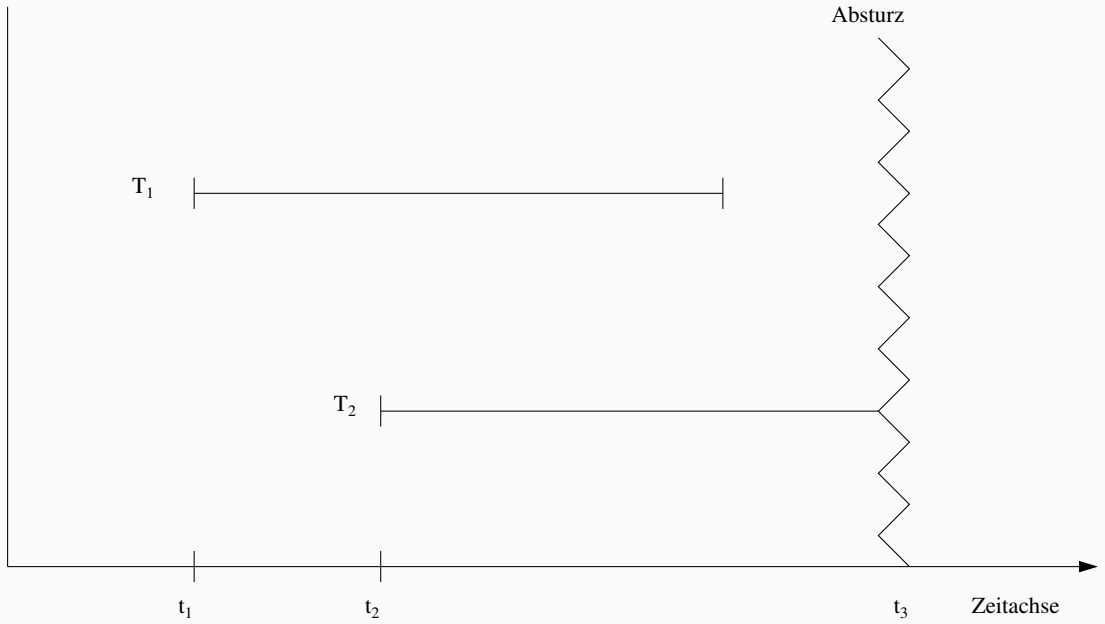
Beispiel für Transaktion (TA)

- Überweise Geld von Konto A nach Konto B:
  - Lies den Kontostand von A in die Variable a: **read**(A,a);
  - Reduziere den Kontostand um EURO 50,–: $a := a - 50$;
  - Schreibe den neuen Kontostand in die Datenbasis: **write**(A,a);
  - Lies den Kontostand von B in die Variable b: **read**(B,b);
  - Erhöhe den Kontostand um EURO 50,–: $b := b + 50$;
  - Schreibe den neuen Kontostand in die Datenbasis: **write**(B,b);

Operationen

- begin of transaction (BOT):
  - Kennzeichnet den Beginn einer Transaktion
- commit:
  - Erfolgreiche Beendigung einer Transaktion
  - Dauerhafte Einbringung aller Änderungen in die Datenbasis
- abort:
  - Selbstabbruch der Transaktion, erfolglose Beendigung
  - Zurücksetzen der Datenbasis in den Zustand vor Beginn der Transaktion

Systemabsturz



- Änderungen der zum Zeitpunkt t_3 abgeschlossenen TA T_1 müssen in der Datenbasis vorhanden sein
- Änderungen der zu t_3 noch nicht abgeschlossenen TA T_2 müssen vollständig aus der Datenbasis entfernt werden (Durchführung von T_2 durch Neustart)

ACID

- Transaktionen sollten ACID-Eigenschaften haben
- ACID:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

ACID (2)

- Atomicity (Atomarität): “alles oder nichts”, d.h. entweder werden all Operationen einer TA ausgeführt oder keine
- Consistency (Konsistenz): wenn eine TA auf einem konsistenten Datenbankzustand aufsetzt, ist dieser nach Beendigung der TA immer noch konsistent
- Isolation: nebenläufig ausgeführte Transaktionen dürfen keine Seiteneffekte aufeinander haben
- Durability (Dauerhaftigkeit): alle mit commit festgeschriebenen Änderungen müssen bestehen bleiben (selbst bei Systemabsturz)

Transaktionen und SQL

- **start transaction** oder **begin**:
  - Starte Transaktion
  - Laut SQL Standard nicht explizit nötig
  - Achtung: In PostgreSQL und viele andere Systeme benutzten standardmäßig “auto-commit”, d.h. jedes SQL Statement wird als eigene Transaktion aufgefasst.
- **commit**:
  - Beende Transaktion und schreibe Änderungen fest
  - funktioniert nur, wenn keine anderen Fehler aufgetreten sind (z.B. Konsistenzverletzung durch Transaktion)
- **rollback**:
  - Beende Transaktion und setze alle Änderungen zurück
  - Anders als **commit** muss DBMS die ”erfolgreiche” Ausführung von **rollback** immer garantieren können

Transaktionen und SQL(2)

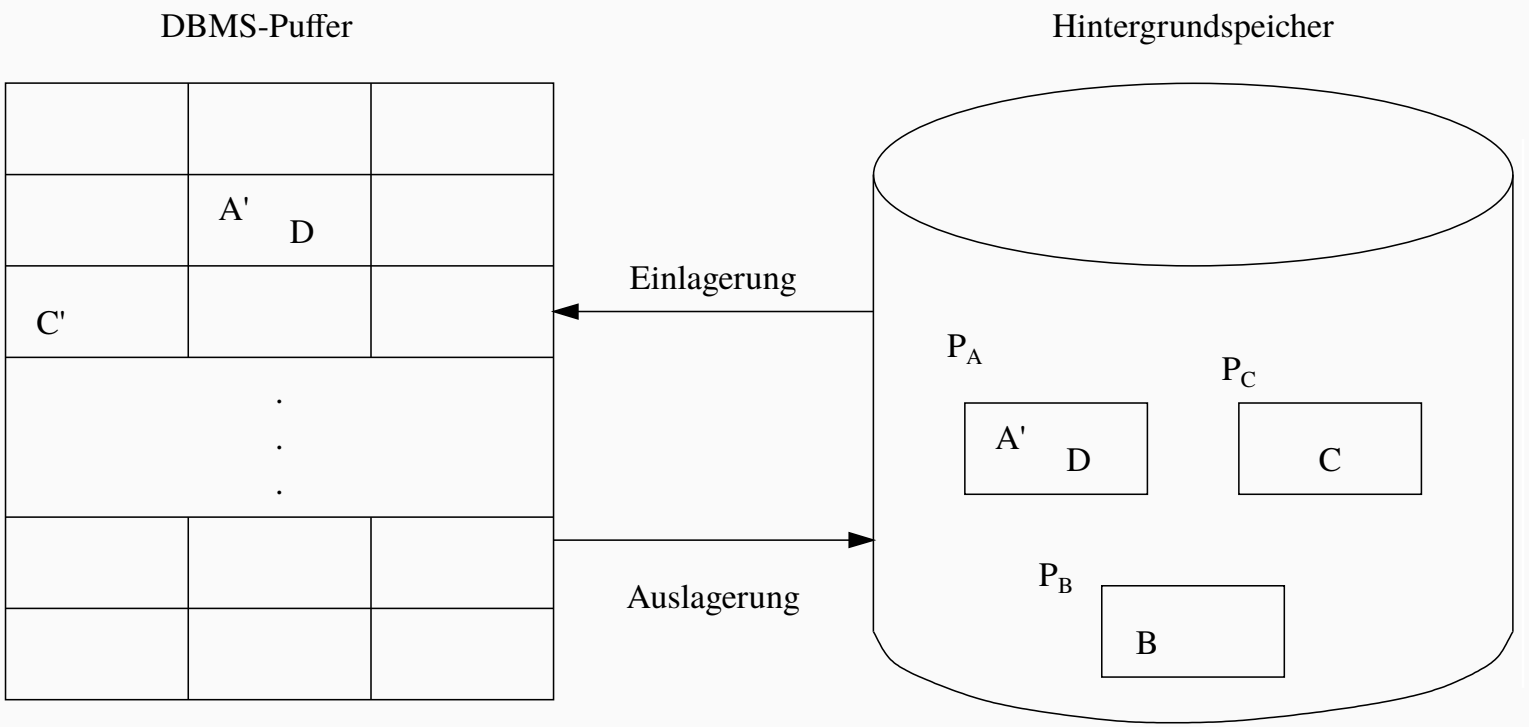
```
insert into Vorlesungen
    values (5275, 'Kernphysik', 3, 2141);

insert into Professoren
    values (2141, 'Meitner', 'C4', 205);

commit work
```

- **commit** nach dem ersten **insert** könnte nicht erfolgreich durchgeführt werden, da zu diesem Zeitpunkt referentielle Integrität verletzt
- In PostgreSQL muss hierzu am Anfang der Transaktion **set constraints all deferred** und **begin** ausgeführt werden.

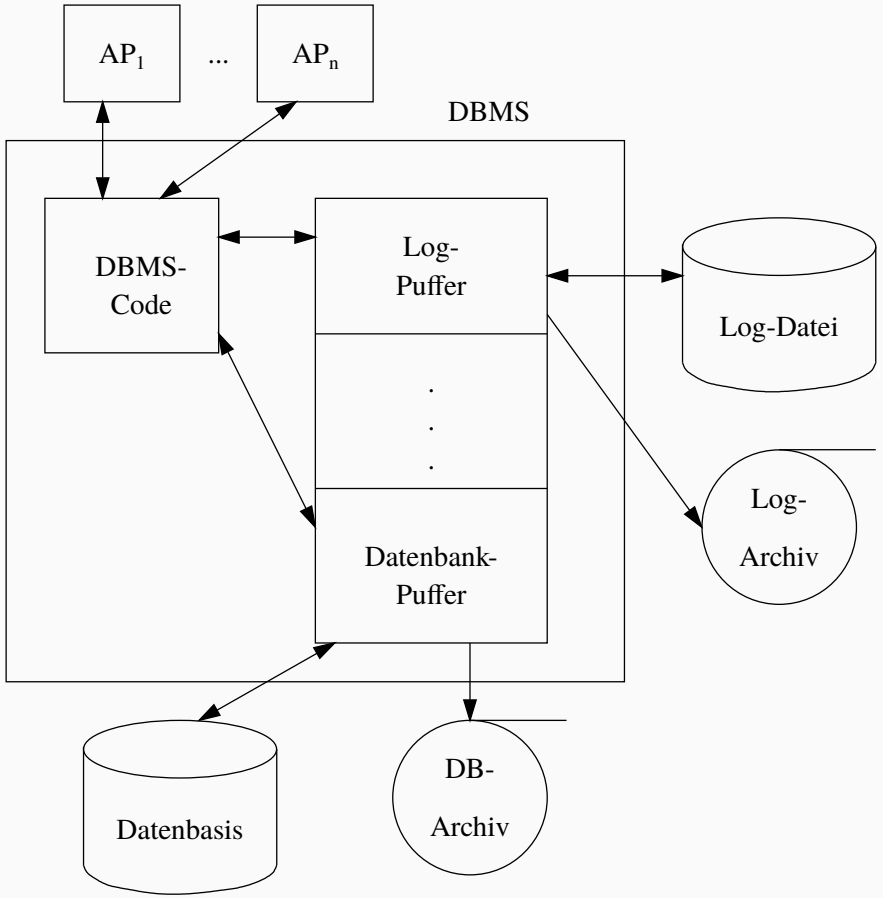
Arten von Fehlern

- Wichtige Aufgabe eines DBMS ist das Verhindern von Datenverlust durch Systemabstürze
  - Abbruch einer Transaktion (z.B. Softwarefehler in der Anwendung)
  - Verlust des Hauptspeichers (z.B. Softwarecrash DBMS oder Betriebssystem)
  - Hintergrundspeicher fällt aus (z.B. Festplatte defekt, Datenbankserver explodiert)

Rückblick: DBMS-Puffer



Schreiben der Log-Daten



- Die Log-Information wird zweimal geschrieben
  - Log-Datei für schnellen Zugriff
  - Log-Archiv für Redundanz

Recovery

- Die zwei wichtigsten Recovery-Mechanismen sind:
  - Sicherungspunkte (Backups)
  - Log-Dateien

Recovery (2)

- Ein *Sicherungspunkt* ist ein Schnappschuss des Datenbankinhalts zu einem bestimmten Zeitpunkt
- In einer *Log-Datei* werden alle Änderungen an der Datenbasis mitprotokolliert
- Sicherungspunkte und Log-Dateien sollten nicht auf der gleichen Maschine gespeichert werden

Schreiben der Log-Daten

- **W**rite **A**head **L**og-Prinzip (WAL-Prinzip)
  - Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle ”zu ihr gehörenden” Log-Einträge ausgeschrieben werden
  - Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das temporäre und das Log-Archiv ausgeschrieben werden

Abbruch einer Transaktion

- Kann durch Fehler in Anwendung ausgelöst werden (oder explizit mit **rollback**)
- Änderungen der Transaktion werden aus der Datenbasis entfernt indem die in der Log-Datei protokollierten Änderungen “rückwärts” rückgängig gemacht werden (aus insert wird delete und umgekehrt)
- Log-Datei muss alle dazu nötigen Informationen enthalten (z.B. beim delete wird das alte Tupel ins Log geschrieben)

Verlust des Hauptspeichers

- Meist durch Softwarecrash verursacht
- Neustart und Wiederanlauf des Systems
- Mit Hilfe der Log-Datei muss sichergestellt werden:
  - abgeschlossene TAs müssen erhalten bleiben
  - noch nicht abgeschlossene TAs müssen zurückgesetzt werden

Festplattenausfall

- Meist durch Hardwaredefekt verursacht
- Mit Hilfe eines Sicherungspunktes und der Log-Datei oder des Log-Archivs kann die Datenbasis wiederhergestellt werden:
    1. Einspielen des Backups
    2. Ausführung der Operationen aus der Log-Datei bzw. des Log-Archivs

Mehrbenutzersynchronisation

- Alle TAs strikt seriell (also nacheinander) auszuführen ist sicher, aber langsam
- Oft werden Systemressourcen nicht voll ausgenutzt, da eine TA auf Plattenzugriff oder Benutzereingabe wartet
- Diese TA blockiert dann alle anderen TAs
- Um Systemressourcen auszunutzen bietet sich Nebenläufigkeit an

Mehrbenutzersynchronisation (2)

- Nicht abgesicherte Nebenläufigkeit kann aber zu folgenden Problemen führen:
  - lost update
  - dirty read
  - non-repeatable read
  - phantom problem

Dirty Read

T_1	T_2
bot	
$\hookrightarrow$	bot
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	$\hookleftarrow$
$w_1(y)$	
commit	
$\hookrightarrow$	abort

T_1 liest einen Wert für x der so nicht gültig ist!

Lost Update

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$r_2(x)$
$w_1(x)$	$\hookleftarrow$
$\hookrightarrow$	$w_2(x)$
commit	$\hookleftarrow$
$\hookrightarrow$	commit

Das Ergebnis der Transaktion T_1 ist verlorengegangen!

Non-Repeatable Read

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$w_2(x)$
	commit
$r_1(x)$	$\hookleftarrow$
...	

T_1 liest x zweimal mit verschiedenem Ergebnis!

Phantom Problem

T_1	T_2
bot	
select count(*) from R;	
$\hookrightarrow$	bot
	insert into R ...;
	commit
select count(*) from R;	$\hookleftarrow$
...	

T_1 findet ein weiteres Tupel beim Abarbeiten der zweiten Anfrage!

Vermeidung der Probleme

- Im Idealfall sollten alle diese Probleme vermieden werden
- Man muss dabei allerdings einen Kompromiss zwischen Performanz und Genauigkeit schließen
- Je mehr Sicherheit, desto langsamer wird die Ausführung
- Über die *Isolation Levels* kann man DBMS mitteilen, welche Sicherheit erwünscht ist

Transaktionen und SQL

- Festsetzen von Eigenschaften einer TA:

  set transaction isolation level *Stufe, Zugriffsmodus*

- Folgende Stufen für den Isolation Level sind möglich:
  - read uncommitted
  - read committed
  - repeatable read
  - serializable
- Achtung: in vielen Systemen ist read committed die Standardeinstellung
- Mögliche Zugriffsmodi:
  - read only
  - read write

Transaktionen und SQL (2)

	dirty read	lost update	nonrep. read	phant. probl.
read uncommitted				
read committed	√			
repeatable read	√	√	√	
serializable	√	√	√	√

default in PostgreSQL: read committed

Transaktionen und SQL (3)

- “read only” sagt dem DBMS, dass eine TA nur Leseoperationen enthält
- Das hat Auswirkungen auf die Performanz
- Nebenläufiges Ausführen von TAs die nur lesen ist unkritisch, d.h. beliebig vieler solcher TAs können völlig uneingeschränkt parallel laufen
- Erst wenn eine TA dazukommt, die auch schreibt müssen Vorkehrungen getroffen werden

Implementierung von Mehrbenutzersynchronisation

- Die meisten Datenbanksysteme verwenden Sperren (“Locks”) um Mehrbenutzersynchronisation zu implementieren
- Jede Zeile hat eine Sperre, vor dem Zugriff wird die jeweilige Sperre angefordert
- Falls die Sperre schon “vergeben” ist, muss gewartet werden
- Beim Ende der Transaktion können alle Sperren wieder freigegeben werden
- Details unterscheiden sich je nach isolation level (z.B. bei read committed werden nur beim Schreiben Sperren gehalten)
- Es gibt auch Verfahren, die nicht auf Sperren basieren

Zusammenfassung

- Transaktionen sind einer der wichtigsten Gründe für den Einsatz von Datenbanksystemen
- Kapselt nebenläufigen Zugriff auf Datenbank, stellt Persistenz sicher
- Benutzung ist sehr einfach, Implementierung ist komplex (mehr dazu in Datenbanksysteme II):
  - Atomicity
  - Consistency: Prüfung der Konsistenzbedingungen (beim commit)
  - Isolation: Mehrbenutzersynchronisation durch Sperren
  - Durability: Log-Datei, Sicherungspunkte