Inhalt [1]

Inhalt [2]

Allgemeines

Das Konzept der Transaktion wird oftmals als einer der größten Beiträge der Datenbankforschung für andere Informatikbereiche angesehen.

Unter **Transaktion** versteht man die „Bündelung“ mehrerer Datenbankoperationen, die in einem Mehrbenutzersystem ohne unerwünschte Einflüsse durch andere Transaktionen oder durch unerwartet eintretende Fehlersituationen fehlerfrei ausgeführt werden sollen.

Aus Sicht des Datenbankbenutzers ist eine Transaktion eine Arbeitseinheit in einer Anwendung, die eine bestimme Funktion erfüllt.

Aus Sicht des Datenbankverwaltungssystems ist eine Transaktion eine Folge von Datenverarbeitungsbefehlen, die die Datenbank von einem konsistenten Zustand in einen anderen überführt. Wesentlich dabei ist, dass diese Folge logisch ununterbrechbar d.h. atomar ausgeführt wird.

Beispiel

Der Transfer von DM 50,- von Konto A nach Konto B:

1)	Lese Kontostand von A in Variable a:	**read(A,a);**
2)	Reduziere den Kontostand um 50:	**a := a -50;**
3)	Schreibe den neuen Kontostand in die Datenbasis:	**write(A,a);**
4)	Lese den Kontostand von B in die Variable b:	**read(B,b);**
5)	Erhöhe den Kontostand um 50:	**b := b + 50;**
6)	Schreibe den neuen Kontostand in die Datenbasis:	**write(B,b);**

Eigenschaften von Transaktionen (ACID-Paradigma)

- **Atomicity (Atomarität):** Eine Transaktion ist die kleinste, nicht mehr zerlegbare Einheit. Entweder werden alle Änderungen einer Transaktion in der Datenbank festgeschrieben oder gar keine (alles-oder-nichts-Prinzip)
- **Consistency:** Eine Transaktion hinterlässt nach Beendigung einen konsistenten Zustand (d.h. alle Konsistenzbedingungen wie z.B. referentielle Integrität sind eingehalten), andernfalls wird sie komplett zurückgesetzt (Atomarität). Zwischenzustände können inkonsistent sein.
- **Isolation:** Parallel ausgeführte Transaktionen beeinflussen sich nicht. Jede Transaktion muss so ausgeführt werden, als wäre sie die einzige, die während der gesamten Ausführungszeit aktiv ist.
- **Durability (Dauerhaftigkeit):** Die Wirkung einer erfolgreich abgeschlossenen Transaktion muss dauerhaft im System erhalten bleiben.

Allgemeines

Das Konzept der Transaktion wird oftmals als einer der größten Beiträge der Datenbankforschung für andere Informatikbereiche angesehen.

Unter **Transaktion** versteht man die „Bündelung“ mehrerer Datenbankoperationen, die in einem Mehrbenutzersystem ohne unerwünschte Einflüsse durch andere Transaktionen oder durch unerwartet eintretende Fehlersituationen fehlerfrei ausgeführt werden sollen.

Grundlegende Anforderungen an eine Transaktionsverwaltung:

- **Recovery:** Behebung von eingetretenen, oft unvermeidbaren Fehlersituationen
- **Synchronisation** von mehreren gleichzeitig auf der Datenbank ablaufenden Transaktionen

Operationen auf Transaktionsebene

Aus Sicht eines DBMS sind ausschließlich die Operationen **read** und **write** einer Transaktion interessant. Für die Steuerung der Transaktionsverarbeitung sind noch weitere Operationen notwendig:

- **begin of transaktion (BOT)**
- **commit:** Damit wird die Beendigung einer Transaktion eingeleitet. Alle Änderungen in der Datenbank werden durch diesen Befehl festgeschrieben, d.h. sie werden dauerhaft in die Datenbank eingebaut.
- **abort:** Dieser Befehl führt zum Selbstabbruch der Transaktion. Das DBMS muss sicherstellen, dass die Datenbank in den Zustand vor den Beginn der Transaktion zurückgesetzt wird.

In neueren DBMS stehen vor allem für Transaktionen langer Dauer noch folgende Operationen zur Verfügung:

- **define savepoint:** Sicherungspunkt, auf den sich eine noch aktive Transaktion zurücksetzen lässt.
- **backup transaktion**: Zurücksetzen einer noch aktiven Transaktion auf den zuletzt angelegten Sicherungspunkt

Abschluss einer Transaktion

Erfolgreicher Abschluss:

BOT
op_1
...
op_n
commit

Als Operationen sind - wie schon erwähnt - nur Interaktionen mit der Datenbank relevant. Alle Operationen auf z.B. lokale Variablen sind nicht von Interesse.

Erfolgloser Abschluss:

BOT
op_1
...
op_j
abort

Transaktion wurde vom Benutzer oder der Anwendung bewusst abgebrochen.

BOT
op_1
...
op_k
~~~~~

Transaktion wurde durch einen unerwarteten Fehler abgebrochen. (z.B. Hardwaredefekt, Stromausfall)
~~~~~

Transaktionsverwaltung in SQL

- **Beginn:** In SQL werden Transaktionen implizit begonnen (beim Anmelden an die Datenbank, sofort nach dem Ende einer Transaktion beginnt die Nächste)
- **Ende:** Der Befehl **commit** beendet eine Transaktion.Es werden alle Änderungen festgeschrieben, sofern keine Konsistenzverletzungen oder andere Probleme aufgedeckt werden.
- **Abbruch:** Transaktionen werden mit dem Befehl **rollback** abgebrochen. Alle Änderungen werden zurückgesetzt.

Beispiel:

```
insert into Vorlesungen values (5275,'Kernphysik',3,2141);
insert into Professoren values (2141,'Meitner','C4'205);
commit;
```

Ein ‚commit' nach dem ersten ‚insert' könnte nicht erfolgreich abgeschlossen werden, da eine Konsistenzverletzung vorliegen würde - vorausgesetzt es wurde ein Foreign-Key zwischen dem Attribut ‚Professor' in ‚Vorlesungen' und der Tabelle Professoren definiert.

Zustandsübergänge einer Transaktion [1]

Eine Transaktion kann sich prinzipiell in einem der folgenden Zustände befinden:

- **potentiell:** Die Transaktion ist codiert und wartet darauf, in den Zustand ‚aktiv' zu wechseln. Diesen Übergang nennt man auch ‚inkarnieren'.
- **aktiv:** Die aktiven (derzeit rechnenden) Transaktionen konkurrieren untereinander um die Betriebsmittel, wie z.B. Prozessor, Hauptspeicher, Ausführung von Aktionen.
- **wartend:** Bei Überlastung des Systems kann die Transaktionsverwaltung einige aktive Transaktionen in den Zustand ‚wartend' verdrängen.
- **abgeschlossen:** Durch den ‚commit'-Befehl abgeschlossenen Transaktionen. Die Wirkung kann aber nicht gleich in der Datenbank festgeschrieben werden. Es müssen ev. noch Konsistenzbedingungen geprüft werden.
- **persistent:** Die Wirkung einer abgeschlossenen Transaktion sind dauerhaft in die Datenbank eingebracht - einer der zwei möglichen Endzustände einer Transaktionsverarbeitung.

Zustandsübergänge einer Transaktion [2]

- **gescheitert:** Transaktionen können aufgrund vielfältiger Ereignisse scheitern. Z.B. kann der Benutzer, die Anwendung selbst durch ein ‚abort' eine aktive Transaktion abbrechen; oder Systemfehler können zum Scheitern aktiver und wartender Transaktionen führen. Bei abgeschlossenen Transaktionen können Konsistenzfehler ein Scheitern veranlassen.
- **wiederholbar:** Einmal gescheiterte Transaktionen sind unter Umständen wiederholbar (wenn sie z.B. von der Transaktionsverwaltung selbst abgebrochen wurden - wegen Konfliktsituationen mit anderen Transaktionen). Die bisherigen Auswirkung dieser Transaktionen auf die Datenbasis müssen zurückgesetzt werden. Dann können sie wieder durch Neustarten aktiviert werden.
- **aufgegeben:** Eine gescheiterte Transaktion kann sich aber auch als ‚hoffnungslos' herausstellen. In diesem Fall wird ihre bisherige Wirkung zurückgesetzt und die Transaktionsverwaltung führt sie in den zweiten möglichen Endzustand - ‚aufgegeben'. Eine entsprechende Fehlermeldung wird dem Benutzer, der Anwendung zurückgeschickt.

Zustandsübergänge einer Transaktion [3]

Graphische Darstellung:

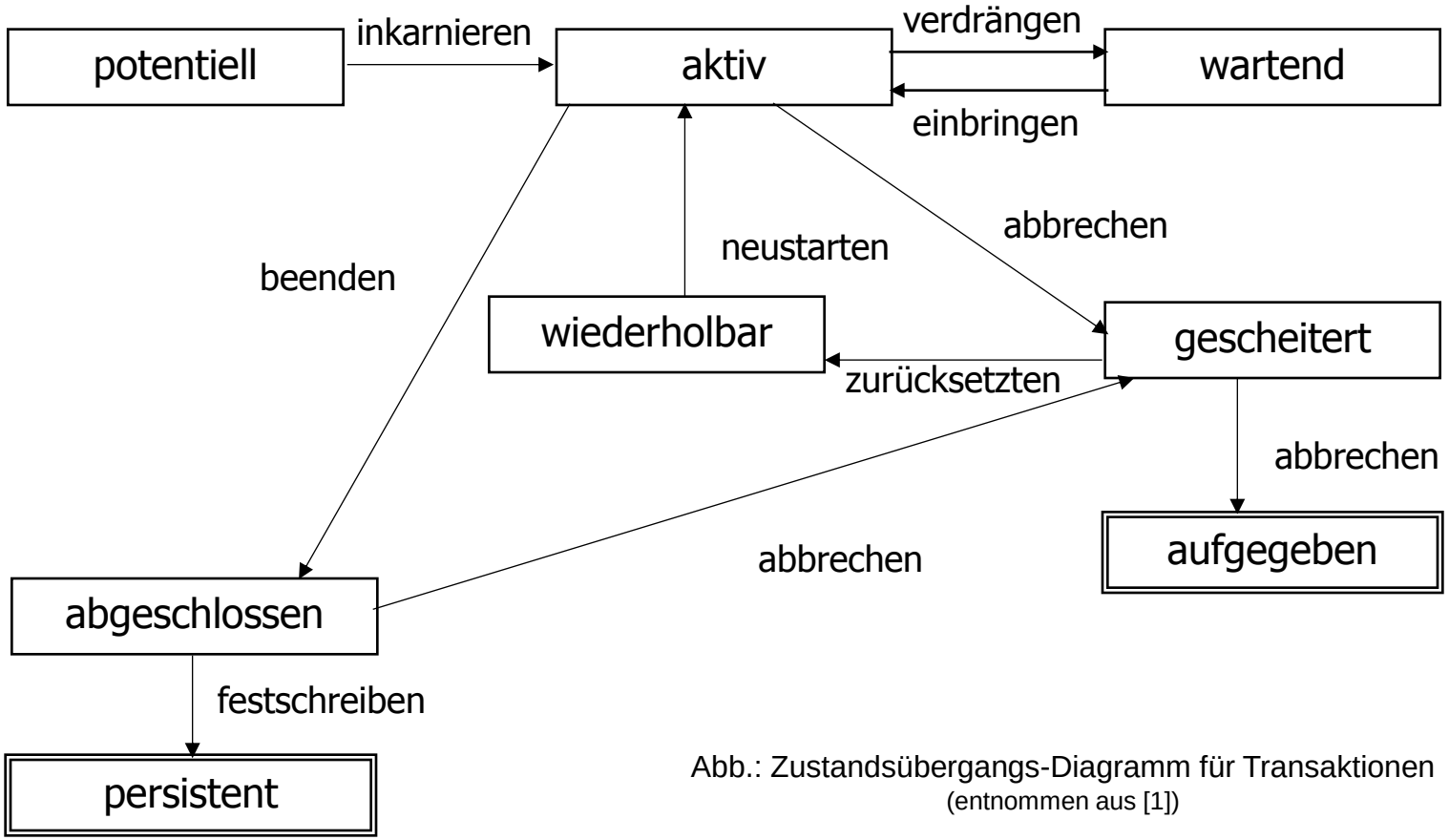


Abb.: Zustandsübergangs-Diagramm für Transaktionen
(entnommen aus [1])

Fehlerklassifikation [1]

Lokaler Fehler einer Transaktion

Diese Fehler führen zum Scheitern der entsprechenden Transaktion, beeinflussen aber nicht den Rest des Systems hinsichtlich der Datenbankkonsistenz.

- Fehler im Anwendungsprogramm
- expliziter Abbruch (abort) der Transaktion durch den Benutzer
- systemgesteuerter Abbruch durch die Transaktionsverwaltung um z.B. einen Deadlock zu beheben

Lokale Fehler treten relativ häufig auf und müssen von der Recovery-Komponente entsprechend schnell behoben werden. Diesen Vorgang bezeichnet man auch als ‚**lokales undo**‘.

Fehlerklassifikation [2]

Fehler mit Hauptspeicherverlust [1]

Ein Datenbankmanagementsystem bearbeitet Daten im sogenannten Datenbankpuffer im Hauptspeicher. Dieser ist in Seiten aufgeteilt:

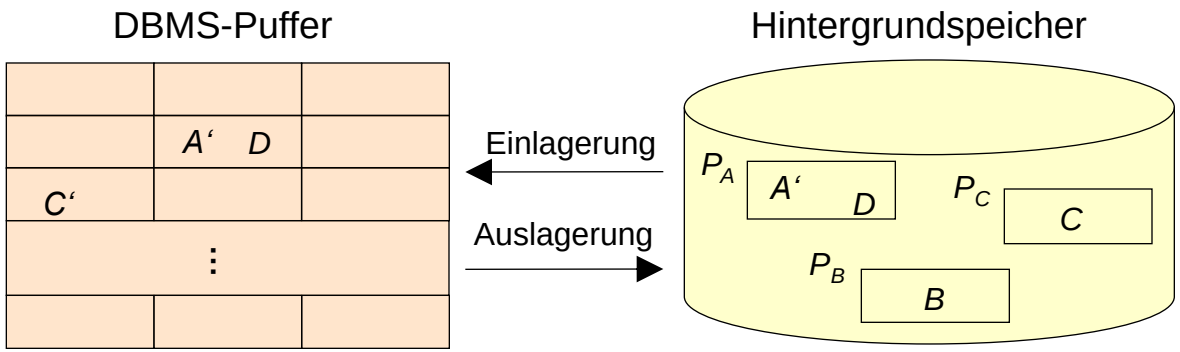


Abb.:
Schematische Darstellung der Speicherhierarchie
(entnommen aus [1])

In diesem Beispiel liegen die Datensätze A, B, C und D in den Seiten P_A, P_B und P_C . Eine Seite enthält normalerweise viele Datensätze, hier durch A und D in P_A symbolisiert. Beim Zugriff einer Transaktion auf ein Datenobjekt, das sich noch nicht im Puffer befindet, muss vorher die entsprechende Seite geladen (eingelagert) werden. Die geänderten Seiten werden dann „früher oder später“ in die Datenbasis zurückgeschrieben. C‘ stellt das geänderte Datenobjekt C dar. Es ist noch nicht auf den Hintergrundspeicher geschrieben, während dies bei A‘ bereits der Fall ist.

Fehlerklassifikation [3]

Fehler mit Hauptspeicherverlust [2]

Diese Art von Fehlern tritt hauptsächlich nach Abstürzen des Betriebssystems oder nach Stromausfall auf. Ein Recovery muss nun

- alle durch noch nicht abgeschlossene Transaktionen Änderungen, die bereits in den Hintergrundspeicher ausgelagert wurden, rückgängig machen (globales **undo**) und
- alle noch nicht ausgelagerten Änderungen von abgeschlossenen Transaktionen nachvollziehen (globales **redo).**

Dazu sind Zusatzinformationen aus Protokolldateien (Log-Dateien) notwendig

Fehlerklassifikation [4]

Fehler mit Hintergrundspeicherverlust

Solche Fehler treten z.B. in folgenden Situationen auf:

- „head crash“, der die Festplatte mit der Datenbank zerstört
- Feuer / Erdbeben
- Fehler in Systemprogrammen (z.B. Plattentreiber) oder Viren, die zu einem Datenverlust führen.

Obwohl solche Fehler sehr selten auftreten, sind unbedingt Vorkehrungen zu treffen. Der Schaden eines Verlusts einer gesamten Datenbank ist normalerweise enorm. Zur Fehlerbehebung werden meist eine Archivkopie der gesamten Datenbasis und ein Log-Archiv verwendet.

Management des Datenbankpuffers [1]

Die grundsätzliche, Seiten-orientierte Architektur DBMS ↔ DB-Puffer ↔ Hintergrundspeicher ist in der folgenden Graphik dargestellt (siehe auch Folie 14). In diesem Abschnitt soll nun die Wechselwirkung zwischen Transaktionsverwaltung und Ein- und Auslagerung von Datenseiten behandelt werden.

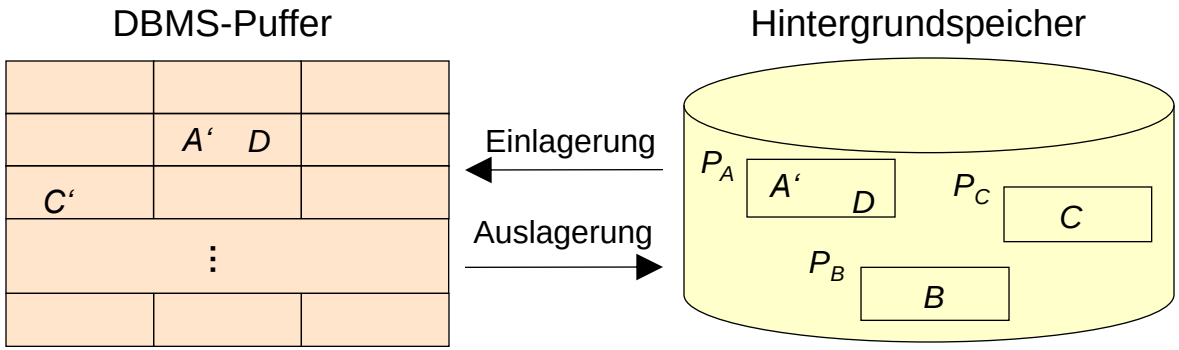


Abb.: Schematische Darstellung der (zweistufigen) Speicherhierarchie
(entnommen aus [1])

Management des Datenbankpuffers [2]

Operationen auf die Seiten im Puffer

Eine Transaktion benötigt im allgemeinen mehrere Datenseiten, die sich entweder (zufällig) bereits im Puffer befinden oder erst eingelagert werden müssen.

- Für die Dauer eines Zugriffs, bzw. einer Änderungsoperation wird die jeweilige Seite im Puffer durch setzen eines FIX-Vermerkes **fixiert**, damit die betreffende Seite während dieser Zeit nicht aus dem Puffer verdrängt wird.
- Werden Daten dieser Seite durch die Operation geändert, wird die Seite aus modifiziert (**dirty**) gekennzeichnet. D.h. der Inhalt dieser Seite stimmt nicht mehr mit dem Hintergrundspeicher überein.
- Nach Beendigung der Operation wird der FIX-Vermerk wieder gelöscht. Die Seite wird also für eine mögliche Ersetzung freigegeben.

Management des Datenbankpuffers [3]

Ersetzung von Seiten aktiver Transaktionen

Salopp gesagt herrscht im Datenbankpuffer ein „Kommen und Gehen“ von Seiten. In Bezug auf aktive, noch nicht abgeschlossene Transaktionen gibt es zwei Strategien:

- ¬**steal**: Eine Ersetzung von Seiten, die von einer noch aktiven Transaktion geändert wurden ist verboten. Somit kann es nie vorkommen, dass Änderungen einer aktiven Transaktion bereits auf den Hintergrundspeicher geschrieben werden. Bei einem „rollback“ braucht man sich also nicht um den Hintergrundspeicher kümmern.
- **steal**: Jede nicht fixierte Seite ist Kandidat für eine Ersetzung, falls neue Seiten eingelagert werden müssen. In diesem Fall müssen bei einem „rollback“ eventuell auch schon in den Hintergrundspeicher geschriebene Änderungen rückgängig gemacht werden, um den Zustand vor Transaktionsbeginn wiederherzustellen.

Management des Datenbankpuffers [4]

Ersetzung von Seiten abgeschlossener Transaktionen

- **force**: Beim „commit“ werden alle von der Transaktion geänderten Seiten in die Datenbasis zurückgeschrieben.
- ¬ **force**: Änderungen abgeschlossener Transaktionen können längere Zeit im Puffer verbleiben, bis sie irgendwann zurückgeschrieben werden (Problem bei Fehler mit Hauptspeicherverlust).

Auswirkung auf Recovery:

	force	¬force
¬steal	•kein Undo •kein Redo	•Redo •kein Undo
steal	•kein Redo •Undo	•Redo •Undo

Auf einen ersten Blick scheint es ideal, ¬steal und force anzuwenden. Dies hat aber auch viele Nachteile:

- die Änderungen einer Transaktion müssen beim „commit“ atomar in den Hintergrundspeicher kopiert werden. Was ist bei einem Systemabsturz während dieses Kopiervorganges?.
- schlecht verteilte Systemauslastung
- eine Seite kann Datensätze enthalten, die von mehreren, unabhängigen Transaktionen geändert wurden.

Management des Datenbankpuffers [5]

Einbringstrategie

Darunter versteht man die Methodik, nach der die Änderungen in die Datenbasis geschrieben (eingebracht) werden.

- **direkte Einbringstrategie (update-in-place):** Jeder Seite wird genau ein Platz im Hintergrundspeicher zugeordnet. Beim Zurückschreiben wird genau dieser Platz ‚überschrieben'. Dies ist die heute gängigste Strategie.
- **indirekte Einbringstrategie:** Geänderte Seiten werden an einen separaten Platz gespeichert, und nur zu bestimmten vom System initiierten Zeitpunkten werden die alten Zustände durch die neuen ersetzt.

Die einfachste Methode ist die sogenannte **Twin-Block-Strategie**. Jeder Seite sind zwei Blöcke im Hintergrundspeicher zugeordnet und ein Flag gibt an, welcher gerade aktuell ist. Auf den anderen wird zurückgeschrieben, dann das Flag umgesetzt. Die atomare Einbringung wird gut unterstützt, jedoch wird viel Speicherplatz benötigt. Beim **Schattenspeicherkonzept** werden nur die tatsächlich modifizierten Seiten verdoppelt.

Management des Datenbankpuffers [6]

Systemkonfiguration für die weiteren Überlegungen:
Die nachfolgende Diskussion der Recoverykomponente geht von folgender (der allgemeinsten) und für das Recovery schwierigsten Konfiguration aus.

- **steal:** Nicht fixierte Seiten können jederzeit ersetzt bzw. auch nur propagiert werden (d.h. ohne Ersetzung zurückgeschrieben) werden.
- ¬ **force:** Geänderte Seiten werden kontinuierlich (bzw. nach einer eigenen Strategie), unabhängig von Transaktionsabschlüssen in die Datenbasis propagiert.
- **update-in-place:** Jede Seite hat genau einen Platz in der Datenbasis. Dorthin wird sie unter Umständen auch vor dem ‚commit' einer Transaktion zurückgeschrieben.
- **kleine Sperrgranulate**: Transaktionen können auch kleinere Objekte als eine ganze Seite exklusiv sperren und verändern. Eine Seite kann also zu einem Zeitpunkt sowohl Änderungen von abeschlossenen Transaktionen als auch von nicht abgeschlossenen Transaktionen enthalten.

Protokollierung von Änderungen [1]

Der Hintergrundspeicher enthält meist nicht den jüngsten konsistenten Zustand, im allgemeinen nicht einmal einen konsistenten Zustand. Deshalb benötigt man Zusatzinformationen, die an anderer Stelle gespeichert werden als die Datenbasis - die sogenannte **Log-Datei** (Protokolldatei).

Struktur der Log-Einträge [1]

Man benötigt für jede Änderungsoperation, die von einer Transaktion durchgeführt wurden zwei Protokollinformationen:

- Die **Redo-Information** gibt an, wie die Änderung nachvollzogen (wiederholt) werden kann.
- Die **Undo-Information** beschreibt, wie die Änderungen rückgängig gemacht werden kann..

Protokollierung von Änderungen [2]

Struktur der Log-Einträge [2]

Neben der Redo- und Undo-Information enthält bei dem vorgestellten Recoveryverfahren jeder normale Log-Eintrag noch:

- **LSN (Log Sequence Number):** Eine eindeutige Identifizierung des Log-Eintrags. Die LSN muss monoton aufsteigen, damit die chronologische Reihenfolge der Protokolleinträge ermittelt werden kann.
- **TA (Transaktionskennung)** der Transaktion, die die Änderung durchgeführt hat.
- **PageID:** die Kennung der Seite, auf der die Änderungsoperation durchgeführt wurde. Betrifft eine Änderung mehrere Seiten, müssen entsprechend viele Log-Einträge geschrieben werden
- **PrevLSN:** Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion. (Wird aus Effizienzgründen mitgeführt.)

Protokollierung von Änderungen [3]

Beispiel einer Log-Datei

Verzahnte Ausführung von zwei Transaktionen und die erstellte Log-Datei:

(entnommen aus [1])

Schritt	T_1	T_2	Log
			[LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1,T_1,**BOT**,0]
2.	$r(A,a_1)$		
3.		**BOT**	[#2,T_2,**BOT**,0]
4.		$r(C,c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A,a_1)$		[#3,T_1,P_A,A-=50,A+=50,#1]
7.		$c_2 := c_2 + 100$	
8.		$w(C,c_2)$	[#4,T_2,P_C,C+=100,C-=100,#2]
9.	$r(B,b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B,b_1)$		[#5,T_1,P_B,B+=50,B-=50,#3]
12.	**commit**		[#6,T_1,**commit**,#5]
13.		$r(A,a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A,a_2)$	[#7,T_2,P_A,A-=100,A+=100,#4]
16.		**commit**	[#8,T_2,**commit**,#7]

Protokollierung von Änderungen [4]

Logische oder physische Protokollierung

- Logische Protokollierung: Es werden die Redo- und Undo-Informationen logisch protokolliert, d.h. es werden jeweils Operationen angegeben (wie im Beispiel auf der vorigen Folie).
- Physische Protokollierung: Für das Undo wird das sogenannte **Before-Image** des Objektes abgespeichert, für das Redo das **After-Image**.
  (In der Logischen Protokollierung kann mit der in der Logdatei geschriebenen Information beim Redo aus dem Bevor-Image das After-Image berechnet werden. Analog beim Undo)

Bei einem Wiederanlauf ist es notwendig zu erkennen, ob sich nun in Bezug auf einen Log-Eintrag das Before-Image oder das After-Image im Hintergrundspeicher befindet.

Dazu dient die LSN: Beim Anlegen eines neuen Log-Eintrags wird die LSN in einen reservierten Bereich der betreffenden Seite geschrieben. Dieser wird dann auch in die Datenbasis mitkopiert. Nachdem die LSN monoton wachsend ist gilt: Ist die LSN der Seite im Hintergrundspeicher kleiner als die des betrachteten Log-Eintrages, ist das Before-Image im Hintergrundspeicher, sonst das After-Image.

Protokollierung von Änderungen [5]

Speicherhierarchie mit Berücksichtigung des Log-Puffers

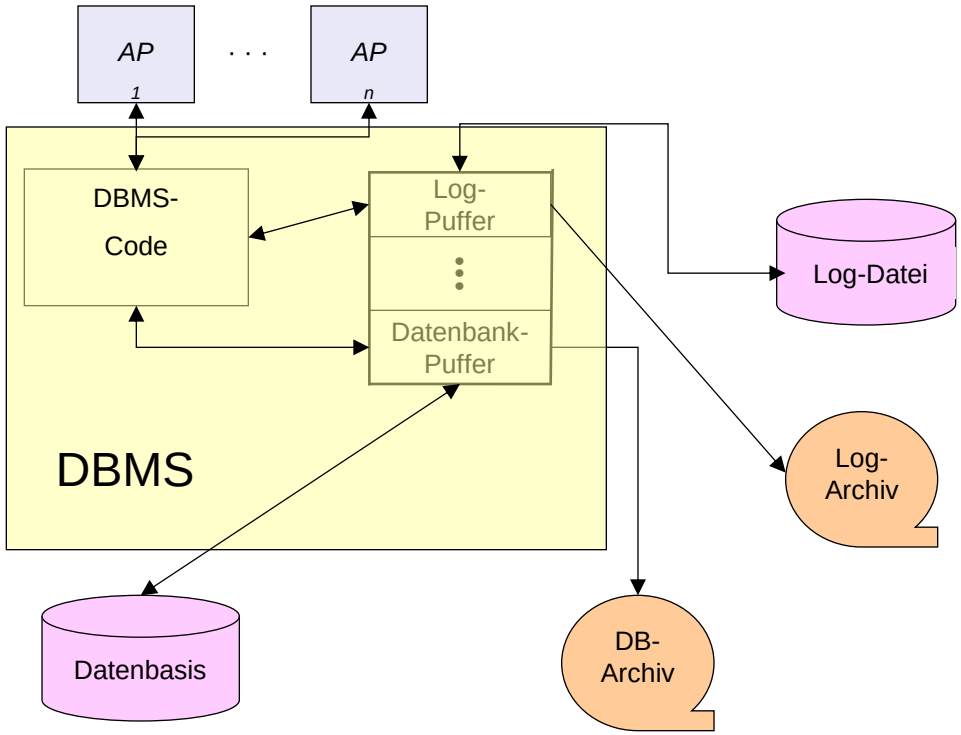


Abb.: Speicherhierarchie eines DBMS
(entnommen aus [1] / Härder und Reuter(1983))

Der Log-Puffer wird zweimal geschrieben: Einmal in die Logdatei (für ein Recovery bei Fehler mit Hauptspeicherverlust), einmal ins Log-Archiv auf Magnetband oder einer anderen Festplatte (für ein Recovery mit Hintergrundspeicherverlust).

Protokollierung von Änderungen [6]

Schreiben der Log-Information [1]

Bevor eine Änderungsoperation durchgeführt wird, muss der zugehörige Log-Eintrag geschrieben werden.

Die Log-Einträge werden im sogenannten Log-Puffer im Hauptspeicher zwischengelagert. In modernen DBMS ist dieser als Ringpuffer angelegt (am einen Ende wird kontinuierlich eingeschrieben, am anderen ausgelesen). Dies hat den Vorteil einer gleichmäßigen Systemlast.

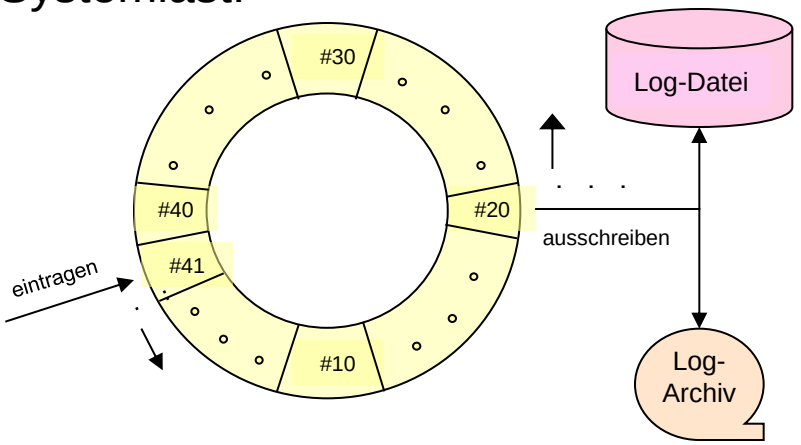


Abb.: Anordnung des Log-Ringpuffers
(entnommen aus [1])

Protokollierung von Änderungen [7]

Schreiben der Log-Information [2]

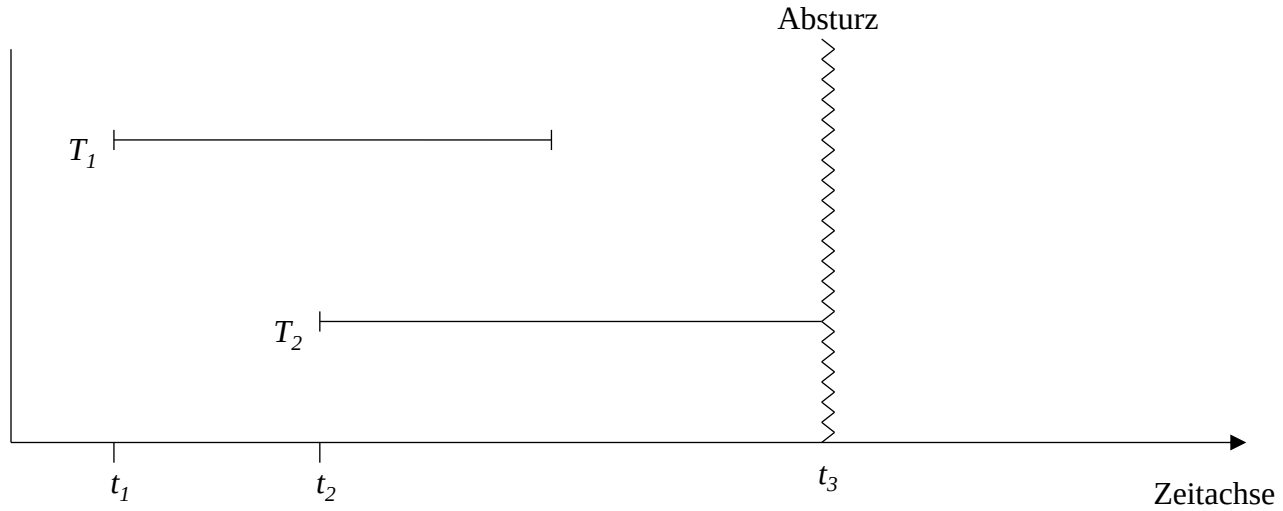
Das **WAL (Write Ahead Log) - Prinzip**

Damit bei einem Recovery wirklich nichts verloren geht, muss beim ausschreiben des Log-Puffers das WAL-Prinzip angewandt werden:

- Bevor eine Transaktion festgeschrieben (committed) wird, müssen alle zu ihr gehörigen Log-Einträge ausgeschrieben werden.
- Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge die zu dieser Seite gehören, ausgeschrieben werden.

Man schreibt natürlich jeweils alle Log-Einträge bis zum letzten notwendigen aus, um die chronologische Reihenfolge der Einträge im Ringpuffer wahren zu können.

Wiederanlauf nach einem Fehler mit Hauptspeicherverlust [1]



- Transaktionen der Art T_1 müssen hinsichtlich ihrer Wirkung vollständig nachvollzogen werden. (**Winner**)
- Transaktionen der Art T_2, die zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden (**Loser**)

Wiederanlauf nach einem Fehler mit Hauptspeicherverlust [2]

Der Wiederanlauf geschieht in drei Phasen:

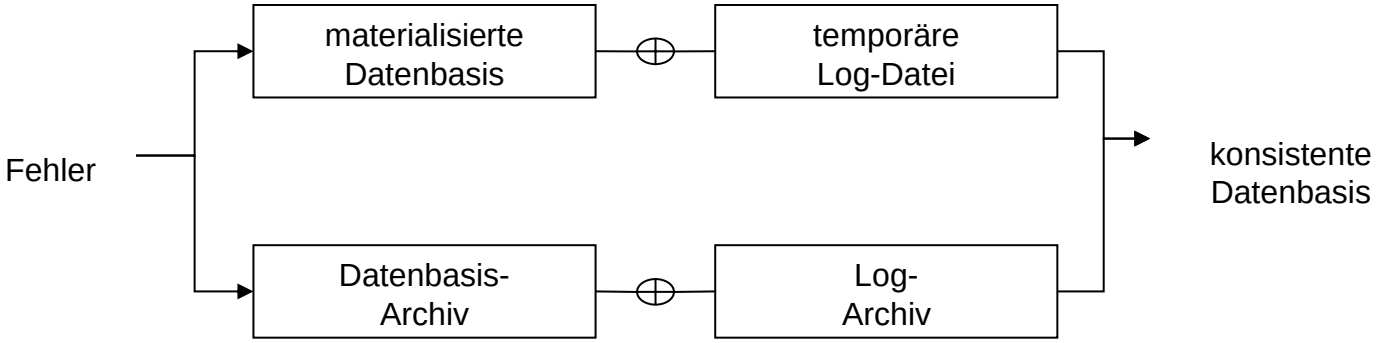
- **Analyse:** Die Logdatei wird von Anfang bis Ende analysiert, um die Winner-Menge von Transaktionen des Typs T_1 und die Loser-Menge von Transaktionen des Typs T_2 zu ermitteln.
- **Wiederholung der Historie:** Es werden alle protokollierten Änderungen in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht.
- **Undo der Loser:** Die Änderungsoperationen der Loser-Transaktionen werden in umgekehrter Reihenfolge ihrer ursprünglichen Ausführung rückgängig gemacht.

Wiederanlauf nach einem Fehler mit Hintergrundspeicherverlust

Hoffentlich wurde für diesen Fall Vorsorge getroffen und das DBMS so konfiguriert, dass eine Archiv-Kopie mit zugehörigem Log-Archiv zur Verfügung steht!

Archiv-Kopie und zugehöriges Log-Archiv erlauben analog dem Recovery bei Hauptspeicherverlust den letzten konsistenten Zustand der Datenbank herzustellen.

Als Zusammenfassung nochmals die zwei möglichen Recovery-Arten nach einem Systemabsturz:



Allgemeines [1]

Begriffsbestimmung

Unter **Mehrbenutzerbetrieb** (Multiprogramming) versteht man allgemein die gleichzeitige (nebenläufige, parallele) Ausführung mehrerer Programme - in DBMS – die parallele, nebenläufige Ausführung mehrerer Transaktionen.

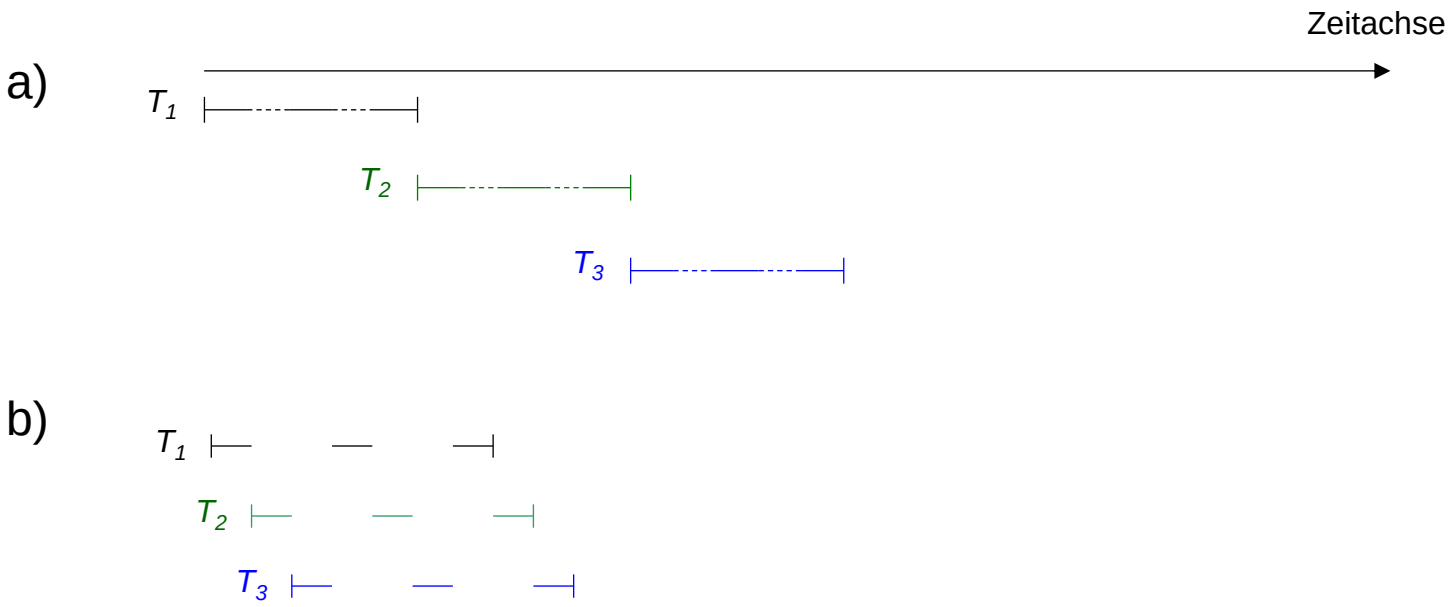
Vor allem Datenbankanwendungen (Transaktionen in DBMS) müssen sehr oft auf langsame Betriebsmittel (z.B. Zugriff auf Hintergrundspeicher oder Benutzereingaben) warten. Im Mehrbenutzerbetrieb kann somit das DBMS in der Zwischenzeit eine andere Anwendung (Transaktion) bedienen.

Im Hinblick auf das ACID-Paradigma ist im Mehrbenutzerbetrieb besonders das einhalten des „I“ (der Isolation) eine Herausforderung. Aber auch das „A“ (Atomarität) kann durch ungeschickten Mehrbenutzer-betrieb gefährdet sein. (Ein „lokales undo“ könnte eine parallel laufende Transaktion beeinflussen.)

Mehrbenutzersynchronisation

Allgemeines [2]

Abstraktes Beispiel



Die Transaktionen T_1, T_2 und T_3 Transaktionen (a) im Einbenutzerbetrieb und (b) im Mehrbenutzerbetrieb (gestrichelte Linien bedeuten Wartezeiten)

Fehler bei unkontrolliertem Mehrbenutzerbetrieb [1]

„Lost Update“ (Verlorengegangene Änderungen)

Z.B. zwei Transaktionen aus dem Bankenbereich: T_1 bucht 300,-- von Konto A auf Konto B; T_2 schreibt Konto A 3% Zinseinkünfte gut.

Ein möglicher verzahnter Ablauf (wenn keine Mehrbenutzer-synchronisation durchgeführt würde).

Schritt	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.		read(A,a_2)
4.		$a_2 := a_2 * 1.03$
5.		write(A,a_2)
6.	write(A,a_1)	
7.	read(B,b_1)	
8.	$b_1 := b_1 + 300$	
9.	write(B,b_1)	

Die im dem Konto A zugeschriebenen Zinsen (update von T_2 in Schritt 5) gehen verloren (werden von T_1 in Schritt 6 überschrieben).

Fehler bei unkontrolliertem Mehrbenutzerbetrieb [2]

„Dirty Read“ (Lesen von nicht freigegebenen Änderungen)

Ein anderer möglicher verzahnter Ablauf von T_1 und T_2 aus Folie 35:

Schritt	T_1	T_2
1.		
2.	$a_1 := a_1 - 300$	
3.	write(A,a_1)	
4.		read(A,a_2)
5.		$a_2 := a_2 * 1.03$
6.		write(A,a_2)
7.	read(B,b_1)	
8.	. . .	
9.	abort	

Die Zinsberechung von T_2 ergibt ein falsches Ergebnis. die vorher durchgeführte Änderung (Schritt 2 in T_1) wurde später zurück-genommen (Schritt 9 in T_1), war also nicht freigegeben.

Fehler bei unkontrolliertem Mehrbenutzerbetrieb [3]

„Phantomproblem“

Zwei neue Transaktionen T_1 und T_2:

T_1	T_2
	select count(*)
	from Konten
insert into Konten	
values (*C*,1000,...)	
	select sum(Kontostand)
	from Konten

In T_2 stimmt die Anzahl der Konten nicht mit der Summe überein, weil in der Zwischenzeit T_1 das „Phantom“ C erzeugt hat..

Serialisierbarkeit

Allgemeines

Obwohl in DBMS Transaktionen parallel abgearbeitet werden, muss das Ergebnis dasselbe sein, wie wenn die Transaktionen hintereinander durchgeführt worden wären.

Etwas allgemein gesagt:

Eine serialisierbare Ausführung einer Menge von Transaktionen ist eine verzahnte, oder wirklich parallele (in Mehrprozessorsystemen) Abarbeitung dieser Transaktionen, in der sichergestellt ist, dass das Ergebnis einer möglichen seriellen Ausführung dieser Transaktionen entspricht.

Die exakte Definition der Serialisierbarkeit erfolgt im **Serialisierbarkeits-Theorem** (siehe in der Literatur (z.B „Kemper“) oder in Informationssysteme 2).

Der Datenbank-Scheduler

Allgemeines

Seine Aufgabe besteht darin, die Einzeloperationen der verzahnt / parallel auszuführenden Transaktionen in eine derartige Reihenfolge zu bringen dass die daraus resultierende Historie (Schedule) „vernünftig“ ist.

Unter „vernünftig“ versteht man zumindest, dass die Historie serialisierbar ist, aber meistens auch, dass sie ohne „kaskadierendes Rollback“ rücksetzbar ist (das Rollback einer einzelnen Transaktion bewirkt nicht ein Rollback weiterer Transaktionen).

Mögliche Techniken (Vorgangsweisen) eines Schedulers:

- Pessimistische Verfahren (sperrbasiert oder zeitstempelbasierte)

  Potentielle Konfliktsituationen werden von vornherein nicht zugelassen.

- Optimistische Verfahren

  Erst beim Commit wird untersucht, ob es einen Konflikt zwischen zwei Transaktionen gab, und falls ja, die Transaktion zurückgesetzt.

Sperrbasierte Synchronisation [1]

Allgemeines

Es wird während des laufenden Betriebs sichergestellt, dass die resultierende Historie serialisierbar ist. Dies geschieht dadurch, dass eine Transaktion erst nach Erhalt einer entsprechenden Sperre auf ein Datum zugreifen kann.

Sperrmodi

Es gibt zwei Sperrmodi (je nach Operation, read oder write):

- S (shared, read lock, Lesesperre)

  Wenn eine Transaktion eine S-Sperre auf A besitzt, darf sie A lesen. Auch andere Transaktionen können gleichzeitig eine S-Sperre auf A besitzten (dürfen A lesen).

- X (exclusive, write lock, Schreibsperre)

  Ein write(A) darf nur eine Transaktion ausführen, die die X-Sperre auf A besitzt.

Verträglichkeitsmatrix:
(Kompabilitätsmatrix)

	NL	S	X
S	✓	✓	-
X	✓	-	-

NL: no lock
waagrecht: existierende Sperre
senkrecht: Sperranforderung

Sperrbasierte Synchronisation [2]

Zwei-Phasen-Sperrprotokoll [1]

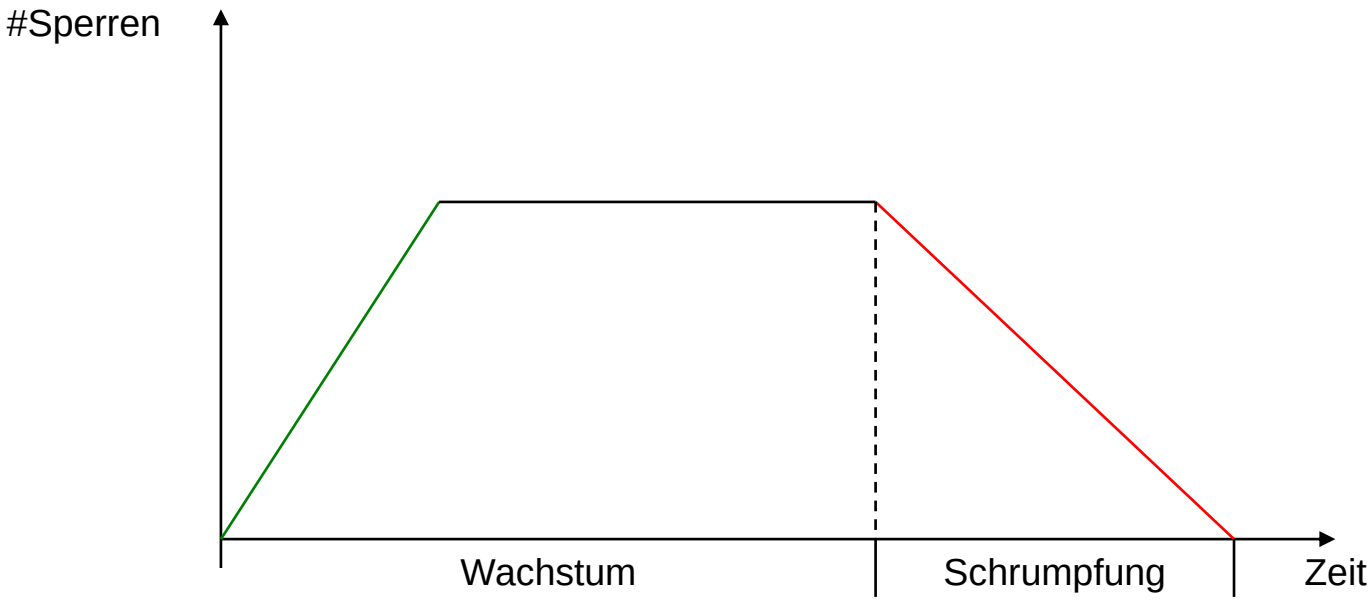
Bezogen auf eine individuelle Transaktion wird folgendes verlangt.

1. Jedes Objekt, das benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an
3. Eine Transaktion muss die Sperren anderer Transaktionen gemäß Verträglichkeitstabelle beachten und muss gegebenenfalls warten (wird in entsprechende Warteschlange eingereiht), bis die Sperre gewährt werden kann.
4. Jede Transaktion durchläuft zwei Phasen:
   - **Wachstumsphase**, in der sie Sperren anfordern, aber keine bisher erworbenen freigeben darf.
   - **Schrumpfungsphase**, in der sie ihre bisher erworbenen Sperren freigeben, aber keine weiteren anfordern darf.
5. Bei Transaktionsende muss die Transaktion all ihre Sperren zurückgeben

Sperrbasierte Synchronisation [3]

Zwei-Phasen-Sperrprotokoll [2]

Graphische Darstellung:



Sperrbasierte Synchronisation [4]

Zwei-Phasen-Sperrprotokoll [3]

Beispiel:

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(*A*)		
3.	read(*A*)		
4.	write(*A*)		
5.		**BOT**	
6.		**lockS**(*A*)	T_2 muss warten
7.	**lockX**(*B*)		
8.	read(*B*)		
9.	**unlockX**(*A*)		T_2 wecken
10.		read(*A*)	
11.		**lockS**(*B*)	T_2 muss warten
12.	write(*B*)		
13.	**unlockX**(*B*)		T_2 wecken
14.		read(*B*)	
15.	**commit**		
16.		**unlockS**(*A*)	
17.		**unlockS**(*B*)	
18.		**commit**	

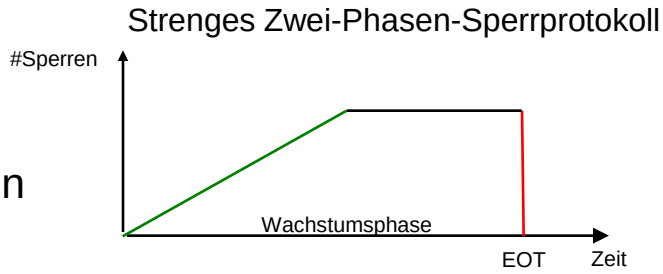
Sperrbasierte Synchronisation [5]

Zwei-Phasen-Sperrprotokoll [4]

Beurteilung:

Es garantiert auf jeden Fall die Serialisierbarkeit, vermeidet jedoch nicht das kaskadierende Rollback und erzeugt Deadlocks.

- **Vermeidung von kaskadierenden Rollbacks – strenges Zwei-Phasen-Sperrprotokoll**,
  Die Anforderungen (1) bis (5) bleiben gleich,
  es gibt jedoch keine Schrumpfungsphase mehr.
  (Alle Sperren werden erst am Ende der Transaktion freigegeben.)



- **Umgang mit Deadlocks**

  a) Erkennung (mittels „Time-out-Strategie“ oder „Analyse des Wartegraphs“ (besser aber aufwendiger)) und Beseitigung, indem mindestens eine beteiligte Transaktion zurückgesetzt wird.

  b) Vermeidung durch Zeitstempelverfahren (siehe in Literatur)
  (Nachteil: Ineffizient; es müssen viele Transaktionen zurückgesetzt werden, ohne dass tatsächlich ein Deadlock auftreten würde.)