Kapitel 10: Transaktionsverwaltung

- Umfasst diejenigen Komponenten eines Datenbankmanagementsystems, deren Aufgabe die Gewährleistung der Atomizität, Isolation und Dauerhaftigkeit der Transaktionen ist.
- *Mehrbenutzerkontrolle*: Isolation der einzelnen Transaktionen.
- *Fehlerbehandlung*: Atomizität und Dauerhaftigkeit einer Transaktion.

10.1 Grundlagen

- Eine Datenbank sei gegeben als eine Menge von Objekten.

  logische Größen: Relation, Tupel einer Relation,

  physische Größen: Blöcke, Seiten einer physischen Datenbank.
- Transaktionen T greifen lesend und schreibend zu den Objekten der Datenbank zu: *Lese-* und *Schreiboperationen*. Wir repräsentieren sie durch ihre Folge von Lese- und Schreiboperationen.
  - Bei Ausführung einer Leseoperation zu einem Datenbankobjekt A, RA, wird der Wert des Objektes A aus der Datenbank in den lokalen Arbeitsbereich der Transaktion übertragen.
  - Bei Ausführung einer Schreiboperation zu A, WA, wird der Wert des Objektes A aus dem lokalen Arbeitsbereich der Transaktion in die Datenbank übertragen.
- Lese- und Schreiboperationen betrachten wir als atomar.

Schedule

- Sei $\mathcal{T} = \{T_1, \ldots, T_n\}$ eine Menge von Transaktionen.
- Die Folge der Lese- und Schreiboperationen einer Transaktion $T_i \in \mathcal{T}$ bezeichnen wir als ihre *Historie* h_i.
- Einen Ablauf der Transaktionen aus $\mathcal{T}$ nennen wir einen *Schedule* S zu $\mathcal{T}$.
- Ein Schedule ist eine Folge von Lese- und Schreiboperationen der einzelnen Transaktionen aus $\mathcal{T}$.
- Die relative Reihenfolge der Operationen einer Transaktion T in einem Schedule S entspricht der Reihenfolge in der zugehörigen Historie h von T.
- Ein *serieller* Schedule zu $\mathcal{T}$ ergibt sich als Konkatenation der Historien der einzelnen Transaktionen aus $\mathcal{T}$.

Beispiel

- Sei $\mathcal{T} = \{T_1, T_2, T_3\}$, wobei $T_1 = R_1A\ W_1A\ R_1B\ W_1B$, $T_2 = R_2A\ W_2A\ R_2B\ W_2B$ und $T_3 = R_3A\ W_3B$.
- Es existieren sechs unterschiedliche serielle Schedule zu $\mathcal{T}$, beispielsweise $S_1 = h_1h_2h_3$, oder auch $S_2 = h_2h_3h_1$.
- Beispiele für nicht serielle Schedule sind:

$$S_3 = R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B,$$
$$S_4 = R_3A\ R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B\ W_3B.$$

augmentierter Schedule

- Sei T_0 eine Transaktion, die gerade zu jedem Objekt der Datenbank eine Schreiboperation enthält. T_0 erzeugt einen Startzustand der Datenbank
- Sei T_∞ eine Transaktion, die zu jedem Objekt eine Leseoperation enthält. T_∞ liest den Endzustand der Datenbank.
- Sei S ein Schedule zu $\mathcal{T}$. Sei $\widehat{S} = T_0\ S\ T_\infty$.

  $\widehat{S}$ nennen wir den *augmentierten* Schedule zu S.

10.2 Mehrbenutzerkontrolle

Problematik

Seien $T_1 = R_1A\ W_1A$ und $T_2 = R_2A\ W_2A$ zwei Transaktionen, die beide dasselbe Objekt A lesen und schreiben. Nehmen wir, dass A ein Lagerkonto repräsentiert und T_1 den Bestand um 100 Einheiten erhöht und T_2 den Bestand um 50 verringert. Zu Beginn betrage der Bestand 80 Einheiten.

S_1	S_2	S_3	S_4	S_5	S_6
$A = 80$	$A = 80$	$A = 80$	$A = 80$	$A = 80$	$A = 80$
R_1A	R_1A	R_1A	R_2A	R_2A	R_2A
W_1A	R_2A	R_2A	W_2A	R_1A	R_1A
R_2A	W_1A	W_2A	R_1A	W_2A	W_1A
W_2A	W_2A	W_1A	W_1A	W_1A	W_2A
$A = 130$	$A = 30$	$A = 180$	$A =$	$A =$	$A =$

Welche der sechs Schedule sind korrekt?

10.2.1 Serialisierbarkeit

Definition

Ein Schedule heißt *serialisierbar* genau dann, wenn zu ihm ein äquivalenter serieller Schedule derselben Transaktionen existiert.

Definition

Zwei Schedule S und S' über derselben Transaktionsmenge heißen *äquivalent*, wenn für jeden Startzustand der Datenbank und jede Semantik der Transaktionen die folgenden beiden Bedingungen erfüllt sind.

- Die Transaktionen lesen in S und S' jeweils dieselben Werte.
- Desweiteren erzeugen S und S' dieselben Endzustände der Datenbank.

Formalisierung der Semantik einer Transaktion?

Beispiel

- Seien $T_1 = R_1A\ W_1A\ R_1B\ W_1B$ und $T_2 = R_2A\ W_2A\ R_2B\ W_2B$.
- Seien S_1 und S_2 Schedule wie folgt:

$$S_1 = R_1A\ W_1A\ R_2A\ W_2A\ R_2B\ W_2B\ R_1B\ W_1B$$
$$S_2 = R_1A\ W_1A\ R_2A\ W_2A\ R_1B\ W_1B\ R_2B\ W_2B$$

	Schedule $T_1\,T_2$		Schedule $T_2\,T_1$
R_1A	A_0	R_2A	A_0
W_1A	$f_{T_1,A}(A_0)$	W_2A	$f_{T_2,A}(A_0)$
R_1B	B_0	R_2B	B_0
W_1B	$f_{T_1,B}(A_0, B_0)$	W_2B	$f_{T_2,B}(A_0, B_0)$
R_2A	$f_{T_1,A}(A_0)$	R_1A	$f_{T_2,A}(A_0)$
W_2A	$f_{T_2,A}(f_{T_1,A}(A_0))$	W_1A	$f_{T_1,A}(f_{T_2,A}(A_0))$
R_2B	$f_{T_1,B}(A_0, B_0)$	R_1B	$f_{T_2,B}(A_0, B_0)$
W_2B	$f_{T_2,B}(f_{T_1,A}(A_0), f_{T_1,B}(A_0, B_0))$	W_1B	$f_{T_1,B}(f_{T_2,A}(A_0), f_{T_2,B}(A_0, B_0))$

Sind S_1 und S_2 serialisierbar?

Abhängigkeitsgraph

Der *Abhängigkeitsgraph* von S ist ein gerichteter Graph $AG(S) = (V, E)$, wobei V die Menge aller Operationen in $\widehat{S}$ und E die Menge der Kanten gemäß den folgenden Bedingungen ($i \neq j$):

- $\widehat{S} = \ldots R_i B \ldots W_i A \ldots \Rightarrow R_i B \rightarrow W_i A \in E$,
- $\widehat{S} = \ldots W_i A \ldots R_j A \ldots \Rightarrow W_i A \rightarrow R_j A \in E$, sofern zwischen $W_i A$ und $R_j A$ in $\widehat{S}$ keine weitere Schreiboperation zu A existiert.

Satz

Zwei Schedule S und S' einer gemeinsamen Menge von Transaktionen sind genau dann äquivalent, wenn $AG(S) = AG(S')$ gilt.

Konfliktgraph

Der *Konfliktgraph* von S ist ein gerichteter Graph $KG(S) = (V, E)$, wobei V die Menge aller Transaktionen in $\widehat{S}$ und E die Menge der Kanten gemäß den folgenden Bedingungen ($i \neq j$):

- $\widehat{S} = \ldots W_i A \ldots R_j A \ldots \Rightarrow T_i \rightarrow T_j \in E$, sofern zwischen $W_i A$ und $R_j A$ in $\widehat{S}$ keine weitere Schreiboperation zu A existiert. *(WR-Konflikt)*
- $\widehat{S} = \ldots W_i A \ldots W_j A \ldots \Rightarrow T_i \rightarrow T_j \in E$, sofern zwischen $W_i A$ und $W_j A$ in $\widehat{S}$ keine weitere Schreiboperation zu A existiert. *(WW-Konflikt)*
- $\widehat{S} = \ldots R_i A \ldots W_j A \ldots \Rightarrow T_i \rightarrow T_j \in E$, sofern zwischen $R_i A$ und $W_j A$ in $\widehat{S}$ keine weitere Schreiboperation zu A existiert. *(RW-Konflikt)*

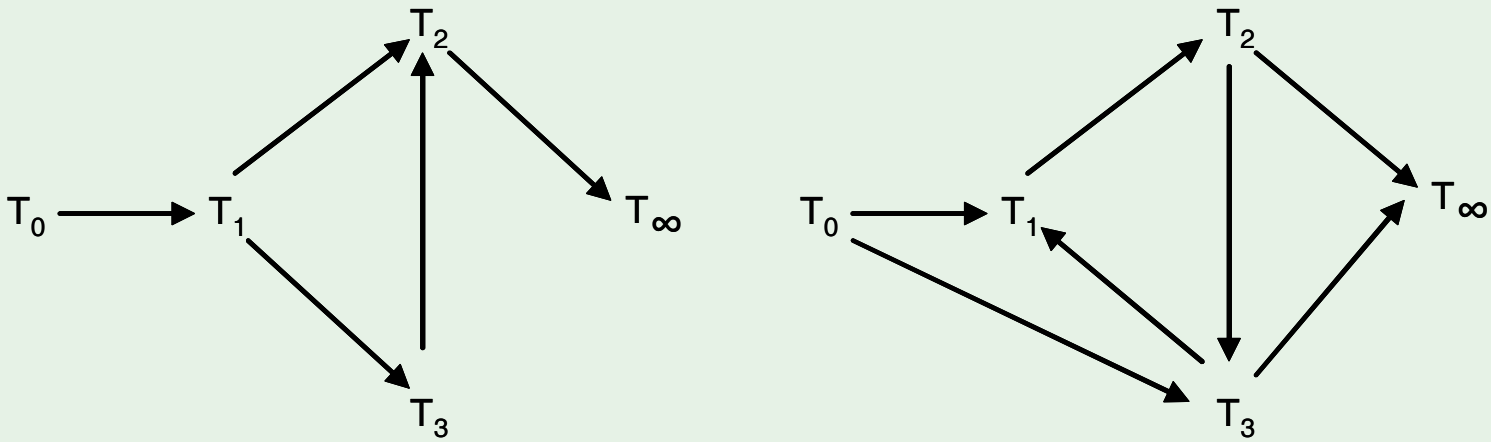
Satz und Definition

- Ein Schedule S ist serialisierbar, wenn $KG(S)$ zyklenfrei ist.
- Ein Schedule S heißt *konfliktserialisierbar* genau dann, wenn $KG(S)$ zyklenfrei ist.

Beispiel

Schedule S_1: $R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B$

Schedule S_2: $R_3A\ R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B\ W_3B$



10.2.2 Sperrverfahren

- Bevor eine Transaktion lesend oder schreibend zu einem Objekt zugreifen darf, muss ihr ein entsprechendes Privileg gewährt werden.
- Sperroperation (*Lock*):
  - *Leseprivileg* $L^R A$
  - *Lese- und Schreibprivileg* LA
- Freigabeoperation (*Unlock*): UA, bzw. $U^R A$.
- *Sperrtabelle*
- *Kompatibilitätsmatrix*:

angefordertes Privileg zu A: / gehaltenes Privileg zu A:

	$L^R A$	LA
$L^R A$	J	N
LA	N	N

- *Livelock* und *Deadlock* können auftreten.

Vermeidung von Livelocks und Deadlocks

- Vermeidung von Livelocks: *first-come-first-served*-Strategie
- Vermeidung von Deadlocks:
  - Jede Transaktion bewirbt sich zu Beginn um alle benötigten Privilegien auf einmal (in einer atomaren Operation).
  - Auf den Objekten wird eine lineare Ordnung definiert. Die Transaktionen fordern ihre jeweiligen Privilegien gemäß dieser Ordnung an.
- *Wartegraph*: Ein Wartegraph hat eine Kante $T_i \longrightarrow T_j$, wenn T_i sich um ein Privileg bewirbt, das T_j besitzt und das, aufgrund der Kompatibilitätsmatrix, nicht zugeteilt werden kann.

  Ein Deadlock liegt nun genau dann vor, wenn der Wartegraph einen Zyklus hat.

Wie kann ein Deadlock aufgelöst werden?

2-Phasen Sperren 2PL

Hat eine Transaktion eine Freigabeoperation ausgeführt, dann darf sie keine Sperroperation mehr ausführen.

mögliche Lock- und Unlock-Operationen gemäß 2PL der Transaktion
RA WA RB WB RC WC

LA RA WA LB RB WB LC RC WC UA UB UC,
LA RA WA LB LC UA RB WB UB RC WC UC,
LA LB LC RA WA UA RB WB UB RC WC UC,
LA LB LC RA WA RB WB RC WC UA UB UC.

2PL ist *strikt*, wenn alle Freigabeoperationen am Transaktionsende ausgeführt werden.

Satz

Das 2-Phasen Sperrprotokoll garantiert serialisierbare Schedule.

Beweise!

Optimalität und Mächtigkeit von 2PL

- 2PL ist ein optimales Sperrverfahren in dem Sinn, dass zu jeder nicht 2-phasigen Transaktion T_1 eine 2-phasige Transaktion T_2 konstruiert werden kann, so dass zu T_1 und T_2 ein nicht serialisierbarer Schedule existiert.
- Es existieren konfliktserialisierbare Schedule, die bei Einhaltung von 2PL nicht entstehen können.

Zeige anhand von Beispielen!

10.2.3 Verfahren mit und ohne Sperren

- Sperrverfahren sind nicht die einzige Technik zur Gewährleistung serialisierbarer Schedule.
- Eine Mehrbenutzerkontrolle wird formal durch eine Abbildung Φ beschrieben, die eine von den Transaktionen angeforderte (Eingabe-) Folge von Operationen S_I in eine serialisierbare auszuführende (Ausgabe-) Folge von Operationen S_O der Transaktionen transformiert.
- Es gilt $\Phi(S_I) = S_O$, wobei S_I ein Präfix eines Schedules und S_O ein Schedule.

Protokoll einer auf einem Sperrverfahren basierenden Mehrbenutzerkontrolle Φ_{2PL}.
Für die Transaktionen gilt $T_1 = L_1A\ R_1A\ L_1B\ U_1A\ W_1B\ U_1B$,
$T_2 = L_2A\ R_2A\ W_2A\ U_2A$, und $T_3 = L_3^R C\ R_3C\ U_3^R C$.

angeforderte Folge	Sperrtabelle	ausgeführte Folge
L_1A	L_1A	
$L_1A\ R_1A$	L_1A	R_1A
$L_1A\ R_1A\ L_2A$	L_1A	R_1A
$L_1A\ R_1A\ L_2A\ L_3^R C$	$L_1A, L_3^R C$	R_1A
$L_1A\ R_1A\ L_2A\ L_3^R C\ R_3C$	$L_1A, L_3^R C$	$R_1A\ R_3C$
$L_1A\ R_1A\ L_2A\ L_3^R C\ R_3C\ L_1B$	$L_1A, L_3^R C, L_1B$	$R_1A\ R_3C$
$L_1A\ R_1A\ L_2A\ L_3^R C\ R_3C\ L_1B\ U_1A$	$L_3^R C, L_1B$	$R_1A\ R_3C$
$L_1A\ R_1A\ L_2A\ L_3^R C\ R_3C\ L_1B\ U_1A\ R_2A$	$L_3^R C, L_1B, L_2A$	$R_1A\ R_3C\ R_2A$
$L_1A\ R_1A\ L_2A\ L_3^R C\ R_3C\ L_1B\ U_1A\ R_2A\ W_2A$	$L_3^R C, L_1B, L_2A$	$R_1A\ R_3C\ R_2A\ W_2A$
$L_1A\ R_1A\ L_2A\ L_3^R C\ R_3C\ L_1B\ U_1A\ R_2A\ W_2A\ U_2A$	$L_3^R C, L_1B$	$R_1A\ R_3C\ R_2A\ W_2A$
...	...	...

Scheduler ohne Sperren

- Angeforderte Aktionen kommen zur Ausführung, sofern der Scheduler sicher ist, dass kein nicht serialisierbarer Schedule in der Entstehung ist.
- Anderenfalls wird eine *aktive* Transaktion abgebrochen.
- Dies kann den Abbruch anderer, *abhängiger* Transaktionen erfordern.

Aktive und abhängige Transaktionen

- Eine Transaktion T ist von einer Transaktion T' *abhängig*, wenn ein gerichteter Weg $T' \rightarrow \ldots \rightarrow T$ im Konfliktgraphen des zugehörigen Schedules existiert, der durch eine Folge von WR-Konflikten begründet ist.
- Eine Transaktion T heißt *aktiv* in einem Schedule S, wenn eine Operation von T in S enthalten ist und T noch nicht ihr Ende erreicht hat. Eine Transaktion signalisiert ihr Ende, indem sie als letztes die Operation *Commit* ausführt.

Überwachen des Konfliktgraphen Φ_{KG}

Sei S die aktuelle Folge der ausgeführten Operationen und sei *op* die nächste angeforderte Operation einer Transaktion T.

Falls $KG(S\ op)$ zyklenfrei, dann führe *op* aus. Anderenfalls breche T und alle von T abhängigen Transaktionen ab und streiche die entsprechenden Operationen dieser Transaktionen in S.

Vergabe von Zeitmarken Φ_{ZM}

Jeder Transaktion T wird bei ihrem Beginn eine eindeutige Zeitmarke $Z(T)$ zugewiesen.

Sei S die aktuelle Folge der ausgeführten Operationen und sei *op* die nächste angeforderte Operation einer Transaktion T.

Falls für alle Transaktionen T', die bereits eine zu *op* in Konflikt stehende Operation in S ausgeführt haben, gerade $Z(T') \leq Z(T)$, dann führe *op* aus. Anderenfalls breche T und alle von T abhängigen Transaktionen ab und streiche die entsprechenden Operationen dieser Transaktionen in S.

Optimistisches Verfahren Φ_{OP}

Sei S die aktuelle Folge der ausgeführten Operationen und sei *op* die nächste angeforderte Operation einer Transaktion T.

Sei des Weiteren $Readset(T)$ und $Writeset(T)$ die Menge der von einer Transaktion T bereits gelesenen, bzw. geschriebenen Objekte.

Ist *op* verschieden von *Commit*, dann führe *op* aus. Ist *op* die *Commit*-Operation, d.h. die letzte Operation von T, dann breche T und alle von T abhängigen Transaktionen ab, sofern bzgl. einer anderen aktiven Transaktion T' eine der folgenden Bedingungen gilt:

- $Readset(T) \cap Writeset(T') \neq \emptyset$,
- $Writeset(T) \cap Writeset(T') \neq \emptyset$,
- $Writeset(T) \cap Readset(T') \neq \emptyset$.

Streiche desweiteren alle Operationen der abgebrochenen Transaktionen in S.

Beispiel

$S_I = R_1A\ R_2A\ W_2A\ R_3B\ W_3B\ W_1B.$
$T_1 = R_1A\ W_1B,\ T_2 = R_2A\ W_2A,\ T_3 = R_3B\ W_3B.$

S_I	S_O
Φ_{KG}	
Φ_{ZM}	
Φ_{OP}	

10.2.4 Phantomproblem

bisherige implizite Annahme

Die Menge der Objekte in der Datenbank ist konstant über der Zeit.

Verletzung der Annahme kann zu *Phantomen* führen.

Phantom kommt aus dem griechischen und bedeutet Trugbild, unwirkliche Erscheinung.

Schedule mit Phantom

Sei eine Transaktion T_1 eine Ausführung eines Programmes P, das zunächst alle Objekte A liest, die eine gewisse Bedingung p erfüllen und anschließend ein weiteres Objekt B. T hat damit eine Historie der Form $R_1A_1 \ \ldots R_1A_k \ R_1B$.

Laufe zeitlich überlappend zu T_1 eine Transaktion T_2 ab, die ein Objekt C liest, ein neues Objekte A_{k+1} in die Datenbank einfügt, das ebenfalls die Bedingung p erfüllt und anschließend B ändert.

Unter diesen Annahmen ist der folgende Schedule möglich:

$$R_2C \ R_1A_1 \ \ldots R_1A_k \ W_2A_{k+1} \ R_2B \ W_2B \ R_1B$$

Dieser Schedule ist formal äquivalent zu $T_2 \ T_1$; es wird hierbei jedoch ignoriert, dass A_{k+1} auch die Bedingung p erfüllt und somit T_1 eine Leseoperation R_1A_{k+1} enthalten müsste.

Lösung des Phantomproblems

- Vergrößerung der Granularität der betrachteten Objekte.
- Anstatt einer Folge von Leseoperationen $R_1A_1 \ldots R_1A_k$ betrachten wir eine einzige Leseoperation, z.B. in der Form $R_1\{A \mid p(A)\}$.
- Da A_{k+1} die Eigenschaft p erfüllt, kann der Konflikt mit dem Phantom A_{k+1} erkannt werden.
- Sperrverfahren können den Test auf p implementieren, indem ganze Relationen, Schlüsselbereiche oder auch Indexbereiche gesperrt werden.

10.3 Fehlerbehandlung

Im realen Betrieb eines Datenbanksystems muss mit Fehlersituationen gerechnet werden.

Transaktionsfehler: Hierunter verstehen wir einen zum Abbruch führenden Fehler im Anwendungsprogramm, bzw. den Abbruch der Transaktion durch den Benutzer oder durch das Datenbanksystem.

Systemfehler: Dies sind Fehler, die nicht eine einzelne, sondern möglicherweise alle ablaufenden Transaktionen betreffen. Ursache können Hardwarefehler oder falsche Werte in Systemtabellen sein.

Mediumfehler: Dies sind Fehler des externen Speichermediums, typischerweise Plattenfehler.

Recovery, Log und Commit

Es ist die Aufgabe der für die Fehlerbehandlung zuständigen Komponente eines Datenbanksystems, bei Eintreten einer Fehlersituation den abgelaufenen Schedule möglichst umfassend zu rekonstruieren (engl. *Recovery*), um eine weitere Ausführung zu ermöglichen.

- Ein Datenbanksystem führt eine sogenannte *Log-Datei*, in der die Veränderungen der Objekte der Datenbank und der Beginn und das Ende der Transaktionen protokolliert werden.
- Beim Ende einer Transaktion durchläuft sie ihre *Commit*-Phase. Hat sie erfolgreich ihre Commit-Phase durchlaufen, dann muss von diesem Zeitpunkt an die Atomizität und Dauerhaftigkeit gewährleistet sein.
- Alle Transaktionen, von denen die betrachtete über gelesene Werte abhängt, müssen ihre Commit-Phase erfolgreich durchlaufen haben.
- Konsequenzen für eine Mehrbenutzerkontrolle:
  - Sperrverfahren,
  - Verfahren ohne Sperren.

undo- und redo-Situation

- Kritisch ist, ob die von einer Transaktion bewirkten Änderungen bereits in der Datenbank materialisiert wurden, oder ob sie erst im Datenbankpuffer wirksam wurden.
  - Wann kann es erforderlich werden, eine geänderte Seite in der Datenbank zu materialisieren, bevor die Transaktion ihre Commit-Phase durchlaufen hat?
  - Wann kann es sinnvoll sein, geänderte Seiten auch nach Durchlaufen der Commit-Phase noch im Datenbankpuffer zu halten und nicht direkt in der Datenbank zu materialisieren?
- Eine Transaktion schreibt vor Abschluss ihrer Commit-Phase in die Datenbank und bevor das Commit erfolgreich ausgeführt werden konnte, tritt ein Fehler auf. $\Longrightarrow$ *Undo-Situation*.
- Eine Transaktion hat ihr Commit erfolgreich ausgeführt und es tritt ein Fehler auf, bevor alle geänderten Seiten in der Datenbank materialisiert sind. $\Longrightarrow$ *Redo-Situation*.

Log-File

- Bei Beginn einer Transaktion T wird (T, Begin) in das Log geschrieben.
- Für jede Operation WA von T wird (T, A, A_{old}, A_{new}) in das Log geschrieben. A_{new} ist der von T erzeugte Wert von A, das *After-Image* von A und A_{old} ist der Wert von A vor Ausführung von WA in der Datenbank, das *Before-Image* von A.

  Dieser Eintrag im Log muss zwingend erfolgt sein, bevor die entsprechende Änderung in der Datenbank materialisiert wird und bevor die Commit-Phase der Transaktion abgeschlossen wird.

  Warum?
- Nach Abschluss der Commit-Phase von T wird (T, Commit) in das Log geschrieben.
- Bei Abbruch von T wird (T, Abort) in das Log geschrieben.

Beheben eines Transaktionsfehlers

Sei T die zu behandelnde Transaktion.

Das Log wird rückwärts bis zum Finden des Eintrags (T, Begin) gelesen und für jeden davor gefundenen Eintrag der Form (T, A, A_{old}, A_{new}) wird der Wert A_{old} für A wieder in der Datenbank materialisiert.

Restart-Algorithmus zum Beheben eines Systemfehlers

- $\mathit{Redone} := \emptyset$; $\mathit{Undone} := \emptyset$.
- Verarbeite das Log rückwärts. Sei (T, A, A_{old}, A_{new}) der betrachtete Log-Eintrag. Falls $A \notin \mathit{Redone} \cup \mathit{Undone}$:

  Redo: Falls (T, Commit) bereits im Log gefunden, dann materialisiere A_{new} in der Datenbank und setze $\mathit{Redone} := \mathit{Redone} \cup \{A\}$.

  Undo: Anderenfalls materialisiere A_{old} in der Datenbank und setze $\mathit{Undone} := \mathit{Undone} \cup \{A\}$.

Objekte: Seiten oder Tupel?

- Seiten der physikalischen Datenbank.

  Materialisieren von A_{old}, bzw. A_{new} in der Datenbank ist das Ersetzen einer kompletten Seite.
- Tupel von Relationen.

  Materialisieren heißt Lesen der aktuellen Seite, Ersetzen des Tupels in der Seite durch A_{old}, bzw. A_{new} und Zurückschreiben der Seite.

Bezug zur Mehrbenutzerkontrolle

Die für die Mehrbenutzerkontrolle gewählte Granularität der Objekte muss größer sein, als die für die Fehlerbehandlung gewählte.

Warum?

Sicherungspunkt

- Verbiete den Start neuer Transaktionen und warte bis alle Transaktionen abgeschlossen sind.
- Erzwinge dann das Materialisieren aller sich noch im Datenbankpuffer befindenen Änderungen in der Datenbank.
- Schreibe den Eintrag *(Checkpoint)* in das Log.

Der Restart-Algorithmus muss das Log lediglich bis zum nächsten Sicherungspunkt verarbeiten.

Mediumfehler

- Strategie *Halten aktueller Kopien*:
  - Einen weitgehenden Schutz vor Datenverlust können wir durch Schaffung von Redundanz in Form von Kopien der Datenbank, einschließlich des Logs, erreichen.
  - Zu jedem Zeitpunkt müssen die Datenbank und alle Kopien denselben Zustand repräsentieren.
  - Es müssen so viele Kopien gehalten werden, dass die Wahrscheinlichkeit, dass ein Fehler alle Kopien gleichzeitig zerstört, praktisch gleich 0 ist.

Mediumfehler

- Strategie *periodische Archivierung*.
  - Archiviere eine Kopie der Datenbank (*Dump*).
  - Nach dem Erstellen eines Dumps wird an das Ende des Logs der Eintrag (*archive*) geschrieben.

Restart zur Behebung eines Medium-Fehlers.

- Lade den aktuellen Dump in die Datenbank.
- Wende dann den Restart-Algorithmus zur Behebung eines Systemfehlers bis zum Lesen des (*Archive*) Eintrags an.
- Führe dabei ausschließlich Redo von Transaktionen durch.

10.4 Transaktionsmodelle

- Bisher kurz laufende Transaktionen.
- Findet während des Ablaufs einer Transaktion eine Interaktion mit dem Benutzer statt, dann haben Transaktionen eine nennenswerte Dauer, deren Länge im Allgemeinen nicht beschränkt werden kann.
- Für solche Transaktionen sind unsere bisherigen Mechanismen nur bedingt geeignet.

strukturierte Transaktionen

- Die Funktionalität einer Fehlerbehandlung wird ausgenutzt.
- `SAVE` erzeugt von Seiten der Transaktion einen Zwischenzustand der Transaktion (*Sicherungspunkt*).
- Ein gezieltes Undo als Teil der Transaktion `ROLLBACK` ist möglich.
- `COMMIT` signalisiert der Transaktionsverwaltung das Ende der Transaktion.

Skizze einer Transaktion zur Buchung der benötigten Teilstücke einer Reise.

```
BEGIN
    .
    .
    .
    Solange Reiseziel B noch nicht erreicht
                und Reservierung eines weiteren Teilstücks zur Erreichung von B möglich
        führe Reservierung des Teilstücks durch.
    Wenn Reiseziel B erreicht, dann COMMIT, sonst ROLLBACK.
END
```

Wie kann `SAVE` sinnvoll eingesetzt werden?

Saga

Eine Transaktion T sei durch eine Folge von Subtransaktionen $T_1, \ldots, T_n$ gegeben. Jede Subtransaktion T_i wird wie eine konventionelle Transaktion behandelt; Annahme: 2PL.

- Am Ende einer Subtransaktion werden alle Sperren freigegeben, um ein Minimum an Sperrkonflikten zu erreichen.
- Auf eine weitergehende Synchronisation wird verzichtet, so dass die Gesamttransaktion im Allgemeinen nicht serialisierbar abläuft.
- Es besteht die implizite Annahme, dass nur solche Subtransaktionen im Rahmen von Sagas verwendet werden, bei denen das Sichtbarwerden der Änderungen vor Ende der Gesamttransaktion tolerierbar ist.
- Wenn eine laufende Saga abgebrochen werden muss, ist das Zurücksetzen bereits beendeter Subtransaktionen nur durch eine anwendungsabhängige *Kompensation* möglich. Aus diesem Grund muss von Seiten der Anwendung für jede Subtransaktion T_i eine Kompensationstransaktion T_i^C bereitgestellt werden.

10.5 verteilte Transaktionen

- Ein *verteiltes Datenbanksystem (vDBS)* besteht aus einer Menge von Datenbanksystemen, die im Allgemeinen auf unterschiedlichen, autonomen Rechnern liegen, die durch ein Kommunikationssystem mit einander verbunden sind und unter einander kommunizieren können.
- Die einzelnen Rechner nennen wir *Sites*.
- Die Transaktionen eines verteilten Datenbanksystems können sich über mehrere Sites erstrecken.
- Auf jeder Site läuft eine *Subtransaktion* einer *globalen* Transaktion unter der Kontrolle des lokalen Datenbanksystems der betreffenden Site ab.
- Die globalen Transaktionen führen Benutzeraufträge durch, die Daten aus Datenbanken unterschiedlicher Sites benötigen. Die globalen Transaktionen müssen somit die ACID-Eigenschaften erfüllen.
- Annahme: jedes Objekt der Datenbank wird an genau einer Site gespeichert.

Modell

- Die Menge der an einer Site j gespeicherten Objekte sei D_j.
- Seien m Sites und eine Menge Transaktionen $\mathcal{T} = \{T_1, \ldots, T_n\}$ gegeben.
- Seien weiter $S_1, \ldots, S_m$ die an den einzelnen Sites ablaufenden *lokalen* Schedule.
- Eine Transaktion T_i heißt *global*, wenn ihre zugehörigen Operationen in mehr als einer Site in den zugehörigen lokalen Schedules zur Ausführung kommen; anderenfalls nennen wir sie *lokal*.
- Die an einer Site j in dem lokalen Schedule S_j ablaufende Subtransaktion einer globalen Transaktion T_i identifizieren wir durch T_{ij}.

Schedule

- Abhängigkeiten an den einzelnen Sites zwischen lokalen Transaktionen und lokalen Subtransaktionen, und Abhängigkeiten globaler Transaktionen über mehrere Sites hinweg.
- Es können nur die lokalen Schedule der einzelnen Sites beobachtet werden.
- Wir nennen eine Folge $\mathcal{S}$ von Operationen der Transaktionen aus $\mathcal{T}$ einen *globalen* Schedule zu lokalen Schedulen $S_1, \ldots, S_m$, wenn für jede Site j, $1 \leq j \leq m$, die Projektion von $\mathcal{S}$ auf die an der Site j ausgeführten Operationen gerade den lokalen Schedule S_j ergibt.

Beispiel

Es werden zwei Sites betrachtet. Seien $D_1 = \{A\}, D_2 = \{B\}$ und $S_1 = R_1A\ W_2A$, bzw. $S_2 = W_1B\ R_2B$ die lokalen Schedule zu globalen Transaktionen T_1 und T_2.

$$\mathcal{S}_1 = \begin{array}{lllll} Site1: & R_1A & & W_2A & \\ Site2: & & W_1B & & R_2B \end{array}$$

$$\mathcal{S}_2 = \begin{array}{lllll} Site1: & R_1A & & & W_2A \\ Site2: & & W_1B & R_2B & \end{array}$$

$$\mathcal{S}_3 = \begin{array}{lllll} Site1: & R_1A & & & W_2A \\ Site2: & & R_2B & W_1B & \end{array}$$

Welche der Schedule sind global bzgl. S_1 und S_2?

10.5.1 Serialisierbarkeit

Problematik

- Seien $D_1 = \{A\}$ und $D_2 = \{B, C\}$. Seien weiter $T_1 = R_1A\ W_1B$, $T_2 = R_2C\ W_2A$ globale Transaktionen und sei $T_3 = R_3B\ W_3C$ eine lokal auf Site 2 ablaufende Transaktion.

  $$\begin{array}{llllll} S_1: & R_1A & W_2A & & & \\ S_2: & R_3B & W_1B & R_2C & W_3C & \end{array}$$

- Seien $D_1 = \{A, B\}$ und $D_2 = \{C, D\}$. Seien weiter $T_1 = RA\ RD$ und $T_2 = RB\ RC$ globale Transaktionen und seien $T_3 = RA\ RB\ WA\ WB$ und $T_4 = RD\ WD\ RC\ WC$ lokale Transaktionen.

  $$\begin{array}{llllll} S_1: & R_1A & R_3A & R_3B & W_3A & W_3B & R_2B \\ S_2: & R_4D & W_4D & R_1D & R_2C & R_4C & W_4C \end{array}$$

Die lokalen Schedule sind jeweils serialisierbar; es existiert jedoch kein serialisierbarer globaler Schedule.

Mehrbenutzerkontrolle

- *homogen:* auf jeder Site kommt dieselbe Technik zur Gewährleistung der Serialisierbarkeit zum Einsatz.
- *heterogen:* es können unterschiedliche Verfahren zum Einsatz kommen.

2-Phasen Sperrverfahren

- Die Sperrverwaltung wird einer ausgewählten Site übertragen, die alle Sperren einer globalen Transaktion verwaltet und somit überwachen kann, dass die 2PL-Regel global eingehalten wird.
- Vor Beginn der Freigabephase synchronisieren sich die einzelnen Subtransaktionen einer globalen Transaktion, so dass nach Beginn der Freigabephase einer Subtransaktion keine andere Subtransaktion derselben globalen Transaktion mehr eine Sperre erhalten kann.
- Die Subtransaktionen halten Sperren bis zur Einleitung einer zwischen allen Subtransaktionen einer globalen Transaktion synchronisierten gemeinsamen Commit-Phase.

Deadlockerkennung

- Sende die Wartegraphen der einzelnen Sites periodisch an eine ausgewählte Site und vereinige sie dort zu einem globalen Wartegraphen.
- Verwende *Time-outs*.
- Die wartende Transaktion sendet ihre Kennung in einer speziellen Nachricht an die sie blockierende Transaktion. Erhält eine Transaktionen eine solche Nachricht, so sendet sie diese an alle Transaktionen weiter, auf die sie selbst wartet. Ist sie nicht blockiert, so ignoriert sie Nachricht. Erhält eine Transaktion ihre eigene Nachricht, so bricht sie sich ab, um die erkannte Deadlock-Situation aufzulösen.

Vorteile/Nachteile?

Zeitmarken

- Globale und rein lokal ablaufende Transaktionen erhalten zu Beginn eine Zeitmarke; alle Subtransaktionen einer globalen Transaktion erhalten dieselbe Zeitmarke.
- Es muss sichergestellt sein, dass nicht an unterschiedlichen Sites gleiche Zeitmarken vergeben werden: Zeitmarken werden als Paar (*lokale Zeitmarke, Site-Kennung*) dargestellt.
- Die den Zeitmarken zugrunde liegenden Zähler (Systemuhren) an den Sites müssen synchronisiert werden, damit Diskrepanzen vermieden werden, die sonst unnötige Abbrüche der Transaktionen bewirken könnten.

Ticketverfahren

- An den einzelnen Sites kommen unterschiedliche Verfahren zum Einsatz.
- Die lokalen Mehrbenutzerkontrollen haben keine Kenntnis darüber haben, dass auf ihnen Subtransaktionen globaler Transaktionen ablaufen können.
- Jede Site unterhält ein spezielles Objekt in der Datenbank, genannt *Ticket*, zu dem jede Subtransaktion einer globalen Transaktion zugreift. Eine Subtransaktion liest das entsprechende Ticket, erhöht den Wert um 1 und schreibt den neuen Wert zurück (*take-a-ticket*-Operation).
- Auf einer Site existiert eine globale Kontrollinstanz, der *globale Transaktionsverwalter*.
- Der globale Transaktionsverwalter garantiert, dass die Ticketordnung der Subtransaktionen eine lineare Ordnung auf den globalen Transaktionen impliziert.

Beispiele

Das Ticket der Site j sei I_j. Seien $D_1 = \{A, B\}, D_2 = \{C, D\}$. Seien weiter $T_1 = R_1A\ R_1D$ und $T_2 = R_2B\ R_2C$ globale Transaktionen und seien $T_3 = R_3A\ R_3B\ W_3A\ W_3B$ und $T_4 = R_4D\ W_4D\ R_4C\ W_4C$ lokale Transaktionen.

- $S_1: \quad R_1(I_1)\ W_1(I_1)\ R_1A\ R_3A\ R_3B\ W_3A\ W_3B\ R_2(I_1)\ W_2(I_1)\ R_2B$
  $S_2: \quad R_4D\ W_4D\ R_1(I_2)\ W_1(I_2)\ R_1D\ R_2(I_2)\ W_2(I_2)\ R_2C\ R_4C\ W_4C$

  Serialisierbar?

- $S_1: \quad R_1(I_1)\ W_1(I_1)\ R_1A\ R_2(I_1)\ W_2(I_1)\ W_2A$
  $S_2: \quad R_2(I_2)\ W_2(I_2)\ R_2B\ R_1(I_2)\ W_1(I_2)\ W_1B$

  Diskutiere!

10.5.2 Fehlerbehandlung

Problematik

- Eine globale Transaktion besteht aus Subtransaktionen, die an einzelnen Sites lokal ablaufen. Die Atomizität einer einzelnen Subtransaktion ist durch das lokale Datenbanksystem gewährleistet.
- Es kann möglich sein, dass eine vollständig abgelaufene Subtransaktion abgebrochen werden muss, weil eine andere Subtransaktion derselben globalen Transaktion, aufgrund eines Fehlers abgebrochen wurde.
- Jede Subtransaktion muss somit als Teil ihrer Commit-Phase Gewissheit erlangen können, dass sie und alle anderen Subtransaktionen erfolgreich ihr Commit beenden können.
- Gelingt dies nicht, dann müssen alle Subtransaktionen abgebrochen werden. Bereits durchgeführte Änderungen in den lokalen Datenbanken werden dann mittels lokalem Undo wieder entfernt.
- Für die Entscheidung über ein gemeinsames Commit müssen die einzelnen Subtransaktionen mittels Austausch von Nachrichten kommunizieren.

2-Phasen-Commit Protokoll (2PC)

- Für jede globale Transaktion T müssen wir eine Site bestimmen, deren Transaktionsverwaltung die Koordination übernimmt *Koordinator*.
- Die Transaktionsverwalter der übrigen Sites, an denen Subtransaktionen dieser Transaktion ablaufen, nennen wir *Teilnehmer*.
- Typischerweise definiert diejenige Site den Koordinator, an der die Transaktion gestartet wird.
- Das 2-Phasen-Commit Protokoll besteht aus zwei Phasen.
  - Wenn die erste Phase endet, dann sind entweder alle Subtransaktionen bereit, ihre lokale Commit-Phase erfolgreich zu beenden, oder der Koordinator weiß, dass dies nicht möglich ist.
  - In der folgenden zweiten Phase können dann alle Subtransaktionen ihr Commit durchführen, oder alle Subtransaktionen werden abgebrochen.

2PC-Protokoll

1. Der Koordinator schreibt zu Beginn $(T, \mathit{Coordinator})$ in sein Log. Wird er mit der Durchführung eines globalen Commit beauftragt, dann beginnt die erste Phase des Protokolls. Der Koordinator schreibt $(T, \mathit{Prepare})$ in sein Log und sendet an jeden Teilnehmer T^i die Nachricht *Prepare*.
2. Wenn ein Teilnehmer eine *Prepare*-Nachricht vom Koordinator erhält, entscheidet er, ob die zugehörige Subtransaktion T^i in der Lage ist, die Commit-Phase erfolgreich zu durchlaufen, oder nicht. Im ersten Fall schreibt der Teilnehmer (T^i, Yes) in sein Log und antwortet dem Koordinator mit der Nachricht *Yes*, im anderen Fall schreibt der Teilnehmer (T^i, No) in sein Log, bricht die Subtransaktion ab und sendet die Nachricht *No* an den Koordinator.
3. Erhält der Koordinator von allen Teilnehmern die Nachricht *Yes*, dann entscheidet er auf globales Commit, er schreibt (T, Commit) in sein Log und antwortet an alle Teilnehmer mit *Commit*. Erhält er von mindestens einem Teilnehmer die Nachricht *No*, dann entscheidet er auf globales Abort, vermerkt dies in seinem Log durch (T, Abort) und sendet zu allen Teilnehmern die Nachricht *Abort*. Die zweite Phase des Protokolls ist begonnen.
4. Erhält ein Teilnehmer vom Koordinator die Nachricht *Abort*, dann schreibt er (T^i, Abort) in sein Log und bricht die zugehörige Subtransaktion ab. Erhält er die Nachricht *Commit*, dann schreibt er (T^i, Commit) in sein Log und schließt damit die Commit-Phase der zugehörigen Subtransaktion ab.

Probleme bedingt durch Fehler im Kommunikationssystem

- Ein Teilnehmer wartet auf die *Prepare*-Nachricht. Da noch keine Entscheidung über ein gemeinsames Commit vom Koordinator gefällt wurde, kann der Teilnehmer seine Subtransaktion abbrechen.
- Der Koordinator wartet auf Antworten auf seine *Prepare*-Nachricht. Da auch jetzt noch keine Entscheidung erfolgte, kann er auf Abbruch entscheiden und allen Teilnehmern, die bereits mit *Yes* antworteten, mit *Abort* antworten.
- Ein Teilnehmer hat *Yes* geantwortet und wartet auf die Entscheidung des Koordinators. Der Teilnehmer ist *unsicher* und kann nicht selbständig entscheiden. Er kann seine Subtransaktion nicht abbrechen, da er mit *Yes* geantwortet hat und diese Antwort in die Entscheidung des Koordinators eingegangen sein kann. Er kann seine Commit-Phase nicht abschließen, da möglicherweise der Koordinator auf *Abort* entschieden hat. Der Teilnehmer ist somit auf unbestimmte Zeit blockiert.

Was dann?