9.3 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 Festplattenfehler.

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.

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 direkt oder indirekt über gelesene Werte abhängt, müssen ihre Commit-Phase erfolgreich durchlaufen haben.

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 direkt oder indirekt über gelesene Werte abhängt, müssen ihre Commit-Phase erfolgreich durchlaufen haben.

UNDO und REDO bei Vorliegen eines Transaktions- bzw. Systemfehlers

- Kritisch ist, ob die von einer Transaktion bewirkten Änderungen bereits in der Datenbank materialisiert wurden, oder ob sie erst im Datenbankpuffer wirksam wurden.
  - Kann es erforderlich werden, eine geänderte Seite in der Datenbank zu materialisieren, bevor die Transaktion ihre Commit-Phase durchlaufen hat?
    Ja, wenn der Datenbankpuffer voll ist und sonst keine Seite mehr geladen werden kann.

UNDO und REDO bei Vorliegen eines Transaktions- bzw. Systemfehlers

- Kritisch ist, ob die von einer Transaktion bewirkten Änderungen bereits in der Datenbank materialisiert wurden, oder ob sie erst im Datenbankpuffer wirksam wurden.
  - Kann es erforderlich werden, eine geänderte Seite in der Datenbank zu materialisieren, bevor die Transaktion ihre Commit-Phase durchlaufen hat?
    Ja, wenn der Datenbankpuffer voll ist und sonst keine Seite mehr geladen werden kann.
  - 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?

UNDO und REDO bei Vorliegen eines Transaktions- bzw. Systemfehlers

- Kritisch ist, ob die von einer Transaktion bewirkten Änderungen bereits in der Datenbank materialisiert wurden, oder ob sie erst im Datenbankpuffer wirksam wurden.
  - Kann es erforderlich werden, eine geänderte Seite in der Datenbank zu materialisieren, bevor die Transaktion ihre Commit-Phase durchlaufen hat?
    Ja, wenn der Datenbankpuffer voll ist und sonst keine Seite mehr geladen werden kann.
  - 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?
    Ja, wenn diese Seite auch von anderen noch laufenden Transaktionen benötigt wird.

UNDO und REDO bei Vorliegen eines Transaktions- bzw. Systemfehlers

- Kritisch ist, ob die von einer Transaktion bewirkten Änderungen bereits in der Datenbank materialisiert wurden, oder ob sie erst im Datenbankpuffer wirksam wurden.
  - Kann es erforderlich werden, eine geänderte Seite in der Datenbank zu materialisieren, bevor die Transaktion ihre Commit-Phase durchlaufen hat?
    Ja, wenn der Datenbankpuffer voll ist und sonst keine Seite mehr geladen werden kann.
  - 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?
    Ja, wenn diese Seite auch von anderen noch laufenden Transaktionen benötigt wird.
- Eine Transaktion schreibt vor Abschluss ihrer Commit-Phase in die Datenbank und bevor das Commit erfolgreich ausgeführt werden konnte, tritt ein Fehler auf
  $\Rightarrow$ *UNDO-Situation*
- Eine Transaktion hat ihr Commit erfolgreich ausgeführt und es tritt ein Fehler auf, bevor alle geänderten Seiten in der Datenbank materialisiert sind
  $\Rightarrow$ *REDO-Situation*

UNDO und REDO bei Vorliegen eines Transaktions- bzw. Systemfehlers

- Kritisch ist, ob die von einer Transaktion bewirkten Änderungen bereits in der Datenbank materialisiert wurden, oder ob sie erst im Datenbankpuffer wirksam wurden.
  - Kann es erforderlich werden, eine geänderte Seite in der Datenbank zu materialisieren, bevor die Transaktion ihre Commit-Phase durchlaufen hat?
    Ja, wenn der Datenbankpuffer voll ist und sonst keine Seite mehr geladen werden kann.
  - 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?
    Ja, wenn diese Seite auch von anderen noch laufenden Transaktionen benötigt wird.
- Eine Transaktion schreibt vor Abschluss ihrer Commit-Phase in die Datenbank und bevor das Commit erfolgreich ausgeführt werden konnte, tritt ein Fehler auf
  $\Rightarrow$ *UNDO-Situation*
- Eine Transaktion hat ihr Commit erfolgreich ausgeführt und es tritt ein Fehler auf, bevor alle geänderten Seiten in der Datenbank materialisiert sind
  $\Rightarrow$ *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 sogenannte *After-Image* von A) und A_{old} ist der Wert von A vor Ausführung von WA in der Datenbank (das sogenannte *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.

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 sogenannte *After-Image* von A) und A_{old} ist der Wert von A vor Ausführung von WA in der Datenbank (das sogenannte *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.
- 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.

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 sogenannte *After-Image* von A) und A_{old} ist der Wert von A vor Ausführung von WA in der Datenbank (das sogenannte *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.
- 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

- $Redone := \emptyset$; $Undone := \emptyset$.
- Verarbeite das Log rückwärts.
  Sei (T, A, A_{old}, A_{new}) der betrachtete Log-Eintrag.
  Falls $A \notin (Redone \cup Undone)$:
  - Redo: Falls $(T, Commit)$ bereits im Log gefunden, dann materialisiere A_{new} in der Datenbank und setze $Redone := Redone \cup \{A\}$
  - Undo: Anderenfalls materialisiere A_{old} in der Datenbank und setze $Undone := Undone \cup \{A\}$

Objekte: Seiten oder Tupel?

- Seiten der physikalischen Datenbank:

  Materialisieren von A_{old} bzw. A_{new} in der Datenbank ist das (blinde) Überschreiben 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.

Objekte: Seiten oder Tupel?

- Seiten der physikalischen Datenbank:

  Materialisieren von A_{old} bzw. A_{new} in der Datenbank ist das (blinde) Überschreiben 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.

Sicherungspunkte begrenzen das Log-File

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

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

Mediumfehler

(a) Strategie 1: *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.

(b) Strategie 2: *periodische Archivierung*

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

Mediumfehler

(a) Strategie 1: *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.

(b) Strategie 2: *periodische Archivierung*

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

Dump-basierter 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.

9.4 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 und Auswahl eines neuen Reiseziels.
END
```

Mittels `SAVE` können Zwischenzustände gerettet werden.

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 und Auswahl eines neuen Reiseziels.
END
```

Mittels `SAVE` können Zwischenzustände gerettet 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.

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.

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.

empfohlene Lektüre

Management Applications — H. Morgan Editor

The Notions of Consistency and Predicate Locks in a Database System

K.P. Eswaran, J.N. Gray,
R.A. Lorie, and I.L. Traiger
IBM Research Laboratory
San Jose, California

In database systems, users access shared data under the assumption that the data satisfies certain consistency constraints. This paper defines the concepts of transaction, consistency and schedule and shows that consistency requires that a transaction cannot request new locks after releasing a lock. Then it is argued that a transaction needs to lock a logical rather than a physical subset of the database. These subsets may be specified by predicates. An implementation of predicate locks which satisfies the consistency condition is suggested.

Key Words and Phrases: consistency, lock, database, concurrency, transaction
CR Categories: 4.32, 4.33

1

The Serializability of Concurrent Database Updates

CHRISTOS H. PAPADIMITRIOU

Massachusetts Institute of Technology, Cambridge, Massachusetts

ABSTRACT A sequence of interleaved user transactions in a database system may not be *serializable*, i e, equivalent to some sequential execution of the individual transactions Using a simple transaction model, it is shown that recognizing the transaction histories that are serializable is an NP-complete problem. Several efficiently recognizable subclasses of the class of serializable histories are therefore introduced; most of these subclasses correspond to serializability principles existing in the literature and used in practice Two new principles that subsume all previously known ones are also proposed Necessary and sufficient conditions are given for a class of histories to be the output of an efficient history scheduler, these conditions imply that there can be no efficient scheduler that outputs all of serializable histories, and also that all subclasses of serializable histories studied above have an efficient scheduler Finally, it is shown how these results can be extended to far more general transaction models, to transactions with partly interpreted functions, and to distributed database systems

KEY WORDS AND PHRASES database management, concurrent update problem, transactions, serializability, schedulers, concurrency control

2

[1] In: Communication of the ACM, Vol. 19, No. 11, 1976.

[2] In: Journal of the ACM, Vol. 26, No. 4, 1979.