# Recovery

Udo Kelter

17.04.2003

**Zusammenfassung dieses Lehrmoduls**

Unter dem Begriff Recovery faßt man diverse Maßnahmen zusammen, die eine Datenbank vor Schäden infolge von Störungen schützen oder die Schäden beheben. In diesem Lehrmodul untersuchen wir zunächst die Ursachen und Folgen von Störungen und bilden drei wichtige Fehlerklassen, nämlich Medienfehler, Systemfehler und Transaktionsfehler. Wir diskutieren die generellen Anforderungen an das Recovery und stellen dann allgemeinere Grundprinzipien des Recovery und für alle drei Fehlerklassen Datenstrukturen und Algorithmen vor, die Schäden vorbeugen oder sie beheben.

**Vorausgesetzte Lehrmodule:**

obligatorisch:
- Transaktionen und die Integrität von Datenbanken
- Architektur von DBMS

empfohlen:
- Datenverwaltungssysteme

**Stoffumfang in Vorlesungsdoppelstunden:** 2.5

# Inhaltsverzeichnis



©2003 Udo Kelter **Stand: 17.04.2003**
Dieser Text darf für nichtkommerzielle Nutzungen als Ganzes und unverändert in elektronischer oder gedruckter Form beliebig weitergegeben werden und in WWW-Seiten, CDs und Datenbanken aufgenommen werden. Jede andere Nutzung, insb. die Veränderung und Überführung in andere Formate, bedarf der expliziten Genehmigung. Die jeweils aktuellste Version ist über http://kltr.de erreichbar.

©2003 Udo Kelter **Stand: 17.04.2003**
Dieser Text darf für nichtkommerzielle Nutzungen als Ganzes und unverändert in elektronischer oder gedruckter Form beliebig weitergegeben werden und in WWW-Seiten, CDs und Datenbanken aufgenommen werden. Jede andere Nutzung, insb. die Veränderung und Überführung in andere Formate, bedarf der expliziten Genehmigung. Die jeweils aktuellste Version ist über http://kltr.de erreichbar.

# 1 Einführung

Dieses Lehrmodul erhebt nicht den Anspruch, eine auch nur annähernd vollständige Darstellung des Themenkomplexes Recovery in Datenbanken zu sein; eine solche würde leicht ein eigenes Buch füllen. Ursache ist die Vielfalt der Einzelaspekte des Recoverys; diese Vielfalt mag überraschen, da die Grundideen des Recoverys recht einfach sind: in [TID] wurden bereits die beiden wichtigsten Prinzipien vorgestellt, wie eine Datenbank nach einer Beschädigung mit Hilfe von sogenannten Recovery-Daten wiederhergestellt werden kann.

Von den Grundideen wird jedoch in vielerlei Weise abgewichen, was auch die Gliederung des Stoffes erschwert:

- Teilweise wird verhindert, daß überhaupt Schäden an der Datenbank eintreten, es handelt sich also nicht um Wiederherstellung, sondern um *Präventivmaßnahmen*. Die Präventiv-Prinzipien und -Techniken werden deswegen unter Recovery subsumiert, weil die Mechanismen fast die gleichen sind wie im Falle einer wirklichen Wiederherstellung des Zustands der Datenbank.
- Teilweise werden auch außerhalb der Datenbank liegende Beschädigungen mitbehoben, vor allem Schäden an laufenden Prozessen, deren Behebung genausogut *Aufgabe des Betriebssystems* sein könnte. Der Grund ist, daß die gemeinsame Behebung dieser Schäden effizienter ist. Diese Einbeziehung führt jedoch zu einer Aufblähung des Stoffes und Involvierung mit diversen Betriebssystemkonzepten; für detaillierte Algorithmen sind dann auch detaillierte Angaben über das Betriebssystem erforderlich. Dies macht auch eine Vorstellung realer Systeme sehr aufwendig.
- Teilweise ist die Trennung zwischen den Recovery-Daten und der *Arbeitsversion* der Datenbank schwierig; eine Unterscheidung ist dann eher willkürliche Interpretation anhand der Algorithmen auf den gesamten Daten. Dies gilt besonders bei Präventivtechniken.
- Teilweise ist das Recovery der Datenbank eng verzahnt mit dem *Recovery der Recovery-Daten*. Beschädigt werden können natürlich auch die Recovery-Daten.

©2003 Udo Kelter 

Der Bereich Recovery hat eine Vielzahl von Berührungspunkten zu anderen Problembereichen, vor allem in Realisierungsfragen. Dies führt dazu, daß viele Konzepte aus diesen Nachbargebieten eindringen und zu einer zusätzlichen Erweiterung des Themenbereichs führen:

- Für das Recovery sind viele komplexe Funktionen zu erfüllen; diese sind letztlich in Programmen zu realisieren, wodurch sich eigene programmiertechnische Probleme ergeben. (Meist ist ein wesentlicher Teil des DBMS, etwa 10 - 30 %, den Recovery-Aufgaben gewidmet.)
- Für eine optimale Effizienz müssen Recovery-Techniken Eigenschaften der Implementierung der Datenbank und des Betriebssystems ausnutzen, z.T. sogar Eigenschaften der Hardware.
- Alle Recovery-Maßnahmen können auch dahingehend interpretiert werden, daß sie die *Zuverlässigkeit* des Datenbanksystems oder des gesamten DV-Systems erhöhen. Viele Maßnahmen sind allgemein verwendbar zur Zuverlässigkeitssteigerung beliebiger DV-Systeme, eventuell sind sie angepaßt an die speziellen Verhältnisse in Datenbanken. Daher ergibt sich in einer Vielzahl von Detailthemen eine Überschneidung mit dem Bereich "Zuverlässigkeit von DV-Systemen".
- In Datenbanksystemen, die parallel benutzt werden, sind Recovery-Probleme konzeptionell und in der Realisation mehr oder weniger stark beeinflußt vom *Concurrency-Control-Problem* und den dort vorhandenen Konzepten.

Auf das Recovery in verteilten Datenbanken gehen wir hier überhaupt nicht ein; eine sehr ausführliche Behandlung findet sich in [BeHG87].

**Bemerkungen zur Stoffgliederung.** Eine geradlinige Gliederung des Stoffes sollte ausgehen von einer Analyse des Problems bzw. der Aufgabenstellung und aus dieser systematisch die zu ergreifenden Maßnahmen bzw. zu benutzenden Algorithmen herleiten. An diesem roten Faden orientiert sich die Gliederung des Stoffes in diesem Lehrmodul:

©2003 Udo Kelter 

Zuerst wird untersucht, welche Störungen in DV-Systemen auftreten können, die die Integrität der Datenbank verletzen (Abschnitt 2). Die Möglichkeiten für Störungen und resultierende Schäden an der Datenbank sind sehr vielfältig; es stellt sich allerdings heraus, daß die Schäden in ein einfacheres Schema gepreßt werden können, welches in etwa die reparierbaren Einheiten enthält (welche letztlich durch die verfügbaren Reparaturtechniken bestimmt werden). Als nächstes wird untersucht, was nach der Störung das Ziel und die Randbedingungen des Recoverys sind, genauer, welcher neue Zustand der Datenbank angestrebt wird und welche sonstigen Anforderungen zu beachten sind (Abschnitt 3). Danach werden die Grundprinzipien des Recoverys (Abschnitt 4) vorgestellt. In den Abschnitten 6, 7 und 8 werden besonders die Techniken des transaktionsbezogenen Recoverys genauer behandelt. Zuvor diskutieren wir in Abschnitt 5 Alternativen bei der Erzeugung von Logdaten.

Die schon oben erwähnten Abweichungen von der Grundidee des Recoverys führen zu gelegentlichen Abweichungen vom roten Faden durch den Stoff. Eine weitere Schwierigkeit, den roten Faden beizubehalten, liegt in der starken Rückkopplung von den Realisationsformen des Recoverys auf seine Ziele und damit auf seine Prinzipien:

- Teilweise wird das Ziel des Recoverys dadurch bestimmt, was in einer gegebenen Umgebung effektiv und effizient machbar ist.
- Durch die Notwendigkeit des Recoverys der Recovery-Daten werden die Ziele des Recoverys ebenfalls durch die Techniken bestimmt. Hierdurch sind gelegentlich Vorwärtsverweise im Text nötig.

# 2 Quellen und Auswirkungen von Störungen

Recovery-Maßnahmen richten sich gegen Störungen in irgendwelchen Teilen des gesamten Systems aus Hardware, Betriebssystem, DBMS und Benutzerprogrammen, die zum Verlust von Daten bzw. zum Verlust der Korrektheit der Daten führen können. Unter einer **Störung** verstehen wir ein Ereignis, bei dem irgendwelche Teile des Systems nicht erwartungsgemäß bzw. gemäß ihrer Spezifikation arbeiten; man

©2003 Udo Kelter 

kann dies auch einen Fehler oder ein Fehlverhalten des entsprechenden Systemteils nennen. Der Effekt einer Störung besteht in einer unerwünschten Veränderung des Inhalts der Datenbank; dies nennen wir einen **Schaden** in der Datenbank. Die Fehlerquellen sollen in diesem Abschnitt genauer untersucht werden; die interessierenden Merkmale sind:

- Wo in der Systemarchitektur entstand die Störung und aus welchem Grund?
- Welche Schäden (in den Daten bzw. Aktivitäten) sind eingetreten?
- Wie werden die Störungen gemeldet, d.h. wie erhalten die Teile des DBMS, die sich mit Recovery befassen, Informationen über Art und Umfang von Schäden?

## 2.1 Systemarchitektur

Ein DBMS ist im engeren Sinne nur eine Komponente eines DV-Systems, welches auf einem Rechner ausgeführt wird. Nur wenige Arten von Störungen "entstehen" im DBMS. Um eine grobe Zuordnung der Arten zu ermöglichen, nehmen wir folgende Schichten einer Systemarchitektur an (eine ausführlichere Darstellung findet sich in [DBSA]):

Ein Benutzerprogramm kann on-line oder off-line ablaufen. Es startet i.d.R. mehrere Transaktionen; z.B. kann bei einer on-line Sitzung zur Datenerfassung jede einzelne Dateneingabe eine Transaktion auslösen.

Transaktionen werden in dieser Systemarchitektur als Programme (bzw. als spezielle Unterprogramme von Benutzerprogrammen) verstanden. Eine Transaktion kann mehrere Datenbankzugriffe ausführen[1]. Transaktionen werden zwar vom Benutzer programmiert, sind jedoch dem DBMS als solche bekannt.

Wir gehen i.f. davon aus, daß das DBMS nicht selbst direkt auf den Platten arbeitet, sondern das Betriebssystem beauftragt, Daten zwischen Arbeitsspeicher und permanenten Speichern zu transportieren.

[1] Den Fall geschachtelter Transaktionen betrachten wir hier nicht.

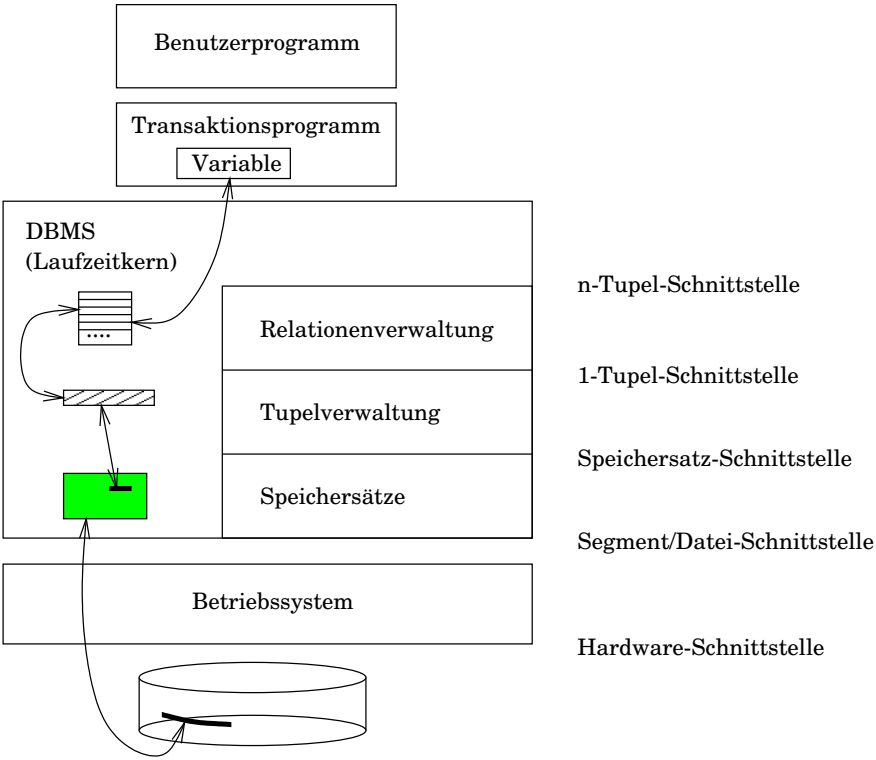


Abbildung 1: Eine Schichtenarchitektur für DBMS und Transaktionsprogramme

Die Komponente des DBMS, die die Recovery-Maßnahmen durchführt, nennen wir **Recovery-Manager**.

Die höheren Schichten in der Schichtenarchitektur veranlassen Transporte der auf der jeweiligen Schicht vorhandenen Datengranulate zwischen Arbeitsspeicher und Permanentspeichern. Hierbei werden (vor allem bei satzorientiertem Transfer) Pufferungstechniken angewandt, durch die dem DBMS der Abschluß eines Transports zum Permanentspeicher signalisiert wird, obwohl die Daten effektiv noch nicht dort vorhanden sind. Nehmen wir zur Illustration eine Operati-

on `write(X,v)` an, die den Wert `v` im Objekt oder Tupel `X` speichert. Hierbei sind folgende Schritte auszuführen:

- Sofern die entsprechende Seite noch nicht geladen ist, muß sie erst geladen werden.
- Der Wert `v` wird in die entsprechende Stelle der Seite kopiert.
- Anschließend muß die Seite noch auf Platte zurückgeschrieben werden. Dies ist aber u.U. nicht sofort möglich, weil z.B. noch andere, nicht beendete Transaktionen in anderen Teilen der Seite arbeiten.

Es kann also längere Zeit dauern, bis eine Änderung "materialisiert" wird. Wenn währenddessen ein Systemfehler eintritt, sind diese Änderungen verloren. Aufgrunddessen ist es sinnvoll, folgende Varianten des Begriffs Datenbank zu unterscheiden:

- Die **physische Datenbank** dies ist der *physische* Zustand, der sich allein aufgrund des Inhalts der persistenten Medien ergibt. Er ist relevant für Rettungsprogramme nach Medienfehlern
- Die **materialisierte Datenbank** ist der *logische* Zustand, der sich aufgrund des Inhalts der persistenten Medien ergibt. Er ist relevant für den Neustart nach einem Systemfehler.
- Die **aktuelle Datenbank** ist der logische Zustand, der sich aufgrund des Inhalts der persistenten *und* der flüchtigen Medien ergibt. Er ist relevant für den normalen Betrieb.

## 2.2 Fehlerklassen

Es gibt vielfältige Arten von Störungen und resultierenden Schäden. Es lohnt sich aber nicht, alle erdenklichen Arten separat zu behandeln, weil man letztlich wegen der verfügbaren Methoden zur Reparatur der Schäden i.w. nur drei Arten von Schäden (bzw. Fehlern) unterscheiden muß. Musterbeispiele von Störungen für die jeweiligen Klassen sind:

- Mediendefekte
- Systemabstürze
- ungeplante Transaktionsabbrüche

Im folgenden beschreiben wir die Fehler und die im Prinzip erforderlichen Schritte zur Behebung der Schäden für jede der drei Klassen. Diese Schritte bezeichnen wir als **Recovery-Grundfunktionen**.

### 2.2.1 Medienfehler

Störungen dieser Art sind recht selten; bei ihnen kann auf Speichermedien nicht mehr zugegriffen werden. Ursachen sind z.B.:

- Gerätedefekte;
- Defekte auf dem eigentlichen Datenträger (Kopfaufsetzer bei einer Magnetplatte, Bandriß, Zerstörung von Datenträgern usw.);
- Organisatorische Fehler (Verlust von Datenträgern, versehentliche Löschung, falsche Anordnung auf Geräten).

In einfachen Fällen kann die Störung durch mehrfache Zugriffsversuche oder andere Maßnahmen im Betriebssystem behoben werden, ohne daß sich die Störung auf das DBMS auswirkt. Die erforderlichen Mechanismen sollen hier nicht diskutiert werden.

In schwererwiegenden Fällen sind mehr oder weniger große Teile der Datenbank verloren oder ihre Zugriffspfade zerstört. Dies nennen wir einen **Medienfehler**. Die Datenbank wird i.d.R. hierdurch *physisch inkonsistent*. Die verlorenen Daten können im laufenden Betrieb nicht rekonstruiert werden. Daher treten in gewisser Weise Folgeschäden ein: Das DBMS muß alle Aktivitäten auf der Datenbank, also alle laufenden Transaktionen und Benutzerprogramme, abbrechen.

Die Reparatur der Schäden ist nur möglich, indem die Datenbank (ggf. nur die betroffenen Teile) rekonstruiert werden. Hierzu benötigt man die beiden folgenden Recovery-Grundfunktionen:

**Neuladen** einer früher erzeugten Sicherungskopie der Datenbank

**globales Redo:** Nachholen aller Änderungen, die seit dem Erzeugen der Sicherungskopie stattgefunden haben.

Bei sehr großen Datenmengen kann es durchaus Stunden dauern, eine Datenbank auf diese Weise zu rekonstruieren, d.h. bei unternehmens-

kritischen Anwendungen muß durch entsprechende Maßnahmen die Wahrscheinlichkeit eines Medienfehlers extrem klein gemacht werden.

### 2.2.2 Systemfehler

Ursache für Störungen im Betriebssystem, die zu Systemabstürzen führen, sind, wenn man so will, Programmierfehler des DBMS-Herstellers. Eine weitere Ursache für Systemfehler sind Stromausfälle.

Die Schäden, die derartige Störungen verursachen können, liegen in drei Bereichen:

- Aktivitäten auf der Datenbank, insbesondere Transaktionen, werden unvollständig ausgeführt; dies läuft auf Transaktionsfehler hinaus, die unten diskutiert werden.
- Der Inhalt der flüchtigen Speicher, also insb. der Puffer, ist verloren. Nach einem Neustart des Systems ist zunächst nur die materialisierte Datenbank verfügbar. Sogar der Effekt bereits abgeschlossener Transaktionen kann verloren gehen.
- Daten, die sich bereits auf Permanentspeichern befinden, werden bei Systemfehlern normalerweise nicht direkt beschädigt. Wenn allerdings das DBMS oder das unterliegende Betriebssystem innerhalb einer Aktion unterbrochen wurde, können Zugriffsstrukturen beschädigt werden und damit große Teile der Datenbank auf den Permanentspeichern physisch inkonsistent sein. Dies ist dann wie ein Medienfehler zu behandeln.

Sofern die Datenbank nur logisch inkonsistent wird, aber physisch konsistent bleibt, sprechen wir von einem **Systemfehler**.

Um die möglichen Schäden in der Datenbank genauer zu analysieren, betrachten wir die Graphik in Bild 2:

- Die Transaktionen T3 und T5 werden abgebrochen und hinterlassen möglicherweise die Datenbank in einem inkonsistenten Zustand.
- Die Transaktionen T1, T2 und T4 sind zwar schon abgeschlossen, aber ihre Effekte sind möglicherweise noch nicht materialisiert und gehen daher durch den Systemfehler verloren.

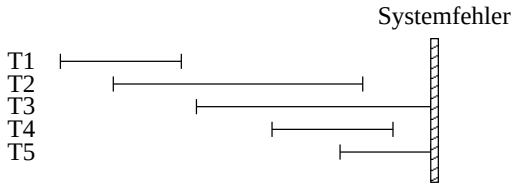


Abbildung 2: Abbruch von Transaktionen bei einem Systemfehler

Zur Behebung dieser Schäden werden die folgenden Recovery-Grundfunktionen benötigt:

**globales Undo:** macht die Wirkung aller unterbrochenen Transaktionen rückgängig; dies beinhaltet für jede Aktion, die eine dieser Transaktionen durchgeführt hat, ein ...

**Undo einer Aktion:** macht die Wirkung einer Aktion rückgängig

**partielles Redo:** holt die Änderungen der Transaktionen nach, die vor dem Systemfehler beendet wurden und deren Wirkung nicht in der materialisierten Datenbank enthalten ist; dies beinhaltet für jede betroffene Aktion ein ...

**Redo einer Aktion:** wiederholt die Wirkung einer Aktion

Ausgehend von der Annahme, daß ein Betriebssystemabsturz durchaus öfter vorkommt – über diese Annahme soll hier nicht diskutiert werden – kommt man zur Forderung, daß das globale Undo und partielle Redo sehr rasch, d.h. innerhalb von Minuten durchgeführt werden müssen.

Weiterhin ist der Fall denkbar, daß während des globalen Undos oder des partiellen Redos erneut ein Systemfehler auftritt, so daß diese Grundfunktionen nicht vollständig ausgeführt werden. Hierfür müssen entsprechende Vorsorgemaßnahmen getroffen werden.

©2003 Udo Kelter 

### 2.2.3 Transaktionsfehler

Unter dem Begriff Transaktionsfehler faßt man alle Störungen zusammen, die zum Abbruch einer Transaktion führen, z.B.:

- Laufzeitfehler, z.B. eine Division durch Null, fehlende Autorisierung bei einem Zugriff auf die Datenbank
- Abbruch der Transaktion durch das DBMS, u.a.
  - wegen einer anderen Störung wie einem Medienfehler
  - zur Deadlockauflösung
  - zur Synchronisation; es gibt Concurrency-Control-Verfahren, bei denen Transaktionen mit Hilfe von Rücksetzungen synchronisiert werden
- programmierter Abbruch einer Transaktion (z.B. in JDBC mittels `rollback`-Operation), i.d.R. als Folge negativ ausgegangener Konsistenzprüfungen

Bei der Deadlockauflösung und beim Zurücksetzen zu Synchronisationszwecken darf das Benutzerprogramm nichts von dem Transaktionsfehler bemerken, die Transaktion muß hier vom DBMS automatisch neu gestartet werden. Hierzu muß die Transaktion analog zu gespeicherten Prozeduren serverseitig (!) als ausführbares Programm vorliegen, d.h. es muß hier eine deutlich andere Architektur vorliegen als bei den grundlegenden Formen der Nutzung von JDBC.

Der Schaden in der Datenbank besteht darin, daß die Wirkungen der bis zum Abbruch der Transaktion ausgeführten Aktionen i.a. zu einem logisch inkonsistenten Datenbankzustand führen.

Zur Behebung dieser Schäden wird die folgende Recovery-Grundfunktion benötigt:

**Rollback einer Transaktion:** macht die Wirkung der abgebrochenen Transaktion rückgängig; dies beinhaltet wie schon beim globalen Undo für jede Aktion, die diese Transaktionen durchgeführt hat, ein Undo dieser Aktion.

## 2.3 Ausgaben von Transaktionen

Die Wirkung einer Transaktion besteht i.a. nicht nur in der Veränderung des Inhalts von Datenbankobjekten, sondern auch in Ausgaben und Meldungen, die während der Ausführung der Transaktion auf den Bildschirm des Benutzers oder in ein Batch-Protokoll ausgegeben werden. Da eine abgebrochene Transaktion keinerlei Wirkung haben soll, können Meldungen nicht unmittelbar ausgegeben werden, sondern müssen zunächst in einer Warteschlange gepuffert werden. Hierfür muß bei serverseitig ausgeführten Transaktionen eine eigene Verwaltungskomponente im Recovery-Manager oder im Rahmen der allgemeinen Kommunikationsmechanismen des Systems vorgesehen werden. Bei Erreichen des Commit-Punktes wird die Meldungsschlange ausgegeben, bei Abbruch der Transaktion gelöscht.

Dieses Verhalten ist jedoch bei programmierten Abbrüchen, die i.d.R. durch eine negativ ausgegangene Konsistenzprüfung ausgelöst werden nicht angemessen, weil hier trotz Rollback eine Beschreibung des Fehlers ausgegeben und ggf. eine Korrekturmöglichkeit angeboten werden sollte. Dies ist ein Sonderfall des allgemeineren Problems interaktiver Transaktionen.

Die Pufferung von Meldungen führt im Prinzip dazu, daß während der Laufzeit einer Transaktion keine Dialoge mit einem Benutzer möglich sind. Es bieten sich zwei Auswege an, beide sind nicht ganz befriedigend:

1. Der Dialog wird in Stücke aufgeteilt, die aus je einer Eingabe und den darauf folgenden Ausgaben bestehen. Jedes dieser Stücke bildet eine Transaktion. Commit oder Rollback der gesamten Änderungen während des Dialogs sind jedoch erst am Ende möglich; hierdurch kann es erforderlich werden, bereits abgeschlossene Transaktionen wieder rückgängig zu machen, was keine normale Aufgabe des Recoverys und technisch meist nicht möglich ist. Probleme bereiten auch Interferenzen mit parallel ablaufenden Transaktionen, die auf die gleichen Daten zugreifen.
2. Die Meldungen werden nicht gepuffert, sondern direkt ausgegeben. Bei einem Rollback ist dann eine Benachrichtigung des Benutzers

erforderlich, welche der vergangenen Meldungen und Änderungen jetzt hinfällig sind. Dies ist in der Praxis höchstens dann akzeptabel, wenn die ersten Dialogschritte nur unkritische Sachverhalte betreffen.

# 3 Anforderungen an das Recovery

## 3.1 Die Integrität der Datenbank aus Sicht des DBMS

Das übergeordnete Ziel aller Recovery-Maßnahmen ist die Sicherstellung der Integrität der Datenbank. Dieses Ziel ist so noch recht grob formuliert. Ein DBMS kann nur für bestimmte Aspekte der Integrität sorgen und Verantwortung tragen:

Alle Änderungen am Inhalt der Datenbank sind durch das DBMS gemäß Spezifikation auszuführen. Eine Datenbank, die diese Bedingung erfüllt, nennen wir **technisch korrekt**.

Die bereits in [TID] gegebene Definition bezog sich allerdings auf eine sequentielle Ausführung der Transaktionen. Diese Annahme ist jetzt zu einschränkend. Da andererseits eine Verallgemeinerung des Begriffes technische Korrektheit für den Fall parallel ausgeführter Transaktionen nicht einfach ist, sei für dieses Lehrmodul folgende Arbeitsdefinition von technischer Korrektheit vereinbart:

Der technisch korrekte Zustand der Datenbank ist derjenige, der sich ergeben hätte, wenn alle von der Störung betroffenen Aktivitäten nicht stattgefunden hätten. Die Datenbank ruht somit gedanklich zum Zeitpunkt der Störung.

Ferner muß die Datenbank während der Zeiträume, zu denen das DBMS nicht aktiv ist, auf permanenten Speichermedien erhalten werden. Die Verhinderung von Störungen und Behebung von Schäden in dieser Zeit wird man eher als Aufgabe des Betriebssystems ansehen, wenngleich auch andere Mittel zur Behebung solcher Schäden benutzt werden können.

©2003 Udo Kelter 

## 3.2 Speicherteile der Datenbank

In der Einleitung wurde bereits postuliert, daß man sich eine Datenbank so vorstellen kann, als ob sie aus einer Menge von Datenbankobjekten bestünde, die einzeln gelesen, verändert, erzeugt oder gelöscht werden können. Dies beschreibt das Bild eines Programmierers oder Benutzers von der Datenbank. Es besagt im Prinzip nichts darüber, in welchen Strukturen die Datenbank gespeichert wird; diese sind Geheimnis des Zugriffssystems. Wenn wir annehmen, die Datenbank wäre zwecks Geheimhaltung verschlüsselt gespeichert, so wirkt die Datenbank wie eine einzige, unstrukturierte Variable mit einer völlig strukturlosen Form der Speicherung. In realen Systemen kann man sich die Datenbank jedoch sehr oft als aus getrennt gespeicherten Teilen zusammengesetzt vorstellen. Ein typischer Speicherteil einer Datenbank ist eine Datei, wie sie durch das Betriebssystem verwaltet wird. Wir nehmen an, daß jedes Datenbankobjekt der Benutzersicht in einem Speicherteil der Datenbank von der jeweils kleinsten Einheit enthalten ist.

Speicherteile der Datenbank sind insofern für das Recovery relevant, als sie unabhängig voneinander beschädigt und wiederhergestellt werden können.

Statt auf die gesamte Datenbank kann man stets die Recovery-Mechanismen auf Teile der Datenbank anwenden, sofern sich der Aufwand lohnt.

## 3.3 Qualitätsmerkmale von Recovery-Mechanismen

Nach einer Störung muß es das Ziel des DBMS sein, die Datenbank wieder in einen Zustand zu versetzen, der für die Benutzer akzeptabel ist; dies soll

- möglichst schnell geschehen,
- unter Verlust von möglichst wenig bereits geleisteter Arbeit,
- möglichst automatisch, d.h. ohne zusätzliche Handarbeit,
- mit möglichst geringen Kosten.

Im einzelnen ergeben sich folgende “Qualitätsanforderungen” für Recovery-Mechanismen:

1. Es soll möglichst wenig Arbeit der Benutzer verloren gehen. Anders gesagt soll das Recovery möglichst vollständig sein. Bei interaktivem Zugriff zur Datenbank sollen z.B. möglichst wenig Dateneingaben verloren gehen, bei Batch-Verarbeitung sollen aufwendige Programme nicht von vorne laufen müssen.
2. Die beschädigten Teile der Datenbank stehen den Benutzern nicht zur Verfügung. Das Recovery soll also schnell sein. Die Recovery-Maßnahmen sollen nicht unnötig aufwendig sein. (Vgl. unten den Kostenaspekt; die Größe des Schadens und der Aufwand für das Recovery sollen in einem vernünftigen Verhältnis zueinander stehen.)
3. Die Wiederherstellung beschädigter Teile der Datenbank soll lokal erfolgen, d.h. die Recovery-Maßnahmen sollen sich möglichst nur auf die beschädigten Teile der Datenbank auswirken. Während der Wiederherstellung soll der Zugriff auf die unbeschädigten Teile nicht behindert werden. Insbesondere soll transaktionsbezogenes Recovery möglich sein.
4. Recovery-Maßnahmen sollten je nach Bedarf automatisch durchgeführt werden. Die Benutzer bzw. Administratoren sollen so weit wie möglich von manueller Arbeit entlastet werden.
5. Die Hilfsdaten für das Recovery können natürlich ebenfalls beschädigt werden. Für ihre Sicherheit muß ebenfalls gesorgt werden.
6. Die Kosten des Recoverys, letztlich Qualität und Quantität des Aufwands, sollen möglichst gering sein. Grob gesagt sind die Kosten umso höher, je besser die vorstehenden Qualitätsanforderungen erfüllt werden. In der Praxis müssen daher oft Kompromisse eingegangen werden; mit fallenden Kosten für die Hardware werden jedoch tendenziell leistungsfähigere Recovery-Mechanismen realisierbar.

## 3.4 Zielzustand des Recoverys

Das Recovery soll zu einem für die Benutzer akzeptablen Zustand der Datenbank führen. Was akzeptabel ist, hängt von vielen Umständen

ab, vor allem von

- Art und Umfang des Schadens,
- erforderlicher Geschwindigkeit und den übrigen oben erwähnten Qualitätsmerkmalen.

Je nach den Eigenschaften des durch das Recovery hergestellten Zustands der Datenbank können wir einige *Arten des Zielzustandes des Recoverys* unterscheiden (nach [Ve77]). Das Wort "Ziel" ist leider mit zwei Ausnahmen nicht so zu verstehen, daß der Benutzer an den Eigenschaften dieser Zielzustände besonders interessiert wäre, sondern daß Zustände mit diesen Eigenschaften durch die Recovery-Techniken effizient hergestellt werden können.

1. Der (technisch) *korrekte Zustand*: alle Effekte der Störung in der Datenbank (und ggf. in den Aktivitäten auf der Datenbank) werden vollständig behoben. Dies setzt voraus, daß die Datenbank während des Recoverys nicht verändert wird bzw. daß das Recovery schneller als die laufenden Änderungen ist.
2. Ein *früherer korrekter Zustand*: die Datenbank wird in einen früher korrekten, jetzt aber nicht mehr aktuellen Zustand versetzt. Die Änderungen in der Zwischenzeit sind verloren.
3. Ein *hypothetischer früherer korrekter Zustand*: die Datenbank wird in einen Zustand versetzt, der früher hätte eintreten können, sofern gewisse Änderungen ihres Inhalts in anderer Reihenfolge stattgefunden hätten. Dies ist dann der Fall, wenn einzelne Teile der Datenbank in einen früheren Zustand zurückversetzt werden, der nie gleichzeitig mit den derzeitigen Zuständen anderer Teile existierte, z.B. beim Rollback von Transaktionen.
4. Ein *früherer korrekter Teilzustand*: Die Datenbank wird in einen Zustand versetzt, der zum Teil mit einem früher oder jetzt noch korrekten Zustand übereinstimmt. Der Inhalt der restlichen Teile ist verloren bzw. wird neu initialisiert.
5. Ein *logisch konsistenter Zustand*: die Datenbank wird in irgendeinen Zustand versetzt, der die statischen Integritätsbedingungen

erfüllt, und der dem korrekten Zustand möglichst nahe kommt. Dieser Zustand kann "schlimmer" inkorrekt sein als infolge des Fehlens von früheren Änderungen oder des Verlustes von Teilen, d.h. nicht infolge von derartigen Verfälschungen entstanden sein.

Die o.g. Arten von Zielzuständen beschreiben Eigenschaften des Zustands der Datenbank nach Beendigung der *vom DBMS* angebotenen Recovery-Maßnahmen, die entweder automatisch oder unter Kontrolle des Benutzers oder Operateurs stattfinden. Darüber hinaus bleibt es bei den Fällen 2 bis 5 dem Benutzer natürlich unbenommen, weitere Wiederherstellungsmaßnahmen *von Hand* durchzuführen.

Ein ungefähres Maß für die Korrektheit eines Datenbankzustandes ist die Zahl der Änderungen einzelner Datenbankobjekte von Hand, die erforderlich wären, um einen völlig korrekten Zustand herzustellen. Ganz grob gesagt wird dann von Fall 1 bis 5 gehend der Zustand, der durch das Recovery erreicht wird, immer inkorrekter, die Vollständigkeit des Recoverys ist immer geringer; exakt kann das Maß der Korrektheit bzw. die Vollständigkeit des Recoverys bei den Alternativen 2 - 5 weder absolut noch relativ zueinander angegeben werden.

Es sei noch explizit darauf hingewiesen, daß zwischen der Größe des Schadens, den man ebenfalls in der erforderlichen Handarbeit zu seiner Reparatur messen könnte, und der Inkorrektheit des Zielzustands des Recoverys kein direkter Zusammenhang bestehen muß. So kann eine Datenbank, die wegen eines einzigen Records inkonsistent geworden ist, durch Recovery der 2. Art auf den Stand von vor einer Woche zurückgesetzt werden, obwohl dann 1000 Änderungen (in anderen Teilen) der Datenbank verlorengehen. Intuitiv wird man hier jedoch ein Mißverhältnis zwischen der Größe des Schadens und der Größe der durch das Recovery betroffenen Teile der Datenbank sowie der Inkorrektheit des Datenbankzustandes nach dem Recovery empfinden.

# 4 Grundprinzipien des Recoverys

Im letzten Abschnitt wurde eine Auswahl von Zuständen der Datenbank angegeben, die nach einer Störung angestrebt werden können.

©2003 Udo Kelter 

Nunmehr erhebt sich die Frage, wie dies effektiv erreicht werden soll.

## 4.1 Recovery-Daten

Recovery ist letzten Endes immer nur deshalb möglich, weil neben der Arbeitsversion der Datenbank zusätzlich andere Daten für Recovery-Zwecke, sogenannte **Recovery-Daten**, gespeichert werden. Sofern die enthaltenen Informationen identisch sind, liegt Redundanz vor, in vielen Fällen enthalten die Recovery-Daten jedoch andere Informationen (z.B. für Zielzustandstypen 2 und 4).

Viele der Störungen, die die Arbeitsversion der Datenbank beschädigen, können ebenso die Recovery-Daten beschädigen. Man muß somit, je nach der Wichtigkeit der Recovery-Daten, auch für diese ein Recovery vorsehen, sozusagen eine Sekundär-Recovery-Maßnahme mit neuen Sekundär-Recovery-Daten. Diese Daten können natürlich auch wieder beschädigt werden, usw.; letztlich kann nie eine völlige Sicherheit gegen alle denkbaren Fälle erreicht werden, denn irgendwann muß dieser Kreis in der Praxis abgebrochen werden. Die Sicherheit ist immer nur endlich.

## 4.2 Risikostreuung

Recovery-Daten und die Mechanismen, die sie benutzen, helfen nie bei allen denkbaren Arten von Störungen, sondern immer nur bei bestimmten. Hieraus folgt:

1. Man soll möglichst *verschiedene Arten von Recovery-Daten und -Mechanismen* vorsehen, die komplementäre Risiken abdecken, um die verschiedenen Arten von Störungen, mit denen gerechnet werden muß (eventuell sogar mehrfach) abzudecken. Durch die Redundanz in den Recovery-Daten wird ebenfalls die *Zuverlässigkeit des Recoverys* erhöht.
2. Die Recovery-Daten dürfen nicht so gespeichert werden, daß sie bei Störungen, bei denen sie helfen sollen, selbst verletzt wären. So sollten beispielsweise zum Schutz gegen physische Schäden die

Recovery-Dateien auf anderen Geräten oder Medien gespeichert werden als die Arbeitsversion der Datenbank.

Ein wichtiges Mittel zur Risikostreuung ist die Verwendung verschiedener Speichermedien. Bei den heute verfügbaren Technologien kommen in Großrechnern für die Massendatenhaltung nur Platten oder Bänder / Bandkassetten in Betracht. Typischerweise sind beide Medien gleichzeitig verfügbar, so daß es sich anbietet,

- die Arbeitsversion der Datenbank auf Platte zu speichern und
- die Recovery-Daten auf Band, zumindest längerfristig; bei manchen Arten von Recovery-Daten ist es erforderlich, sie zumindest zeitweise auf Medien mit schnellem Zugriff zu halten, also Platte.

Wir werden *Platte* und *Band* im folgenden als Synonym benutzen für *irgendwelche* permanenten Speichermedien, die die erforderliche Zugriffsgeschwindigkeit haben und die auf unterschiedlichen Geräten montiert sind, also eine Risikostreuung gegen Gerätefehler ermöglichen. Das Band kann also auch eine andere Platte sein.

## 4.3 Reparaturprinzipien

Im Sinne von Wiederherstellung bedeutet Recovery, daß man von einem defekten Zustand der Datenbank ausgeht und daß dieser Zustand anschließend in einen akzeptablen anderen Zustand verändert wird. Dieses Denkschema ist allerdings bei Präventivmaßnahmen, die man ja ebenfalls zum Recovery zählt, nicht anwendbar. Im Falle einer Störung, gegen die die Präventivmaßnahme schützt, ist die Datenbank gerade *nicht* defekt und diesbezügliche Reparaturaktivitäten sind nicht erforderlich. Wir trennen daher zwischen *Reparaturprinzipien* und *Präventionsprinzipien*.

Wir können die Reparaturprinzipien grob danach unterscheiden, ob sie von der Arbeitsversion der Datenbank oder von den Recovery-Daten aus versuchen, den gewünschten neuen Zustand zu konstruieren.

©2003 Udo Kelter 

### 4.3.1 Arbeitsversion der Datenbank als Ausgangsbasis

Diese Prinzipien sind immer nur dann anwendbar, wenn die Arbeitsversion der Datenbank nach der Störung überwiegend erhalten geblieben ist. Es wird versucht, durch lokale Änderungen den erwünschten Zustand zu konstruieren. Hierbei wird nach folgenden Prinzipien verfahren.

**Rückwärts-Recovery** (**backward recovery**): Die beschädigten Teile der Datenbank werden in einen früheren Zustand zurückversetzt, möglichst in den unmittelbar vor der Störung. In den meisten Fällen bedeutet dies, den Effekt von vorzeitig abgebrochenen Transaktionen rückgängig zu machen (Rollback). Vorbereitend müssen die früheren Inhalte der durch Transaktionen veränderten Datenbankobjekte in den Recovery-Daten gespeichert werden. Je nach den Umständen wird die Datenbank in Zustände des Typs 1, 2 oder 3 gebracht.

**Fehlerkompensation:** Dieses Prinzip ist nur dann anwendbar, wenn die Störung so interpretierbar ist, daß sie den eigentlich gewünschten, korrekten Zustand der Datenbankobjekte *invertierbar funktional* in den nun vorhandenen, inkorrekten Zustand umformt. Der korrekte Zustand kann nun dadurch herbeigeführt werden, daß eine *inverse* Funktion auf die Zustände der betroffenen Datenbankobjekte angewendet wird. Beispiel: Eine Zahl wurde um 1000 erhöht, sollte aber nur um 100 erhöht werden. Die Abweichung beträgt 900, sie kann durch Subtraktion kompensiert werden.

Dieses Recovery-Prinzip ist offensichtlich nur in wenigen Fehlerklassen anwendbar. Es hat allerdings den Vorteil, auch bei schon abgeschlossenen Transaktionen anwendbar zu sein. Unter Umständen ist es auch effizienter als ein Rollback der Transaktion.

Bei richtiger Anwendung wird die Datenbank durch die Fehlerkompensation in den korrekten Zustand (Typ 1) gebracht.

**Rettung:** In manchen Fällen sind die üblichen Recovery-Prinzipien nicht mehr anwendbar, weil entweder die Recovery-Daten ebenfalls

zerstört sind oder gar keine vorgesehen sind, insbesondere für die (Primär-) Recovery-Daten. In dem Fall muß man "retten, was zu retten ist". Dies ist zwar Flickschusterei, aber besser als ein völliger Verlust der Datenbank. Eine Reihe von realisierten Systemen (vgl. [Ve77]) bietet Programme an, die die Datenbank selbständig oder in Interaktion mit dem Benutzer in einen Zustand versetzen, der zumindest physisch und logisch konsistent ist (Zustandstyp 3 oder 5). Denkbare Maßnahmen sind:

- Die beschädigte Arbeitsversion der Datenbank (oder die beschädigten Recovery-Daten) werden daraufhin untersucht, welche Teile unbeschädigt geblieben sind und weiterverwendet werden können.
- Bei beschädigten Zugriffspfaden und ausreichender Redundanz in diesen kann der *wahrscheinliche* Zustand der Datenbank vor der Störung rekonstruiert werden.

Fehlerkompensation und Rettung sollten sehr zurückhaltend angewandt werden. Sie sind mit relativ viel Handarbeit verbunden und die Gefahr ist groß, daß neue Fehler eingeführt werden.

### 4.3.2 Recovery-Daten als Ausgangsbasis

Bei diesem Prinzip benötigt man in jedem Fall sogenannte Dumps als Teil der Recovery-Daten. Ein (Backup-) **Dump** ist eine Kopie des Zustands der Datenbank oder eines Teils von ihr in der Vergangenheit, der in der Regel zum damaligen Zeitpunkt korrekt war und für Recovery-Zwecke aufgezeichnet wurde.

Die Benutzung des Dumps als neue Arbeitsversion der Datenbank entspricht Zielzustandstyp 2 oder 3, je nachdem, ob die gesamte Datenbank oder nur Teile durch die alte Version ersetzt wurden.

Zusätzlich können nach dem Laden des Dumps Änderungen an der Datenbank, die seit dem Zeitpunkt der Erstellung des Dumps durchgeführt wurden, nachgeholt werden. Hierzu ist es erforderlich, daß vorbereitend geeignete Informationen über diese Veränderungen gespeichert werden. Je nach der Vollständigkeit dieser Informationen kommt Zielzustandstyp 1, 2 oder 3 zustande.

©2003 Udo Kelter 

Die wichtigste Technik in diesem Zusammenhang ist das *Wiederholen von Transaktionen* auf dem Dump. Da hier im wesentlichen der zeitliche Ablauf der Ereignisse in der Datenbank wiederholt wird, spricht man von **Vorwärts-Recovery** (*forward recovery*), im Gegensatz zum Rückwärts-Recovery, wo die Zeit "ein Stück zurückgedreht" wird.

## 4.4 Präventionsprinzipien

Merkmal von Maßnahmen nach dem Präventionsprinzip ist, daß bei gewissen Störungen gar kein echter Schaden in der Arbeitsversion der Datenbank eintritt. Meist sind nach der Störung überhaupt keine Reparaturaktivitäten in der Datenbank erforderlich oder nur ein "Aufräumen", welches jedoch keine weiteren Hilfsdaten erfordert.

Bei den Präventivmaßnahmen ist oft keine klare Trennung zwischen Arbeitsversion der Datenbank und Recovery-Daten möglich.

**Verzögertes Schreiben** (*deferred update*): Dieses Prinzip ist nur in bestimmten Situationen bei Transaktions- und Systemfehlern wirksam. Alle Schreibaktionen (Änderungen) einer Transaktion werden möglichst erst unmittelbar vor dem Commit-Zeitpunkt ausgeführt. Die zu schreibenden Werte müssen bis dahin aufgehoben werden. Praktisch müssen Kopien der betroffenen Objekte im Arbeitsspeicher angelegt werden, auf denen die Änderungen zunächst stattfinden.

Solange noch keine Schreibaktion ausgeführt wurde, ist das Rollback einer Transaktion trivial: lediglich die Kopien der Datenbankobjekte sind zu löschen und die Transaktion ist beim Recovery- Manager abzumelden.

Zur Vermeidung von gewissen Parallelitätsanomalien empfiehlt es sich sogar, alle Schreibaktionen im Rahmen der Abarbeitung des Commit-Befehls auszuführen. Da man das Commit als atomares Ereignis ansieht, ist die Zeitdauer vom ersten Schreiben bis zum Commit Null, jedenfalls in dieser idealisierenden Annahme.

Es ergibt sich folgender Vorteil: Bei einem Systemfehler ist die Arbeitsversion der Datenbank auf Platte nach einem Neustart des Sy-

stems sofort in einem früher korrekten Zustand (Typ 2), lediglich die vorher noch aktiven und abgebrochenen Transaktionen sind nachzuholen, um den korrekten Zustand der Datenbank herzustellen. Insbesondere sind keine Maßnahmen erforderlich, um einen logisch und physisch konsistenten Zustand der Datenbank herzustellen. Für viele Zwecke kann die Datenbank unmittelbar weiterverwendet werden.

Einziger Nachteil des verzögerten Schreibens ist der Aufwand für die Kopien. Bei "kleinen" Transaktionen ist dieser Aufwand erträglich, so daß das verzögerte Schreiben (bzw. das Schreiben erst bei Commit) unbedingt zu empfehlen ist.

**Vorsichtiges Ändern** (*careful replacement*): Dieses Prinzip wird vor allem in der Verbindung mit dem verzögerten Schreiben eingesetzt und behebt weitgehend dessen Schwachpunkt, denn es schützt gegen physische Inkonsistenzen infolge von Systemfehlern beim Ändern der Datenbank (also beim Rückschreiben der Kopien). Es kann nur durch eine spezielle Programmiertechnik bei den verändernden Aktionen realisiert werden.

Die zu verändernden Einheiten können Sätze, Speicherseiten o.ä. sein. Die Gefahr bei konventionellem Ändern besteht darin, daß stets eine Zeitspanne lang der alte Zustand vor der Änderung schon zerstört, der neue Zustand jedoch noch nicht vollständig hergestellt ist, vor allem wenn mehrere Einheiten von einer Änderung betroffen sind.

Beim vorsichtigen Ändern werden zunächst alle Änderungen auf Kopien gemacht, die Kopien in die Datenbank eingefügt (auf Platte!), so daß während des Änderns zwei Versionen (sogenannte "virtuelle Kopien") derselben Einheiten nebeneinander existieren können. Dann muß noch zur neuen Version "umgeschaltet" werden, und die alten Versionen werden gelöscht. Während des Umschaltens besteht kein Schutz gegen Systemfehler, die Dauer des Umschaltens ist jedoch um Größenordnungen kleiner als die der gesamten Änderung.

©2003 Udo Kelter 

# 5 Logging

Die in Abschnitt 2.2 beschriebenen Grundfunktionen sind nicht ohne vorbereitende Maßnahmen denkbar; konkret müssen alle relevanten Daten in einem Log protokolliert werden.

Unter einem (Recovery-) **Log** versteht man eine Aufzeichnung der Änderungen in der Datenbank für Recovery-Zwecke, insb. für die Grundfunktionen, aber ggf. auch diverse andere Zwecke. Andere Bezeichnungen sind *audit trail*, *system log* und *journal*. Der Log enthält für jede ändernde Aktion einen Eintrag (für lesende Aktionen sind i.a. keine Einträge nötig) mit folgenden Angaben:

- Identifikation der Transaktion
- Identifikation des Objekts
- Art der Aktion
- Undo- und/oder Redo-Daten

sowie ggf. zusätzliche Angaben wie Datum und Uhrzeit, Identifikation betroffener Seiten der Datenbank oder Verweis auf den vorhergehenden Logeintrag der gleichen Transaktion.

Neben den Einträgen für ändernde Aktionen gibt es zusätzlich Einträge für Beginn, Commit und Rücksetzung von Transaktionen (BOT-, EOT- und Abort-Einträge) und weitere interne Zwecke.

Ferner erhält jeder Log-Eintrag eine laufende Nummer (*log sequence number*), die verschiedenen Zwecken dient.

Aus Sicherheitsgründen kann ein Log doppelt geführt werden, z.B. parallel auf Platte und einem Band.

## 5.1 Undo- vs. Redo-Logging

Für das Undo bzw. Redo einer Aktion werden andere Daten benötigt; daher kann man reine **Undo-Logs** bzw. **Redo-Logs** benutzen, die keine Daten für den jeweils anderen Zweck enthalten. Man kann natürlich auch einen gemeinsamen Log für beide Zwecke benutzen; wegen der gemeinsamen Daten ist dies zwar platzsparend, aber dennoch nicht immer sinnvoll, weil die Logs für unterschiedliche Zwecke benutzt werden.

Unter **Vorwärts-Logging** bzw. **Redo-Logging** versteht man die Erzeugung von Logdaten für die Redo-Funktionen. Beim globalen bzw. partiellen Redo werden vollständige Transaktionen in ihrer ursprünglichen Reihenfolge nachgeholt; für diese Zwecke reicht es aus, den Log auf einem sequentiellen Medium wie einem Band zu speichern. Für die Erzeugung der Redo-Logdaten gilt folgende Regel:

> *Die Redo-Logdaten müssen vor dem Abschluß der Commit-Anweisung materialisiert sein.*

Andernfalls wäre einerseits der Applikation und damit dem Benutzer der erfolgreiche Abschluß der Transaktion angezeigt worden, andererseits könnten bei einem sofort folgenden Systemabsturz solche Änderungen der Transaktion, die noch nicht in der materialisierten Datenbank enthalten sind, nicht wiederhergestellt werden.

Unter **Rückwärts-Logging** bzw. **Undo-Logging** versteht man die Erzeugung von Logdaten für die Undo-Funktionen. Da das Rollback von Transaktionen sehr schnell sein muß, kommt als Speicherungsmedium für einen Rückwärts-Log nur die Platte in Frage. Für die Erzeugung der Undo-Logdaten gilt die folgende (sogenannte *write ahead log-*) Regel:

> *Die Undo-Logdaten müssen vor der Änderungen der materialisierten Datenbank im Log materialisiert sein.*

Andernfalls ist nach einem Systemfehler kein Undo mehr möglich.

## 5.2 Archiv-Logs

Bei großen Systemen und hohem Änderungsaufkommen können die Logs sehr groß werden. An dieser Stelle hilft die Beobachtung weiter, daß die Undo-Logdaten sehr schnell überflüssig werden: diese werden nur für das Rollback von Transaktionen benötigt, von denen wir annehmen, daß sie relativ kurz sind. Sobald eine Transaktion ihr Commit erreicht, können im Prinzip alle zu dieser Transaktion gehörigen Undo-Daten gelöscht werden. Bei einem sequentiell organisierten Log

ist das aber nicht ohne weiteres möglich. Eine Lösung besteht darin, den gesamten Log zu teilen in

- einen **aktiven Log**, der auf Platte steht, der die neuen Einträge enthält und der sowohl Undo- als auch Redo-Daten enthält, und in
- einen oder mehrere **Archiv-Logs**, die alle alten Einträge enthalten, die nur noch Redo-Daten enthalten und die auf anderen Medien stehen können (z.B. CD-ROM oder Band).

In bestimmten Zeitabständen oder nach Erreichen einer bestimmten Größe wird ein neuer aktiver Log eingerichtet und der bisherige aktive Log zum nächsten Archiv-Log. Da alle Logeinträge noch nicht abgeschlossener Transaktionen auf Platte vorhanden sein müssen, muß der alte aktive Log solange auf Platte bleiben, wie er noch Einträge aktiver Transaktionen enthält. (Alternativ könnte man diese Transaktionen abbrechen, rücksetzen und neu starten.) In den alten aktiven Log werden auch alle noch folgenden Einträge dieser Transaktionen eingeschrieben. Alle Einträge von neu gestarteten Transaktionen kommen in den neuen aktiven Log.

Nachdem alle Transaktionen, die noch dem alten aktiven Log zugeordnet sind, beendet sind, kann dieser komprimiert und ggf. schon parallel für eine eventuelle Verwendung in einem globalen Redo präpariert werden:

- Falls es ein gemeinsamer Undo- und Redo-Log ist, können die Undo-Daten entfernt werden.
- Mehrfache Wertzuweisungen für das gleiche Datenelement können mit Ausnahme der letzten gelöscht werden.
- Einträge von zurückgesetzten Transaktionen können entfernt werden.
- Die Änderungen können ggf. schon so vorsortiert werden, daß alle Änderungen, die eine Seite betreffen, hintereinander liegen, also beim globalen Redo eine mehrfache Übertragung der gleichen Seite vermieden wird.

©2003 Udo Kelter 

## 5.3 Zustands- vs. Transitions-Logging

Wir hatten bisher völlig offengelassen, woraus die Undo- bzw. Redo-Daten überhaupt bestehen. Hier sind zwei Ansätze denkbar:

Beim **Zustands-Logging** werden *Zustände* protokolliert:

- für das Redo: der Zustand nach der Aktion
- für das Undo: der Zustand vor der Aktion

Die Implementierung des Redo bzw. Undo ist hier sehr einfach: der vorhandene Wert wird mit dem neuen bzw. alten Wert überschrieben.

Beim **Transitions-Logging** werden *Zustandsübergänge* protokolliert:

- für das Redo: die beobachtete Zustandsveränderungsfunktion
- für das Undo: die invertierende Funktion der beobachteten Zustandsveränderungsfunktion

Die Implementierung des Redo bzw. Undo besteht hier darin, die Zustandsveränderungsfunktion bzw. ihre invertierende Funktion erneut auszuführen.

## 5.4 Logging auf verschiedenen Abstraktionsebenen

Logging ist unabhängig von der Wahl zwischen Zustands- und Transitions-Logging auf beliebigen Abstraktionsebenen (s. Schichtenarchitektur in Bild 1 in Abschnitt 2.1) möglich. Bei der Entscheidung sind folgende Aspekte zu berücksichtigen:

- der Umfang der Logdaten; dieser sollte möglichst klein sein.
- die Bestimmbarkeit der Undo-Daten: zunächst kennt das System nur die Aktion
- der Aufwand zur Implementierung der Grundfunktionen

Das Logging auf Seitenebene ist sehr einfach zu realisieren, die Undo-Daten sind leicht bestimmbar, nachteilig ist aber das hohe Datenvolumen: selbst wenn nur eine einzige Zahl, die nur wenige Byte beansprucht, geändert wird, muß eine ganze Seite von z.B. 1 kB in den

Log aufgenommen werden. Dieses Problem kann durch Transitions-Logging vermieden werden, da hier Veränderung viel kompakter beschrieben werden kann. Beim Logging auf Seitenebene muß ferner verhindert werden, daß mehrere Transaktionen parallel auf der gleichen Seite arbeiten, es kommt also zu logisch nicht notwendigen Verzögerungen.

Das Logging auf Speichersatzebene ist ebenfalls einfach zu realisieren, die Undo-Daten sind leicht bestimmbar, das Problem des Datenvolumens entfällt hier, wenn die Sätze kurz sind.

Wenn man auf der Ebene der Tupel/Objekte relativ feingranulare Operationen wie "setze Attribut A an Objekt O" loggt, läßt sich, sofern die Applikationen häufig einzelne Attribute ändern, das Datenvolumen deutlich gegenüber dem Logging auf Speichersatzebene reduzieren.

Auf der n-Tupel-Ebene kann das Datenvolumen noch weiter reduziert werden, da hier durch ein einzelnes Kommando Veränderungen an sehr vielen Objekten kompakt beschrieben werden können. Dennoch ist das Logging auf dieser Ebene i.a. aus mehreren Gründen nicht zu empfehlen: (a) Diese Aktionen können mehrere Seiten betreffen und (ohne besondere Maßnahmen) ist die Fehler-Atomarität dieser Aktionen nicht gewährleistet: bei einem Systemfehler ist nicht sichergestellt, daß diese Aktionen ganz oder gar nicht materialisiert sind. Anders gesagt muß die Datenbank immer "aktionskonsistent" gehalten werden; dies ist nicht immer möglich. (b) Die Undo-Daten können i.a. nicht in gleicher Weise kompakt dargestellt werden. (c) Bei objektorientierten oder navigierenden Datenbankmodellen kann es sehr schwierig sein, die Undo-Daten überhaupt auf dieser Ebene mit vertretbarem Aufwand zu bestimmen.

## 5.5 Logging auf Transaktionsebene

Eine weitere Alternative besteht darin, komplette Transaktionsaufrufe zu loggen und beim Redo zu wiederholen. Der Log enthält hier den Namen der Transaktion und die Aufrufparameter. Voraussetzung ist hier, daß die Transaktionsprogramme komplett innerhalb des DBS

verwaltet werden, denn sie müssen bei einem Neustart ja vom DBMS aufgerufen werden können.

Äußerst problematisch ist hier die Gewinnung der Undo-Daten, vgl. die Bemerkungen zum Logging auf der n-Tupel-Ebene. Insgesamt ist das Logging auf Transaktionsebene nur in seltenen Ausnahmefällen sinnvoll.

# 6 Recovery für Transaktionsfehler

**Rückwärts-Recovery.** Für das **Rollback** einer Transaktion wird ein Rückwärts-Log benötigt. Dieser Log wird von hinten aus rückwärts nach Einträgen für diese Transaktion durchlaufen, bis der BOT-Eintrag gefunden wird. Bei jedem gefundenen Eintrag wird das Aktions-Undo ausgeführt. Durch eine Verkettung der Einträge kann die Suche nach den Einträgen stark beschleunigt werden.

**Präventionsmaßnahmen.** Um die Wahrscheinlichkeit bzw. den Umfang von Undo-Maßnahmen klein zu machen, kann man auf allen Ebenen Schreibaktionen möglichst lange aufschieben; man nennt dies **verzögertes Schreiben**. Statt also Schreibaktionen immer sofort auf die nächsttiefere Ebene weiterzugeben, arbeitet man zunächst auf Puffern und propagiert diese Änderungen erst möglichst spät, am besten erst unmittelbar vor dem Commit, "nach unten". Vor der ersten Schreibaktion ist ein Rollback daher trivial, und die materialisierte DB ist bei einem Systemfehler nicht logisch inkonsistent.

Das verzögerte Schreiben ist generell sehr zu empfehlen, lediglich bei sehr großen Mengen geänderter Daten kann es wegen des Platzaufwands sinnvoller sein, auf das verzögerte Schreiben zu verzichten.

# 7 Recovery für Systemfehler

## 7.1 Präventionsmaßnahmen

Ziel ist hier natürlich, den Aufwand für das globale Undo und partielle Redo zu optimieren. Dieser Aufwand hängt sehr vom Zustand der

materialisierten DB nach einem Medien- oder Systemfehler ab, dieser wiederum davon, ob bestimmte Präventionsmaßnahmen durchgeführt werden. Wir besprechen zunächst derartige Präventionsmaßnahmen und Einflußfaktoren. Bei diesen Maßnahmen muß man abwägen, welchen Gewinn sie ggf. bei einem (relativ seltenen) Systemfehler bringen und welchen Aufwand sie im Normalbetrieb verursachen.

**Direkte vs. indirekte Seitenadressierung:** Hier geht es um die Frage, ob einer Seite immer genau ein Block zugeordnet ist (direkte Seitenallokation) oder mehrere Blöcke alternativ zugeordnet sein können (indirekte Seitenallokation).

Eine indirekte Seitenallokation ist Voraussetzung für das vorsichtige Ändern von Blöcken, denn dann müssen während einer Änderung die alte und neue Version der Seite zugleich in der physischen Datenbank existieren.

Bei einer direkten Seitenallokation sind die Änderungen überschreibend (*update in place*), es besteht hier kein Unterschied zwischen materialisierter und physischer Datenbank.

Nachteil der indirekten Seitenadressierung ist der erhöhte Aufwand; für die Umschaltung auf die neue Version ist ein zusätzlicher Plattenzugriff erforderlich.

**Atomares Ändern mehrerer Seiten:** Sofern eine Aktion (z.B. das Löschen eines Tupels) mehrere Seiten verändert, stellt sich die Frage, ob diese Änderungen atomar materialisiert werden.

Wenn solche Änderungen nicht atomar erfolgen, ist der Zustand der materialisierten Datenbank nach einem Systemfehler unvorhersehbar, es können physische Inkonsistenzen auftreten.

Atomares Ändern ist leider praktisch nur bei einer indirekten Seitenadressierung implementierbar, die hohen Aufwand verursachen kann.

**Materialisieren unsicherer Änderungen:** Das Problem ist hier, daß mehrere Transaktionen u.U. mehr Seiten verändern, als Puffer vorhanden sind. In diesem Fall müssen veränderte Blöcke schon vor

EOT in die physische Datenbank zurückgeschrieben werden (analog zum Paging in Betriebssystemen). Beim *update in place* wird dann sogar die materialisierte Datenbank verändert.

Nachteil des Materialisierens unsicherer Änderungen sind hohe Undo-Kosten. Vermeiden kann man dies durch größere Puffer, was heute meist kein Problem ist.

**Materialisieren aller Änderungen bei Commit:** Die Frage ist hier, ob alle veränderten Seiten im Rahmen der Commit-Verarbeitung materialisiert werden oder nicht.

Falls ja, kann das globale Redo nach einem Systemfehler komplett entfallen! Leider verursacht das sofortige Materialisieren aller veränderten Seiten einen hohen Aufwand im laufenden Betrieb und ist daher nicht immer möglich.

Zusammenfassend läßt sich sagen, daß das Materialisieren aller Änderungen bei Commit und das atomare Ändern mehrerer Seiten zwar sehr attraktiv bzgl. des erreichten DB-Zustands sind, aber oft an zu hohem Aufwand scheitern.

## 7.2 Globales Undo

Das globale Undo findet beim erneuten Hochfahren des System nach einem Systemfehler statt; wir gehen davon aus, daß der normale Betrieb zunächst noch gesperrt ist, das globale Undo also die materialisierte Datenbank exklusiv verfügbar hat.

Ziel ist im Prinzip ein Undo aller abgebrochenen Transaktionen. Man könnte dies dadurch erreichen, daß man die Menge der abgebrochenen Transaktionen bestimmt und für jede einzeln ein Undo durchführt. Dies wäre aber ungeschickt, weil man dann für jede Transaktion den Log durchsuchen müßte. Stattdessen ist ein gemeinsames Rollback aller abgebrochenen Transaktionen effizienter. Analog zum Transaktion-Rollback durchläuft man den Log von hinten aus rückwärts und führt ein Aktions-Undo für jede Aktion einer abgebrochenen Transaktionen durch.

Die Menge der abgebrochenen Transaktionen ist anhand des Logs

bestimmbar: es handelt sich um die Transaktionen, für die ein BOT-, aber kein Commit- oder Abort-Eintrag im Log vorhanden ist. Wenn man den Log von hinten durchsucht, findet man einen Aktionseintrag, ohne vorher einen Commit-Eintrag gefunden zu haben.

Ohne vorbereitende Maßnahmen muß man den ganzen (!) Log durchlaufen, denn es könnte ganz vorne noch eine Transaktion stehen, die aus irgendwelchen Gründen nicht beendet worden ist. Dies ist natürlich unerwünscht, denn dies dauert lange und es ist sehr unwahrscheinlich, daß sich vorne noch "liegengebliebene" Transaktionen befinden. Abhilfe schaffen hier Undo-Checkpoints.

Ein **Undo-Checkpoint** ist ein spezieller Log-Eintrag, der die Identifikationen der derzeit aktiven Transaktionen enthält. Sobald beim Rückwärtsdurchlauf ein Undo-Checkpoint gefunden wird, ist die Restmenge der abgebrochenen Transaktionen bekannt. Der Rückwärtsdurchlauf kann beendet werden, sobald alle zugehörigen BOT-Einträge gefunden worden sind.

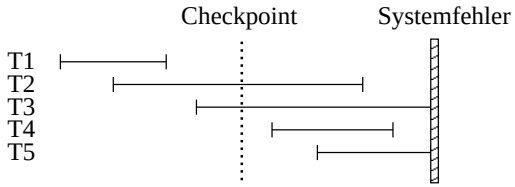


Abbildung 3: Beispiel eines Checkpoints

Erzeugt werden sollten Undo-Checkpoints etwa im Abstand von einigen Minuten oder alternativ nach jeweils 100 gestarteten Transaktionen.

Bei dem Beispiel in Bild 3 würde der Checkpoint festhalten, daß zu diesem Zeitpunkt die Transaktionen T2 und T3 nicht beendet waren.

**Systemfehler beim globalen Undo.** Während der Ausführung eines globalen Undo kann natürlich erneut ein Systemfehler eintreten, so daß das globale Undo unterbrochen würde und in der materialisierte Datenbank nur ein undefinierter Teil der Änderungen rückgängig

gemacht wird.

Beim Zustands-Logging kann nun einfach das globale Undo komplett wiederholt werden; dies liegt daran, daß das Undo einer Aktion hier durch Schreiben des früheren Werts implementiert wird und daß bei dieser Implementierung des Aktions-Undo eine mehrfache Ausführung letztlich den gleichen Effekt erzeugt wie eine einmalige Ausführung. Dies bezeichnet man auch als **Idempotenzeigenschaft**.

Beim Transitions-Logging hat das Aktions-Undo die Idempotenzeigenschaft nicht, deshalb darf das globale Undo nach einer Unterbrechung nicht einfach wiederholt werden. Stattdessen ist eine aufwendigere Implementierung des globalen Undo erforderlich, bei der sichergestellt wird, daß jedes Aktions-Undo nur genau einmal ausgeführt wird.

## 7.3 Partielles Redo

Das partielle Redo findet nach dem globalen Undo statt. Es müssen noch die Änderungen nachgeholt werden, die nicht in der materialisierten Datenbank enthalten sind und die von Transaktionen stammen, die vor dem Systemfehler beendet worden sind. Durch vorbeugende Maßnahmen (Materialisieren aller Änderungen bei Commit) können solche Änderungen ganz vermieden werden, aber diese vorbeugenden Maßnahmen sind nicht immer möglich.

Der Grund, warum man nicht alle Änderungen sofort materialisiert, liegt im Zeitverlust für die Plattenzugriffe, d.h. manche veränderten Seiten können erst einige Zeit nach dem Commit auf Platte zurückgeschrieben werden. Zu einem bestimmten Zeitpunkt kann eine Menge von Seiten darauf warten, zurückgeschrieben zu werden, und es ist zu entscheiden, welche davon als nächste zurückgeschrieben werden soll. Diese Frage stellt sich übrigens auch bei der Pufferverwaltung von Dateien; die in Betriebssystemen üblichen Strategien – die angewandt werden, wenn man die Segmente als Dateien realisiert und die Segmentverwaltung so dem Betriebssystem überläßt – laufen darauf hinaus, stark frequentierte Seiten nicht zurückzuschreiben, weil deren Inhalt ja doch bald wieder verändert wird. Im Sinne des partiellen

Redo sind derartige Strategien gerade falsch: derartige Seiten enthalten Änderungen von vielen, u.U. lange zurückliegenden Transaktionen. Sinnvoller ist es hier, mit erster Priorität Seiten zurückzuschreiben, die Änderungen abgeschlossener Transaktionen enthalten.

Nach einem Systemfehler hat man ohne vorbereitende Maßnahmen keinerlei Daten darüber, welche Änderungen nicht materialisiert worden sind, d.h. man müßte auf Verdacht *alle* vollständigen Transaktionen, die im Log vorkommen, wiederholen. Aus Aufwandsgründen kommen auch keine Verfahren in Frage, bei denen man die komplette Datenbank oder den kompletten Log durchsuchen müßte. Abhilfe schaffen hier Redo-Checkpoints. Bei einem **Redo-Checkpoint** werden alle veränderten Seiten zurückgeschrieben, und im Log wird ein entsprechender Eintrag geschrieben. Der komplette vor dem Redo-Checkpoint liegende Teil des Logs braucht daher beim partiellen Redo nicht mehr berücksichtigt zu werden.

Wenn wir in Bild 3 annehmen, daß der eingezeichnete Checkpoint zugleich ein Redo-Checkpoint ist, dann würden im Rahmen des Redo-Checkpoints die bisherigen Änderungen von T2 und T3 materialisiert werden. Infolge des Rückwärtsdurchlaufs durch den Log im Rahmen des vorherigen globalen Undo ist schon bekannt, daß

- T2 und T4 erfolgreich beendet worden sind; diese Transaktionen werden also beim partiellen Redo berücksichtigt;
- T3 und T5 nicht erfolgreich beendet worden sind; diese Transaktionen werden beim partiellen Redo nicht berücksichtigt.

**Systemfehler beim partiellen Redo.** Während der Ausführung eines partiellen Redo kann erneut ein Systemfehler eintreten, so daß das partielle Redo unterbrochen würde und in der materialisierten Datenbank nur ein undefinierter Teil der Änderungen nachgeholt sind. Hier gelten im Prinzip die gleichen Überlegungen wie oben schon im Zusammenhang mit Systemfehlern beim globalen Undo. Sofern die Implementierung des Redo einer Aktion idempotent ist, kann das partielle Redo einfach wiederholt werden.

## 7.4 Änderungsdateien

Diese Technik ist eine Präventiv-Technik gegen Systemfehler. Nach Neuladen des Systems ist die Datenbank *sofort* und ohne Recovery-Maßnahmen benutzbar, allerdings nur in einem früher korrekten Zustand (Zielzustandstyp 2, vgl. Abschnitt 3.4). Eine saubere Trennung zwischen Arbeitsversion der Datenbank und Recovery-Daten ist bei dieser Technik nicht möglich.

Die Datenbank wird aufgeteilt in eine Hauptdatei und eine Änderungsdatei.

Die **Hauptdatei** enthält einen früher korrekten Datenbankzustand. Insofern ähnelt sie stark einem Backup-Dump. Im Unterschied zu diesem ist sie jedoch direkt zugreifbar auf Platte gespeichert, und es wird auch laufend auf sie zugegriffen, allerdings nur lesend.

Die **Änderungsdatei** hat im wesentlichen den gleichen Inhalt wie ein Vorwärts-Log. Die enthaltenen Änderungen werden jedoch nicht in der Hauptdatei ausgeführt, d.h. es gibt keine Version der Datenbank auf dem aktuellen Stand.

Bei Zugriffen auf die Datenbank muß zunächst der aktuelle Stand für das betreffende Datenbankobjekt rekonstruiert werden. Zuerst wird die Änderungsdatei nach zutreffenden Einträgen durchsucht, danach erst zur Hauptdatei zugegriffen. Der Mehraufwand gegenüber einem Zugriff zu einer Datenbank auf aktuellem Stand kann durch geschickte Wahl der Algorithmen in erträglichem Rahmen gehalten werden (vgl. [SeL76]).

Die Änderungsdatei wird regelmäßig und meist im laufenden Betrieb in die Hauptdatei gemischt. Zeitpunkt und Häufigkeit hängen von einigen Umständen ab, jedoch sind die Mischungen i.d.R. so häufig, daß die Änderungsdatei erheblich kleiner ist als ein Log.

Während des Mischens besteht kein Schutz gegen Systemfehler mehr. Aus diesem Grund werden zwei Kopien der Hauptdatei empfohlen, von denen eine als Backup-Kopie dient. Ferner empfiehlt sich das Prinzip des vorsichtigen Änderns.

©2003 Udo Kelter 

# 8 Recovery für Medienfehler

Grundlage aller Recovery-Maßnahmen für Medienfehler sind Dumps. Ein **Dump** (auch *backup copy* oder *image copy*) ist eine Kopie (eines Teils) der Datenbank. Die Erzeugung eines Dumps ist bei sehr großen Datenbanken aufwendig, selbst bei schnellen Platten und Rechnern kann die Dauer für die Erzeugung in der Größenordnung einer Stunde liegen. Ein Dump sollte daher möglichst außerhalb der normalen Betriebsszeiten erzeugt werden, u.a. um den Normalbetrieb nicht zu beeinträchtigen. Bei Systemen, die rund um die Uhr verfügbar sein müssen, kann für die Erzeugung des Dumps nicht einfach die ganze Datenbank gesperrt werden; hier muß durch spezielle Maßnahmen dafür gesorgt werden, daß der Dump überhaupt einen konsistenten Datenbankzustand enthält.

Ein **inkrementeller Dump** ist eine Kopie der seit einem bestimmten Datum veränderten Teile der Datenbank, i.d.R. auf dem Niveau von Seiten. Bezugszeitpunkt ist der Zeitpunkt der Anfertigung des letzten vollständigen oder inkrementellen Dumps.

**Rücksetzen, der DB** Für das **Rücksetzen der DB** benötigt man nun einen vollständigen Dump und beliebig viele inkrementelle Dumps und geht wie folgt vor:

1. löschen der vorhandenen Arbeitversion
2. laden des vollständigen Dumps
3. einmischen der inkrementellen Dumps in der gleichen Reihenfolge, in der sie erzeugt worden sind

Für das anschließende **globale Redo** benötigt man einen Vorwärts-Log (also ggf. mehrere Archiv-Logs und einen aktiven Log). Zuerst werden die Archiv-Logs in Reihenfolge ihrer Entstehung eingespielt, dann der aktive Log. Beim Einspielen des aktiven Log ist das Vorgehen analog zum partiellen Redo, d.h. es werden nur die vollständigen Transaktionen wiederholt.

©2003 Udo Kelter 

# Literatur

[BeHG87] Bernstein, P.A.; Hadzilacos, V.; Goodman, N.: Concurrency control and recovery in database systems; Addison-Wesley Publishing Company; 1987

[SeL76] Severance, D.G.; Lohman, G.M.: Differential files: their application to the maintenance of large databases; ACM-TODS 1:3, p.256-267; 1976/09

[Ve77] Verhofstad, J.S.M.: Recovery and crash resistance in a filing system; p.158-167 in: SIGMOD77; 1977/08

[TID] Kelter, U.: Lehrmodul "Transaktionen und die Integrität von Datenbanken"; 2003

[DBSA] Kelter, U.: Lehrmodul "Architektur von DBMS"; 2001

# Glossar

**Änderungsdatei:** Aufteilung der Datenbank in eine (nahezu statische) Hauptdatei und eine Änderungsdatei, die Differenzen zwischen aktuellem Stand und Stand gemäß der Hauptdatei enthält

**aktiver Log:** auf Platte stehender Teil des gesamten Logs, in dem ältere Einträge, die in Archiv-Logs ausgelagert worden sind, fehlen

**aktuelle Datenbank:** logischer Zustand, der sich aufgrund des Inhalts der persistenten *und* der flüchtigen Medien ergibt; relevant für den normalen Betrieb

**Archiv-Log:** komprimierter Redo-Log, der ggf. für ein globales Redo benutzt wird

**Dump:** früher angefertigte Kopie des Zustands der Datenbank oder eines Teils von ihr

**Fehler-Atomarität:** Eigenschaft einer Transaktion, daß die Folge von Aktionen ganz oder gar nicht ausgeführt wird

**Fehlerkompensation:** Behebung von Schäden durch Anwendung einer kompensierenden Operation auf die betroffenen Objekte

**globales Redo:** Recovery-Grundfunktion, die alle Änderungen in der Datenbank, die seit dem Erzeugen der letzten Sicherungskopie stattgefunden haben, nachholt

**globales Undo:** Recovery-Grundfunktion, die die Wirkung aller durch einen Systemfehler unterbrochenen Transaktionen rückgängig macht

**Idempotenz:** Eigenschaft von Operationen (hier speziell Aktions-Undos oder -Redos), bei mehrfacher Ausführung den gleichen Effekt zu erzeugen wie einmaliger Ausführung

**inkrementeller Dump:** Dump, der nur die seit einem bestimmten Datum veränderten Teile der Datenbank enthält

**Log:** Aufzeichnung der Änderungen in der Datenbank für Recovery-Zwecke

**materialisierte Datenbank:** logischer Zustand, der sich aus dem Inhalt der persistenten Medien ergibt (die physisch konsistent sein müssen); relevant für den Neustart nach einem Systemfehler

**Medienfehler:** Gruppe von Schäden, bei denen ein Speichermedium so stark beschädigt ist, daß der Inhalt komplett unbrauchbar geworden und vollständig aus den Recovery-Daten rekonstruiert werden muß

**Neuladen:** Recovery-Grundfunktion, die eine früher erzeugte Sicherungskopie als Datenbankinhalt installiert

**partielles Redo:** Recovery-Grundfunktion, die die Änderungen der Transaktionen nachholt, die vor dem Systemfehler beendet wurden und deren Wirkung nicht in der materialisierten Datenbank enthalten ist

**physisch konsistent:** eine Datenbank ist physisch konsistent, wenn sich die internen Speicherungsstrukturen in einem ordnungsgemäßen Zustand befinden; auf einem physisch inkonsistenten Zustand kann der Laufzeitkern i.a. nicht mehr korrekt arbeiten

**physische Datenbank:** physischer Zustand, der sich allein aufgrund des Inhalts der persistenten Medien ergibt; relevant für Rettungsprogramme nach Medienfehlern

**Recovery:** Wiederherstellung der Datenbank nach einer Beschädigung infolge einer Störung

**Redo einer Aktion:** Recovery-Grundfunktion, die die Wirkung einer Aktion nachholt

**Redo-Checkpoint:** spezieller Log-Eintrag, der anzeigt, daß zu diesem Zeitpunkt alle veränderten Seiten zurückgeschrieben worden sind; erlaubt eine effizientere Implementierung des partiellen Redos

**Redo-Log:** Log, der nur Daten für das Vorwärts-Recovery enthält

**Rollback:** Recovery-Grundfunktion, die die bisherigen Wirkungen einer Transaktion aufhebt

©2003 Udo Kelter 

**Rückwärts-Recovery** (*backward recovery*): Recovery-Verfahren, bei denen die beschädigten Teile der Datenbank in einen früheren Zustand zurückversetzt werden, möglichst in den unmittelbar vor der Störung

**Schaden:** Veränderung des Inhalts der Datenbank infolge einer Störung

**Störung:** Ereignis, bei dem irgendwelche Teile des Systems nicht erwartungsgemäß bzw. gemäß ihrer Spezifikation arbeiten

**Systemfehler:** Absturz des DBMS-Kerns bzw. Gruppe von Schäden, die dadurch verursacht werden; die Schäden bestehen insb. im Verlust der Daten in den flüchtigen Speichern (Puffern), aber nicht darin, daß die Datenbank physisch inkonsistent wird

**Transaktionsfehler:** Abbruch einer Transaktion bzw. Gruppe von Schäden, die hierdurch verursacht werden; die Schäden bestehen in den veränderten Objekten, die potentiell einen logisch inkonsistenten Zustand bilden

**Transitions-Logging:** Logging-Verfahren, bei dem in den Logeinträgen Zustandsübergänge protokolliert werden

**Undo einer Aktion:** Recovery-Grundfunktion, die die Wirkung einer Aktion rückgängig macht

**Undo-Checkpoint:** spezieller Log-Eintrag, der die Identifikationen der zur Zeit aktiven Transaktionen enthält; erlaubt eine effizientere Implementierung des globalen Undos

**Undo-Log:** Log, der nur Daten für das Rückwärts-Recovery enthält

**verzögertes Schreiben** (*deferred update*): alle Schreibaktionen (Änderungen) einer Transaktion werden erst im Rahmen des Commits durchgeführt

**vorsichtiges Ändern** (*careful replacement*): Änderungen (in der physischen Datenbank) werden nicht durchgeführt, indem Sektoren überschrieben werden, sondern indem zunächst eine neue Version aller veränderten Seiten eingefügt und dann auf einmal auf die neuen Versionen umgeschaltet wird

**Vorwärts-Recovery** (*forward recovery*): Recovery-Verfahren, bei denen Änderungen in der Datenbank nachgeholt werden, z.B. nach Neuladen der Datenbank

**Zustands-Logging:** Logging-Verfahren, bei dem in den Logeinträgen Zustände vor bzw. nach einer Aktion gespeichert werden

©2003 Udo Kelter 

# Index

©2003 Udo Kelter 