Kapitel 13

Recovery

Aufgabe der Recovery-Komponente des Datenbanksystems ist es, nach einem Fehler den jüngsten konsistenten Datenbankzustand wiederherzustellen.

13.1 Fehlerklassen

Wir unterscheiden drei Fehlerklassen:

1. lokaler Fehler in einer noch nicht festgeschriebenen Transaktion,
2. Fehler mit Hauptspeicherverlust,
3. Fehler mit Hintergrundspeicherverlust.

13.1.1 Lokaler Fehler einer Transaktion

Typische Fehler in dieser Fehlerklasse sind

- Fehler im Anwendungsprogramm,
- expliziter Abbruch (**abort**) der Transaktion durch den Benutzer,
- systemgesteuerter Abbruch einer Transaktion, um beispielsweise eine Verklemmung (Deadlock) zu beheben.

Diese Fehler werden behoben, indem alle Änderungen an der Datenbasis, die von dieser noch aktiven Transaktion verursacht wurden, rückgängig gemacht werden (*lokales Undo*). Dieser Vorgang tritt recht häufig auf und sollte in wenigen Millisekunden abgewickelt sein.

13.1.2 Fehler mit Hauptspeicherverlust

Ein Datenbankverwaltungssystem manipuliert Daten innerhalb eines *Datenbankpuffers*, dessen Seiten zuvor aus dem Hintergrundspeicher *eingelagert* worden sind und nach gewisser Zeit (durch Verdrängung) wieder *ausgelagert* werden müssen. Dies bedeutet, daß die im Puffer durchgeführten Änderungen erst mit dem Zurückschreiben in die materialisierte Datenbasis permanent werden. Abbildung 13.1 zeigt eine Seite P_A, in die das von A nach A' geänderte Item bereits zurückgeschrieben wurde, während die Seite P_C noch das alte, jetzt nicht mehr aktuelle Datum C enthält.

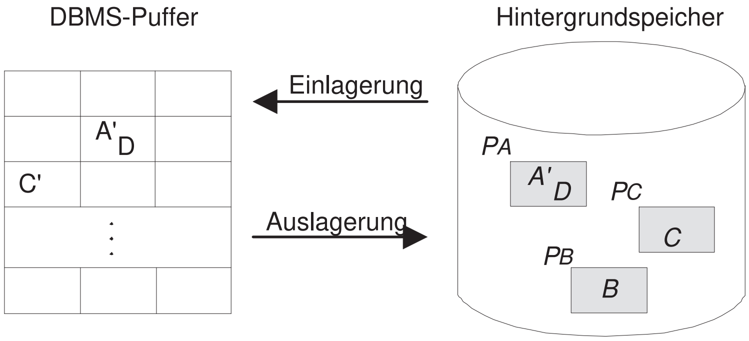

Abbildung 13.1: Schematische Darstellung der zweistufigen Speicherhierarchie

Bei einem Verlust des Hauptspeicherinhalts verlangt das Transaktionsparadigma, daß

- alle durch nicht abgeschlossene Transaktionen schon in die materialisierte Datenbasis eingebrachten Änderungen rückgängig gemacht werden (*globales undo*) und
- alle noch nicht in die materialisierte Datenbasis eingebrachten Änderungen durch abgeschlossene Transaktionen nachvollzogen werden (*globales redo*).

Fehler dieser Art treten im Intervall von Tagen auf und sollten mit Hilfe einer Log-Datei in wenigen Minuten behoben sein.

13.1.3 Fehler mit Hintergrundspeicherverlust

Fehler mit Hintergrundspeicherverlust treten z.B in folgenden Situationen auf:

- *head crash*, der die Platte mit der materialisierten Datenbank zerstört,
- Feuer/Erdbeben, wodurch die Platte zerstört wird,
- Fehler im Systemprogramm (z. B. im Plattentreiber).

Solche Situationen treten sehr selten auf (etwa im Zeitraum von Monaten oder Jahren). Die Restaurierung der Datenbasis geschieht dann mit Hilfe einer (hoffentlich unversehrten) Archiv-Kopie der materialisierten Datenbasis und mit einem Log-Archiv mit allen seit Anlegen der Datenbasis-Archivkopie vollzogenen Änderungen.

13.2 Die Speicherhierarchie

13.2.1 Ersetzen von Pufferseiten

Eine Transaktion referiert Daten, die über mehrere Seiten verteilt sind. Für die Dauer eines Zugriffs wird die jeweilige Seite im Puffer *fixiert*, wodurch ein Auslagern verhindert wird. Werden Daten auf einer fixierten Seite geändert, so wird die Seite als *dirty* markiert. Nach Abschluß der Operation wird der *FIX*-Vermerk wieder gelöscht und die Seite ist wieder für eine Ersetzung freigegeben.

Es gibt zwei Strategien in Bezug auf das Ersetzen von Seiten:

- *¬steal* : Die Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ist ausgeschlossen.
- *steal* : Jede nicht fixierte Seite darf ausgelagert werden.

Bei der *¬steal*-Strategie werden niemals Änderungen einer noch nicht abgeschlossenen Transaktion in die materialisierte Datenbasis übertragen. Bei einem *rollback* einer noch aktiven Transaktion braucht man sich also um den Zustand des Hintergrundspeichers nicht zu kümmern, da die Transaktion vor dem **commit** keine Spuren hinterlassen hat. Bei der *steal*-Strategie müssen nach einem *rollback* die bereits in die materialisierte Datenbasis eingebrachten Änderungen durch ein *Undo* rückgängig gemacht werden.

13.2.2 Zurückschreiben von Pufferseiten

Es gibt zwei Strategien in Bezug auf die Wahl des Zeitpunkts zum Zurückschreiben von modifizierten Seiten:

- *force*: Beim **commit** einer Transaktion werden alle von ihr modifizierten Seiten in die materialisierte Datenbasis zurückkopiert.
- *¬force*: Modifizierte Seiten werden nicht unmittelbar nach einem **commit**, sondern ggf. auch später, in die materialisierte Datenbasis zurückkopiert.

Bei der *¬force*-Strategie müssen daher weitere Protokoll-Einträge in der Log-Datei notiert werden, um im Falle eines Fehlers die noch nicht in die materialisierte Datenbasis propagierten Änderungen nachvollziehen zu können. Tabelle 13.1 zeigt die vier Kombinationsmöglichkeiten.

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

Tabelle 13.1: Kombinationsmöglichkeiten beim Einbringen von Änderungen

Auf den ersten Blick scheint die Kombination *force* und *¬steal* verlockend. Allerdings ist das sofortige Ersetzen von Seiten nach einem **commit** sehr unwirtschaftlich, wenn solche Seiten sehr intensiv auch von anderen, noch aktiven Transaktionen benutzt werden (*hot spots*).

13.2.3 Einbringstrategie

Es gibt zwei Strategien zur Organisation des Zurückschreibens:

- *update-in-place*: Jeder ausgelagerten Seite im Datenbankpuffer entspricht eine Seite im Hintergrundspeicher, auf die sie kopiert wird im Falle einer Modifikation.
- Twin-Block-Verfahren: Jeder ausgelagerten Seite P im Datenbankpuffer werden zwei Seiten P^0 und P^1 im Hintergrundspeicher zugeordnet, die den letzten bzw. vorletzten Zustand dieser Seite in der materialisierten Datenbasis darstellen. Das Zurückschreiben erfolgt jeweils auf den vorletzten Stand, sodaß bei einem Fehler während des Zurückschreibens der letzte Stand noch verfügbar ist.

13.3 Protokollierung der Änderungsoperationen

Wir gehen im weiteren von folgender Systemkonfiguration aus:

- *steal* : Nicht fixierte Seiten können jederzeit ersetzt werden.
- $\neg$*force* : Geänderte Seiten werden kontinuierlich zurückgeschrieben.
- *update-in-place* : Jede Seite hat genau einen Heimatplatz auf der Platte.
- *Kleine Sperrgranulate* : Verschiedene Transaktionen manipulieren verschiedene Records auf derselben Seite. Also kann eine Seite im Datenbankpuffer sowohl Änderungen einer abgeschlossenen Transaktion als auch Änderungen einer noch nicht abgeschlossenen Transaktion enthalten.

13.3.1 Struktur der Log-Einträge

Für jede Änderungsoperation, die von einer Transaktion durchgeführt wird, werden folgende Protokollinformationen benötigt:

- Die *Redo*-Information gibt an, wie die Änderung nachvollzogen werden kann.
- Die *Undo*-Information gibt an, wie die Änderung rückgängig gemacht werden kann.
- Die *LSN (Log Sequence Number)* ist eine eindeutige Kennung des Log-Eintrags und wird monoton aufsteigend vergeben.
- Die *Transaktionskennung TA* der ausführenden Transaktion.
- Die *PageID* liefert die Kennung der Seite, auf der die Änderung vollzogen wurde.
- Die *PrevLSN* liefert einen Verweis auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion (wird nur aus Effizienzgründen benötigt).

13.3.2 Beispiel einer Log-Datei

Tabelle 13.2 zeigt die verzahnte Ausführung zweier Transaktionen und das zugehörige Log-File. Zum Beispiel besagt der Eintrag mit der *LSN* #3 folgendes:

- Der Log-Eintrag bezieht sich auf Transaktion T_1 und Seite P_A.
- Für ein *Redo* muß A um 50 erniedrigt werden.
- Für ein *Undo* muß A um 50 erhöht werden.
- Der vorhergehende Log-Eintrag hat die *LSN* #1.

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

Tabelle 13.2: Verzahnte Ausführung zweier Transaktionen und Log-Datei

13.3.3 Logische versus physische Protokollierung

In dem Beispiel aus Tabelle 13.2 wurden die *Redo*- und die *Undo*-Informationen logisch protokolliert, d.h. durch Angabe der Operation. Eine andere Möglichkeit besteht in der physischen Protokollierung, bei der statt der *Undo*-Operation das sogenannte *Before-Image* und für die *Redo*-Operation das sogenannte *After-Image* gespeichert wird.

Bei der logischen Protokollierung wird

- das *Before-Image* durch Ausführung des *Undo*-Codes aus dem *After-Image* generiert,
- das *After-Image* durch Ausführung des *Redo*-Codes aus dem *Before*-Image generiert.

Um zu erkennen, ob das *Before-Image* oder *After-Image* in der materialisierten Datenbasis enthalten ist, dient die *LSN*. Beim Anlegen eines Log-Eintrages wird die neu generierte LSN in einen reservierten Bereich der Seite geschrieben und dann später mit dieser Seite in die Datenbank zurückkopiert. Daraus läßt sich erkennen, ob für einen bestimmten Log-Eintrag das *Before-Image* oder das *After-Image* in der Seite steht:

- Wenn die LSN der Seite einen kleineren Wert als die LSN des Log-Eintrags enthält, handelt es sich um das *Before-Image*.
- Ist die LSN der Seite größer oder gleich der LSN des Log-Eintrags, dann wurde bereits das *After-Image* auf den Hintergrundspeicher propagiert.

13.3.4 Schreiben der Log-Information

Bevor eine Änderungsoperation ausgeführt wird, muß der zugehörige Log-Eintrag angelegt werden. Die Log-Einträge werden im *Log-Puffer* im Hauptspeicher zwischengelagert. Abbildung 13.2 zeigt das Wechselspiel zwischen den beteiligten Sicherungskomponenten.

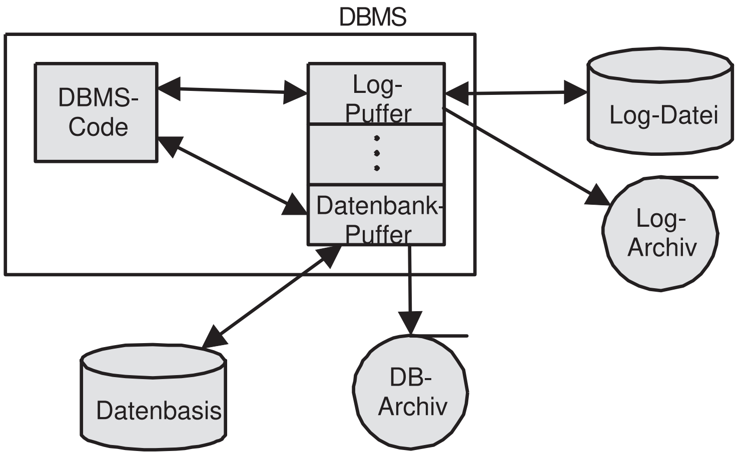

Abbildung 13.2: Speicherhierarchie zur Datensicherung

In modernen Datenbanksystemen ist der Log-Puffer als Ringpuffer organisiert. An einem Ende wird kontinuierlich geschrieben und am anderen Ende kommen laufend neue Einträge hinzu (Abbildung 13.3). Die Log-Einträge werden gleichzeitig auf das temporäre Log (Platte) und auf das Log-Archiv (Magnetband) geschrieben.

13.3.5 WAL-Prinzip

Beim Schreiben der Log-Information gilt das *WAL*-Prinzip (Write Ahead Log):

- Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle zu ihr gehörenden Log-Einträge geschrieben werden. Dies ist erforderlich, um eine erfolgreich abgeschlossene Transaktion nach einem Fehler nachvollziehen zu können (*redo*).
- Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in die Log-Datei geschrieben werden. Dies ist erforderlich, um im

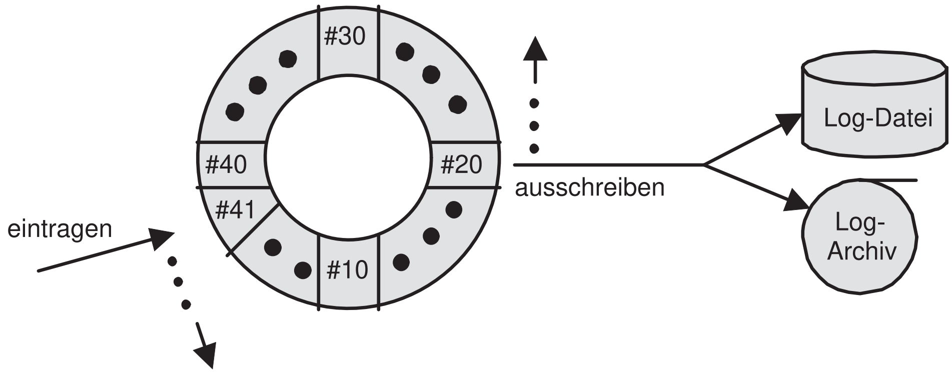

Abbildung 13.3: Log-Ringpuffer

Fehlerfall die Änderungen nicht abgeschlossener Transaktionen aus den modifizierten Seiten der materialisierten Datenbasis entfernen zu können (*undo*).

13.4 Wiederanlauf nach einem Fehler

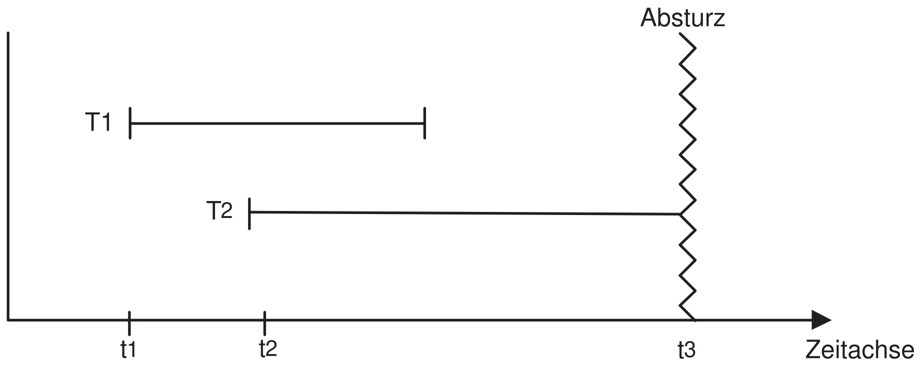

Abbildung 13.4: Zwei Transaktionstypen bei Systemabsturz

Abbildung 13.4 zeigt die beiden Transaktionstypen, die nach einem Fehler mit Verlust des Hauptspeicherinhalts zu behandeln sind:

- Transaktion T_1 ist ein *Winner* und verlangt ein *Redo*.
- Transaktion T_2 ist ein *Loser* und verlangt ein *Undo*.

Der Wiederanlauf geschieht in drei Phasen (Abbildung 13.5):

1. *Analyse*: Die Log-Datei wird von Anfang bis Ende analysiert, um die *Winner* (kann **commit** vorweisen) und die *Loser* (kann kein **commit** vorweisen) zu ermitteln.

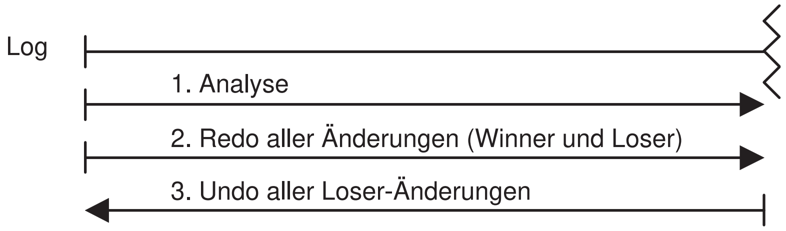

Abbildung 13.5: Wiederanlauf in drei Phasen

2. *Redo*: Es werden alle protokollierten Änderungen in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht, sofern sich nicht bereits das Afterimage des Protokolleintrags in der materialisierten Datenbasis befindet. Dies ist dann der Fall, wenn die *LSN* der betreffenden Seite gleich oder größer ist als die *LSN* des Protokolleintrags.

3. Undo: Die Log-Datei wird in umgekehrter Richtung, d.h. von hinten nach vorne, durchlaufen. Dabei werden die Einträge von *Winner*-Transaktionen übergangen. Für jeden Eintrag einer *Loser*-Transaktion wird die *Undo*-Operation durchgeführt.

Spezielle Vorkehrungen müssen getroffen werden, um auch Fehler beim Wiederanlauf kompensieren zu können. Es wird nämlich verlangt, daß die *Redo*- und *Undo*-Phasen *idempotent* sind, d.h. sie müssen auch nach mehrmaliger Ausführung (hintereinander) immer wieder dasselbe Ergebnis liefern:

$$undo(undo(...(undo(a))...)) = undo(a)$$
$$redo(redo(...(redo(a))...)) = redo(a)$$

13.5 Lokales Zurücksetzen einer Transaktion

Die zu einer zurückzusetzenden Transaktion gehörenden Log-Einträge werden mit Hilfe des *PrevLSN*-Eintrags in umgekehrter Reihenfolge abgearbeitet. Jede Änderung wird durch eine *Undo*-Operation rückgängig gemacht.

Wichig in diesem Zusammenhang ist die Verwendung von *rücksetzbaren Historien*, die auf den Schreib/Leseabhängigkeiten basieren.

Wir sagen, daß in einer Historie H die Transaktion T_i von der Transaktion T_j liest, wenn folgendes gilt:

- T_j schreibt ein Datum A, das T_i nachfolgend liest.
- T_j wird nicht vor dem Lesevorgang von T_i zurückgesetzt.
- Alle anderen zwischenzeitlichen Schreibvorgänge auf A durch andere Transaktionen werden vor dem Lesen durch T_i zurückgesetzt.

Eine Historie heißt *rücksetzbar*, falls immer die schreibende Transaktion T_j vor der lesenden Transaktion T_i ihr **commit** ausführt. Anders gesagt: Eine Transaktion darf erst dann ihr

commit ausführen, wenn alle Transaktionen, von denen sie gelesen hat, beendet sind. Wäre diese Bedingung nicht erfüllt, könnte man die schreibende Transaktion nicht zurücksetzen, da die lesende Transaktion dann mit einem offiziell nie existenten Wert für A ihre Berechnung **commited** hätte.

13.6 Sicherungspunkte

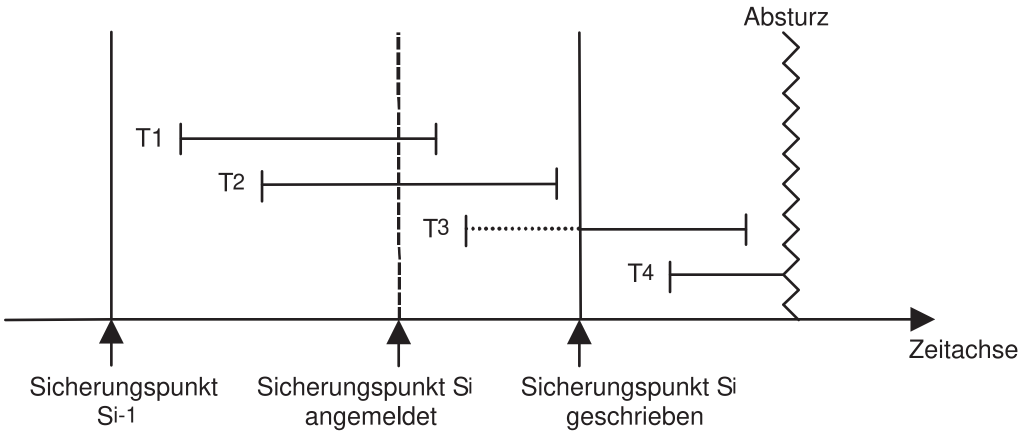

Abbildung 13.6: Transaktionsausführung relativ zu einem Sicherungspunkt

Mit zunehmender Betriebszeit des Datenbanksystems wird die zu verarbeitende Log-Datei immer umfangreicher. Durch einen *Sicherungspunkt* wird eine Position im Log vermerkt, über den man beim Wiederanlauf nicht hinausgehen muß.

Abbildung 13.6 zeigt den dynamischen Verlauf. Nach Anmeldung des neuen Sicherungspunktes S_i wird die noch aktive Transaktion T_2 zu Ende geführt und der Beginn der Transaktion T_3 verzögert. Nun werden alle modifizierten Seiten auf den Hintergrundspeicher ausgeschrieben und ein transaktionskonsistenter Zustand ist mit dem Sicherungspunkt S_i erreicht. Danach kann man mit der Log-Datei wieder von vorne beginnen.

13.7 Verlust der materialisierten Datenbasis

Bei Zerstörung der materialisierten Datenbasis oder der Log-Datei kann man aus der Archiv-Kopie und dem Log-Archiv den jüngsten, konsistenten Zustand wiederherstellen.

Abbildung 13.7 faßt die zwei möglichen Recoveryarten nach einem Systemabsturz zusammen:

- Der obere (schnellere) Weg wird bei intaktem Hintergrundspeicher beschritten.
- Der untere (langsamere) Weg wird bei zerstörtem Hintergrundspeichert beschritten.

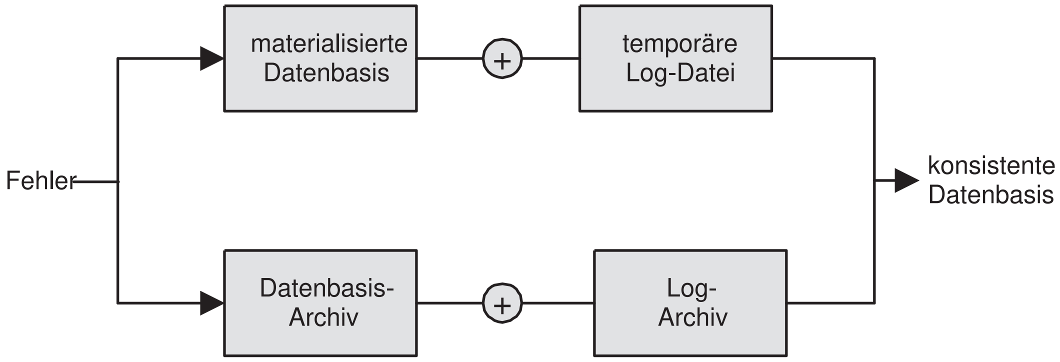

Abbildung 13.7: Zwei Recovery-Arten

Kapitel 14

Sicherheit

In diesem Kapitel geht es um den Schutz gegen absichtliche Beschädigung oder Enthüllung von sensiblen Daten. Abbildung 14.1 zeigt die hierarchische Kapselung verschiedenster Maßnahmen.

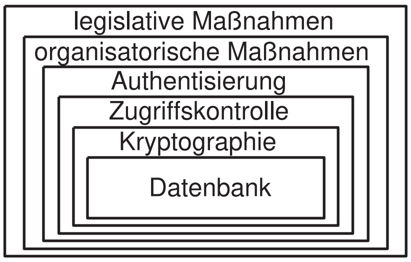

Abbildung 14.1: Ebenen des Datenschutzes

14.1 Legislative Maßnahmen

Im *Gesetz zum Schutz vor Mißbrauch personenbezogener Daten bei der Datenverarbeitung* ist festgelegt, welche Daten in welchem Umfang schutzbedürftig sind.

14.2 Organisatorische Maßnahmen

Darunter fallen Maßnahmen, um den persönlichen Zugang zum Computer zu regeln:

- bauliche Maßnahmen
- Pförtner
- Ausweiskontrolle
- Diebstahlsicherung

- Alarmanlage

14.3 Authentisierung

Darunter fallen Maßnahmen zur Überprüfung der Identität eines Benutzers:

- Magnetkarte
- Stimmanalyse/Fingerabdruck
- Paßwort: w ohne Echo eintippen,
  System überprüft, ob $f(w)$ eingetragen ist, f^{-1} aus f nicht rekonstruierbar
- dynamisches Paßwort: vereinbare Algorithmus, der aus Zufallsstring gewisse Buchstaben heraussucht

Paßwortverfahren sollten mit Überwachungsmaßnahmen kombiniert werden (Ort, Zeit, Fehleingabe notieren)

14.4 Zugriffskontrolle

Verschiedene Benutzer haben verschiedene Rechte bzgl. derselben Datenbank. Tabelle 14.1 zeigt eine Berechtigungsmatrix (wertunabhängig):

Benutzer	Ang-Nr	Gehalt	Leistung
A (Manager)	R	R	RW
B (Personalchef)	RW	RW	R
C (Lohnbüro)	R	R	—

Tabelle 14.1: Berechtigungsmatrix

Bei einer wertabhängigen Einschränkung wird der Zugriff von der aktuellen Ausprägung abhängig gemacht:

$$\text{Zugriff } (A, \text{Gehalt}): \quad \begin{array}{lccr} R: & \text{Gehalt} & < & 10.000 \\ W: & \text{Gehalt} & < & 5.000 \end{array}$$

Dies ist natürlich kostspieliger, da erst nach Lesen der Daten entschieden werden kann, ob der Benutzer die Daten lesen darf. Ggf. werden dazu Tabellen benötigt, die für die eigentliche Anfrage nicht verlangt waren. Beispiel: Zugriff verboten auf Gehälter der Mitarbeiter an Projekt 007.

Eine Möglichkeit zur Realisierung von Zugriffskontrollen besteht durch die Verwendung von Sichten:

```
define view v(angnr, gehalt) as
select angnr, gehalt from angest
where gehalt < 3000
```

Eine andere Realisierung von Zugriffskontrollen besteht durch eine Abfragemodifikation.

- **Beispiel:**
  Die Abfrageeinschränkung

  ```
  deny (name, gehalt) where gehalt > 3000
  ```

  liefert zusammen mit der Benutzer-Query

  ```
  select gehalt from angest where name = 'Schmidt'
  ```

  die generierte Query

  ```
  select gehalt from angest
  where name = 'Schmidt' and not gehalt > 3000
  ```

In statistischen Datenbanken dürfen Durchschnittswerte und Summen geliefert werden, aber keine Aussagen zu einzelnen Tupeln. Dies ist sehr schwer einzuhalten, selbst wenn die Anzahl der referierten Datensätze groß ist.

- **Beispiel:**
  Es habe Manager X als einziger eine bestimmte Eigenschaft, z. B. habe er das höchste Gehalt. Dann läßt sich mit folgenden beiden Queries das Gehalt von Manager X errechnen, obwohl beide Queries alle bzw. fast alle Tupel umfassen:

  ```
  select sum (gehalt) from angest;
  select sum (gehalt) from angest
  where gehalt < (select max(gehalt) from angest);
  ```

In SQL-92 können Zugriffsrechte dynamisch verteilt werden, d. h. der Eigentümer einer Relation kann anderen Benutzern Rechte erteilen und entziehen.

Die vereinfachte Syntax lautet:

```
grant { select | insert | delete | update | references | all }
on <relation> to <user> [with grant option]
```

Hierbei bedeuten

`select`:	darf Tupel lesen
`insert`:	darf Tupel einfügen
`delete`:	darf Tupel löschen
`update`:	darf Tupel ändern
`references`:	darf Fremdschlüssel anlegen
`all` :	`select + insert + delete + update + references`
`with grant option`:	`<user>` darf die ihm erteilten Rechte weitergeben

- **Beispiel:**
  A: `grant read, insert on angest to` B `with grant option`
  B: `grant read on angest to` C `with grant option`
  B: `grant insert on angest to` C

Das Recht, einen Fremdschlüssel anlegen zu dürfen, hat weitreichende Folgen: Zum einen kann das Entfernen von Tupeln in der referenzierten Tabelle verhindert werden. Zum anderen kann durch das probeweise Einfügen von Fremdschlüsseln getestet werden, ob die (ansonsten lesegeschützte) referenzierte Tabelle gewisse Schlüsselwerte aufweist:

```
create table Agententest(Kennung character(4) references Agenten);
```

Jeder Benutzer, der ein Recht vergeben hat, kann dieses mit einer *Revoke*-Anweisung wieder zurücknehmen:

```
revoke { select | insert | delete | update | references | all }
on <relation> from <user>
```

- **Beispiel:**
  B: `revoke all on angest from` C
  Es sollen dadurch dem Benutzer C alle Rechte entzogen werden, die er von B erhalten hat, aber nicht solche, die er von anderen Benutzern erhalten hat. Außerdem erlöschen die von C weitergegebenen Rechte.

Der Entzug eines Grant `G` soll sich so auswirken, als ob `G` niemals gegeben worden wäre!

- **Beispiel:**

```
A: grant read, insert, update  on angest to D
B: grant read, update          on angest to D with grant option
D: grant read, update          on angest to E
A: revoke insert, update       on angest from D
```

  Hierdurch verliert D sein `insert`-Recht, E verliert keine Rechte. Falls aber vorher A Rechte an B gab, z.B. durch

```
A: grant all on angest to B with grant option
```

  dann müssten D und E ihr *update*-Recht verlieren.

14.5 Auditing

Auditing bezeichnet die Möglichkeit, über Operationen von Benutzern Buch zu führen. Einige (selbsterklärende) Kommandos in SQL-92:

```
audit delete any table;
noaudit delete any table;
audit update on erika.professoren whenever not successful;
```

Der resultierende *Audit-Trail* wird in diversen Systemtabellen gehalten und kann von dort durch spezielle Views gesichtet werden.

14.6 Kryptographie

Da die meisten Datenbanken in einer verteilten Umgebung (Client/Server) betrieben werden, ist die Gefahr des Abhörens von Kommunikationskanälen sehr hoch. Zur Authentisierung von Benutzern und zur Sicherung gegen den Zugriff auf sensible Daten werden daher *kryptographische Methoden* eingesetzt.

Der prinzipielle Ablauf ist in Abbildung 14.2 skizziert: Der Klartext x dient als Eingabe für ein Verschlüsselungsverfahren *encode*, welches über einen Schlüssel e parametrisiert ist. Das heißt, das grundsätzliche Verfahren der Verschlüsselung ist allen Beteiligten bekannt, mit Hilfe des Schlüssels e kann der Vorgang jeweils individuell beeinflußt werden. Auf der Gegenseite wird mit dem Verfahren *decode* und seinem Schlüssel d der Vorgang umgekehrt und somit der Klartext rekonstruiert.

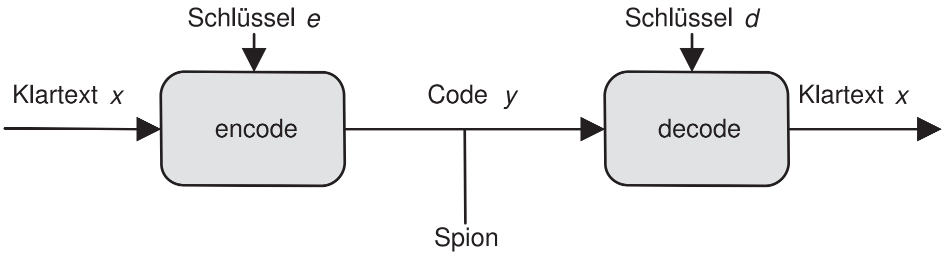

Abbildung 14.2: Ablauf beim Übertragen einer Nachricht

Zum Beispiel kann eine Exclusive-OR-Verknüpfung des Klartextes mit dem Schlüssel verwendet werden, um die Chiffre zu berechnen. Derselbe Schlüssel erlaubt dann die Rekonstruktion des Klartextes.

Klartext	`010111001`	
Schlüssel	`111010011`	
Chiffre	`101101010`	= Klartext ⊕ Schlüssel
Schlüssel	`111010011`	
Klartext	`010111001`	= Chiffre ⊕ Schlüssel

Diese Technik funktioniert so lange gut, wie es gelingt, die zum Bearbeiten einer Nachricht verwendeten Schlüssel e und d auf einem sicheren Kanal zu übertragen, z. B. durch einen Kurier. Ein Spion, der ohne Kenntnis der Schlüssel die Leitung anzapft, ist dann nicht in der Lage, den beobachteten Code zu entschlüsseln (immer vorausgesetzt, der Raum der möglichen Schlüssel wurde zur Abwehr eines vollstängigen Durchsuchens groß genug gewählt). Im Zeitalter der globalen Vernetzung besteht natürlich der Wunsch, auch die beiden Schlüsselpaare e und d per Leitung auszutauschen. Nun aber laufen wir Gefahr, daß der Spion von ihnen Kenntnis erhält und damit den Code knackt.

Dieses (auf den ersten Blick) unlösbare Problem wurde durch die Einführung von *Public Key Systems* behoben.

14.6.1 Public Key Systems

Gesucht sind zwei Funktionen $enc,\ dec : \mathbb{N} \to \mathbb{N}$ mit folgender Eigenschaft:

1. $dec((enc(x)) = x$
2. effizient zu berechnen
3. aus der Kenntnis von *enc* läßt sich *dec* nicht effizient bestimmen

Unter Verwendung dieser Funktionen könnte die Kommunikation zwischen den Partner Alice und Bob wie folgt verlaufen:

1. Alice möchte Bob eine Nachricht schicken.
2. Bob veröffentlicht sein enc_B.
3. Alice bildet $y := enc_B(x)$ und schickt es an Bob.
4. Bob bildet $x := dec_B(y)$.

14.6.2 Das RSA-Verfahren

Im Jahre 1978 schlugen Rivest, Shamir, Adleman folgendes Verfahren vor:

geheim:	Wähle zwei große Primzahlen $p,\ q$ (je 500 Bits)
öffentlich:	Berechne $n := p \cdot q$
geheim:	Wähle d teilerfremd zu $\varphi(n) = (p-1) \cdot (q-1)$
öffentlich:	Bestimme d^{-1}, d.h. e mit $e \cdot d \equiv 1 \bmod \varphi(n)$
öffentlich:	enc(x):= $x^e \bmod n$
geheim:	dec(y):= $y^d \bmod n$

- **Beispiel:**
  $p = 11,\ q = 13,\ d = 23\ \Rightarrow$
  $n = 143,\ e = 47$
  $enc(x) := x^{47} \bmod 143$
  $dec(y) := y^{23} \bmod 143$

14.6.3 Korrektheit des RSA-Verfahrens

Die Korrektheit stützt sich auf den **Satz von Fermat/Euler:**

$$x \text{ rel. prim zu } n \Rightarrow x^{\varphi(n)} \equiv 1 \bmod n$$

14.6.4 Effizienz des RSA-Verfahrens

Die Effizienz stützt sich auf folgende Überlegungen:

a) **Potenziere mod** n

Nicht e-mal mit x malnehmen, denn Aufwand wäre $O(2^{500})$, sondern:

$$x^e := \begin{cases} (x^{e/2})^2 & \text{falls } e \text{ gerade} \\ (x^{\lfloor e/2 \rfloor})^2 \cdot x & \text{falls } e \text{ ungerade} \end{cases}$$

Aufwand: $O\ (\log e)$, d.h. proportional zur Anzahl der Dezimalstellen.

b) **Bestimme** $e := d^{-1}$

Algorithmus von Euklid zur Bestimmung des ggt:

$$ggt(a,b) := \begin{cases} a & \text{falls } b = 0 \\ ggt(b,\ a \bmod b) & \text{sonst} \end{cases}$$

Bestimme $ggt(\varphi(n),\ d)$ und stelle den auftretenden Rest als Linearkombination von $\varphi(n)$ und d dar.

Beispiel:

$$\begin{array}{rcrcl} & & 120 & = & \varphi(n) \\ & & 19 & = & d \\ 120 \bmod 19 & = & 6 & = & \varphi(n) - 6 \cdot d \\ 19 \bmod 6 & = & 1 & = & d - 3 \cdot (\varphi(n) - 6d) = 19d - 3 \cdot \varphi(n) \\ & & & \Rightarrow & e = 19 \end{array}$$

c) **Konstruktion einer großen Primzahl**

Wähle 500 Bit lange ungerade Zahl x.

Teste, ob $x,\ x+2,\ x+4,\ x+6, \ldots$ Primzahl ist.

Sei $\Pi(x)$ die Anzahl der Primzahlen unterhalb von x. Es gilt:
$\Pi(x) \approx \frac{x}{\ln x} \Rightarrow$ Dichte $\approx \frac{1}{\ln x} \Rightarrow$ mittlerer Abstand $\approx \ln x$

Also zeigt sich Erfolg beim Testen ungerader Zahlen der Größe $n = 2^{500}$ nach etwa $\frac{\ln 2^{500}}{4} = 86$ Versuchen.

Komplexitätsklassen für die Erkennung von Primzahlen:

$$\begin{array}{l} \text{Prim} \overset{?}{\in} \mathbb{P} \\ \text{Prim} \in \mathbb{NP} \\ \overline{\text{Prim}} \in \mathbb{NP} \\ \overline{\text{Prim}} \in \mathbb{RP} \end{array}$$

$L \in \mathbb{RP} :\iff$ es gibt Algorithmus A, der angesetzt auf die Frage, ob $x \in L$, nach polynomialer Zeit mit ja oder nein anhält und folgende Gewähr für die Antwort gilt:

$$x \notin L \Rightarrow \text{ Antwort: nein}$$
$$x \in L \Rightarrow \text{ Antwort: } \underbrace{\text{ja}}_{>1-\varepsilon} \text{ oder } \underbrace{\text{nein}}_{<=\varepsilon}$$

Antwort: ja $\Rightarrow x$ ist zusammengesetzt.

Antwort: nein $\Rightarrow x$ ist höchstwahrscheinlich prim.

Bei 50 Versuchen $\Rightarrow$ Fehler $\leq \varepsilon^{50}$.

Satz von Rabin:

Sei $n = 2^k \cdot q + 1$ eine Primzahl, $x < n$

1) $x^q \equiv 1 \bmod n$ oder
2) $x^{q \cdot 2^i} \equiv -1 \bmod n$ für ein $i \in \{0, \ \dots, \ k-1\}$

Beispiel:

Sei $n = 97 = 2^5 \cdot 3 + 1$, sei $x = 2$.

Folge der Potenzen	x	x^3	x^6	x^{12}	x^{24}	x^{48}	x^{96}
Folge der Reste	2	8	64	22	-1	1	1

Definition eines Zeugen:

Sei $n = 2^k \cdot q + 1$.

Eine Zahl $x < n$ heißt Zeuge für die Zusammengesetztheit von n

1) $ggt(x, n) \neq 1$ oder
2) $x^q \not\equiv 1 \bmod n$ und $x^{q \cdot 2^i} \not\equiv -1$ für alle $i \in \{0, \dots, k-1\}$

Satz von Rabin:

Ist n zusammengesetzt, so gibt es mindestens $\frac{3}{4}n$ Zeugen.

```
function prob-prim (n: integer): boolean
z:=0;
repeat
    z=z+1;
    wuerfel x;
until (x ist Zeuge fuer n) OR (z=50);
return (z=50)
```

Fehler: $(\frac{1}{4})^{50} \sim 10^{-30}$

14.6.5 Sicherheit des RSA-Verfahrens

Der Code kann nur durch das Faktorisieren von n geknackt werden.
Schnellstes Verfahren zum Faktorisieren von n benötigt

$$n^{\sqrt{\frac{\ln\ln(n)}{\ln(n)}}} \text{ Schritte.}$$

Für $n = 2^{1000} \Rightarrow \ln(n) = 690,\ \ln\ln(n) = 6.5$

Es ergeben sich $\approx \sqrt[10]{n}$ Schritte $\approx 10^{30}$ Schritte $\approx 10^{21}$ sec (bei 10^9 Schritte pro sec) $\approx 10^{13}$ Jahre.

14.6.6 Implementation des RSA-Verfahrens

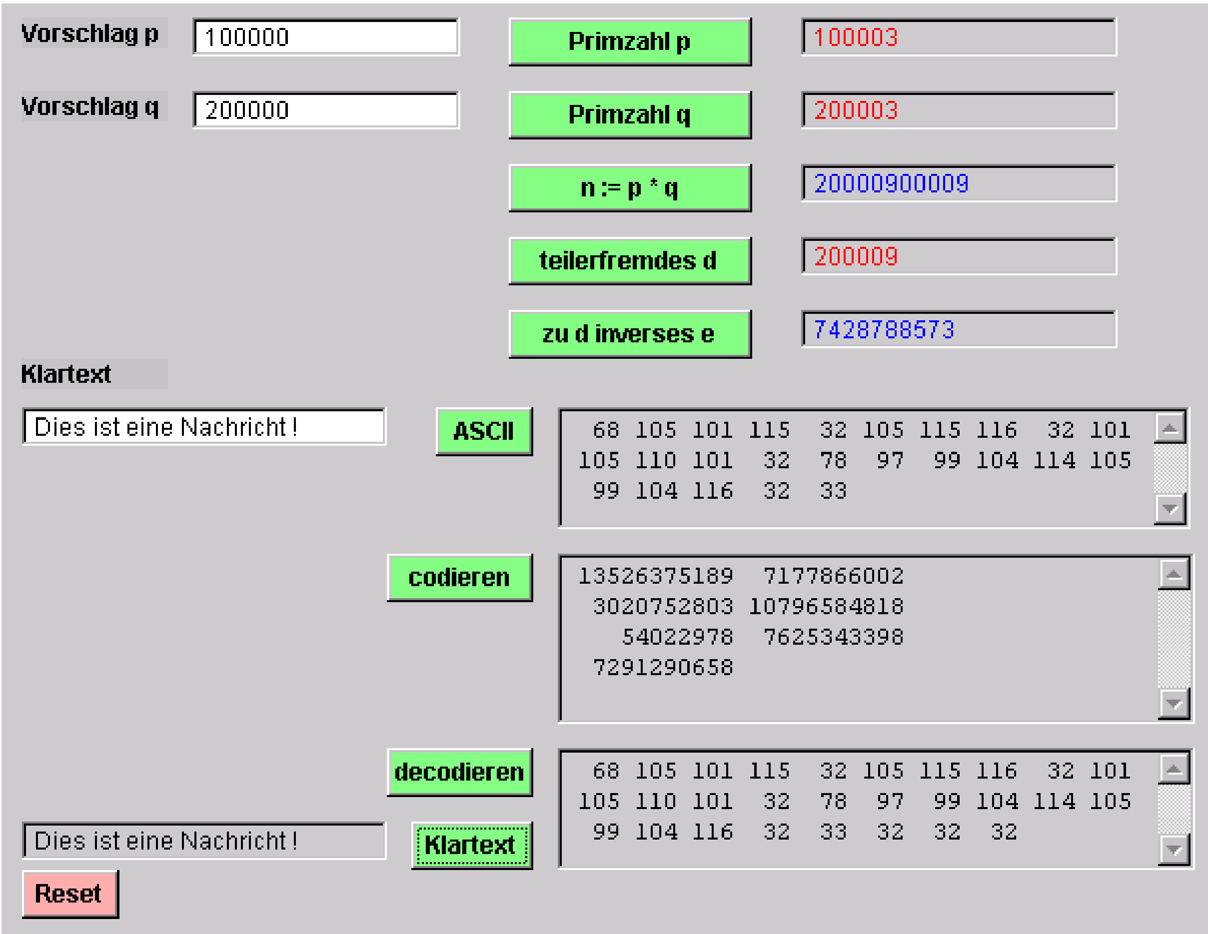

Abbildung 14.3: Java-Applet mit RSA-Algorithmus

14.6.7 Anwendungen des RSA-Verfahrens

Verschlüsseln :

Alice schickt $y := enc_B(x)$ an Bob.
Bob bildet $x := dec_B(y)$.

Unterschreiben einer geheimen Nachricht :

Alice schickt $y := enc_B(dec_A(x))$ an Bob.
Bob bildet $x := enc_A(dec_B(y))$.

Unterschreiben einer öffentlichen Nachricht :

Sei f eine unumkehrbare Hashfunktion.
Alice bildet $z := dec_A(f(x))$.
Alice schickt $< x , z >$ an Bob.
Bob vergleicht $f(x)$ mit $enc_A(z)$.

Anonymer Zahlungsverkehr mit blinder Unterschrift :

Alice würfelt Schecknummer x und Ausblendfaktor r.
Alice schickt $s := x \cdot enc_{Bank}(r)$ zur Bank.
Bank belastet Alice's Konto mit 1,-DM und schickt $z = dec_{Bank}(s)$ zurück.
Alice erhält also $(x \cdot r^e)^d \bmod n = (x^d \cdot r) \bmod n$.
Alice bildet z/r und verfügt nun über $y := x^d \bmod n = dec_{Bank}(x)$.
Alice präsentiert y dem Kaufmann.
Der Kaufmann verifiziert y durch $enc_{Bank}(y)$.
Der Kaufmann schickt y an die Bank.
Die Bank trägt $enc_{Bank}(y)$ in die Liste der verbrauchten Schecks ein.
Die Bank schreibt dem Kaufmann 1,- DM gut.
Der Kaufmann schickt die Ware an Alice.

Zertifizierungscenter :

Bob erzeugt selbst sein Schlüsselpaar enc_B und dec_B.
Bob besorgt sich persönlich von einem Zertifizierungscenter Z
dessen öffentlichen Schlüssel enc_Z und ein Zertifikat $z := dec_Z(enc_B)$.
Bob schickt z an Alice.
Alice bildet $enc_Z(z)$ und erhält somit enc_B.
Alice kann nun sicher sein, Bob's öffentlichen Schlüssel vor sich zu haben.