RECOVERY

"Concurrency Control and Recovery in Database Systems"

Bernstein, Hadzilacos, Goodman

Addison-Wesley

Kapitel 1, 6

Modell eines Datenbanksystems

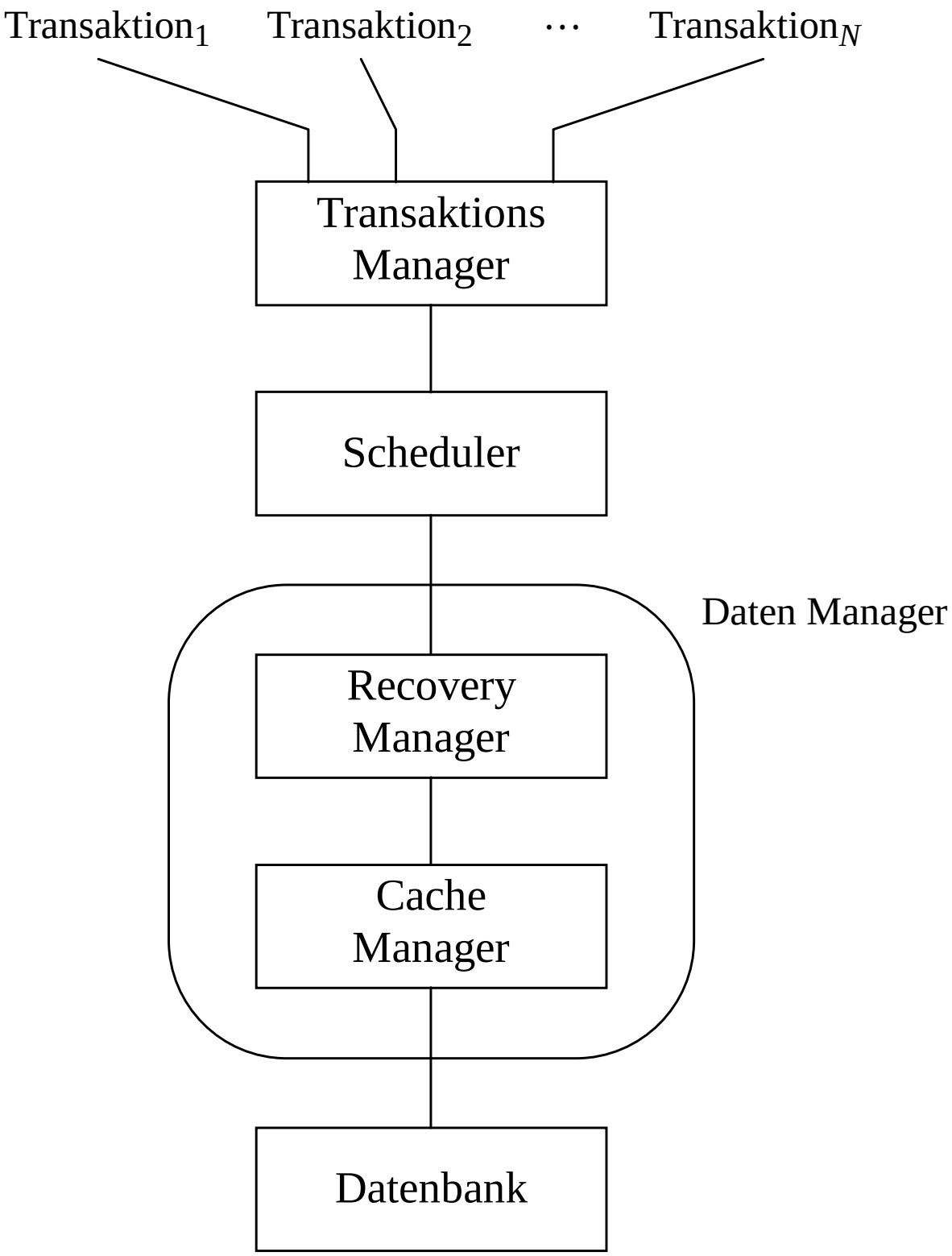
Ein Datenbanksystem besteht aus vier Modulen:

1. Dem **Transaktions-Manager**
   Er führt die Vorverarbeitung der Operationen aus, die von den Transaktionen abgesetzt werden.

2. Dem **Scheduler**
   Er kontrolliert die relative Ordnung zwischen Operationen auf der Datenbank.

3. Dem **Recovery Manager**
   Er ist verantwortlich für das Commitment oder den Abbruch von Transaktionen.

4. Dem **Cache Manager**
   Er operiert direkt auf dem Datenbestand.


Transaktion$_1$
Transaktion$_2$
⋯
Transaktion$_N$
Transaktions Manager
Scheduler
Daten Manager
Recovery Manager
Cache Manager
Datenbank

Der Cache Manager (CM)

In einem Computersystem sind normalerweise flüchtiger (volatile) und nicht-flüchtiger (stable) Speicher vorhanden.

- Flüchtiger Speicher ist schnell, aber teuer und deshalb nicht sehr groß.

- Nicht-flüchtiger Speicher ist langsam, aber billig und daher in großen Mengen verfügbar.

Durch die limitierte Größe des flüchtigen Speichers kann das DBS nur einen Teil der Daten im Speicher halten. Dieser Teil wird **Cache** genannt.

Die Verwaltung des Caches ist die Aufgabe des Cache Managers.

Zur Verwaltung des Caches stehen dem Cache Manager hauptsächlich die Operationen **Fetch(x)** und **Flush(x)** zur Verfügung.

- **Fetch(x)** lädt x (normalerweise eine ganze Page) aus dem nicht-flüchtigen Speicher (Platte) in den flüchtigen Speicher (RAM).

- **Flush(x)** schreibt eine Kopie von x aus dem RAM auf die Platte.

Es gibt Momente, an denen ein Fetch(x) nicht durchgeführt werden kann, da es keinen Platz im RAM mehr gibt. Um dieses Problem zu lösen, muß der Cache Manager Platz schaffen, indem er Flush(x)-Operationen ausführt.

Der Recovery Manager (RM)

Der Recovery Manager ist dafür verantwortlich, dass die Datenbank nur die Ergebnisse von committed Transaktionen beinhaltet und keine Ergebnisse von abgebrochenen Transaktionen.

Der Recovery Manager unterstützt die Operationen

```
START, COMMIT, ABORT, READ, WRITE
```

Diese Operationen werden ausgeführt indem der RM die Operationen Fetch(x) und Flush(x) von Cache Manager benutzt.

Der RM ist so entworfen, dass er resistent gegenüber Fehlern ist, bei denen der Inhalt des flüchtigen Speichers verloren geht. Solche Fehler werden Systemfehler genannt.

Nach einem solchen Systemfehler steht dem RM nur der Inhalt des nicht-flüchtigen Speichers zur Verfügung, um die Datenbank wieder in einen konsistenten Zustand zu bringen.

Die Hauptaufgabe des RM ist es daher, zu entscheiden, wann Daten aus dem RAM auf die Platte geschrieben werden.

Sonst kann es zu folgenden Problemen kommen:

1. Dem nicht-flüchtige Speicher fehlt ein Update einer committed Transaktion.

2. Der nicht-flüchtige Speicher enthält einen Wert der durch eine uncommitted Transaktion geschrieben wurde.

Der Recovery Manager kann auch dazu entworfen sein, resistent gegenüber Medien-Fehlern (z.B. Platten-Crash) zu sein.

Um dies zu erreichen, muß er redundante Kopien der Daten an zumindest zwei verschiedenen Orten halten.

Wegen der engen Zusammenarbeit von Recovery und Cache Manager werden oft beide zusammen genommen als Daten Manager (DM) bezeichnet.

Der Scheduler

Der Scheduler ist ein Programm, das die nebenläufige Abarbeitung mehrerer Transaktionen steuert.

Er führt diese Aufgabe aus, indem er die Reihenfolge, in der der DM die Operationen verschiedener Transaktionen ausführt, einschränkt.

Ziel des Ganzen sind serialisierbare, recoverable Ausführungen.

Er kann ebenfalls kaskadierte Abrüche verhindern, oder zusichern, dass die Ausführung strikt ist.

Nachdem der Scheduler eine Operation zur Bearbeitung bekommen hat, kann er eine von drei möglichen Aktionen ausführen:

1. **Ausführen** (Execute):
   Er gibt die Operationen an den DM weiter. Wenn der DM die Operation ausgeführt hat, informiert er den Scheduler. Wenn die Operation ein READ war, übergibt der DM den Wert.

2. **Abweisen** (Reject):
   Er kann die Operation abweisen, indem er dies der Transaktion mitteilt; die Transaktion wird dadurch abgebrochen.

3. **Verzögern** (Delay):
   Er kann die Operation auf einen späteren Zeitpunkt verschieben, indem er sie in eine Warteschlange (queue) aufnimmt.

Mit diesen drei Aktionen kann der Scheduler die Reihenfolge der Operationen verschiedener Transaktionen kontrollieren.

Beispiel:

```
Read1 (Accounts[13]);
Read2 (Accounts[13]);
Write2 (Accounts[13],101.000);
Commit2;
Write1 (Accounts[13],1.100);
Commit1;
```

Diese nicht serialisierbare Ausführung kann durch den Scheduler vermieden werden, indem er die Operation $Write_1$ zurückweist. In diesem Fall wird die Transaktion T1 abgebrochen.

Eine andere Möglichkeit wäre es, die Operation $Read_2$ zu verzögern, bis $Write_1$ abgearbeitet ist.

- Der Scheduler hat nur wenige Informationen zur Verfügung, um zu entscheiden, wann und ob eine Operation ausgeführt werden kann.

- Es liegen nur die Informationen vor, die von den Transaktionen geliefert werden. Insbesondere hat der Scheduler kein Wissen über Programmdetails oder Operationen, die in der Zukunft abgesetzt werden.

Der Transaktions Manager (TM)

- Transaktionen interagieren mit der Datenbank über den Transaktions Manager.
- Der TM empfängt die Datenbank- und Transaktions-Operationen die von den Transaktionen abgesetzt werden.
- Der TM leitet diese Operationen an den Scheduler weiter.
- Der TM ist verantwortlich für den Abbruch von Transaktionen.

Reihenfolge von Operationen

Zu jedem Zeitpunkt ist es jedem Modul möglich, jede an es übergebene noch nicht ausgeführte Operation auszuführen.

Auch wenn der RM die Operation *p* vor der Operation *q* bekommen hat, kann er *q* vor *p* ausführen.

Wenn ein Modul zwei Operationen in einer bestimmten Reihenfolge ausgeführt haben will, dann ist die Aufgabe dieses Moduls, dafür zu sorgen, dass diese Reihenfolge eingehalten wird.

Beispiel:

Wenn der Scheduler verlangt, dass *p* vor *q* ausgeführt wird, dann muß er zuerst *p* an den RM übergeben und warten bis der RM die Ausführung bestätigt (acknowledged). Dann kann er *q* an den RM übergeben.

Diese Abfolge

- Übergeben einer Operation
- auf Bestätigung warten
- Übergeben einer weiteren Operation

wird als **Handshake** bezeichnet.

Eine andere Möglichkeit ist die Benutzung eines **FIFO Queues**. Jedes Modul bekommt Operationen über eine Queue übertragen.

Nachteil dieses Verfahrens ist, dass eine Reihenfolge in der Abarbeitung erzwungen wird, auch wenn dies in einigen Fällen nicht notwendig ist.

Ein weiters Problem entsteht wenn mehr als zwei Module in die Verarbeitung von Operationen involviert sind.

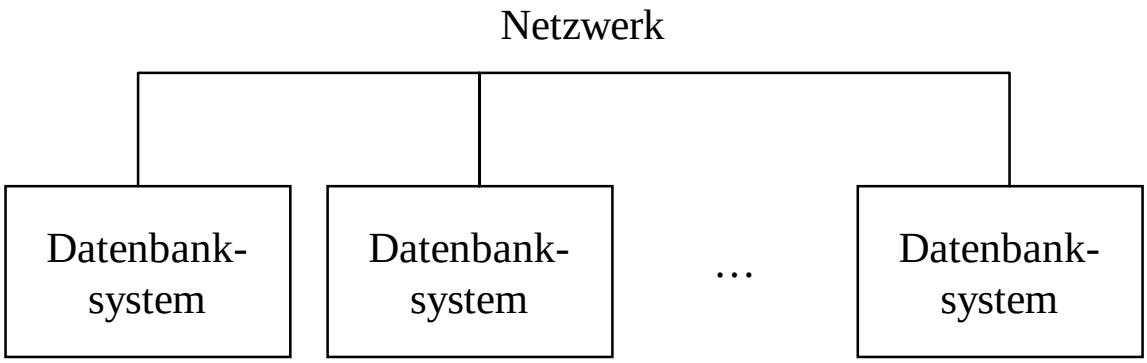
Wenn Beispielsweise zwei Module DM1 und DM2 die Funktion des Daten Managers übernehmen.

Angenommen der Scheduler will, dass DM1 die Operation *p* bearbeitet, bevor DM2 *q* bearbeitet. Der Scheduler kann diese Reihenfolge nur mittels Handshakes aufrechterhalten.

Wir nehmen im folgenden an, dass Handshaking benutzt wird, um die Reihenfolge von Transaktionen zu gewährleisten, es sei denn, es ist ausdrücklich anders definiert.

Verteilte Datenbanksysteme

Ein verteiltes Datenbanksystem ist ein Anzahl von Standorten die über ein Netzwerk miteinander verbunden sind.



Prozesse können miteinander Botschaften austauschen, ob sie an dem selben Standort laufen oder an verschiedenen Standorten.

- Jeder Standort ist ein vollständiges zentrales Datenbanksystem, dass einen Teil der Datenbank speichert.

- Jedes Objekt der Datenbank ist an genau einem Standort gespeichert.

- Jede Transaktion besteht aus einem oder mehreren Prozessen an einem oder mehreren Standorten.

- Wenn ein TM eine Anfrage erhält, die an diesem Standort nicht erfüllt werden kann, reicht er die Anfrage zu dem Scheduler eines anderen Standortes weiter.

- Jeder TM kann mit jedem Scheduler des verteilten Systems in Kontakt treten und ihm Operationen übergeben.

Fehler

Frage: Wie können Transaktionen fehlertolerant bearbeitet werden?

Welche Arten von Fehlern gibt es überhaupt?

- Transaktions-Fehler
- System-Fehler
- Medien-Fehler

- Ein **Transaktions-Fehler** tritt auf, wenn eine Transaktion abbricht.

- Ein **System-Fehler** bezieht sich auf den Verlust von Inhalten des flüchtigen Speichers.
  Im Falle von RAM wäre dies beispielsweise durch einen Stromfall möglich. Oder wenn das Betriebssystem einen Ausfall hat.

- Ein **Medien-Fehler** tritt auf, wenn Teile des nicht-flüchtigen (Fest-) Speichers zerstört werden, wenn beispielsweise Sektoren einer Festplatte beschädigt werden.

Lösung:

Wir betrachten einen Teil des Speichers als unzuverlässig. Den flüchtigen Speicher im Falle des System-Fehlers oder den Festspeicher im Falle des Medien-Fehlers.

Um dem Verlust von Daten vorzubeugen, wird eine Kopie der Daten gehalten. Diese redundante Kopie wird in einem anderen Speicher gehalten.

- Im Festspeicher im Falle des System-Fehlers.
- In einem anderen Festspeicher (zweite Platte) im Falle des Medien-Fehlers.

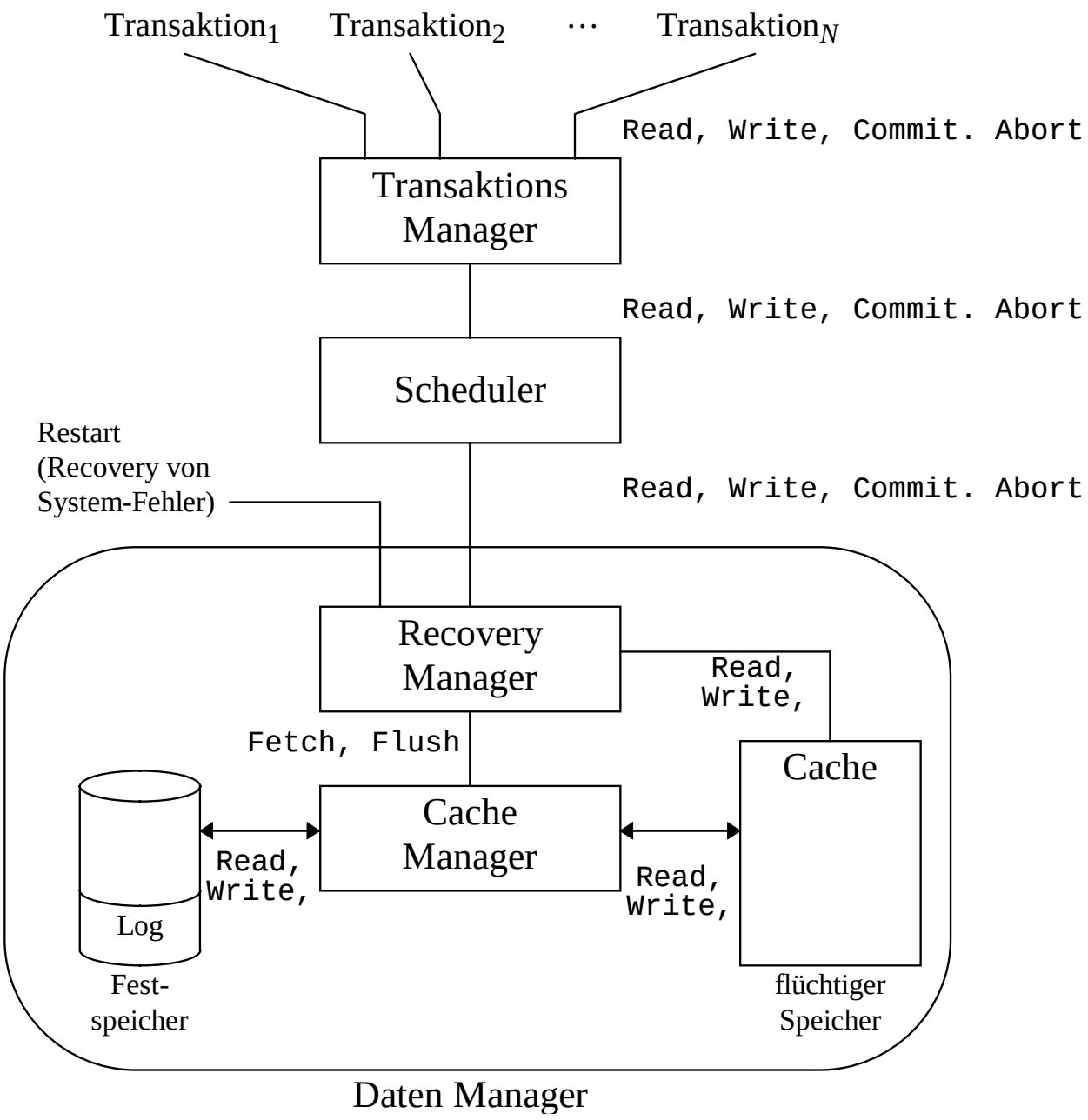
Daten Manager Architektur

Es ist der Daten Manager, der mit dem Speicher umgeht und der Hauptsächlich von Fehlern betroffen ist.

Zusätzlich zu den vorher besprochenen Operationen Read, Write, Abort und Commit (und Start) kann der DM auch auf eine Restart-Operation reagieren.

Diese Operation kommt von einem externen Modul, z.B. dem Betriebssystem, zum Zeitpunkt eines Recovery von einem System-Fehler.

Die Aufgabe des Restart ist es, die Datenbank in einen konsistenten Zustand zu bringen.


Transaktion$_1$
Transaktion$_2$
⋯
Transaktion$_N$
Read, Write, Commit. Abort
Transaktions Manager
Read, Write, Commit. Abort
Scheduler
Restart (Recovery von System-Fehler)
Read, Write, Commit. Abort
Recovery Manager
Read, Write,
Fetch, Flush
Cache
Cache Manager
Read, Write,
Read, Write,
Log
Fest-speicher
flüchtiger Speicher
Daten Manager

Der Festspeicher

Der Daten Manager wird in zwei Teile aufgeteilt, den Recovery Manager, der die Read, Write, Commit, Abort und Restart Operation verarbeitet und den Cache Manager, der den Speicher manipuliert.

- Der CM unterstützt die Operationen Fetch und Flush.

- Der RM kontrolliert teilweise die Flush-Operationen des CM, um sicherzustellen, dass die Daten, die notwendig sind einen Restart durchzuführen, immer im Festspeicher sind.

- Für Festplatten ist die Granularität des Speichers normalerweise ein Block (Seite) fester Größe.

Schreiben des Festspeichers ist ein atomarer Vorgang (d.h. die Seite wird entweder ganz oder gar nicht geschrieben).

Es wird unterschieden zwischen DM's die eine Kopie eines Objektes im Festspeicher halten und solchen mit mehreren Kopien.

- Eine Kopie im Speicher
  - → **in-place updating** (der alte Wert wird überschrieben).

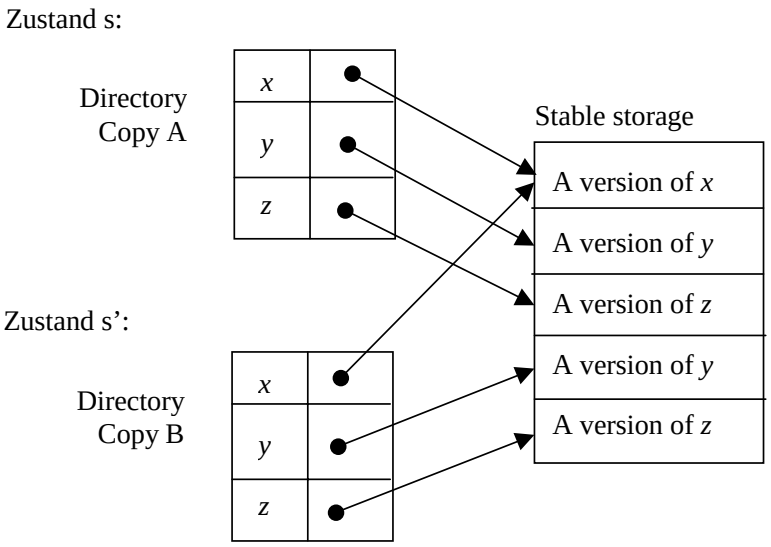
- N Kopien im Speicher
  - → **shadowing** (alte Versionen der Objekte bleiben erhalten).

Shadowing

Im Falle des Shadowing ändert sich die Abbildung von Daten-Objekten zu Speicherplätzen mit der Zeit.

→ Es wird ein Verzeichnis (Directory) benötigt.

Ein solches Verzeichnis definiert einen Stand der Datenbank.
Beim Schadowing gibt es üblicherweise mehrere solcher Verzeichnisse, von denen jedes einen anderen Stand der Datenbank definiert.



An Example of Shadowing
Using shadowing, directory copies A and B each define a database state.

Definition: feste Datenbasis (stable Database)

Als **feste Datenbasis** definiert man den **Zustand** der Datenbank **im Festspeicher**.

Beim in-place Updating gibt es genau einen festen Wert für jedes Objekt im Festspeicher und damit einen definierten Zustand der Datenbank.

Beim Schadowing wird ein spezielles Verzeichnis D angenommen, das den momentanen Zustand der festen Datenbasis enthält.

Der Cache Manager

Motivation für einen Cache Manager: Nicht jedes Read oder Write soll im (langsamen) Festspeicher ausgeführt werden.

→ Es wird eine Kopie der Datenbank im Hauptspeicher gehalten.

Da der Hauptspeicher begrenzt ist kann jeweils nur ein Teil der Datenbank im Speicher gehalten werden.

→ Caching (Buffering)

Caching

- Ein Teil des Hauptspeichers, der Cache, wird für Teile der Datenbank reserviert.

- Der Cache besteht aus einer Anzahl von Slots, die jeder ein Objekt der Datenbank aufnehmen können.

- Die Granularität der Daten-Objekte entspricht der der Einheiten, die atomar geschrieben werden können (Seiten).

- Ein Cache-Slot enthält den Wert eines Objektes und ein sogenanntes **dirty**-**bit**, das gesetzt wird, wenn der Wert des Objektes im Cache sich von dem Wert im Festspeicher unterscheidet.
  Ein Slot mit gesetztem dirty-bit ist **dirty**.

- Es gibt ein Cache Directory, das den Namen jedes Objektes im Cache enthält, sowie den jeweils zugewiesenen Slot.

Cache

Slot Number	*Dirty Bit*	*Data Item Value*
1	1	“October12”
2	0	3.1416
	.	
	.	
	.	

Cache Directory

Data Item Name	*Slot Number*
x	2
y	1
.	
.	
.	

Cache Structure

- Der Transfer von Objekten in und aus dem Cache wird vom Cache Manager durch die Operationen Fetch und Flush ermöglicht.

FLUSH (Cache Slot *c*):

```
if c not dirty then /* do nothing */
else
   copy c-value into stable storage;
   clear c's dirty-bit;
```

FETCH (Data Item Name *x*):

1. Wähle einen leeren Cache Slot, *c;*
   falls alle besetzt → Flush(c).
2. Kopiere den Wert von *x* nach *c*.
3. Lösche das dirty-bit von *c.*
4. Verändere das Cache-Directory
   (*c* enthält *x*).

Falls der Slot *c* in Schritt 1 besetzt war, hat *x* das Objekt in *c* **ersetzt**.

Ersetzungsstrategien:

LRU (Least Recently Used)

FIFO (First In, First Out)

READ (*x*):

- fetch(*x*)

  -- falls *x* noch nicht im Cache ist --

- gebe den Wert von *x* zurück.

WRITE (*x*):

- belege einen Slot im Cache.
- schreibe den neuen Wert in den Cache.
- setzte das dirty-bit.

Wann wird der Wert eines Slots mit gesetztem dirty-bit geflushed?

→ dafür gibt es verschiedene Strategien (RM).

Der RM kontrolliert das Flushen von Slots über die Operationen Pin(*c*) und Unpin(*c*).

- Pin(*c*) : *c* darf nicht geflushed werden.
- Unpin(*c*) : macht vorheriges Pin(*c*) rückgängig.

Der Recovery Manager

Die Schnittstelle des RM ist definiert durch 5 Prozeduren:

1. Read(T_i,x)
2. Write (T_i,x,v)
3. Commit(T_i)
4. Abort(T_i)
5. Restart

Der RM sollte diese Operationen **atomar** ausführen, also äquivalent zu einer seriellen Ausführung.

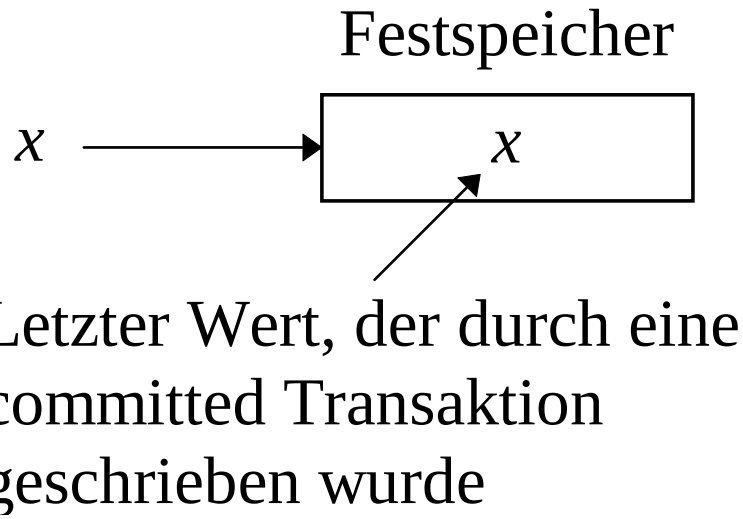
→ 2-Phasen-Sperren, Zeitstempel (Scheduler)

Strikte Ausführung

Logging

- Annahme: RM benutzt in-place Updating.

- Jedes Daten-Objekt hat einen eigenen Platz im Festspeicher.
  Idealerweise ist im Festspeicher für jedes Objekt *x* der letzte Wert der durch eine committed Transaktion geschrieben wurde gespeichert.



In der Praxis wird der Idealfall durch zwei Faktoren verhindert:

- Transaktionen benötigen einige Zeit um zu terminieren.

- Buffering von Werten im Cache.

Daher befinden sich im Festspeicher Werte, die von uncommitted Transaktionen geschrieben worden, und es fehlen Werte, die durch committed Transaktionen geschrieben wurden.

Im Falle eines Fehlers muß der RM beim Restart in der Lage sein, die Datenbank in einen konsistenten, committed Zustand zu bringen.

Dafür stehen dem RM nur die Informationen im Festspeicher zur Verfügung.

Aus diesem Grund speichert der RM zusätzlich zur Datenbank Informationen im Festspeicher.

→ Log-Datei

Die Log-Datei ist eine Historie von ausgeführten Operationen. Sie besteht aus Einträgen der Form $[T_i, x, v]$, mit der Bedeutung, dass die Transaktion T_i den Wert v in das Objekt x geschrieben hat.

Um beim Restart die Datenbank in eine Zustand zu bringen, bei dem jedes Objekt *x* den Wert einer committed Transaktion enthält, muß die Reihenfolge der Write-Operationen im Log gehalten werden.

→ seqentielle Datei

- Zusätzlich zur Datenbank und dem Log, kann der RM *aktiv*-, *commit*- und *abort*-Listen im Festspeicher halten.

- Diese Listen enthalten Identifier für jede Transaktion, die aktiv, committed oder aborted ist.

- In vielen Implementierung ist es die Aktion des Eintragen einer Transaktion in die *commit*-Liste, die eine Transaktion in committed Zustand überführt.

Undo und Redo

Zu manchen Zeiten ist es für den Recovery Manager notwendig, darauf zu bestehen, dass eine Seite geflushed wird.

Die verschiedenen Strategien, wann und wie dies zu tun ist, führen zu einer Kategorisierung der RM Algorithmen.

- Der RM *benötigt ein Undo,* wenn er es nicht-committed Transaktionen erlaubt, Werte in die stable Database zu schreiben.

  System-Fehler → Undo (Restart)

- Der RM *benötigt ein Redo,* wenn er Transaktionen erlaubt, zu committen, bevor alle Werte, die sie geschrieben haben, in die stable Database aufgenommen wurden.

  System-Fehler → Redo (Restart)

Indem der RM den Zeitpunkt eines Commits einer Transaktion, in Relation zum Schreiben der Werte in die stable Database bestimmt, kann er kontrollieren, ob er Redo oder Undo benötigt.

Dies führt zur folgenden Kategorisierung der RM Algorithmen:

1. Undo / Redo
2. Undo / No Redo
3. Redo / No Undo
4. No Undo / No Redo

Um sicherzustellen, dass im Festspeicher immer die Informationen verfügbar sind, die von Undo und/oder Redo benötigt werden, müssen alle Implementierungen von RMs bestimmten Regeln folgen:

Undo Regel: (Write Ahead Log Protocol)

Falls der Speicherplatz des Objektes *x* in der stable Database momentan den letzten committed Wert von *x* enthält, dann muß dieser Wert im Festspeicher (Log) gesichert werden, bevor er durch einen uncommitted Wert überschrieben wird.

Write Ahead Log Protocol:
Das Before-Image eines Write muß geloggt werden, bevor (**logged ahead**) das Write in die Datenbasis eingebracht wird.

Redo Regel:

Bevor eine Transaktion committen kann, müssen alle Werte, die sie geschrieben hat, im Festspeicher gesichert sein (Log oder stable Database).

Die Undo und Redo Regeln

→ sichern zu, dass der jeweils letzte, committed Wert eines Objektes *x* im Festspeicher verfügbar ist.

- Die Undo Regel sichert zu, dass der letzte committed Wert von *x* im Festspeicher gesichert wird, bevor er überschrieben wird.

- Die Redo Regel sichert zu, dass ein Wert in dem Moment im Festspeicher gesichert ist, indem er committed wird.

Garbage Collection

- Obwohl der Sekundärspeicher im Vergleich zum Hauptspeicher groß ist, ist er doch limitiert.

- Der Platz, den die stable Database einnimmt, ist natürlich durch die Anzahl der Datenobjekte begrenzt.

- Um den Undo und Redo Regeln gerecht zu werden, muß der RM Informationen in der Log-Datei speichern.

- Um das Wachstum der Log-Datei(en) zu beschränken, muß der RM Speicher freigeben, von dem er sicher ist, dass er nicht mehr benötigt wird.
  → **Garbage Collection.**

Anforderungen an die Log-Datei durch Restart:

Für jedes Datenobjekt *x* gilt, wenn die stable Database nicht den letzten committed Wert von *x* enthält, muß die Restart-Prozedur in der Lage sein, den Wert im Log zu finden.

Garbage Collection Regel:

Ein Eintrag $[T_i, x, v]$ kann aus dem Log entfernt (removed) werden, wenn

1. T_i abgebrochen wurde, oder

2. T_i committed hat, aber eine andere committed Transaktion einen Wert nach *x* geschrieben hat, nachdem T_i dies tat.
   (v also nicht der letzte committed Wert von *x* ist)

Anmerkung:

Selbst wenn v der letzte committed Wert von *x* ist und sicher in der stable Database gespeichert ist, kann der Eintrag $[T_i, x, v]$ nicht aus dem Log entfernt werden.

Wenn eine Transaktion T_j nach T_i einen Wert für *x* in die Datenbank geschrieben hat, und danach abbricht, wird der Eintrag benötigt, um T_js Write rückgängig zu machen (Undo).

Wenn der RM allerdings kein Undo benötigt, dann kann T_js Wert für *x* nicht in der stable Database eingetragen sein, bevor T_j committed ist. Daher kann die Garbage Collection Regel um einen dritten Fall erweitert werden:

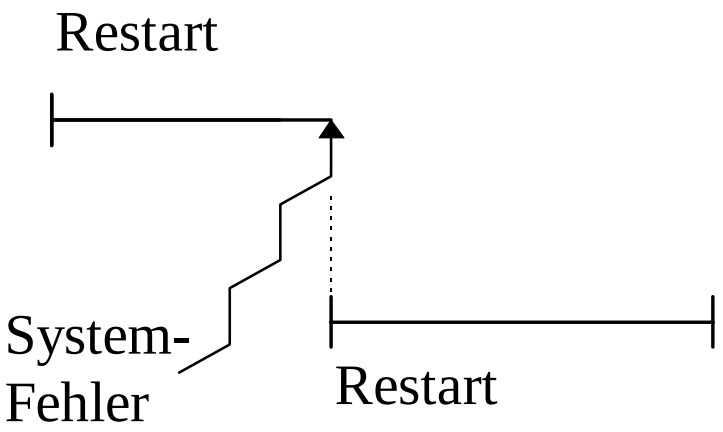
Garbage Collection Regel (Fortsetzung)

3. $[T_i, x, v]$ kann aus dem Log entfernt werden, wenn der RM kein Undo benötigt, v der letzte committed Wert von *x* ist, v der Wert von *x* in der stable Database ist und $[T_i, x, v]$ der einzige Eintrag im Log für *x* ist.

Diese letzte Bedingung sichert zu, dass die Restart Prozedur keinen vorherigen Eintrag im Log als den letzten committed Wert von *x* interpretiert, wenn $[T_i, x, v]$ entfernt wird.

Fehlertoleranz (Idempotenz) des Restart

Obwohl die Operationen Read, Write, Commit und Abort als atomar definiert sind, können sie durch einen System-Fehler und den darauf folgenden Restart unterbrochen werden, tatsächlich kann sogar der Restart dadurch unterbrochen werden.



→ Der Restart muß idempotent sein !

Wenn die Restart Prozedur irgendwann unterbrochen wird und wieder neu startet, muß das Ergebnis das gleiche sein, als wenn der Restart schon beim erstenmal bis zum Ende gelaufen wäre.

Der Undo/Redo Algorithmus

Vorteile:

- Ein Recovery Manager, der Undo/Redo benutzt, vermeidet es, den Cache Manager unnötig oft zu Flush() Operationen zu zwingen.
  → dies minimiert I/O.

- Ein No-Redo Algorithmus dagegen flushed häufiger, da er sicherstellen muss, dass alle, von einer Transaktion veränderten, Werte in der stable Database sind, bevor die Transaktion committed.

- Ein Recovery Manager, der Undo/Redo benutzt, erlaubt es einem Cache Manager, der In-Place Updating durchführt, einen Slot zu ersetzen (zu flushen), der zuletzt von einer uncommitted Transaktion geschrieben wurde.

- Ein No-Undo Algorithmus kann in diesem Fall den Slot nicht flushen, da er ein uncommitted Update in der stable Database verankern würde.

Generell:

- Der Undo/Redo Algorithmus maximiert die Effizienz im normalen Betrieb.

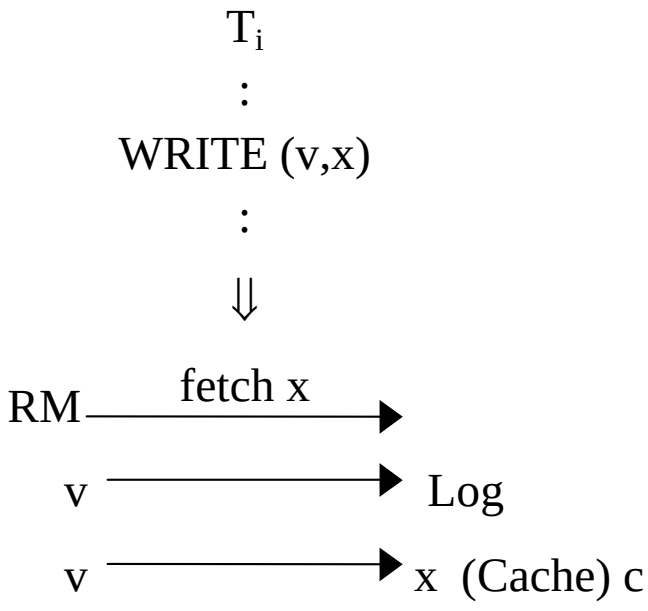
- Er ist dafür weniger effizient bei der Verarbeitung von Fehlern und dem damit verbundenen Restart.

Der Undo/Redo Algorithmus ist gegenüber den anderen möglichen RM-Algorithmen

- am flexibelsten, was den Zeitpunkt des Flushens eines dirty Slots betrifft.
  Leider ist er auch der komplizierteste der Algorithmen.

- Diese Entscheidung wird fast völlig dem Cache Manager überlassen.

Informell:



- Der RM verlangt vom CM nicht, den Slot c zu flushen

- CM flushed c, wenn der Platz benötigt wird.

- **Falls** der CM Slot c flushed, **und** die Transaktion T_i wegen eines Systemfehlers abbricht (vor dem Commit), **dann** ist ein **Undo** notwendig.

- **Falls** die Transaktion T_i committed hat **und** ein Systemfehler auftritt **bevor** der CM den Slot c geflushed hat, **dann** ist ein **Redo** notwendig.

Annahmen:

- Die Granularität der Datenobjekte ist die gleiche, wie für den atomaren Transfer zum Festspeicher (also eine Seite).

- Das Log besteht aus Einträgen der Form $[T_i, x, v]$.

- Für das Log wird die Garbage Collection Regel angewandt, um Speicher zu sparen.

- Der initiale Wert jedes Datenobjektes wird ins Log geschrieben, bevor irgendeine Transaktion bearbeitet wird.

- Jedes Update des Log oder der Commit-Liste geht direkt in den Festspeicher und muß bestätigt sein, bevor mit dem nächsten Schritt fortgefahren wird.

Undo/Redo Algorithmus

Die fünf RM-Prozeduren:

RM-WRITE(t_i,x,v):

1. Füge T_i zur Aktiv-Liste hinzu, falls sie noch nicht dort ist.

2. Wenn *x* noch nicht im Cache ist, fetche *x*.

3. Füge $[T_i, x, v]$ in die Log-Datei ein.

4. Schreibe *v* in den Cache Slot, der von *x* belegt ist.

5. Bestätige (Acknowledge) dem Scheduler die Verarbeitung des RM-WRITE.

Anmerkung zu Schritt 4:

Hier und in allen weiteren Fällen gilt, dass, wenn ein Wert in einen Cache Slot geschrieben wird, das dirty-bit gesetzt wird.

Außerdem nehmen wir an, dass der RM einen Cache Slot pint, bevor er ihn schreibt oder liest und danach wieder unpint.

RM-READ(T_i,x):

1. Wenn *x* nicht im Cache ist, fetche *x.*

2. Gib den Wert von *x* an den Scheduler.

RM-COMMIT(T_i):

1. Füge T_i der Commit-Liste hinzu.

2. Bestätige dem Scheduler das Commitment von T_i.

3. Lösche T_i aus der Aktiv-Liste.

Anmerkung zu Schritt 1:

Dies ist der Moment in dem T_i committed, wenn ein Systemfehler auftritt, bevor dieser Schritt beendet ist, ist T_i uncommitted. Sonst ist T_i committed, selbst wenn die Schritte 2 und 3 nicht beendet werden.

RM-ABORT(T_i):

1. Für jedes Datenobjekt *x*, das durch T_i verändert wurde
   - Wenn *x* nicht im Cache ist, belege einen Slot für *x.*
   - Schreibe das Before-Image von *x,* in Bezug auf T_i , in den Slot von *x*.

   Anmerkung: Der Wert ist jetzt nur im Cache wiederhergestellt, Der CM schreibt diesen Wert erst beim Flushen in die stable Database.

1. Füge T_i der Abort-Liste hinzu.

2. Bestätige dem Scheduler die Durchführung des RM-ABORT.

3. Lösche T_i aus der Aktiv-Liste.

RESTART:

1. Lösche alle Cache Slots.
    - Bei einem Systemfehler ist der Hauptspeicher betroffen, Werten im Cache kann also nicht vertraut werden.

2. Setzte redone = {}
   undone = {}
    - Dies sind lokale Variablen der Restart-Prozedur, sie halten fest, für welche Datenobjekte der letzte committed Wert wiederhergestellt wurde (durch Undo oder Redo).

3. Beginne mit dem letzten Eintrag im Log und gehe rückwärts bis zum Anfang des Log.

   Wiederhole die folgenden Schritte bis entweder
   redone $\cup$ undone = {alle Datenobjekte}
   oder es keine Einträge mehr im Log gibt.

   Für alle Einträge $[T_i, x, v]$ im Log,
   falls $x \notin$ redone $\cup$ undone, dann

   - Falls *x* nicht im Cache ist, fetche *x.*
   - wenn T_i in der Commit-Liste ist, kopiere *v* in den Slot von *x* und setzte redone = redone $\cup$ { *x* }.
     - Der letzte committed Wert von *x* ist jetzt nur im Cache wiederhergestellt.
   - sonst (T_i in Abort- oder Aktiv-Liste aber nicht in der Commit-Liste) kopiere das Before-Image von *x* bezüglich T_i (im log), in den Slot von *x* und setzte
     undone = undone $\cup$ { *x* }.

4. Entferne jede Transaktion T_i die sowohl in der Commit-Liste als auch in der Aktiv-Liste ist, aus der Aktiv-Liste.
5. Bestätige dem Scheduler die Durchführung des RESTART.
   (Der Scheduler wartet, bis der Restart beendet ist)

Der Undo/Redo Algorithmus befolgt

- die Undo Regel
- die Redo Regel
- und der Restart ist Idempotent

Undo und Redo Regeln

Undo Regel

Annahme: Der Platz von *x* im Festspeicher enthält den letzten committed Wert *v* von *x,* geschrieben durch die Transaktion T_i.

- Zum Zeitpunkt als T_i *x* geschrieben hat, hat der RM den Eintrag $[T_i,x,v]$ in das Log eingefügt.

- Wegen der Garbage Collection Regel gilt, dass dieser Eintrag (*v* ist der letzte committed Wert von *x*) nicht aus dem Log entfernt worden sein kann.

→ Der Eintrag ist also immer noch im Log, wenn der CM *v* in der stable Database überschreibt.

Redo Regel:

- Alle Updates einer Transaktion werden im Log gespeichert, bevor die Transaktion committed, unabhängig davon, ob sie auch in der stable Database eingetragen sind oder nicht.

- Wegen der Gabage Collection Regel sind sie immer noch im Log, wenn die Transaktion committed.

Der Algorithmus genügt beiden Regeln

→ Die Restart Prozedur findet im Festspeicher immer die Informationen, um den letzten committed Wert jedes Datenobjektes in der stable Database wiederherzustellen.

In Schritt 3 der Restart Prozedur

- wird ein **Redo** eines Update einer committed Transaktion auf ein Objekt *x* durchgeführt, oder

- ein **Undo** eines Update einer uncommitted Transaktion auf ein Objekt *x* durchgeführt.

- allerdings nur dann, wenn nicht eine andere committed Transaktion danach ein Update auf *x* durchgeführt hat.

→ Also enthält, nachdem die Restart Prozedur beendet ist, jedes Datenobjekt seinen letzten committed Wert.

Der Restart ist Idempotent:

Wenn der Restart einmal durch einen Systemfehler unterbrochen wird und wieder neu (von Beginn an) gestartet wird,

- dann sind die Updates die in Schritt 3 Redone oder Undone werden, die gleichen, die vom Restart gemacht worden wären, wenn er nicht unterbrochen worden wäre.

Checkpointing

Die Restart Prozedur muß unter Umständen alle Sätze im Log lesen, dies ist aus zwei Gründen ineffizient.

1. Das Log kann sehr groß sein, da die Garbage Collection eine teure (d.h. langsame) Operation ist, und daher selten ausgeführt wird.

2. Die meisten Datenobjekte in der stable Database enthalten zum Zeitpunkt der Systemfehlers wahrscheinlich schon ihren letzten committed Wert. Die Restart Prozedur macht also mehr als eigentlich notwendig.

Technik des Checkpointig

Das Checkpointing ist eine Methode, die Informationen während der normalen Operation in das Log schreibt, um die Arbeit für den Restart zu reduzieren.

1. Setzte eine Markierung im Log, in der Commit-Liste und in der Abort-Liste, um anzuzeigen, welche Updates bereits in die stable Database geschrieben oder rückgängig gemacht wurden.

2. Schreibe die After-Images von committed Updates oder die Before-Images von abgebrochenen Updates in die stable Database.

- Technik 1 gibt dem Restart Informationen darüber, welche Updates nicht noch einmal bearbeitet (Undo oder Redo) werden müssen.

- Technik 2 reduziert die Arbeit, die von der Restart Prozedur ausgeführt werden muß, indem dies zum Zeitpunkt des Checkpointing erledigt wird.

- Technik 1 ist notwendig für jedem Form des Checkpointing, Technik 2 ist optional.

Commit Consistent Checkpointing

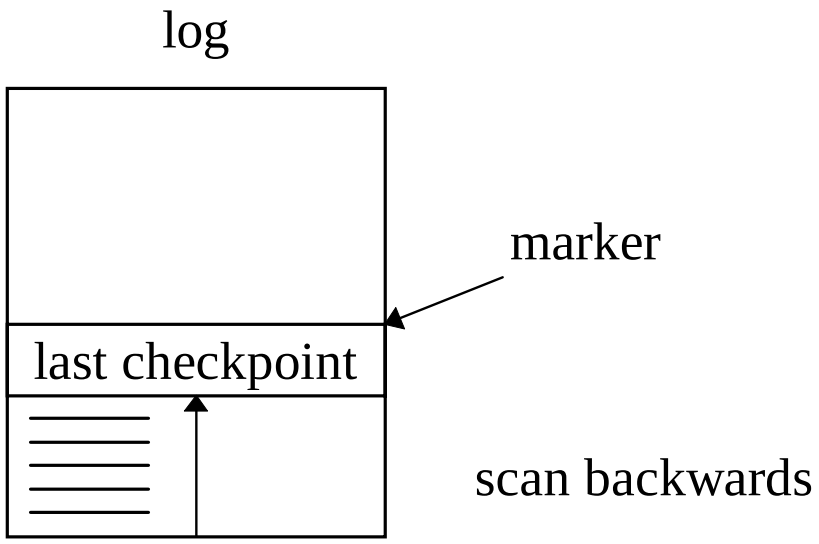
Ein sehr einfaches Verfahren des Checkpointing ist es,

- Periodisch jede Verarbeitung von Transaktionen zu stoppen,
- zu warten bis, alle Transaktionen T_i entweder committed sind oder abgebrochen wurden,
- alle dirty Cache Slots zu flushen und
- das Ende des Log zu markieren, um anzuzeigen, dass ein Checkpointing zu diesem Zeitpunkt stattgefunden hat.

→ Die stable Database enthält nun die letzten committed Werte der Datenobjekte bezüglich der Transaktionen T_i, deren Aktivitäten im Log aufgezeichnet ist.

- Die Restart Prozedur muß jetzt das Log vom Ende an rückwärts scannen bis zum letzten Checkpoint.

- Es muß weiterhin nach Daten hinter dem Checkpoint gesucht werden, im speziellen nach den Before-Images von uncommitted Transaktionen.



Das Hauptproblem dieser Form des Checkpointing ist Performanz.

Benutzer müssen unter Umständen sehr lange warten, bis alle Transaktion committed sind und der Cache geflushed ist (und damit das Checkpointing durchgeführt werden kann).

Cache Consistent Checkpointing

- Sichert zu, dass alle Writes, die in den Cache geschrieben wurden, auch in die stable Database geschrieben werden.

- Periodisch wird die Verarbeitung aller Transaktionen gestoppt, wobei aktive Transaktionen angehalten (blocked) werden.

- Flushe alle dirty Cache Slots.

- Markiere das Ende des Log, um anzuzeigen, dass ein Checkpointing zu diesem Zeitpunkt stattgefunden hat.

Bei dieser Form des Checkpointing muß nicht darauf gewartet werde, dass die Transaktionen committen.

Aufgaben des Restart nach einem Systemfehler:

- Alle Updates von committed Transaktionen, die vor dem letzten Checkpoint stattfanden, sind in der stable Database.
  → Kein Redo (für diese Updates)

→ Der Restart muß nur die Updates von committed Transaktionen wiederherstellen, die im Log nach dem letzten Checkpoint auftreten.

- Alle Updates von Transaktionen, die vor dem letzten Checkpoint abgebrochen wurden, wurden während des Checkpointing rückgängig gemacht.
  → Kein Undo (für diese Updates)

→ Der Restart muß nur die Updates von den Transaktionen rückgängig machen, die in der Aktiv- aber nicht in der Commit-Liste sind, oder in der Abort-Liste sind und nach dem letzten Checkpoint-Marker in dieser Liste stehen.

Medienfehler

Problem: Verlust von Daten im Sekundärspeicher, d.h. der stable Database oder dem Log.

Ziel: wiederherstellen des letzten committed Wertes aller Datenobjekte.

- Im Gegensatz zum Recovery von Systemfehlern, ist der Inhalt des Sekundärspeichers von dem Ausfall betroffen.

→ **Redundante Kopien** des letzten committed Wertes jeden Datenobjektes.

- Die Fähigkeit, Medienfehler zu überstehen steigt mit der **Anzahl der Kopien**.

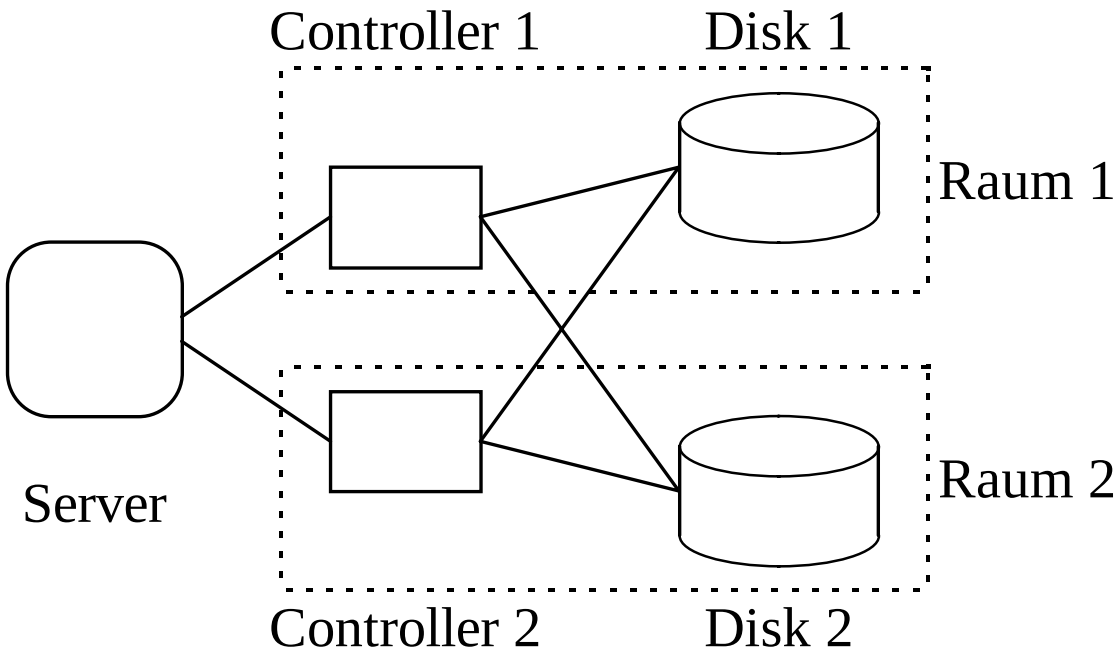
- Die Kopien auf verschiedenen, voneinander unabhängigen Laufwerken zu halten ist besser.

- Sind die Laufwerke mit mit unabhängen Controllern versehen, erhöht dies die Toleranz gegenüber Controller-Fehlern.

- Sind die Laufwerke in verschiedenen Räumen (Gebäuden) untergebracht, erhöht dies die Toleranz gegenüber Feuer, Überschwemmung, etc.

- In der Praxis reichen zwei Kopien auf verschiedenen Laufwerken meist aus.

Beispiel einer Konfiguration:

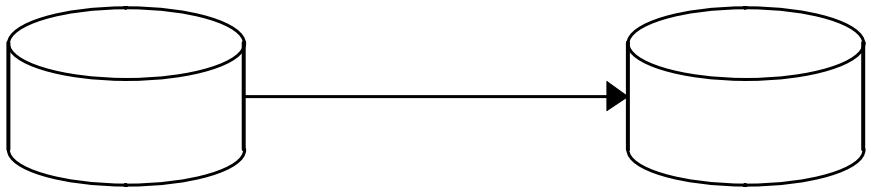


Zwei mögliche Ansätze:

- Spiegeln (Mirroring)
- Archivierung (Archiving)

Spiegeln (Mirroring)

Mirroring: Eine Kopie jeder (Fest-)Platte online, in Form einer zweiten Platte.



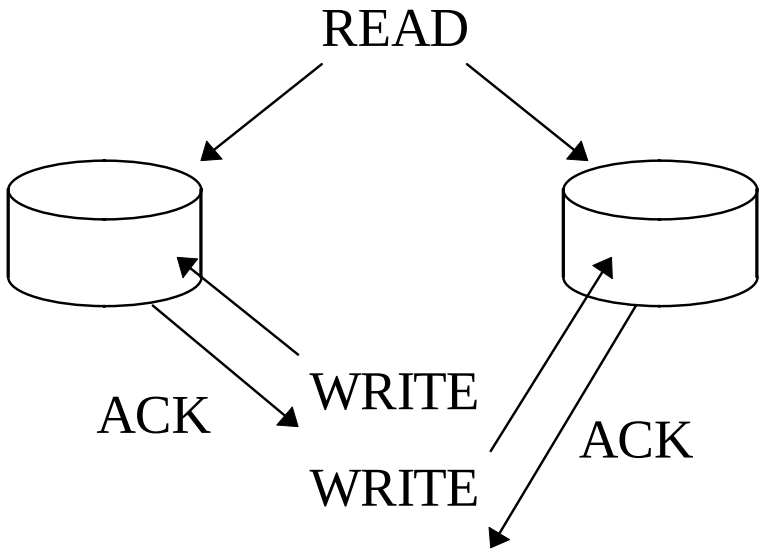
- Jede WRITE-Operation muß nun auf beiden Platten durchgeführt werden.

- Um sicherzustellen, dass beide Kopien identisch sind, muß jedes WRITE in der gleichen Reihenfolge durchgeführt werden.

    - Schreibe zuerst auf eine Platte, warte auf die Bestätigung

    - Schreibe dann auf die andere Platte

- READ-Operationen können zwischen den Platten aufgeteilt werden.



- Wenn eine Festplatte ausfällt, werden alle Operationen vom Mirror übernommen.

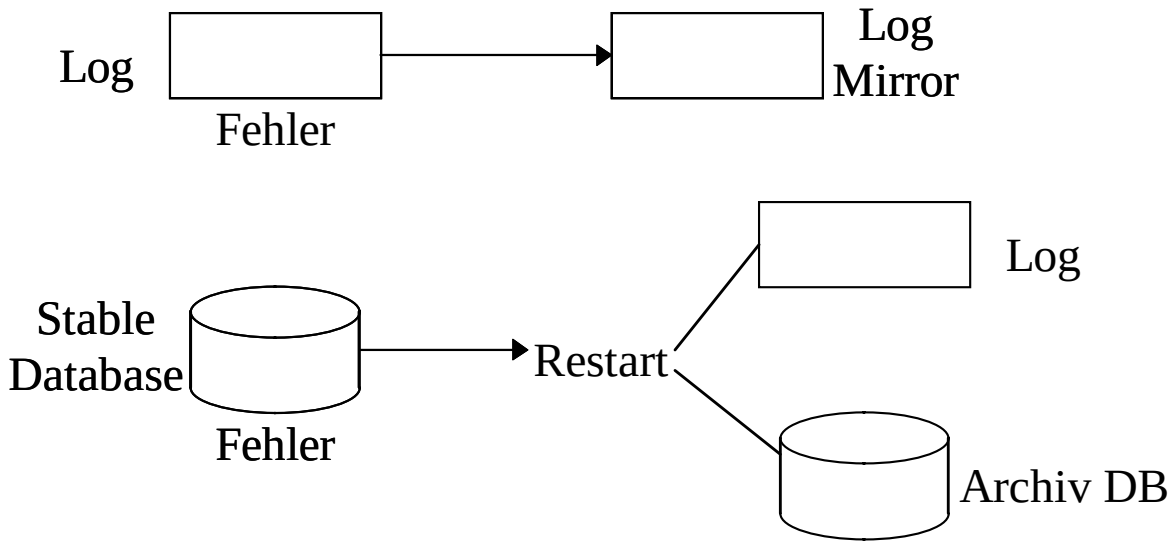
- Wenn die ausgefallene Platte ersetzt oder wiederhergestellt wird, muß sie auf den neuesten Stand (up-to-date) gebracht werden. Die geschieht durch Kopieren des Inhalts.

Archivierung (Archiving)

- Beim Archiving wird periodisch der Wert jedes Datenobjekts in eine Archiv-Datenbank geschrieben (dumped).

- Das Log enthält alle Updates, die seit dem letzten Dump gemacht wurden.

- Da das Log hier eine wichtige Rolle spielt wird es zur Sicherheit gespiegelt.



- Archivieren und Checkpointig sind ähnliche Aktivitäten. Das Archiving wird deshalb auch als **Archiv Checkpointing** bezeichnet.

- Insbesondere können die veschiedenen Checkpointing Techniken auch beim Archiving verwendung finden.
  → Commit Consistent, Cache Consistent

- Sehr häufig werden Archiving und Checkpointing zum gleichen Zeitpunkt durchgeführt.

- Archiving ist sehr teuer (langsam). Zusätzlich zu den Problemen, die Checkpointing mit sich bringt, müssen beim Archiving sehr viele Daten geschrieben werden. Und viele dieser Daten müssen zuvor aus der stable Database gelesen werden.

- Medienfehler betreffen oft nur einen Teil Datenbank (einige Zylinder eine Festplatte, oder eine aus einer Gruppe von Festplatten).

  → Restore nur auf einer bestimmten Menge von Datenobjekten.

- Um die Komplexität des RM zu reduzieren, ist es nützlich, Archiving und Checkpointing so zu entwerfen, dass dieselbe Restart-Prozedur sowohl für Systemfehler als auch für Medienfehler benutzt werden kann.