Kapitel 12: Transaktionsverwaltung – Recovery –

12.1 Prinzipien der Recovery in DBS

Fehlerkategorien (die die Atomarität oder Persistenz von Transaktionen gefährden)**:**

1) Fehler im Anwendungsprogramm
2) Fehler in der Systemsoftware (BS oder DBS),
   die zum "Systemabsturz" (engl.: system failure, (soft) crash) führen
   - "Bohrbugs": deterministisch und reproduzierbar;
     sind in gut ausgetester Systemsoftware so gut wie eliminiert
   - "Heisenbugs": schwer einzugrenzen und praktisch nicht reproduzierbar;
     treten vor allem in Überlastsituationen im Mehrbenutzerbetrieb auf
3) Stromausfall
4) Transienter Hardwarefehler (CPU, Speicher)
5) Fehlbedienung (Systemadministrator, Operateur, Techniker)
6) Plattenfehler (Media−Fehler),
   der zum Verlust von permanent gespeicherten Daten führt
   (engl.: disk failure, hard crash)
7) Katastrophe (Feuer, Erdbeben, etc.)

Fehlerbehandlung durch das DBS:

1) Unvollendete Transaktionen von fehlerhaft beendeten Programmen werden zurückgesetzt.
2) − 5)
   Das DBS wird im *Warmstart* neu gestartet und führt dabei eine *Crash−Recovery* durch.
   Dabei werden alle vor dem Absturz unvollendeten Transaktionen zurückgesetzt
   (*Undo* von Transaktionen).
   Änderungen von beendeten Transaktionen werden nötigenfalls DBS−intern wiederholt
   (*Redo* von Transaktionen).
   Bemerkungen:
   - Transaktionen, für die die Recovery ein Undo durchgeführt hat, müssen vom Anwendungsprogramm oder sogar vom Endbenutzer neu gestartet werden.
   - Das Prinzip einer "Backward Recovery" (Undo mit anschließender Fortsetzung der Verarbeitung) hat sich gegenüber einer "Forward Recovery" (z.B. instruktionssynchrone Verarbeitung auf einem zweiten Rechner, der im Fehlerfall die Verarbeitung fortsetzt) vor allem im Hinblick auf "Heisenbugs" sehr bewährt.
   - Die Fehlerkategorie 3 kann durch ununterbrechbare Stromversorgung eliminiert werden.
6) Wiederherstellung der verlorenen Daten durch Aufsetzen auf einer Archivkopie der Datenbank und Wiederholung der Änderungen abgeschlossener Transaktionen (*Media−Recovery, Archiv−Recovery*).
7) Änderungen abgeschlossener Transaktionen müssen auf einem zweiten, genügend weit entfernten Rechner doppelt erfaßt werden; im Katastrophenfall übernimmt der zweite Rechner die Verarbeitung.

Arbeitsprinzipien der Recovery−Komponente:

- Um im Fehlerfall Undo und Redo durchführen zu können, müssen Änderungen auf einem hinreichend stabilen Speichermedium (für die Crash−Recovery auf Platte, für die Media−Recovery auf mehreren Platten und/oder Band) protokolliert werden. Für jede Änderung gibt es eine *Undo−Information*, um die Änderung rückgängig machen zu können, und eine *Redo−Information*, um die Änderung wiederholen zu können.

- *Atomaritätsprinzip:*
  Vor einer Veränderung der permanenten Datenbank (durch Zurückschreiben geänderter Seiten aus dem Puffer auf die Platte) muß die entsprechende Undo−Information auf stabilen Speicher geschrieben werden.

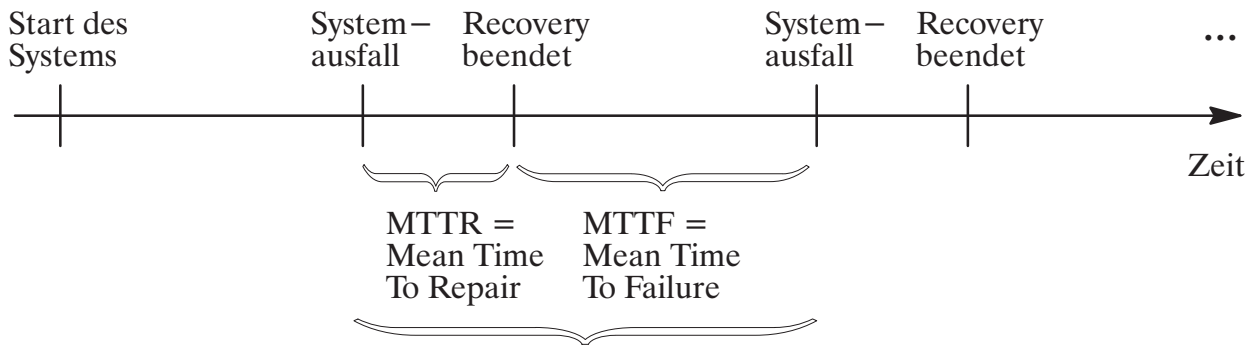
- *Persistenzprinzip:*
  Vor dem Commit einer Transaktion muß die entsprechende Redo−Information auf stabilen Speicher geschrieben werden.

- *Idempotenzprinzip:*
  Da es während eines Warmstarts zu einem erneuten Ausfall kommen kann, müssen Recovery−Maßnahmen beliebig oft wiederholbar sein.
  Dabei muß sichergestellt sein, daß
  - zweimaliges Undo derselben Änderung denselben Effekt hat wie einmaliges Undo,
  - zweimaliges Redo derselben Änderung denselben Effekt hat wie einmaliges Redo,
  - Undo einer Änderung, die durch den Systemabsturz bereits verloren gegangen ist, keinen Effekt hat und
  - Redo einer Änderung, die durch den Systemabsturz gar nicht verloren gegangen ist, keinen Effekt hat.

Generelle Zielsetzungen der Recovery−Komponente:

- Hohe Verfügbarkeit = $\frac{MTTF}{MTTF + MTTR}$

  (Eine Verfügbarkeit von 99 % impliziert beispielsweise eine "Auszeit" von ca. 5000 Minuten pro Jahr, während der das System nicht verfügbar ist;
  eine Verfügbarkeit von 99.9 % impliziert eine "Auszeit" von ca. 500 Minuten pro Jahr.)



- Schnelle Recovery beim Warmstart → kurze MTTR
- Geringer Overhead im laufenden Normalbetrieb

12.2 Implementierung der Crash−Recovery mittels Logging

Prinzipien zur Implementierung von Logging:

- Änderungen werden in Form von Logsätzen in einer Logdatei protokolliert.
  Um nicht für alle Änderung einen Plattenzugriff durchführen zu müssen, werden Logsätze zunächst in einen Logpuffer im Hauptspeicher geschrieben und nur bei Bedarf (s.u.) sequentiell in die Logdatei auf Platte geschrieben.

  Bemerkung:
  Um sicherzustellen, daß für die Logdatei tatsächlich immer sequentielle I/Os stattfinden, sollte die Logdatei auf einer separaten Platte liegen.

- Ein Logsatz enthält in der Regel sowohl die Undo−Information als auch die Redo−Information einer Änderung.
  Im einfachsten Fall bezieht sich ein Logsatz bzw. die darin protokollierte Änderung auf genau eine Seite der Datenbank. Der Logsatz beschreibt entweder den Zustand der Seite bzw. der geänderten Bytes vor und nach der Änderung (physisches Logging, zustandsorientiertes Logging) oder eine Operation auf der Byte−Ebene (physiologisches Logging, operationsorientiertes Logging auf der Byte−Ebene).
  Ein Spezialfall des zustandsorientierten Logging ist das Protokollieren ganzer Seiten jeweils vor und nach einer Änderung (”Before−Images” und ”After−Images”).

  Bemerkung:
  Operationsorientiertes Logging ist auch für komplexere Operationen möglich, die mehrere Seiten ändern (logisches Logging). Diese Variante ist aber schwieriger zu realisieren und daher in kommerziellen Systemen selten zu finden.

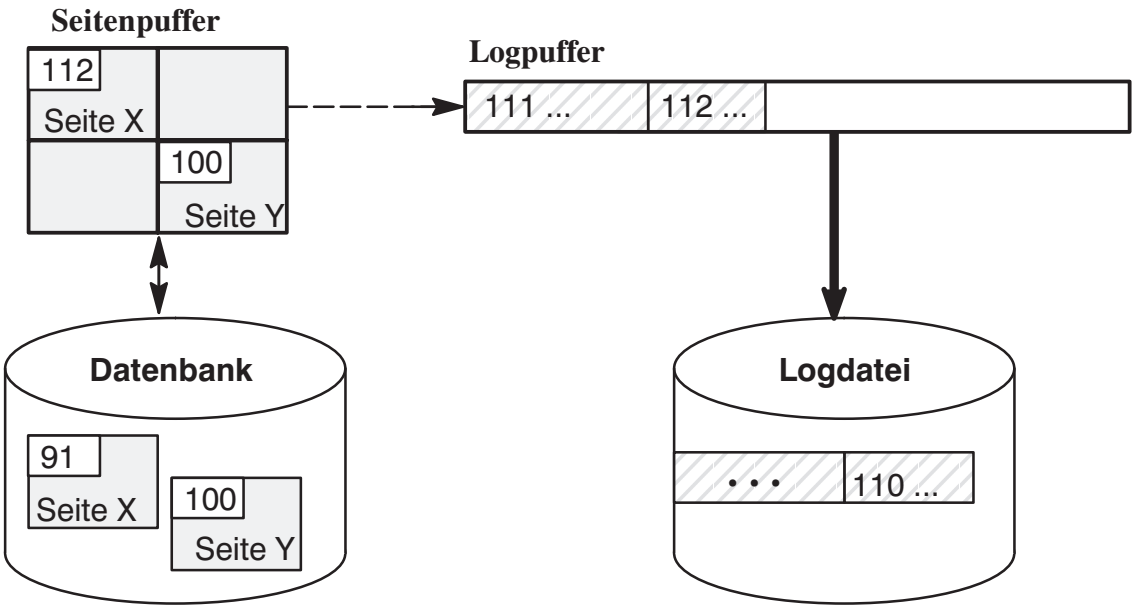
- Zusätzlich werden Logsätze für die Beendigung von Transaktionen (EOT und RBT) geschrieben, sowie optional auch für den Beginn von Transaktionen (BOT).

- Logsätze sind − über Logpuffer und Logdatei hinweg − transaktionsweise rückwärts verkettet. Dadurch können einzelne Transaktionen auch im laufenden Normalbetrieb einfach zurückgesetzt werden.

- Logsätze sind fortlaufend numeriert. Die Nummer eines Logsatzes − genannt ”*Log Sequence Number” (LSN)* − gibt den relativen Zeitpunkt einer Änderung an. Sie kann auch − je nach Implementierung − zur Adressierung der Logsätze in der Logdatei dienen.
  Die LSN der jeweils jüngsten Änderung einer Seite wird in einem speziellen Feld im Header der Seite gespeichert.

- Der Logpuffer muß in den folgenden Fällen in die Logdatei auf Platte geschrieben werden:
  - vor dem Zurückschreiben einer geänderten Seite aus dem Puffer in die Datenbank auf der Platte, sofern die Seite Änderungen von noch nicht abgeschlossenen Transaktionen enthält (*Write Ahead Logging, WAL−Prinzip*)
  - unmittelbar vor dem Commit einer Transaktion, die Änderungen durchgeführt hat.

Architektur für das Logging im laufenden Normalbetrieb:



Vorgehen beim Warmstart nach einem Crash:

1) *Analysephase:*
Die Logdatei wird sequentiell gelesen (in aufsteigender LSN−Reihenfolge), und dabei wird bestimmt, welche Transaktionen vor dem Crash beendet wurden (”Gewinner”) und welche unvollendet waren (”Verlierer”).

2) *Redo−Phase:*
Die Logdatei wird nochmals sequentiell gelesen (in aufsteigender LSN−Reihenfolge), und dabei werden die Änderungen der Gewinner wiederholt, sofern die Änderungen noch nicht in der Datenbank enthalten sind. Dazu wird getestet, ob die LSN des Logsatzes größer (d.h. jünger) ist als die LSN im Header der betreffenden Seite; und nur wenn dies der Fall ist, wird die Änderung wiederholt.
Dieser Test stellt die Idempotenz der Redo−Phase sicher.

3) *Undo−Phase:*
Die Logdatei wird nochmals sequentiell gelesen (in absteigender LSN−Reihenfolge), und dabei werden die Änderungen der Verlierer rückgängig gemacht, sofern die Änderungen in der Datenbank enthalten sind. Dazu wird getestet, ob die LSN des Logsatzes kleiner oder gleich der LSN im Header der betreffenden Seite ist (d.h. nicht jünger); und nur wenn dies der Fall ist, wird die Änderung rückgängig gemacht.
Dieser Test stellt die Idempotenz der Undo−Phase sicher.

Verbesserung der Effizienz:

- *im laufenden Normalbetrieb:*
  Wegen des Persistenzprinzips muß eigentlich bei jedem Commit einer Änderungs−transaktion der Logpuffer in die Logdatei auf Platte geschrieben werden. Man kann die Anzahl der Log−I/Os reduzieren, indem man das Schreiben mehrerer EOT−Logsätze zusammenfaßt. Dazu muß das Commit von Transaktionen (genauer: das Melden des SQLCODE 0 für die COMMIT−WORK−Anweisung des Anwendungsprogramms) ggf. verzögert werden, bis der Logpuffer voll ist, der Logpuffer wegen des WAL−Prinzips auf Platte geschrieben werden muß oder ein Commit bereits sehr lange verzögert wurde. Diese Technik nennt man *Group−Commit*.

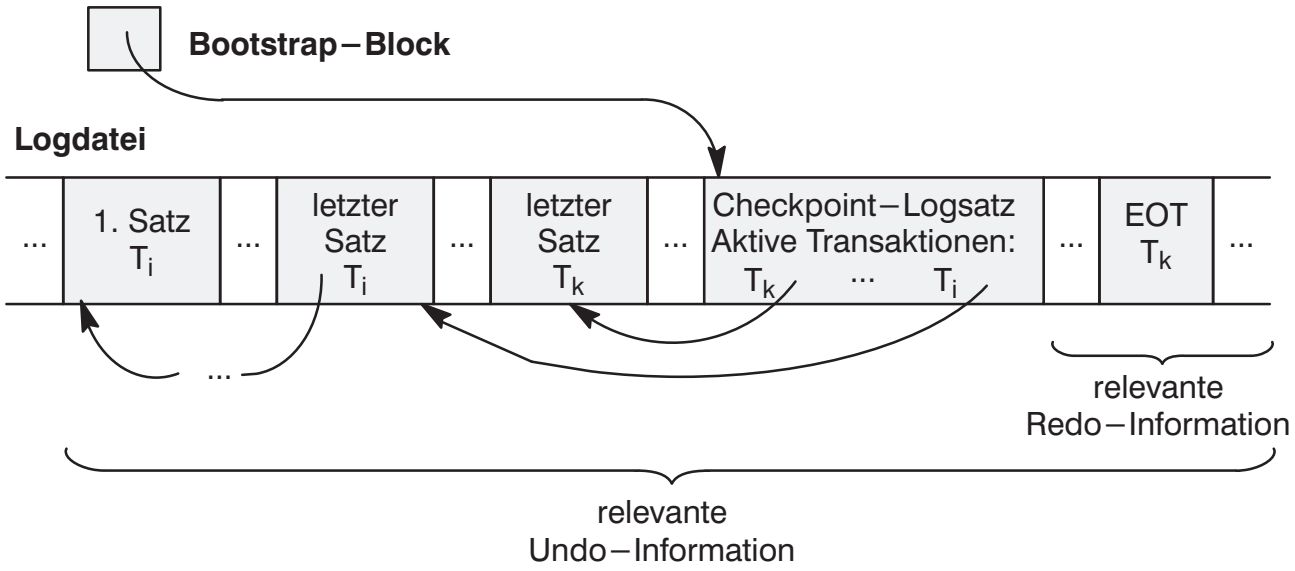
- *beim Warmstart:*
  Die Analysephase und die Redo−Phase der Recovery müssen im Prinzip die gesamte Logdatei lesen, obwohl vermutlich nur der jüngere Teil der Logdatei relevant ist.
  Dies kann verbessert werden, indem man im laufenden Normalbetrieb periodisch *Sicherungspunkte (Checkpoints)* erzeugt, die dazu beitragen, den Analyse− und Redo−Aufwand beim Warmstart zu begrenzen. Da Sicherungspunkte zusätzlichen Overhead im laufenden Normalbetrieb verursachen, kann der Systemadministrator bei den meisten kommerziellen DBS die Frequenz der Sicherungspunkterzeugung einstellen (typische Werte liegen in der Größenordnung von 10 Minuten).

  Variante 1: Synchrone Sicherungspunkte
  Beim Sicherungspunkt werden alle geänderten Seiten aus dem Datenbankpuffer in die Datenbank auf Platte zurückgeschrieben. Zusätzlich wird die Tatsache, daß ein Sicherungs−punkt erzeugt wurde in der Logdatei durch einen Checkpoint−Logsatz vermerkt. Die Plattenadresse (und/oder LSN) des jeweils jüngsten Checkpoint−Logsatzes wird in einem speziellen ”Bootstrap−Block” mit einer festen Plattenadresse festgehalten.
  Bei einem eventuellen späteren Warmstart muß die Redo−Phase die Logdatei dann nur ab dem jüngsten Checkpoint−Logsatz lesen.

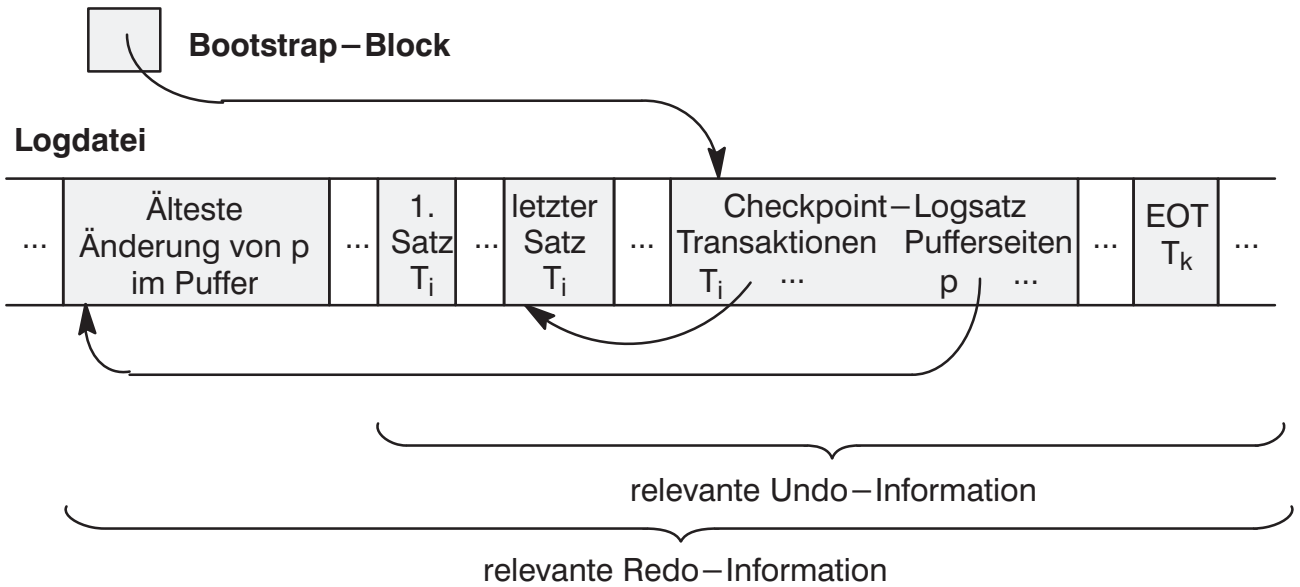
  Die Analysephase kann ebenfalls optimiert werden, indem man bei einem Sicherungspunkt in den Checkpoint−Logsatz eine Liste der zu diesem Zeitpunkt aktiven Transaktionen aufnimmt sowie für jede aktive Transaktion die LSN des jeweils letzten Logsatzes der Transaktion vor dem Sicherungspunkt. Beim Warmstart braucht die Analysephase dann ebenfalls die Logdatei nur ab dem jüngsten Checkpoint−Logsatz lesen.

  Variante 2: Asynchrone Sicherungspunkte
  Synchrone Sicherungspunkte belasten den laufenden Normalbetrieb vor allem dadurch, daß sie sehr viele Pufferseiten ”impulsartig” innerhalb kurzer Zeit in die Datenbank schreiben. Bei asynchrone Sicherungspunkten werden zum Zeitpunkt des Sicherungspunktes gar keine Seiten in die Datenbank zurückgeschrieben. Stattdessen läßt man einen ständig mitlaufenden Hintergrundprozeß veränderte Pufferseiten immer dann in Datenbank zurückschreiben, wenn die entsprechende Platte gerade unbelegt ist.
  Um trotzdem den Analyse− und Redo−Aufwand beim Warmstart gering zu halten, werden bei einem asynchronen Sicherungspunkt Informationen über den aktuellen Pufferinhalt in den Checkpoint−Logsatz geschrieben (zusätzlich zur Information über die aktiven Trans−aktionen): Für jede veränderte Pufferseite wird die LSN des ältesten Logsatzes dieser Seite seit dem letzten Zurückschreiben der Seite im Checkpoint−Logsatz eingetragen.
  Beim Warmstart braucht die Redo−Phase die Logdatei dann nur ab der ältesten LSN aller entsprechenden Seiten−Einträge im jüngsten Checkpoint−Logsatz lesen.

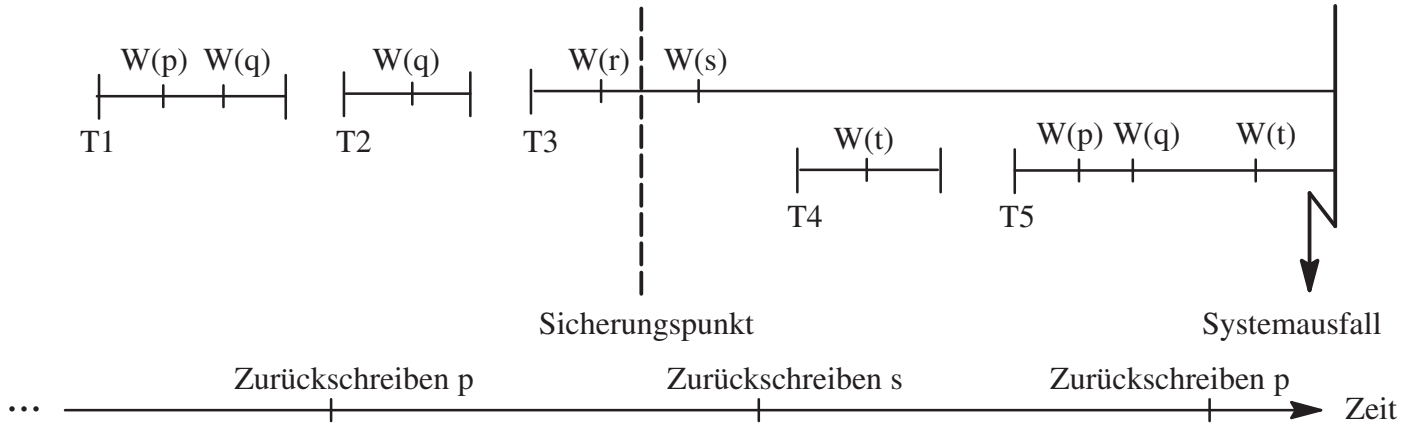
Logorganisation bei synchronen Sicherungspunkten:



Logorganisation bei asynchronen Sicherungspunkten:



Beispiel mit synchronen Sicherungspunkten:



Aktion	**Seitenpuffer [Seite (LSN)]**	**Datenbank [Seite (LSN)]**	**Logpuffer [LSN (Seite/Transaktion)]**	**Logdatei [LSN (Seite/Transaktion)]**
BOT (T1)		p (0), q (0), ...	–	
W1 (p)	p (1)		1 (p/T1)	
W1 (q)	q (2)		1 (p/T1), 2 (q/T1)	
EOT (T1)			–	> 1 (p/T1), 2 (q/T1), 3 (T1)
Flush (p)		p (1)		
BOT (T2)				
W2 (q)	q (4)		4 (q/T2)	
EOT (T2)			–	> 4 (q/T2), 5 (T2)
BOT (T3)				
W3 (r)	r (6)		6 (r/T3)	
Checkpoint		q (4), r (6)	–	> 6 (r/T3), 7 (CP:{T3})
W3 (s)	s (8)		8 (s/T3)	
Flush (s)		s (8)	–	> 8 (s/T3)
BOT (T4)				
W4 (t)	t (9)		9 (t/T4)	
EOT (T4)			–	> 9 (t/T4), 10 (T4)
BOT (T5)				
W5 (p)	p (11)		11 (p/T5)	
W5 (q)	q (12)		11 (p/T5), 12 (q/T5)	
Flush(p)		p (11)	–	> 11 (p/T5), 12 (q/T5)
W5 (t)	t (13)		13 (t/T5)	
Crash	–		–	

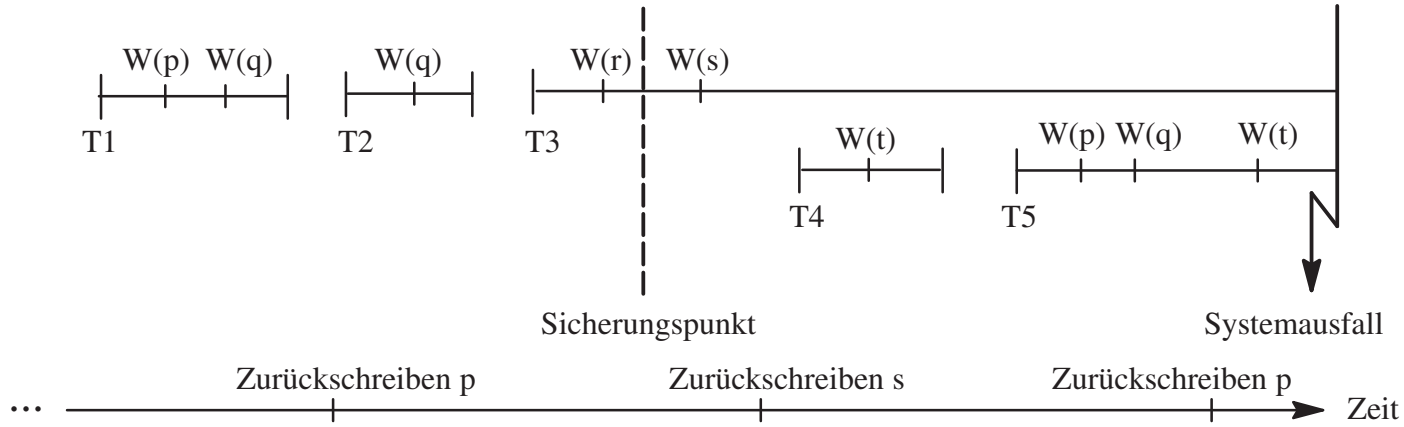
Warmstart:

Analysephase: Verlierer = {T3, T5},
Gewinner = {T4}

Redo−Phase: 9 (t/T4)

Undo−Phase: 11 (p/T5), 8 (s/T3), 6 (r/T3)

Beispiel mit asynchronen Sicherungspunkten:



Aktion	**Seitenpuffer [Seite (LSN)]**	**Datenbank [Seite (LSN)]**	**Logpuffer [LSN (Seite/Transaktion)]**	**Logdatei [LSN (Seite/Transaktion)]**
BOT (T1)				
W1 (p)				
W1 (q)				
EOT (T1)				
Flush (p)		p(1)		
BOT (T2)				
W2 (q)				
EOT (T2)				
BOT (T3)				
W3 (r)				
Checkpoint		—		> 6 (r/T3), 7 (CP:{T3},2)
W3 (s)				
Flush (s)		s(8)		
BOT (T4)				
W4 (t)				
EOT (T4)				
BOT (T5)				
W5 (p)				
W5 (q)				
Flush(p)		p(11)		
W5 (t)				
Crash				

Warmstart:

Analysephase:	Verlierer = {T3, T5}, Gewinner = alle anderen
Redo−Phase:	2 (q/T1), 4 (q/T2), 9 (t/T4)
Undo−Phase:	11 (p/T5), 8 (s/T3)

Pseudocode für Crash−Recovery mit asynchronen Sicherungspunkten:

/* Basic data structures of page−model redo−winners recovery algorithms: */

```
type Page: record of
      PageNo: identifier;
      PageSeqNo: identifier;
      Status: (clean, dirty);
      Contents: array [PageSize] of char;
   end;
persistent var StableDatabase:
      set of Page indexed by PageNo;
var DatabaseCache:
      set of Page indexed by PageNo;
type LogEntry: record of
      LogSeqNo: identifier;
      TransId: identifier;
      PageNo: identifier;
      ActionType: (write, full−write, begin, commit, rollback, checkpoint);
      ActiveTrans: set of TransInfo;
          /* present only in log entries of type checkpoint */
      DirtyPages: set of DirtyPageInfo;
          /* present only in log entries of type checkpoint */
      UndoInfo: array of char;
      RedoInfo: array of char;
      PreviousSeqNo: identifier;
   end;
persistent var StableLog:
      ordered set of LogEntry indexed by LogSeqNo;
var LogBuffer:
      ordered set of LogEntry indexed by LogSeqNo;
persistent var MasterRecord: record of
      StartPointer: identifier;
      LastCP: identifier;
   end;
type TransInfo: record of
      TransId: identifier;
      LastSeqNo: identifier;
   end;
var ActiveTrans:
      set of TransInfo indexed by TransId;
type DirtyPageInfo: record of
      PageNo: identifier;
      RedoSeqNo: identifier;
   end;
var DirtyPages:
      set of DirtyPageInfo indexed by PageNo;
```

/* Actions during normal operation */

```
write or full−write (pageno, transid, seqno):
  DatabaseCache[pageno].Contents := modified contents;
  DatabaseCache[pageno].PageSeqNo := seqno;
  DatabaseCache[pageno].Status := dirty;
  newlogentry.LogSeqNo := seqno;
  newlogentry.ActionType := write or full−write;
  newlogentry.TransId := transid;
  newlogentry.PageNo := pageno;
  newlogentry.UndoInfo := information to undo update
     (before−image for full−write);
  newlogentry.RedoInfo := information to redo update
     (after−image for full−write);
  newlogentry.PreviousSeqNo :=
      ActiveTrans[transid].LastSeqNo;
  ActiveTrans[transid].LastSeqNo := seqno;
  LogBuffer += newlogentry;
  if pageno not in DirtyPages then
       DirtyPages += pageno;
       DirtyPages[pageno].RedoSeqNo := seqno;
  end /*if*/;

fetch (pageno):
  DatabaseCache += pageno;
  DatabaseCache[pageno].Contents :=
     StableDatabase[pageno].Contents;
  DatabaseCache[pageno].PageSeqNo :=
     StableDatabase[pageno].PageSeqNo;
  DatabaseCache[pageno].Status := clean;

flush (pageno):
  if there is logentry in LogBuffer with logentry.PageNo = pageno
  then
     force ( );
  end /*if*/;
  StableDatabase[pageno].Contents :=
     DatabaseCache[pageno].Contents;
  StableDatabase[pageno].PageSeqNo :=
     DatabaseCache[pageno].PageSeqNo;
  DatabaseCache[pageno].Status := clean;
  DirtyPages −= pageno;

force ( ):
    StableLog += LogBuffer;
    LogBuffer := empty;
```

```
begin (transid, seqno):
  ActiveTrans += transid;
  ActiveTrans[transid].LastSeqNo := seqno;
  newlogentry.LogSeqNo := seqno;
  newlogentry.ActionType := begin;
  newlogentry.TransId := transid;
  newlogentry.PreviousSeqNo := nil;
  LogBuffer += newlogentry;

commit (transid, seqno):
  newlogentry.LogSeqNo := seqno;
  newlogentry.ActionType := commit;
  newlogentry.TransId := transid;
  newlogentry.PreviousSeqNo :=
      ActiveTrans[transid].LastSeqNo;
  LogBuffer += newlogentry;
  ActiveTrans −= transid;
  force ( );
```

/* **Checkpointing and log truncation during normal operation** */

```
log truncation ( ):
   OldestUndoLSN :=
       min {i | StableLog[i].TransId is in ActiveTrans};
   SystemRedoLSN := min {DirtyPages[p].RedoSeqNo};
   OldestRedoPage := page p such that
       DirtyPages[p].RedoSeqNo = SystemRedoLSN;
   NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
   OldStartPointer := MasterRecord.StartPointer;
   while OldStartPointer − NewStartPointer
            is not sufficiently large
   and SystemRedoLSN < OldestUndoLSN
   do
      flush (OldestRedoPage);
      SystemRedoLSN := min{DirtyPages[p].RedoSeqNo};
      OldestRedoPage := page p such that
          DirtyPages[p].RedoSeqNo = SystemRedoLSN;
      NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
   end /*while*/;
   MasterRecord.StartPointer := NewStartPointer;

checkpoint ( ):
   log truncation ( );
   logentry.ActionType := checkpoint;
   logentry.ActiveTrans :=
       ActiveTrans (as maintained in memory);
   logentry.DirtyPages := DirtyPages;
   logentry.LogSeqNo := new sequence number;
   LogBuffer += logentry;
   force ( );
   MasterRecord.LastCP := logentry.LogSeqNo;
```

```
/* Simple three−pass crash−crash recovery */

restart ( ):
  analysis pass ( ) returns losers;
  redo pass ( );
  undo pass ( );

analysis pass ( ) returns losers, DirtyPages:
  cp := MasterRecord.LastCP;
  losers := StableLog[cp].ActiveTrans;
  DirtyPages := StableLog[cp].DirtyPages;
  max := LogSeqNo of most recent log entry in StableLog;
  for i := cp to max do
    case StableLog[i].ActionType:
      begin:
        losers += StableLog[i].TransId;
        losers[StableLog[i].TransId].LastSeqNo := nil;
      commit:
        losers −= StableLog[i].TransId;
      full−write:
        losers[StableLog[i].TransId].LastSeqNo := i;
    end /*case*/;
    if StableLog[i].ActionType = write or full−write
       and StableLog[i].PageNo not in DirtyPages
    then
       DirtyPages += StableLog[i].PageNo;
       DirtyPages[StableLog[i].PageNo].RedoSeqNo := i;
    end /*if*/;
  end /*for*/;
```

```
redo pass ( ):
  cp := MasterRecord.LastCP;
  SystemRedoLSN := min{cp.DirtyPages[p].RedoSeqNo};
  max := LogSeqNo of most recent log entry in StableLog;
  for i := SystemRedoLSN to max do
       if StableLog[i].ActionType = write or full−write
          and StableLog[i].TransId not in losers
       then
            pageno := StableLog[i].PageNo;
            if pageno in DirtyPages
               and i >= DirtyPages[pageno].RedoSeqNo
            then
               fetch (pageno);
               if DatabaseCache[pageno].PageSeqNo < i
               then
                  read and write (pageno) according to StableLog[i].RedoInfo;
                  DatabaseCache[pageno].PageSeqNo := i;
               else
                  DirtyPages[pageno].RedoSeqNo :=
                     DatabaseCache[pageno].PageSeqNo + 1;
               end /*if*/;
            end /*if*/;
       end /*if*/;
  end /*for*/;


undo pass ( ):
  while there exists t in losers
       such that losers[t].LastSeqNo <> nil
  do
      nexttrans = TransNo in losers
          such that losers[nexttrans].LastSeqNo =
          max {losers[x].LastSeqNo | x in losers};
     nextentry = losers[nexttrans].LastSeqNo;
     if StableLog[nextentry].ActionType = full−write
     then
       pageno = StableLog[nextentry].PageNo;
       fetch (pageno);
       if DatabaseCache[pageno].PageSeqNo >= nextentry
       then
         read and write (pageno) according to StableLog[nextentry].UndoInfo;
         DatabaseCache[pageno].PageSeqNo := nextentry−1;
       end /*if*/;
       losers[nexttrans].LastSeqNo :=
          StableLog[nextentry].PreviousSeqNo;
     end /*if*/;
  end /*while*/;
```

12.3 Verteilte Commit−Protokolle

Problem der Atomarität in einem verteilten System:

Transaktion können Daten in mehreren Systemen ändern (mehrere Datenbanken desselben DBS oder mehrere DBS). Es ist erheblich schwieriger, die Atomarität einer solchen verteilten Transaktion zu gewährleisten. Zusätzlich zu den Logging−Maßnahmen der beteiligten Systeme ist ein verteiltes Protokoll notwendig, um das Commit oder Rollback der Transaktion in allen Systemen einheitlich durchzuführen. Es darf nicht passieren, daß ein System einen Commit−Logsatz für die Transaktion schreibt, während ein anderes System seinen Teil der Transaktion zurücksetzt, beispielsweise weil dieses System abstürzt. Zusätzlich müssen verschiedene Arten von Nachrichtenfehlern (verlorene Nachrichten, duplizierte Nachrichten) verkraftet werden.

2−Phasen−Commit−Protokoll (2PC)

Grundprinzip:

Eines der beteiligten Systeme wird zum *Koordinator* für den Ablauf des Commit einer Transaktion bestimmt. Es kann z.B. immer das System gewählt werden, von dem aus die Transaktion initiiert wurde. Alle Systeme, in denen die Transaktion Änderungen durchgeführt hat, werden als *Agenten* bezeichnet. Das Protokoll läuft dann in zwei Phasen ab:

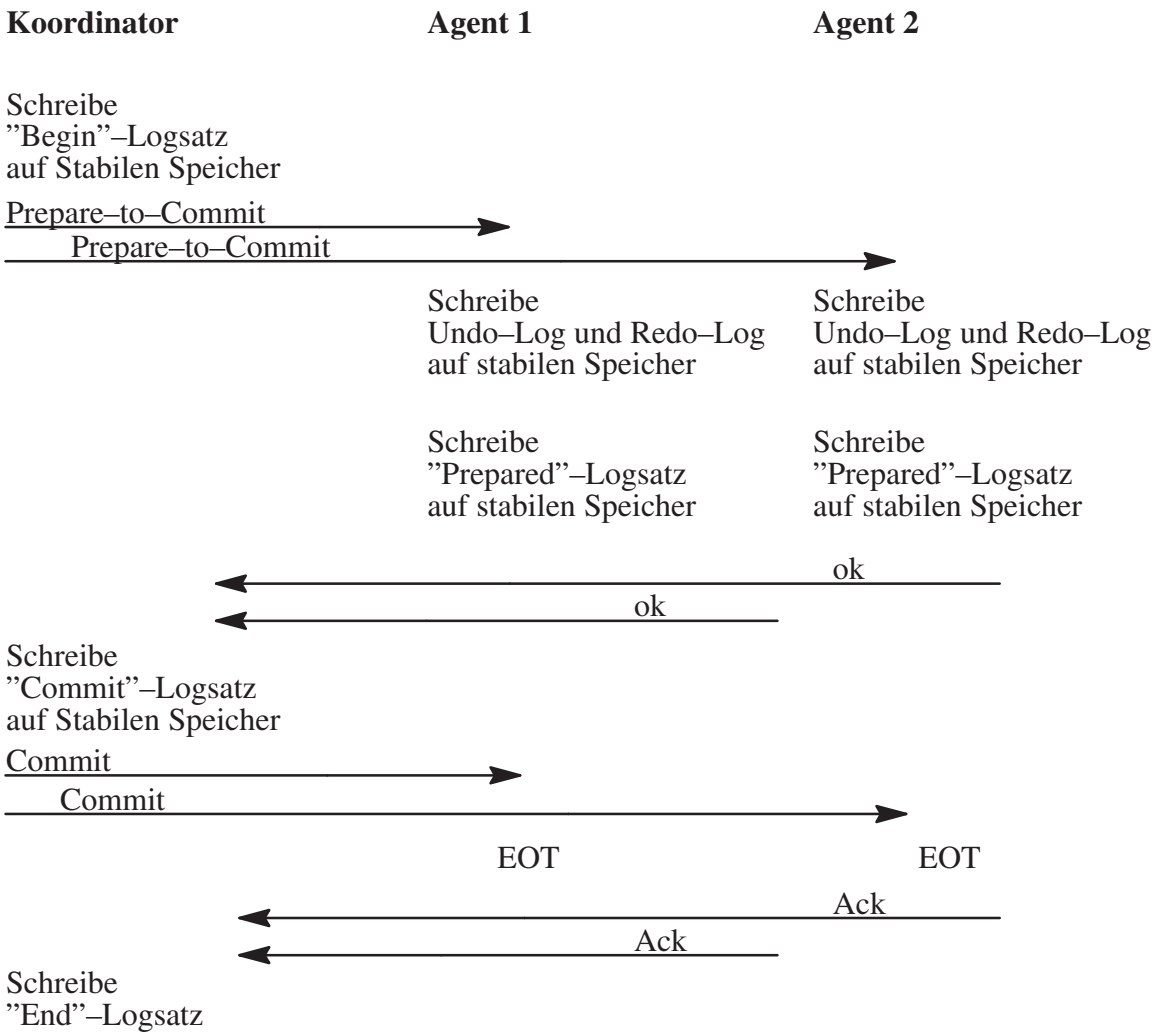
- Phase 1: Der Koordinator fragt alle Agenten, ob sie in der Lage sind, die Transaktion mit Commit zu beenden, und bittet die Agenten gleichzeitig, sich sowohl auf ein Commit als auch ein Rollback vorzubereiten (”Prepare”−Nachricht). Im positiven Fall gehen dann alle Agenten in den ”Prepared”−Zustand über.
- Phase 2: Aufgrund des ”Umfrageergebnisses” trifft der Koordinator dann eine globale Entscheidung für die Transaktion und teilt diese allen Agenten mit. Nur wenn alle Agenten positiv geanwortet haben, kann der Koordinator eine Commit−Entscheidung treffen; andernfalls muß seine Entscheidung ”Rollback” lauten.

In der Zeit zwischen dem Antworten eines Agenten und dem Erhalt der Koordinatorentscheidung kann ein Agent keine eigenmächtige Entscheidung treffen. Insbesondere muß der Agent weiterhin alle Sperren für die Transaktion halten, da auch ein Rollback noch möglich ist. Da der Koordinator ausfallen kann oder die Nachricht mit seiner Entscheidung verloren gehen kann, können Agenten auf diese Weise in eine gewisse Blockierungssituation geraten, die sich sehr negativ auf die Leistung des betroffenen Systems auswirken kann. Man bezeichnet aus diesem Grund das 2PC auch als ein ”blockierendes” Commit−Protokoll. Man kann beweisen, daß es − unter Berücksichtigung aller möglichen Fehlerfälle − kein nichtblockierendes verteiltes Commit−Protokoll geben kann.

Bemerkung:

- Es gibt optimierende Varianten des 2PC, die Nachrichten und/oder Log−I/Os in bestimmten Fällen vermeiden.
- Das 2PC ist standardisiert (ISO/OSI, X/Open). Es wird von praktisch allen kommerziellen DBS unterstützt, wobei diese sowohl in die Koordinator− als auch in die Agentenrolle schlüpfen können. In vielen Systemumgebungen, insbesondere wenn verschiedene DBS zum Einsatz kommen, kann die Koordinatorrolle von einem *TP−Monitor (Transaction Processing Monitor)* übernommen werden.
  TP−Monitore sind Systemsoftware−Komponenten, die eine transaktionsorientierte Kommunikation zwischen Clients und Servers steuern.

Nachrichten und Zustandsänderungen des 2PC im Erfolgsfall:



Behandlung von Nachrichtenverlusten:

Falls bei einem Knoten eine – aufgrund des Protokolls – erwartete Nachricht nicht innerhalb einer bestimmten "Timeout"–Frist eintrifft, wird angenommen, daß der Sender ausgefallen ist, und der Empfänger verhält sich wie bei einem Knotenausfall.

Behandlung von Knotenausfällen:

Ausfallender Knoten	**Reaktion Koordinator**	**Reaktion des Agenten**
Koordinator in Phase 1 (vor dem Schreiben des "Commit"–Logsatzes)	Nach dem Restart: Wiederholung des Sendens der "Prepare"–Nachricht	– Falls noch nicht "prepared": Warten oder einseitiges Abort – Falls bereits "prepared": Warten (Blockierung) oder ggf. andere Agenten fragen
Koordinator in Phase 2 (vor dem Schreiben des "End"–Logsatzes)	Nach dem Restart: Wiederholung des Sendens der "Commit"– oder "Abort"–Nachricht	– Vor dem EOT: Warten (Blockierung) oder ggf. andere Agenten fragen – Nach dem EOT: Wiederholung des "Ack" bei wiederholtem Empfang der "Commit"– oder "Abort"–Nachricht
Agent in Phase 1 (vor dem Schreiben des "Prepared"–Logsatzes)	Bei ausbleibendem Votum: Wiederholung des Sendens der "Prepare"–Nachricht	Nach dem Restart: Einseitiges Abort
Agent in Phase 2 (vor dem EOT)	Bei ausbleibendem "Ack": Wiederholung des Sendens der "Commit"– oder "Abort"–Nachricht	Nach dem Restart: Warten auf Nachricht des Koordinators (Wiederholte "Prepare"–Nachricht oder (wiederholte) "Commit"/"Abort"–Nachricht) oder Nachfragen beim Koordinator

Ergänzende Literatur zu Kapitel 12:

J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993

P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison−Wesley, 1987

G. Weikum, Transaktionen in Datenbanksystemen, Addison−Wesley, 1988

T. Özsu, P. Valduriez, Principles of Distributed Database Systems, 2nd Edition, Prentice−Hall, 1998