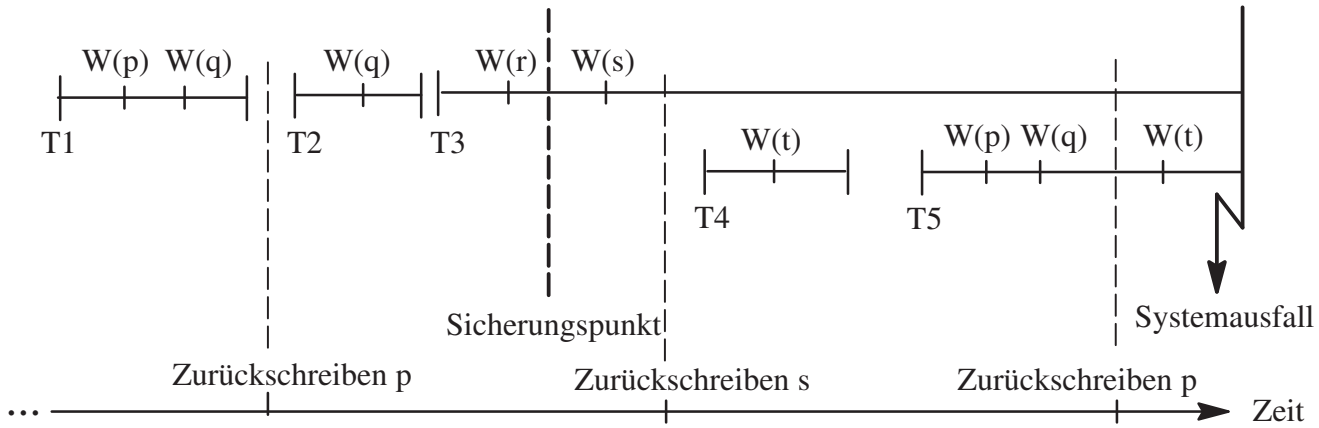
UNIVERSITÄT DES SAARLANDES
Fachbereich 14– Informatik
Prof. Dr.–Ing. G. Weikum
Dipl.–Inform. R. Schenkel

Datenbanksysteme WS 1999/2000

Übung 11 — Beispiellösung

Aufgabe 1: Recovery

Vor einem Systemausfall seien in einem Datenbanksystem die Transaktionen T1, T2, T3, T4 und T5 abgelaufen, die auf die Datenbankseiten p,q,r,s und t zugegriffen hatten. Der genaue zeitliche Ablauf der Transaktionen sowie Sicherungspunkte und Zurückschreib–Aktionen vom Puffer in die Datenbank seien durch den folgenden Schedule gegeben.



a) Machen Sie sich den genauen Ablauf bei Verwendung synchroner Sicherungspunkte klar. Geben Sie dazu nach jeder Operation den Inhalt des Seitenpuffers, der Datenbank, des Logpuffers und der Logdatei an. Verwenden Sie dazu eine Tabelle analog zum Skript.

b) Beschreiben Sie die Aktionen, die beim Warmstart ausgeführt werden müssen.

c) Nehmen Sie nun an, die Sicherungspunkte seien asynchron, und verfahren Sie wie in a) und b).

Lösung zu 1a):

Aktion	**Seitenpuffer [Seite (LSN)]**	**Datenbank [Seite (LSN)]**	**Logpuffer [LSN (Seite/Trans.)]**	**Logdatei [LSN (Seite/Trans.)]**
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	Inhalt verloren		Inhalt verloren	

Lösung zu 1b):

Beim Warmstart wird zunächst mit der ***Analysephase*** begonnen. Der Logsatz wird ab dem letzten Checkpoint gelesen und es wird festgestellt, daß die Transaktion T3 zum Zeitpunkt des Checkpoints aktiv war. Durch sequentielles Lesen der weiteren Einträge wird festgestellt, daß die Transaktion T4 begonnen und committed wurde, die Transaktionen T3 und T5 jedoch nicht bis zum COMMIT gekommen sind. dadurch ergibt sich nach der Analysephase: Verlierer= {T3, T5}, Gewinner= {T4}

In der anschließenden ***REDO–Phase*** müssen nun alle Gewinner–Transaktionen (hier nur T4) nachgeführt werden, soweit ihre Änderungen noch nicht in die Datenbank eingebracht wurden. Ob dies der Fall ist oder nicht, wird anhand der LSNs festgestellt. Bei LSN 9 wird das After–Image (bzw. die logische Änderungsoperation) von t gelesen. In die Datenbank wurde t bisher nicht eingebracht, so daß die LSN dort kleiner als 9 sein muß. Wir müssen die Änderung von T4 in der Seite t also nachführen.

In der nun folgenden ***UNDO–Phase*** müssen alle Änderungen von Verlierer–Transaktionen, die in die Datenbank eingebracht wurden, rückgängig gemacht werden. Wir beginnen mit dem Lesen der

Logdatei ab dem Ende der Datei in Richtung Anfang. Bei LSN 12 fällt zunächst keine UNDO–Operation an, da die LSN von q in der Datenbank kleiner ist als 12 und damit die ursprüngliche Änderung von T5 nicht materialisiert wurde. Bei LSN 11 jedoch wird erkannt, daß die Änderung der Transaktion T5 auf Seite p rückgängig gemacht werden muß, da sie vor dem Crash in die Datenbank eingebracht wurde. Die Änderung von T3 auf s (LSN 8) muß aus denselben Gründen rückgängig gemacht werden. Wir gelangen nun zum Checkpoint und erkennen, daß T3 eine Verlierer–Transaktion ist, deren vorherige Änderungen rückgängig gemacht werden müssen. Da Log–Einträge von Transaktionen rückwärts verkettet sind, finden wir im Eintrag bei LSN 6 einen NIL–Zeiger, da die Transaktion T3 vorher keine Änderungen hatte. Wir können die UNDO–Phase deshalb hier beenden.

Lösung zu 1c):

Aktion	**Seitenpuffer [Seite (LSN)]**	**Datenbank [Seite (LSN)]**	**Logpuffer [LSN (Seite/Trans.)]**	**Logdatei [LSN (Seite/Trans.)]**
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		**–**	–	.. 6 (r/T3), 7 (CP:{T3}, ***2***)
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	Inhalt verloren		Inhalt verloren	

Die Aktionen unterscheiden sich während des Normalbetriebs nur darin, daß zum Zeitpunkt des Checkpoints keine Seiten in die Datenbank eingebracht werden, und daß stattdessen nur die Information in den Logsatz eingetragen wird, ab welcher LSN die REDO–Phase beginnen muß. Die REDO–Phase muß bei der ältesten Aktion einer committed Transaktion beginnen, die noch nicht in die Datenbank eingebracht wurden. Wir haben zum Zeitpunkt des Checkpoints die Seiten geänderten Seiten p,q und r im Puffer. Die Seite p wurde zum ’Zeitpunkt’ 1 geändert, die Änderung ist jedoch bereits

in die Datenbank eingebracht worden. Die Änderung auf q hingegen ist noch nicht eingebracht, und da die Transaktion 1 committed ist, muß das REDO dort aufsetzen. Deshalb wird LSN 2 als Aufsetzpunkt in den Checkpoint–Satz eingetragen.

In der ***Analysephase*** lesen wir zunächst den Checkpoint–Satz und stellen fest, daß Transaktion T3 zu diesem Zeitpunkt noch aktiv war. T3 gehört also noch zu den Transaktionen vor dem Checkpoint, die zu einem Verlierer geworden sein könnte. Alle anderen vor dem Checkpoint liegenden Transaktionen müssen Gewinner sein. Deshalb brauchen wir in der Analysephase auch hier nur den Log ab dem letzten Checkpoint zu lesen, um alle Verlierer–Transaktionen {T3, T5} zu bestimmen. Alle anderen Transaktionen sind Gewinner.

In der **REDO–Phase** beginnen wir bei LSN 2 und verfahren ansonsten wie bei Teilaufgabe b). Wir haben hier die Änderungen auf q von T1 (LSN 2), q von T2 (LSN 4) und t von T4 (LSN 9) nachzuführen.

Die **UNDO–Phase** wird analog wie in Teilaufgabe b) ausgeführt. Dadurch, daß beim Checkpoint keine Seiten in die Datenbank eingebracht wurden, sparen wir hier jedoch eine UNDO–Aktion. Die Aktion von T3 auf r braucht nicht rückgängig gemacht zu werden, da sie nicht in den Puffer eingebracht wurde. Nur LSN 11 und LSN 8 führen zu UNDO–Operationen.

Aufgabe 2: Recovery & Concurrency Control

Warum kann man bei Verwendung von Tupelsperren als Concurrency–Control–Protokoll die Recovery nicht einfach mittels Before– und After–Images von Seiten realisieren?

Könnte man wenigstens nur die Undo–Phase mittels Seiten–Before–Images durchführen oder nur die Redo–Phase mittels Seiten–After–Images?

Welche Vorteile hätten diese eingeschränkten Varianten gegenüber einer Recovery, die sowohl bezüglich Undo als auch Redo auf tupelorientierten Operationen basiert? Geben Sie ggf. Beispiele an, die Ihre Überlegungen veranschaulichen.

Lösung zu 2:

Betrachten wir die Seite *X*, die die beiden Tupel *p* und *q* enthält. Vor dem Crash der Datenbank seien u.a. die folgenden Aktionen durchgeführt worden:

T1:	*W1(p)*			|	
T2:		*W2(q)*	*EOT*	|	**CRASH!**

Während der Redo–Phase wird die Seite *X* so geschrieben, wie sie nach *W2(q)* ausgesehen hat, da *T2* eine Gewinnertransaktion ist. Führen wir jetzt ein seitenorientiertes Undo aus, so wird die Seite *X* wieder überschrieben und zwar mit dem Before–Image von *W1(p)*. Dadurch wird jedoch die Änderung von Transaktion *T2* wieder rückgängig gemacht.

Die Undo–Phase kann bei Tupelsperren nie durch Seiten–Images ausgeführt werden, da entweder bereits erfolgte Änderungen der Redo–Phase wieder rückgängig gemacht werden (Undo–Phase nach Redo–Phase), oder es können nach der Undo–Phase inkonsistente Datenbankzustände existieren, die

es unmöglich machen anschließend ein tupelorientiertes Redo zu machen. Betrachten wir z.B. folgende Transaktionen:

T1:	*W1(p)*		*W1(p)*	*EOT*	|	
T2:		*W2(q)*			|	**CRASH!**

Nach dem seitenorientierten Undo sieht die Seite *X* aus wie zwischen dem ersten *W1(p)* und *W2(q)*. Da einzelne Tupel keine LSN besitzen, ist es in der Redo–Phase für Transaktion *T1* unmöglich festzustellen, welche Aktionen wiederholt werden müssen und welche nicht.

Im Gegensatz zum Undo ist es jedoch möglich ein seitenorientiertes Redo für Gewinnertransaktionen mit anschließendem tupelorientierten Undo durchzuführen.

Aufgabe 3: Rollbacks

a) Ergänzen Sie den in den Vorlesungsunterlagen angegebenen Pseudocode für Logging und Crash-Recovery um Pseudocode für die Operation *rollback(transid, seqno)*.

b) Erweitern Sie den Pseudocode für die Analyse-, Undo- und Redo-Phasen der Recovery um die Behandlung von Transaktionen, die vor dem Crash mit Rollback beendet waren.

c) Welche Schwierigkeiten treten bei der ‘‘Log Truncation’’ in Verbindung mit solchen Rollback–Transaktionen auf? Erweitern Sie ggf. auch den Pseudocode für das Checkpointing entsprechend.

Lösung zu 3a):

Problem: Abgebrochene Transaktionen als Verlierer behandeln

Eine scheinbar offensichtliche Variante, abgebrochene Transaktionen einfach als “Verlierer” im Log “stehenzulassen” - d.h. ohne einen Commit-Logsatz im Log einzutragen - stößt hier auf die folgenden Probleme.

- Beim Warmstart müßte zum Ermitteln der Verlierer-Transaktionen nun die ganze Logdatei durchsucht werden, da evtl. auch vor dem Checkpoint noch relevante UNDO-Informationen für abgebrochene Transaktionen stehen können. Da abgebrochene Transaktionen als potentiell “unerledigte” Verlierer im Log verzeichnet sind, ist ein einfaches Abschneiden der Logdatei (Log Truncation) ebenfalls nicht mehr möglich.
- Ohne Berücksichtigung abgebrochener Transaktionen gilt für die Logdatei, daß einem Zugriff auf eine Seite durch eine Verlierer-Transaktion niemals ein anderer Zugriff auf dieselbe Seite durch eine andere Transaktion folgen darf. Diese Invariante ist jedoch notwendig, um den REDO-Lauf vor dem UNDO-Lauf durchführen zu können.

Lösung 1: Abgebrochene Transaktionen als Gewinner behandeln

Eine einfache und elegante Möglichkeit zum Zurücksetzen von Transaktionen ist, eine abgebrochene Transaktion durch normales Schreiben der Before-Images erfolgreich zu Ende zu führen und die Transaktion dadurch zu den Gewinnern gehören zu lassen. Der Pseudo-Code für das Rollback würde in diesem Fall folgendermaßen aussehen:

```
rollback(transid, lastSeqNo)
  logEntry:= log entry with index lastSeqNo;
  while logEntry is not nil and logEntry.actionType= write or full-write
     newlogentry.LogSeqNo := new log sequence number;
     newlogentry.ActionType := write;
     newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
     newlogentry.RedoInfo := logEntry.UndoInfo;
     newlogentry.UndoInfo := logEntry.RedoInfo;
     ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
     LogBuffer += newlogentry;
     write page logEntry.Pageno according to logEntry.UndoInfo;
     DatabaseCache[logEntry.Pageno].Status:=dirty;
     ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
     lastSeqNo := logEntry.PreviousSeqNo;
     logEntry:= log entry with index lastSeqNo;
  end
  newlogentry.LogSeqNo:=new sequence number;
  newlogentry.ActionType:=commit;
  newlogentry.TransId:=transid;
  newlogentry.PreviousSeqNo:= ActiveTrans[transId].LastSeqNo;
  LogBuffer+=newlogentry;
  ActiveTrans–=transid;
  force();
```

Die Eleganz der Lösung liegt darin, daß keine weiteren Änderungen an den Recovery-Prozeduren vorgenommen werden müssen. Der Nachteil liegt in der zusätzlichen Erzeugung von Logeinträgen beim Rollback.

Lösung 2: Abgebrochene Transaktionen zurücksetzen und aus Sicht der Recovery “erledigen” Um zu vermeiden, daß beim Zurücksetzen einer Transaktion zusätzliche Logeinträge erzeugt werden, kann folgender Algorithmus angewendet werden: Die Kette der Logsätze wird rückwärts verfolgt, und es wird für jede Seite ein UNDO durchgeführt (ähnlich wie in undo pass()). Nach dem UNDO einer Seite wird die LSN in der Seite auf die um Eins dekrementierte LSN des Logsatzes gesetzt. Dadurch wird die Idempotenz des UNDOs im Falle eines Systemcrashes während des Zurücksetzens garantiert. Wenn alle Seiten zurückgesetzt wurden, schreibt die Transaktion alle betroffenen Seiten persistent in die Datenbank. Dadurch wird sichergestellt, daß nach einem evtl. folgenden Systemabsturz keine UNDO-Aktion mehr während des Warmstarts für die abgebrochene Transaktion durchgeführt werden muß. Die Transaktion kann damit als “erledigt” betrachtet werden, und es wird ein Rollback-Eintrag in den Log eingefügt, um einem potentiellen Warmstart zu signalisie-

ren, daß die Transaktion weder ein u.U. noch zurückzusetzender Verlierer noch ein auszuführender Gewinner ist. In diesem Fall sieht der Pseudo-Code der Prozedur *rollback* folgendermaßen aus:

```
rollback(transid, seqno)
  nextLSN:=ActiveTrans[transid].LastSeqNo;
  do
     logEntry:= log entry with index nextLSN;
      if(logEntry.ActionType==write or fullwrite)
      then
        write page logEntry.pageno according to logEntry.UndoInfo;
        DatabaseCache[logEntry.PageNo]:=(logEntry.LogSeqNo)–1;
        DatabaseCache[logEntry.PageNo].Status:=dirty;
        touchedPages += logEntry.PageNo;
      end
      nextLSN:=logEntry.PreviousSeqNo;
   while(nextLSN is not nil)
   forall touchedPages p do
    flush(p);
   end;
   newlogentry.LogSeqNo:=seqno;
   newlogentry.ActionType:=rollback;
   newlogentry.TransId:=transid;
   newlogentry.PreviousSeqNo:=
   ActiveTrans[transId].LastSeqNo;
   LogBuffer+=newlogentry;
   ActiveTrans–=transid;
   flush();
```

Lösung zu 3b):

Nur bei Lösung 2 muß die Bestimmung der Menge der Gewinner- und Verlierer-Transaktionen modifiziert werden. Hier muß neben den Verlierern noch die Menge *rolledback* der zurückgesetzten Transaktionen bestimmt werden, die beim Warmstart ignoriert werden können, da die zurückgesetzten Seiten bereits in die Datenbank eingebracht sind.

```
analysis pass ( ) returns losers, rolledback, 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;
         rollback:
               losers –= StableLog[i].TransId;
               rolledback += StableLog[i].TransId;
       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 and not in rolledback
      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 ( ): (bleibt unverändert)

Lösung zu 3c):

Bei Lösung 1 bleibt die Log-Truncation unverändert, da abgebrochene Transaktionen als normale Gewinner angesehen werden. Bei Lösung 2 können Logeinträge von abgebrochenen Transaktionen einfach entfernt werden, da die Änderungen garantiert bereits in die Datenbank eingebracht wurden und deshalb kein REDO mehr für die Seiten durchgeführt werden muß.