# Kapitel 16: Daten-Recovery – Wie Systemausfälle behandelt werden

Fehlerkategorien:

1. Fehler im Anwendungsprogramm
2. Ausfall der Systemsoftware (BS, DBS, usw.): Bohrbugs, Heisenbugs
3. Stromausfall und transiente Hardwarefehler
4. Plattenfehler
5. Katastrophen

Behandlung durch das DBS:

1 → Rollback
2, 3 → Crash Recovery (basierend auf Logging)
4 → Media Recovery (basierend auf Backup und Logging)
5 → Remote Backup/Log, Remote Replication

# Goal of Crash Recovery

**Failure-resilience:**

- **redo** recovery for committed transactions
- **undo** recovery for uncommitted transactions

**Failure model:**

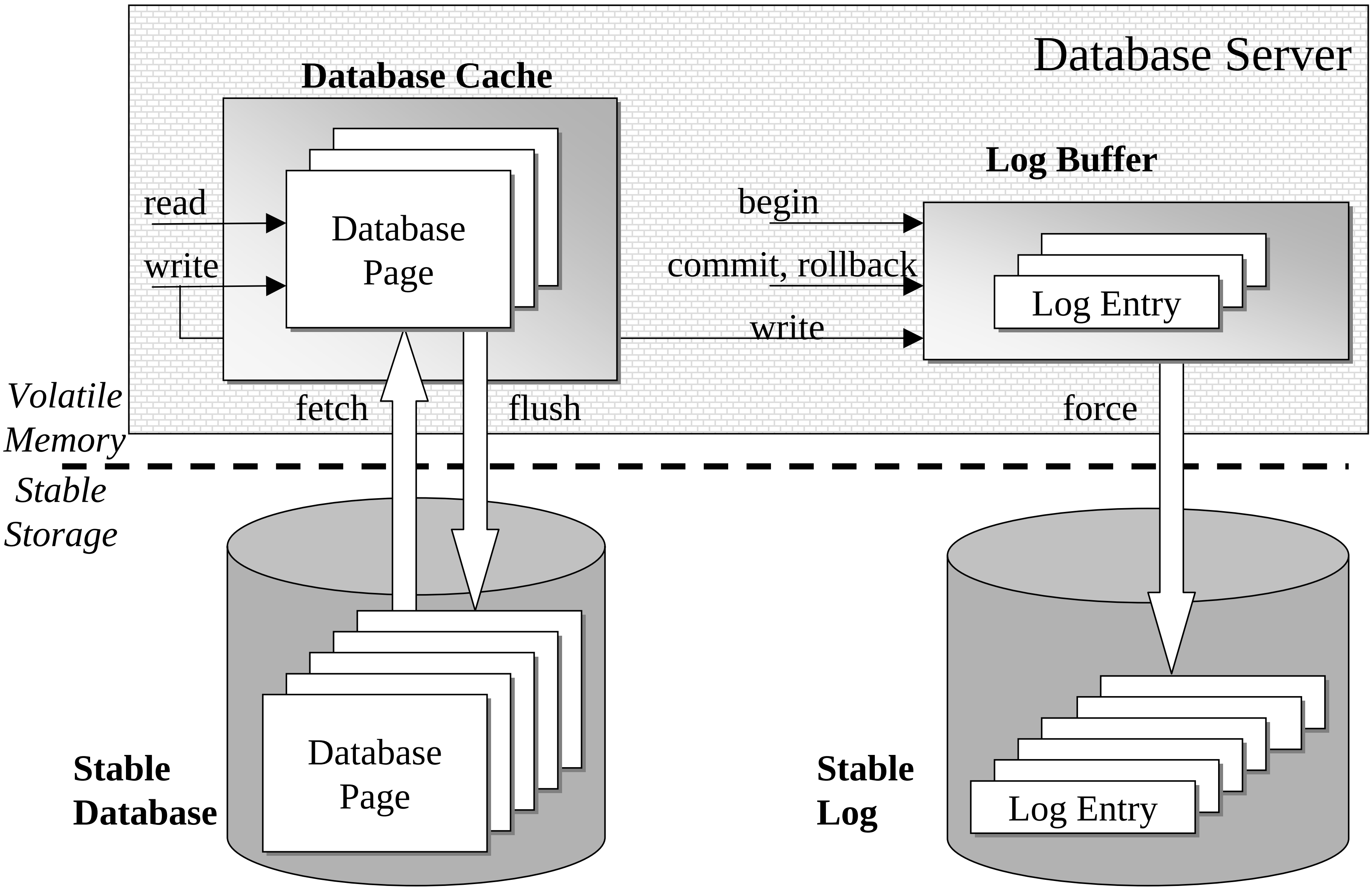
- soft (no damage to secondary storage)
- fail-stop (no unbounded failure propagation)

captures most (server) software failures,
both Bohrbugs and Heisenbugs

**Requirements:**

- fast restart for high availability (= MTTF / (MTTF + MTTR)
- low overhead during normal operation
- simplicity, testability, very high confidence in correctness

# Overview of System Architecture



# Overview of Simple Three-Pass Algorithm

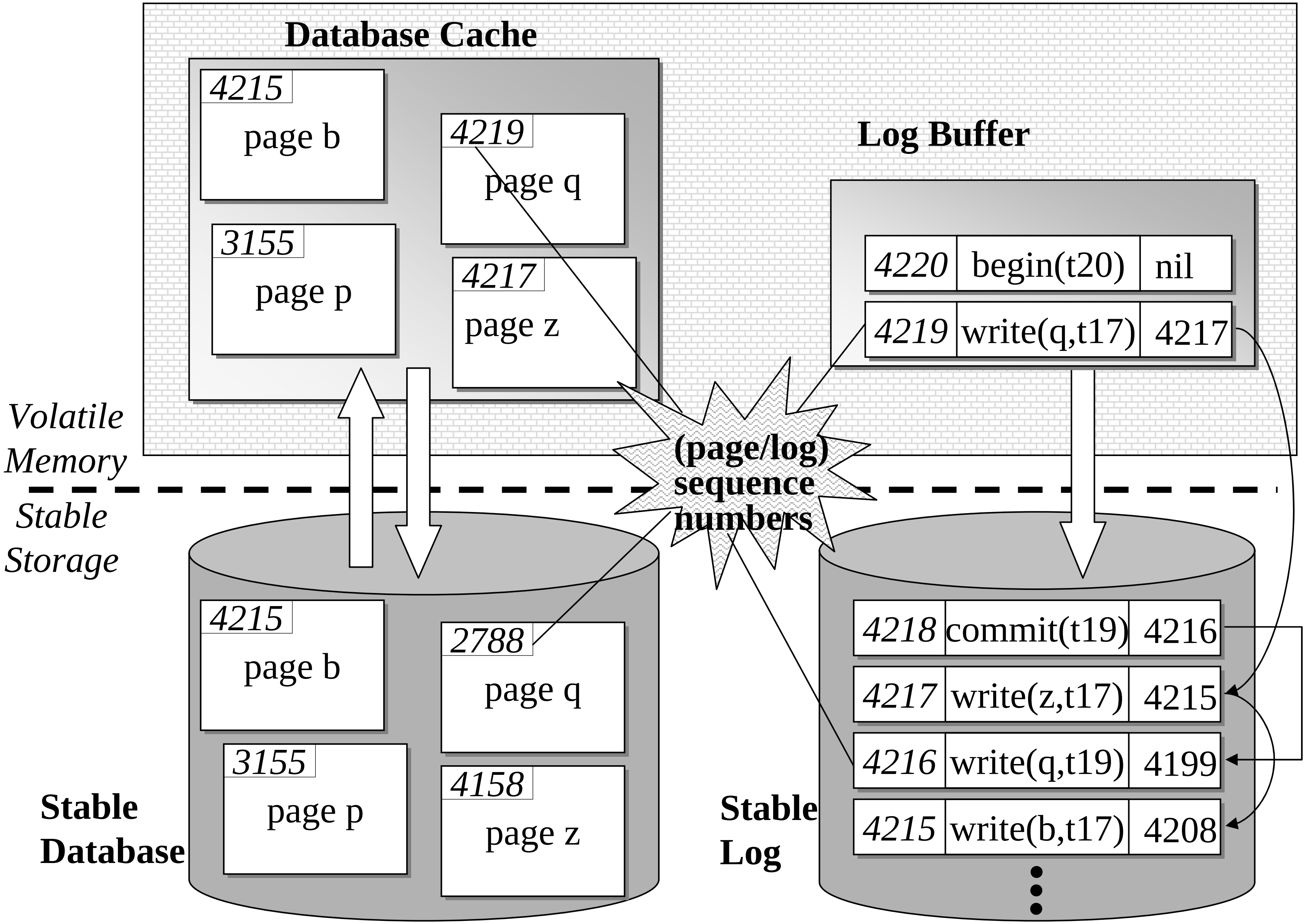
- **Analysis pass:**
  determine start of stable log from master record
  perform forward scan
  to determine winner and loser transactions
- **Redo pass:**
  perform forward scan
  to redo all winner actions in chronological (LSN) order
  (until end of log is reached)
- **Undo pass:**
  perform backward scan
  to traverse all loser log entries in reverse chronological order
  and undo the corresponding actions

# Incorporating General Writes As Physiological Log Entries

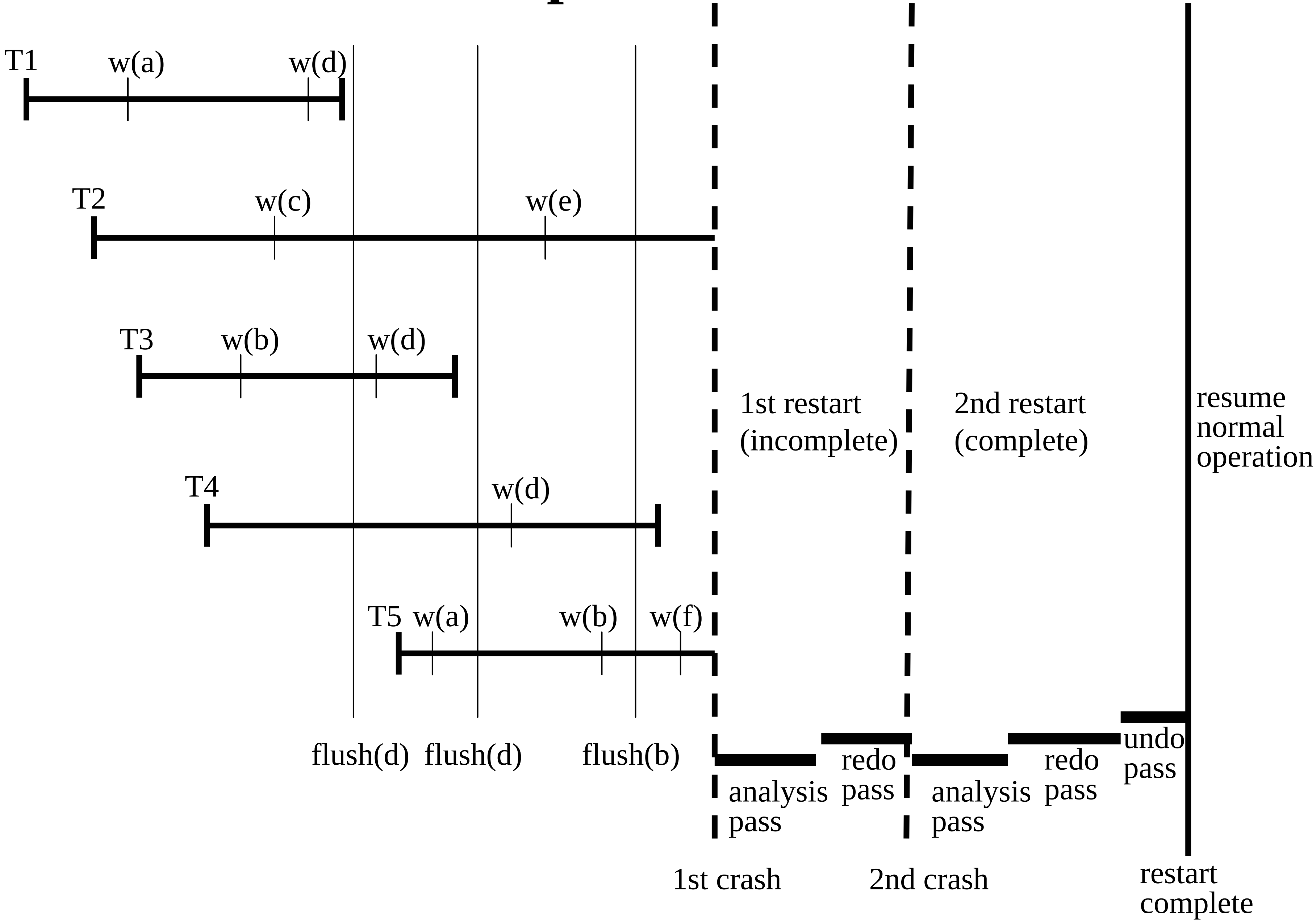
Principle:

- state testing during the redo pass:
  for log entry for page p with log sequence number i,
  redo write only if i > p.PageSeqNo
  and subsequently set p.PageSeqNo := i
- state testing during the undo pass:
  for log entry for page p with log sequence number i,
  undo write only if i ≤ p.PageSeqNo
  and subsequently set p.PageSeqNo := i-1

# Usage of (Log) Sequence Numbers



# Example Scenario



# Example under Simple Three-Pass Algorithm with General Writes

| Sequence number: action | Change of cached database [PageNo: SeqNo] | Change of stable database [PageNo: SeqNo] | Log entry added to log buffer [LogSeqNo: action] | Log entries added to stable log [LogSeqNo's] |
|---|---|---|---|---|
| 1:begin(T1) | | | 1: begin(T1) | |
| 2: begin(T2) | | | 2: begin(T2) | |
| 3: write(a,T1) | a: 3 | | 3: write(a,T1) | |
| 4: begin(T3) | | | 4: begin(T3) | |
| 5: begin(T4) | | | 5: begin(T4) | |
| 6: write(b,T3) | b: 6 | | 6: write(b,T3) | |
| 7: write (c,T2) | c: 7 | | 7: write(c,T2) | |
| 8: write(d,T1) | d: 8 | | 8: write(d,T1) | |
| 9: commit(T1) | | | 9: commit(T1) | 1,2,3,4,5,6,7,8,9 |
| 10: flush(d) | | d:8 | | |
| 11: write(d,T3) | d: 11 | | 11: write(d,T3) | |
| 12: begin(T5) | | | 12: begin(T5) | |
| 13: write(a,T5) | a: 13 | | 13: write(a,T5) | |
| 14: commit(T3) | | | 14: commit(T3) | 11,12,13,14 |
| 15: flush(d) | | d: 11 | | |
| 16: write(d,T4) | d: 16 | | 16: write(d,T4) | |
| 17: write(e,T2) | e: 17 | | 17: write(e,T2) | |
| 18: write(b,T5) | b: 18 | | 18: write(b,T5) | |
| 19: flush(b) | | b: 18 | | 16,17,18 |
| 20: commit(T4) | | | 20: commit(T4) | 20 |
| 21: write(f,T5) | f: 21 | | 21: write(f,T5) | |
| system crash | | | | |

| restart | | | | |
|---|---|---|---|---|
| analysis pass: losers = {T2,T5} | | | | |
| redo(3) | a: 3 | | | |
| consider-redo(6) | b: 18 | | | |
| flush (a) | | a: 3 | | |
| consider-redo(8) | d: 11 | | | |
| consider-redo(11) | d: 11 | | | |
| second system crash | | | | |

| second restart | | | | |
|---|---|---|---|---|
| analysis pass: losers = {T2,T5} | | | | |
| consider-redo(3) | a:3 | | | |
| consider-redo(6) | b: 18 | | | |
| consider-redo(8) | d: 11 | | | |
| consider-redo(11) | d: 11 | | | |
| redo(16) | d: 16 | | | |
| undo(18) | b: 17 | | | |
| consider-undo(17) | e: 0 | | | |
| consider-undo(13) | a: 3 | | | |
| consider-undo(7) | c: 0 | | | |
| second restart complete: resume normal operation | | | | |

# Need and Opportunity for Log Truncation

Major cost factors and potential availability bottlenecks:

1) analysis pass and redo pass scan entire log
2) redo pass performs many random I/Os on stable database

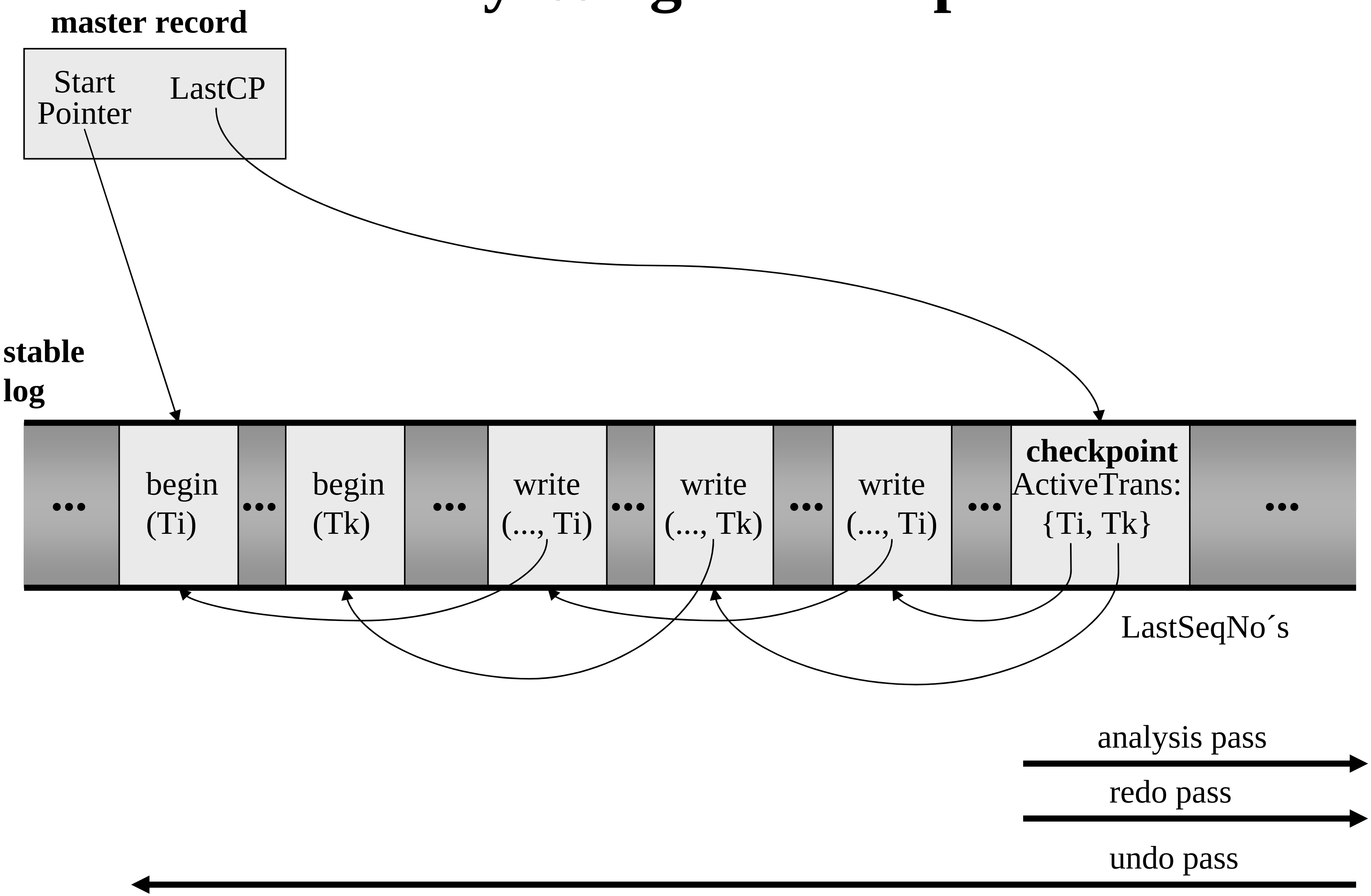
Improvement:

continuously advance the log start pointer (garbage collection)

- for redo, can drop all log entries for page p that
  precede the last flush action for p =: RedoLSN (p);
  min{RedoLSN (p) | dirty page p} =: SystemRedoLSN
- for undo, can drop all log entries that
  precede the oldest log entry of a potential loser =: OldestUndoLSN

*Remarks:*
*for full-writes, all but the most recent after-image can be dropped*
*log truncation after complete undo pass requires global flush*

# Heavy-Weight Checkpoints



# Dirty Page List for Redo Optimization

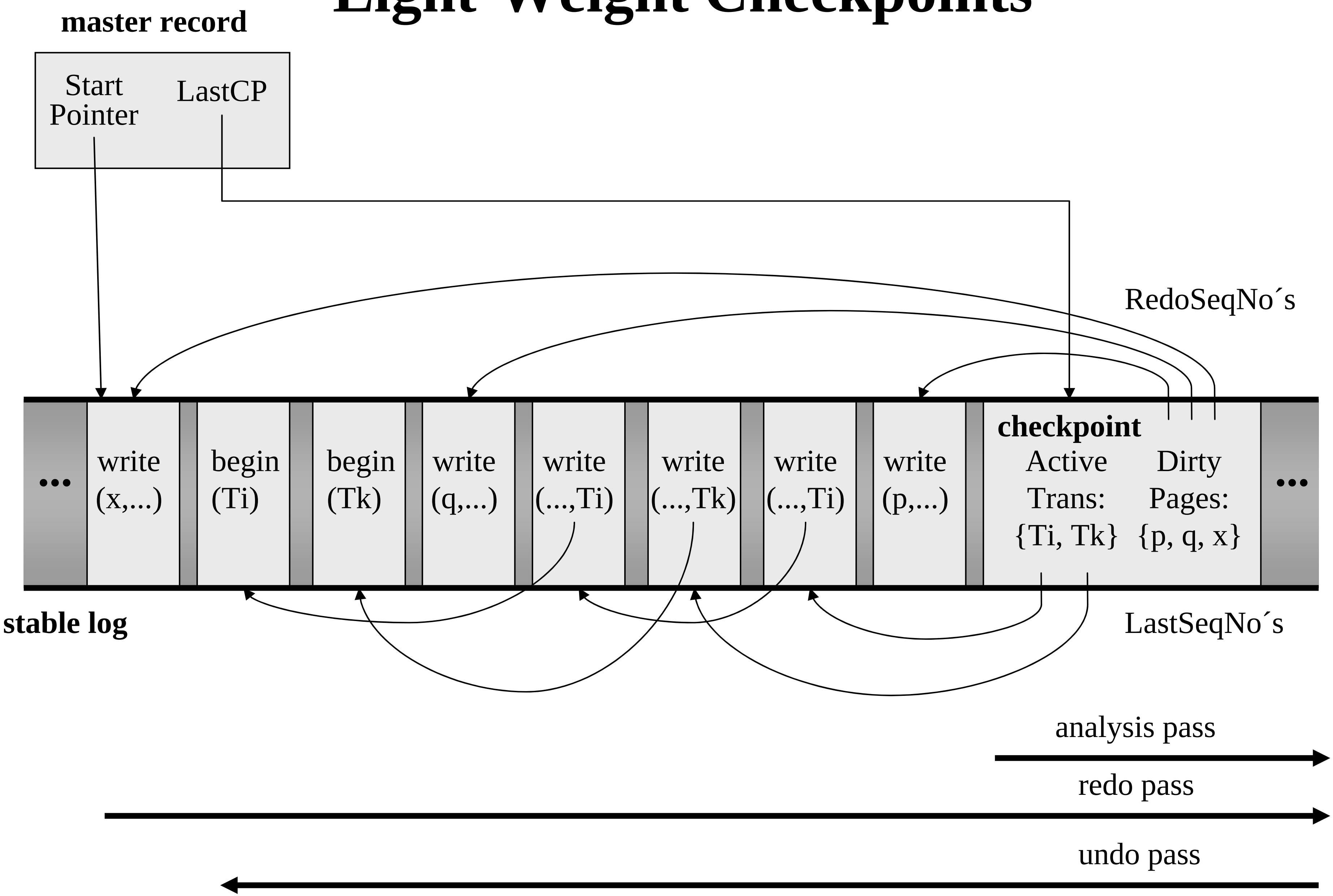
Keep track of
- the set of dirty cached pages
- for each such page the sequence number of the oldest write action that followed the most recent flush action (redo sequence numbers)

Avoid very old RedoSeqNo's by write-behind demon

```
type DirtyPageListEntry: record of
        PageNo: identifier;
        RedoSeqNo: identifier;
     end;
var DirtyPages:
        set of DirtyPageListEntry indexed by PageNo;
```

Record dirty page list in checkpoint log entry and reconstruct (conservative approximation of) dirty page list during analysis pass
→ exploit knowledge of dirty page list and redo sequence numbers for I/O optimizations during redo

# Light-Weight Checkpoints



# Example with Optimizations

| Sequence number: action | Change of cached database [PageNo: SeqNo] | Change of stable database [PageNo: SeqNo] | Log entry added to log buffer [LogSeqNo: action] | Log entries added to stable log [LogSeqNo's] |
|---|---|---|---|---|
| 1:begin(T1) | | | 1: begin(T1) | |
| 2: begin(T2) | | | 2: begin(T2) | |
| 3: write(a,T1) | a: 3 | | 3: write(a,T1) | |
| 4: begin(T3) | | | 4: begin(T3) | |
| 5: begin(T4) | | | 5: begin(T4) | |
| 6: write(b,T3) | b: 6 | | 6: write(b,T3) | |
| 7: write (c,T2) | c: 7 | | 7: write(c,T2) | |
| 8: write(d,T1) | d: 8 | | 8: write(d,T1) | |
| 9: commit(T1) | | | 9: commit(T1) | 1,2,3,4,5,6,7,8,9 |
| 10: flush(d) | | d:8 | **10: flush(d)** | |
| 11: write(d,T3) | d: 11 | | 11: write(d,T3) | |
| 12: begin(T5) | | | 12: begin(T5) | |
| 13: write(a,T5) | a: 13 | | 13: write(a,T5) | |
| **14: checkpoint** | | | 14: CP DirtyPages: {a,b,c,d} RedoLSNs: a:3, b:6, c:7, d:11 ActiveTrans: {T2,T3,T4,T5} | 10,11,12,13,14 |
| 15: commit(T3) | | | 15: commit(T3) | 15 |
| 16: flush(d) | | d: 11 | **16: flush(d)** | |
| 17: write(d,T4) | d: 17 | | 17: write(d,T4) | |
| 18: write(e,T2) | e: 18 | | 18: write(e,T2) | |
| 19: write(b,T5) | b: 19 | | 19: write(b,T5) | |
| 20: flush(b) | | b: 19 | **20: flush(b)** | 16,17,18,19 |
| 21: commit(T4) | | | 21: commit(T4) | 20,21 |
| 22: write(f,T5) | f: 22 | | 22: write(f,T5) | |
| system crash | | | | |

| restart | | | | |
|---|---|---|---|---|
| analysis pass: losers = {T2,T5}<br>**DirtyPages = {a,c,d,e,f}**<br>**RedoLSNs: a:3, c:7, d:17, e:18** | | | | |
| redo(3) | a:3 | | | |
| consider-redo(6) | b: 19 | | | |
| skip-redo(8) | | | | |
| **skip-redo(11)** | | | | |
| redo(17) | d:17 | | | |
| undo(19) | b: 18 | | | |
| consider-undo(18) | e: 0 | | | |
| consider-undo(13) | a: 3 | | | |
| consider-undo(7) | c: 0 | | | |
| restart complete: resume normal operation | | | | |

# Pseudocode: Data Structures (1)

```
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, compensate, checkpoint, flush);
        ActiveTrans: set of TransInfo;
        DirtyPages: set of DirtyPageInfo;
        UndoInfo: array of char;
        RedoInfo: array of char;
        PreviousSeqNo: identifier;
        NextUndoSeqNo: identifier;
     end;
```

# Pseudocode: Data Structures (2)

```
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;
typeDirtyPageInfo: record of
        PageNo: identifier;
        RedoSeqNo: identifier;
        end;
var DirtyPages:
        set of DirtyPageInfo indexed by PageNo;
```

# Pseudocode: Actions During Normal Operation (1)

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

# Pseudocode: Actions During Normal Operation (2)

```
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;
   newlogentry.LogSeqNo := next sequence number;
   newlogentry.ActionType := flush;
   newlogentry.PageNo := pageno;
   LogBuffer += newlogentry;
   DirtyPages -= pageno;
```

# Pseudocode: Actions During Normal Operation (3)

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

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

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

# Pseudocode: Actions During Normal Operation (4)

```
abort (transid):
   logentry :=
         ActiveTrans[transid].LastSeqNo;
   while logentry is not nil and
         logentry.ActionType = write or full-write
   do
      newlogentry.LogSeqNo := new sequence number;
      newlogentry.ActionType := compensation;
      newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
      newlogentry.RedoInfo :=
          inverse action of the action in logentry;
      newlogentry.NextUndoSeqNo := logentry.PreviousSeqNo;
      ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
      LogBuffer += newlogentry;
      write (logentry.PageNo) according to logentry.UndoInfo;
      logentry := logentry.PreviousSeqNo;
   end /*while*/
   newlogentry.LogSeqNo := new sequence number;
   newlogentry.ActionType := rollback;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   newlogentry.NextUndoSeqNo := nil;
   LogBuffer += newlogentry;
   ActiveTrans -= transid;
   force ( );
```

# Pseudocode: Actions During Normal Operation (5)

```
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 large enough
          and SystemRedoLSN < OldestUndoLSN
    do
        flush (OldestRedoPage);
        SystemRedoLSN := min{DatabaseCache[p].RedoLSN};
        OldestRedoPage := page p such that
              DatabaseCache[p].RedoLSN = SystemRedoLSN;
        NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    end /*while*/;
    MasterRecord.StartPointer := NewStartPointer;

checkpoint ( ):
    logentry.ActionType := checkpoint;
    logentry.ActiveTrans := ActiveTrans (as maintained in memory);
    logentry.DirtyPages := DirtyPages (as maintained in memory);
    logentry.LogSeqNo := next sequence number to be generated;
    LogBuffer += logentry;
    force ( ); MasterRecord.LastCP := logentry.LogSeqNo;
```

# Pseudocode: Recovery Procedure (1)

```
restart ( ):
   analysis pass ( ) returns losers, DirtyPages;
   redo pass ( );
   undo pass ( );
```

# Pseudocode: Recovery Procedure (2)

```
analysis pass ( ) returns losers, DirtyPages:
   var losers: set of record
          TransId: identifier; LastSeqNo: identifier;
       end indexed by TransId;
   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 or compensat
           and StableLog[i].PageNo not in DirtyPages
     then
           DirtyPages += StableLog[i].PageNo;
           DirtyPages[StableLog[i].PageNo].RedoSeqNo := i;
     end /*if*/;
     if StableLog[i].ActionType = flush
     then DirtyPages -= StableLog[i].PageNo; end /*if*/;
   end /*for*/;
```

# Pseudocode: Recovery Procedure (3)

```
redo pass ( ):
   SystemRedoLSN := min {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 or compensate
          then
             pageno = StableLog[i].PageNo;
             if pageno in DirtyPages and
                DirtyPages[pageno].RedoSeqNo < i
             then
                fetch (pageno);
                if DatabaseCache[pageno].PageSeqNo < i
                then
                   read and write (pageno)
                        according to StableLog[i].RedoInfo;
                   DatabaseCache[pageno].PageSeqNo := i;
                end /*if*/;
             end /*if*/;
          end /*if*/;
   end /*for*/;
```

# Pseudocode: Recovery Procedure (4)

```
undo pass ( ):
   ActiveTrans := empty;
   for each t in losers
   do
         ActiveTrans += t;
         ActiveTrans[t].LastSeqNo := losers[t].LastSeqNo;
   end /*for*/;
   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 = compensation
       then
          losers[nexttrans].LastSeqNo :=
             StableLog[nextentry].NextUndoSeqNo;
       end /*if*/;
```

# Pseudocode: Recovery Procedure (5)

```
if StableLog[nextentry].ActionType = write or full-write
    then
       pageno = StableLog[nextentry].PageNo;
       fetch (pageno);
       if DatabaseCache[pageno].PageSeqNo >= nextentry.LogSeqNo
       then
          newlogentry.LogSeqNo := new sequence number;
          newlogentry.ActionType := compensation;
          newlogentry.PreviousSeqNo :=
             ActiveTrans[transid].LastSeqNo;
          newlogentry.NextUndoSeqNo := nextentry.PreviousSeqNo;
          newlogentry.RedoInfo :=
               inverse action of the action in nextentry;
          ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
          LogBuffer += newlogentry;
          read and write (StableLog[nextentry].PageNo)
               according to StableLog[nextentry].UndoInfo;
          DatabaseCache[pageno].PageSeqNo := newlogentry.LogSeqNo;
      end /*if*/;
      losers[nexttrans].LastSeqNo =
            StableLog[nextentry].PreviousSeqNo;
    end /*if*/;
```

# Pseudocode: Recovery Procedure (6)

```
if StableLog[nextentry].ActionType = begin
       then
           newlogentry.LogSeqNo := new sequence number;
           newlogentry.ActionType := rollback;
           newlogentry.TransId := StableLog[nextentry].TransId;
           newlogentry.PreviousSeqNo :=
              ActiveTrans[transid].LastSeqNo;
           LogBuffer += newlogentry;
           ActiveTrans -= transid;
           losers -= transid;
       end /*if*/;

   end /*while*/;
   force ( );
```

# Fundamental Problem of Distributed Commit

**Problem:**

- Transaction operates on multiple servers (**resource managers**)
- Global commit needs unanimous local commits of all **participants (agents)**
- Distributed system may fail partially (server crashes, network failures) and creates the potential danger of inconsistent decisions

**Approach:**

- Distributed handshake protocol known as **two-phase commit (2PC)**
- with a **coordinator** taking responsibility for unanimous outcome
- Recovery considerations for in-doubt transactions

# 2PC During Normal Operation

- **First phase (voting):**
  coordinator sends *prepare* messages to participants
  and waits for *yes* or *no* votes
- **Second phase (decision)**
  coordinator sends *commit* or *rollback* messages to participants
  and waits for *ack*s
- **Participants** write *prepared* log entries in voting phase
  and become *in-doubt (uncertain)*
  → potential **blocking** danger, breach of local autonomy
- Participants write commit or rollback log entry in decision phase
- **Coordinator** writes *begin* log entry
- Coordinator writes *commit* or *rollback* log entry
  and can now give return code to the client´s commit request
- Coordinator writes *end (done, forgotten)* log entry
  to facilitate **garbage collection**

→ 4n messages, 2n+2 forced log writes, 1 unforced log write
with n participants and 1 coordinator

# Illustration of 2PC

*Coordinator* | *Participant 1* | *Participant 2*

force-write
begin log entry

send “prepare” (Coordinator → Participant 1)

send “prepare” (Coordinator → Participant 2)

force-write
prepared log entry (Participant 1)

force-write
prepared log entry (Participant 2)

send “yes” (Participant 1 → Coordinator)

send “yes” (Participant 2 → Coordinator)

force-write
commit log entry

send “commit” (Coordinator → Participant 1)

send “commit” (Coordinator → Participant 2)

force-write
commit log entry (Participant 1)

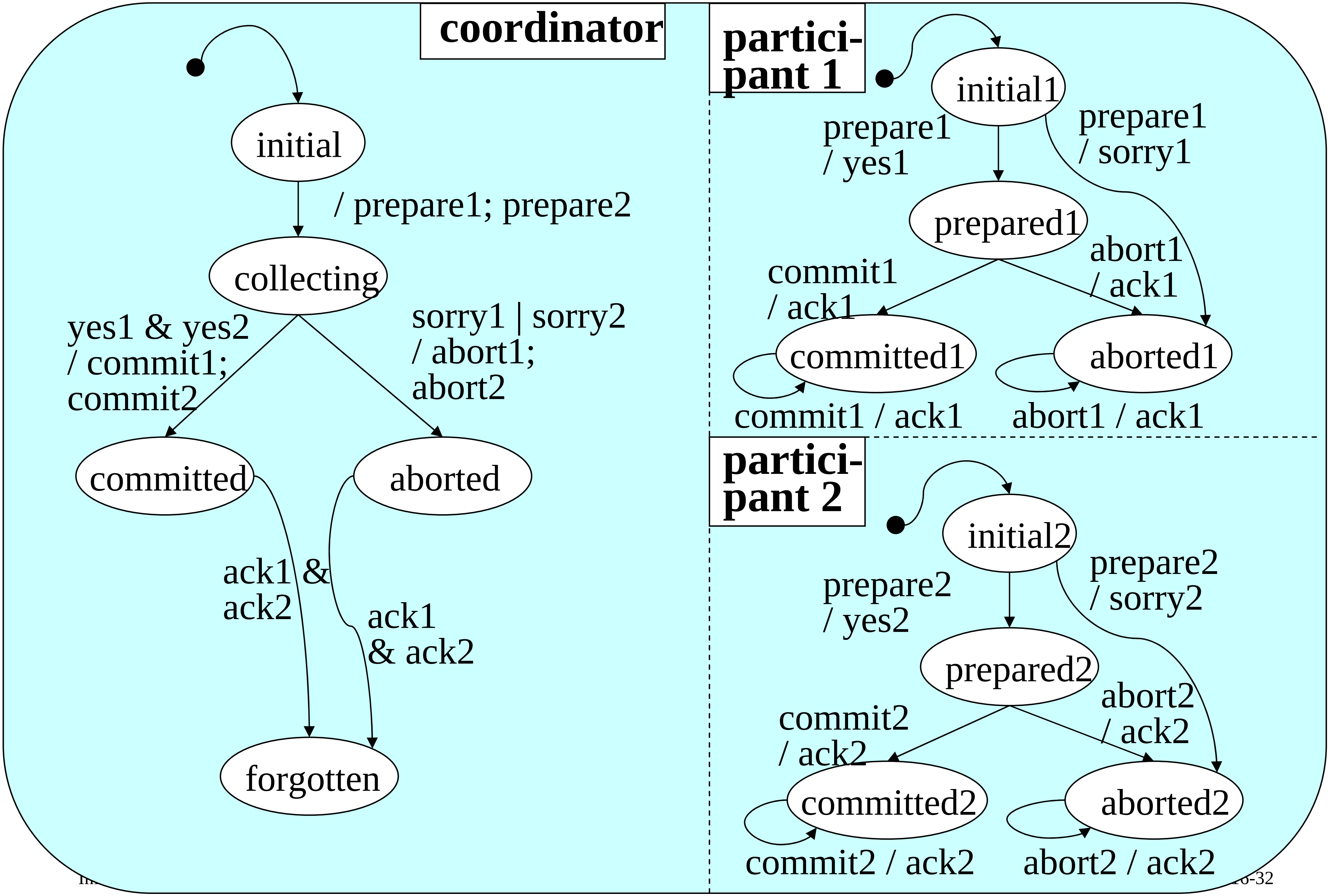
force-write
commit log entry (Participant 2)

send “ack” (Participant 1 → Coordinator)

send “ack” (Participant 2 → Coordinator)

write
end log entry

# Statechart for Basic 2PC



# Restart and Termination Protocol

## Failure model:

- process failures: transient server crashes
- network failures: message losses, message duplications
- assumption that there are no malicious commission failures
  $\rightarrow$ Byzantine agreement
- no assumptions about network failure handling
  $\rightarrow$ can use datagrams or sessions for communication

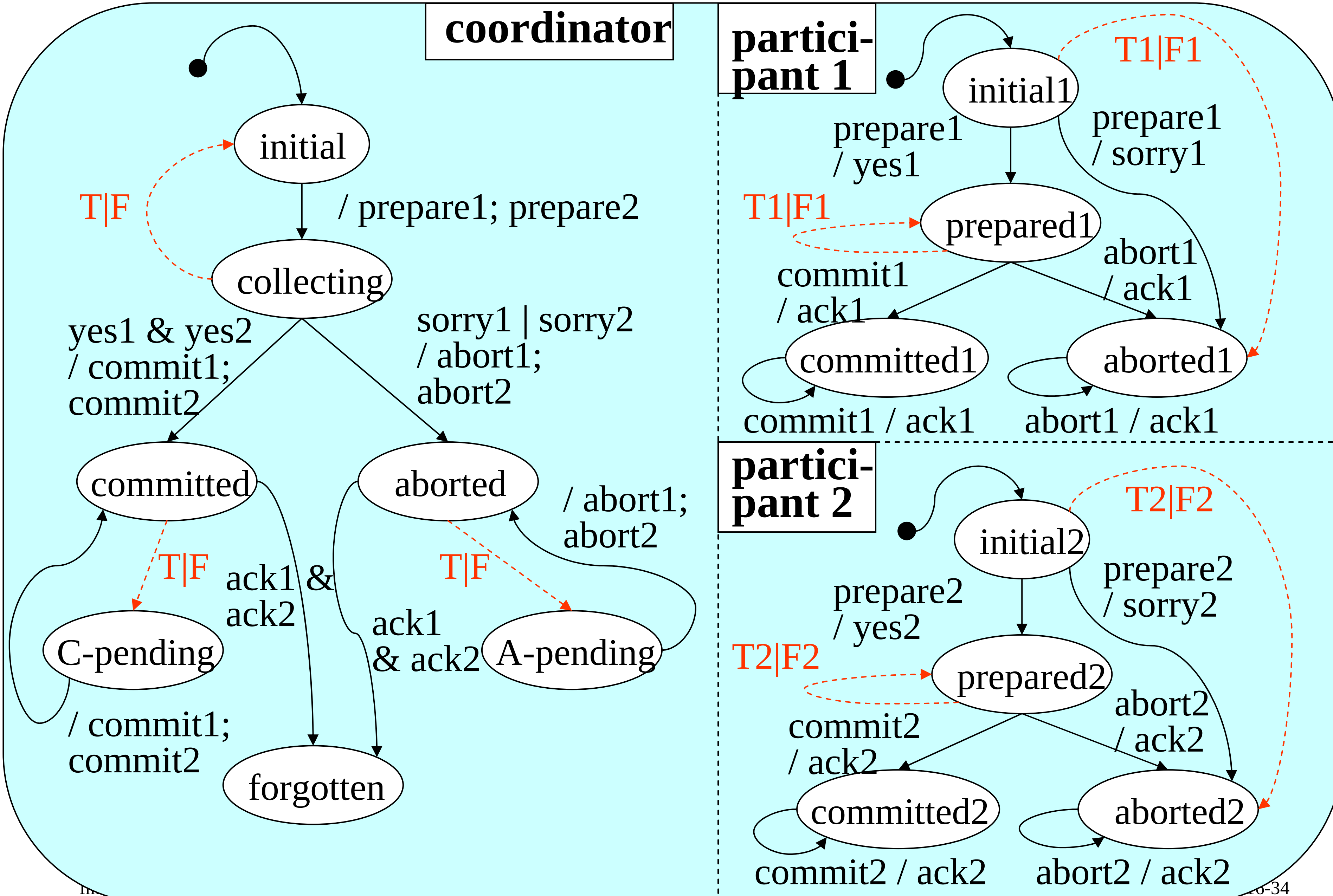
## Restart protocol after failure (F transitions):

- coordinator restarts in last remembered state and resends messages
- participant restarts in last remembered state and resends message or waits for message from coordinator

## Termination protocol upon timeout (T transitions):

- coordinator resends messages and may decide to abort the transaction in first phase
- participant can unilaterally abort in first phase and wait for or may contact coordinator in second phase

# Statechart for Basic 2PC with Restart/Termination



# Correctness of Basic 2PC

**Theorem 19.1 (Safety):**
2PC guarantees that if one process is in a final state, then either all processes are in their committed state or all processes are in their aborted state.

**Proof methodology:**
Consider the set of possible computation paths starting in global state (initial, initial, ..., initial) and reason about invariants for states on computation paths.

**Theorem 19.2 (Liveness):**
For a finite number of failures the 2PC protocol will eventually reach a final global state within a finite number of state transitions.

# Independent Recovery

*Independent recovery:* ability of a failed and restarted process to terminate his part of the protocol without communicating to other processes.

**Theorem:**
There exists no distributed commit protocol that can guarantee independent process recovery in the presence of multiple failures (e.g., network partitionings).