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: Continuous Availability

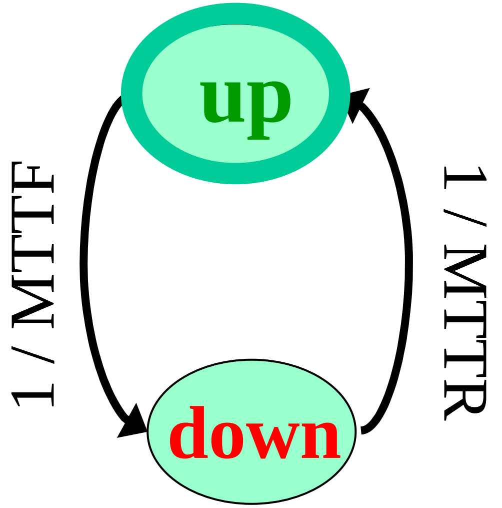
Business apps and e-services demand 24 x 7 availability

99.999 availability would be acceptable (5 min outage/year)

Downtime costs (per hour):

Brokerage operations: $ 6.4 Mio.
Credit card authorization: $ 2.6 Mio.
Ebay: $ 225 000
Amazon: $ 180 000
Airline reservation: $ 89 000
Cell phone activation: $ 41 000

(Source: Internet Week 4/3/2000)



Stationary availability =

$$\frac{MTTF}{MTTF + MTTR}$$

State of the art:

DB servers ≈ 99.99 to 99.999 %

Internet e-service (e.g., Ebay) ≈ 99 %

Heisenbugs and the Recovery Rationale

Failure causes:

- power: 2 000 h MTTF or ∞ with UP
- chips: 100 000 h MTTF
- system software: 200 – 1 000 h MTT
- telecomm lines: 4 000 h MTTF
- admin error: ??? or → ∞ with auto-a
- disks: 800 000 h MTTF
- environment: > 20 000 h MTTF

Transient software failures are the main problem:
Heisenbugs
(non-repeatable exceptions caused by stochastic confluence of very rare events)

→ **Failure model** for crash recovery:

- ***fail-stop*** (no dynamic salvation/resurrection code)
- ***soft failures*** (stable storage survives)

→ **„Self-healing“ recovery =**
amnesia + data repair (from „trusted“ store) + re-init

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)

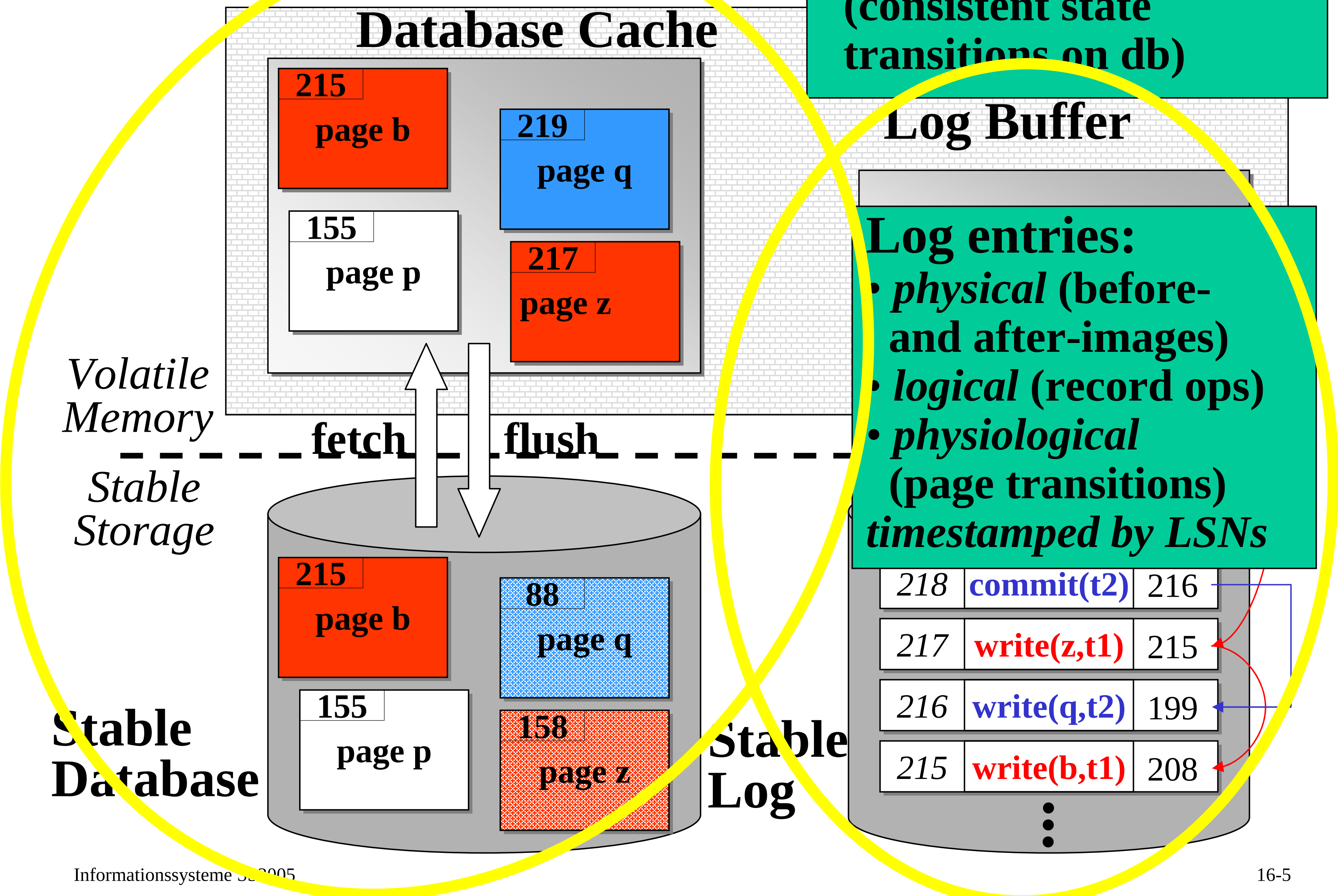
captures most (server) software failures,
both Bohrbugs and Heisenbugs

Requirements:

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

Why Exactly is Recov

Atomic transactions
(consistent state transitions on db)

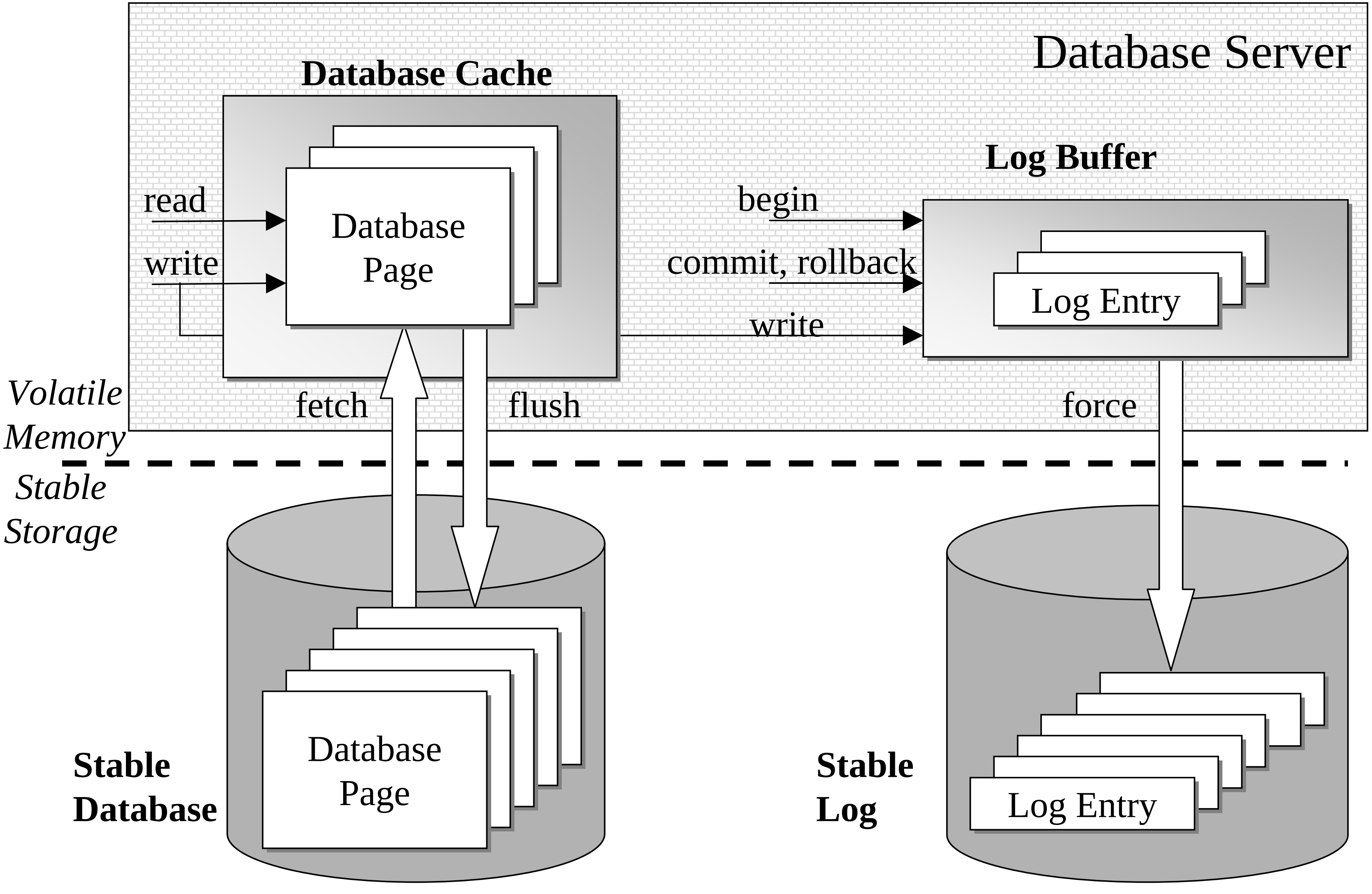


218	**commit(t2)**	216
217	**write(z,t1)**	215
216	**write(q,t2)**	199
215	**write(b,t1)**	208

Log entries:
- ***physical*** **(before- and after-images)**
- ***logical*** **(record ops)**
- ***physiological*** **(page transitions)**

timestamped by LSNs

Overview of System Architecture



How Does Recovery Work?

- **Analysis pass**: determine winners vs. losers (by scanning the stable log)
- **Redo pass**: redo winner writes (b
- **Undo pass**: undo loser writes (by

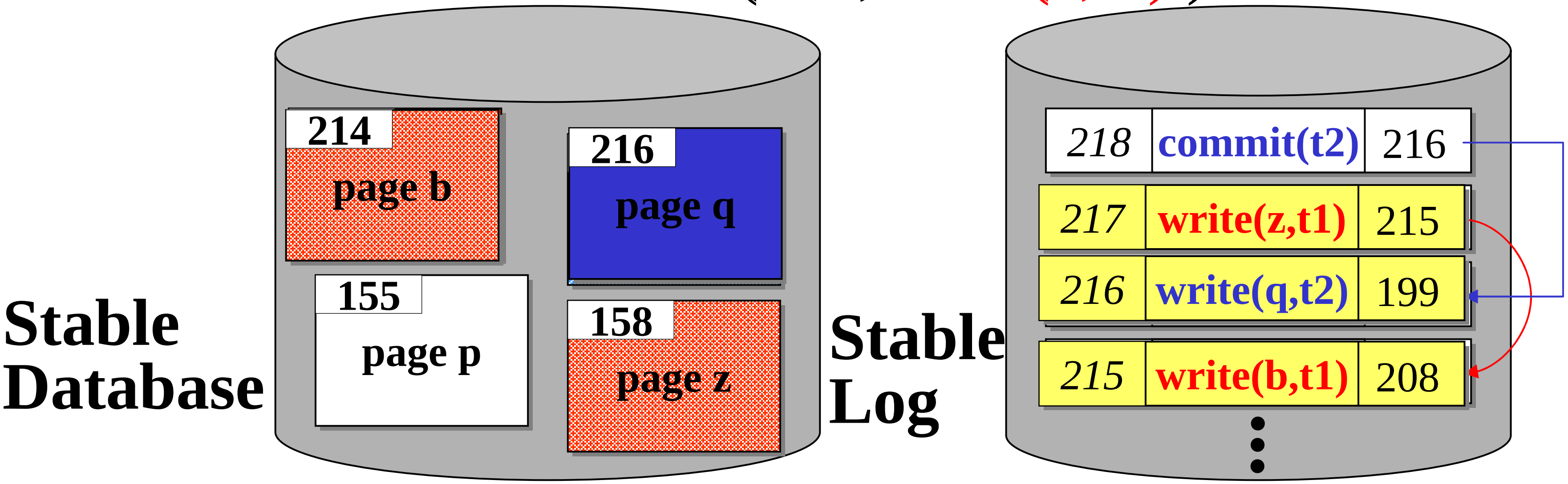
LSN in page header implements testable state

losers: {t1}, winners: {t2}

redo(216, write(q,t2))

undo (217, write(z,t1)) ?

undo (215, write(b,t1))



218	commit(t2)	216
217	write(z,t1)	215
216	write(q,t2)	199
215	write(b,t1)	208

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

Log Operations in Normal Mode

Goals:

- Avoid random writes to stable log, try to batch writes
- Guarantee entry in stable log to undo change of stable database by potential loser transaction
- Guarentee entry in stable log to redo change by winner transaction

Solution: Flush log buffer when

- Dirty page is flushed from DB cache to stable DB
- Transaction commits

Incorporating General Writes As Physiological Log Entries

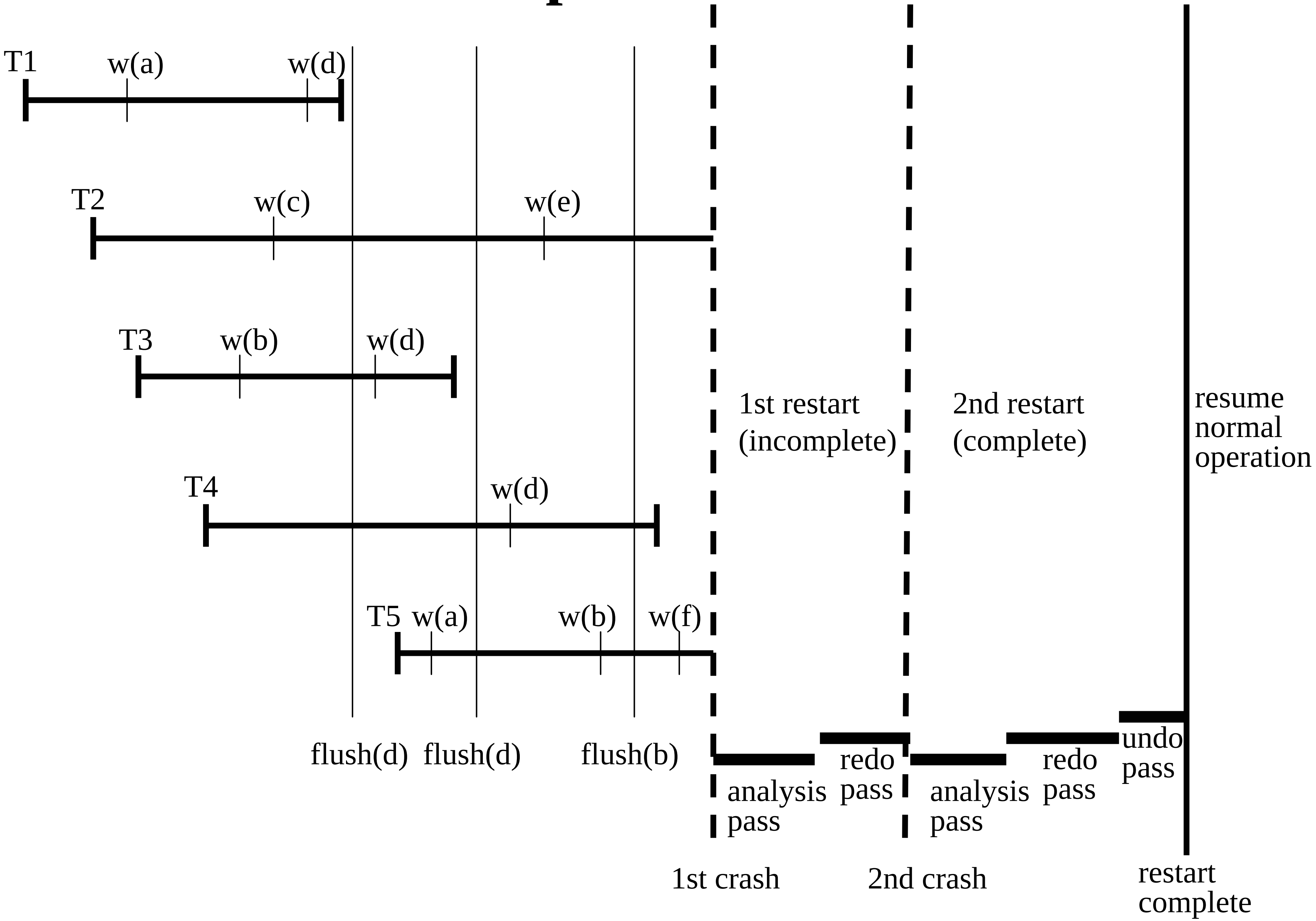
Principle:

- 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

What Do We Need to Optimize for?

- **Fast restart** (for high **availability**) by bounding the log and minimizing page fetches
- **Low overhead** during normal operation by minimizing forced log writes (and page flushes)
- High transaction concurrency during normal operation
- Correctness (and simplicity) in the presence of many subtleties

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				

Data Structures for Logging & Recovery

```
type Page: record of
   PageNo: id; PageSeqNo: id; Status: (clean, dirty);
   Contents: array [PageSize] of char; end;
persistent var StableDatabase: set[PageNo] of Page;
var DatabaseCache: set[PageNo] of Page;
type LogEntry: record of
   LogSeqNo: id; TransId: id; PageNo: id;
   ActionType:(write, full-write, begin, commit, rollback);
   UndoInfo: array of char; RedoInfo: array of char;
   PreviousSeqNo: id; end;
persistent var StableLog: list[LogSeqNo] of LogEntry;
var LogBuffer: list[LogSeqNo] of LogEntry;
```

modeled in functional manner with test *op* $\in$ *state*:

write s on page p $\in$ StableDatabase $\Leftrightarrow$ StableDatabase[p].PageSeqNo $\geq$ s

write s on page p $\in$ CachedDatabase $\Leftrightarrow$
((p $\in$ DatabaseCache $\wedge$ DatabaseCache[p].PageSeqNo $\geq$ s) $\vee$
(p $\notin$ DatabaseCache $\wedge$ StableDatabase[p].PageSeqNo $\geq$ s))

Correctness Criterion

A crash recovery algorithm is **correct**
if it guarantees that, after a system failure,
the **cached database** will eventually be **equivalent** to a
serial order of the **committed transactions**
that coincides with the **serialization order**
of the schedule.

Simple Redo Pass

```
redo pass ( ):
min := LogSeqNo of oldest log entry in StableLog;
max := LogSeqNo of most recent log entry in StableLog;
for i := min to max do
  if StableLog[i].TransId not in losers then
     pageno = StableLog[i].PageNo; fetch (pageno);
     case StableLog[i].ActionType of
        full-write:
           full-write (pageno) with contents
              from StableLog[i].RedoInfo;
        write:
           if DatabaseCache[pageno].PageSeqNo < i then
              read and write (pageno)
                 according to StableLog[i].RedoInfo;
              DatabaseCache[pageno].PageSeqNo := i;
           end /*if*/;
       end /*case*/;
end /*for*/;
```

Correctness of Simple Redo & Undo

Invariants during redo pass (compatible with serialization):

$$\forall s \in StableLog: (\ all\ s' \leq s\ \ have\ been\ \ processed$$

$$\Rightarrow (\ \forall\ pages\ p\ \forall trans\ t\ \forall o \in StableLog:$$

$$(o.transid = t \wedge o.pageid = p \wedge t \in winners \wedge o \leq s)$$

$$\Rightarrow o \in CachedDb\ \ ))$$

$$\forall o \in CachedDb\ \ : o \in StableLog\ \ \vee\ o \in StableDb$$

Invariant of undo pass:

$$\forall s \in StableLog:$$

$$(\ all\ s' \in StableLog\big|_{losers}\ with\ s' \geq s\ have\ been\ \ processed$$

$$\Rightarrow (\ \forall\ pages\ p\ \forall trans\ \ t\ \forall o \in StableLog:$$

$$(o.transid = t \wedge o.pageid = p \wedge t \in losers \wedge o \geq s)$$

$$\Rightarrow o \notin CachedDb\ \ ))$$

Need and Opportunity for Log Truncation

Major cost factors and potential availability bottlenecks:

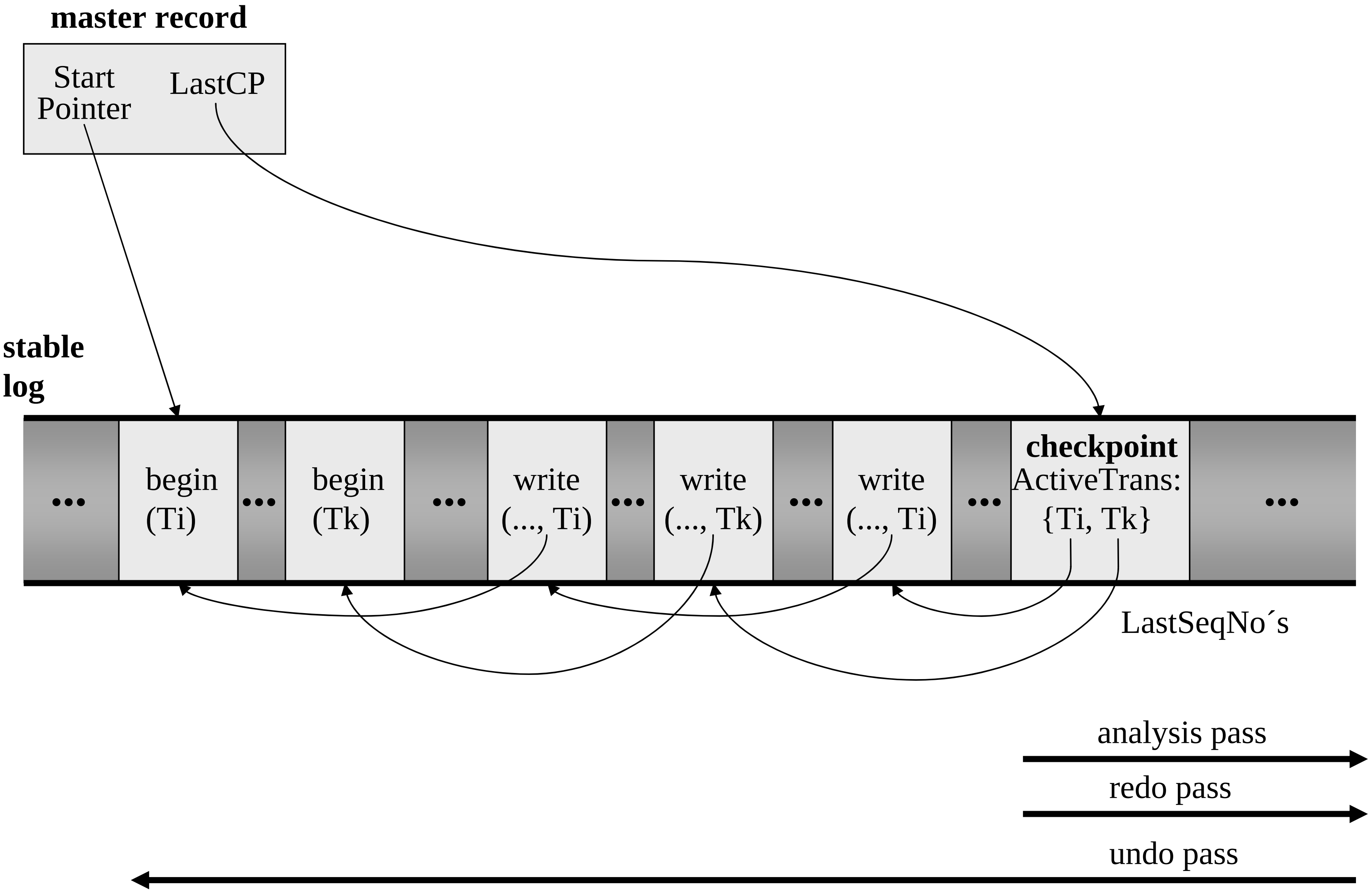
1) analysis pass and redo pass scan entire log
2) redo pass performs many random I/Os on stable database

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

Simplistic Approach: Heavy-Weight Checkpoints



master record

Start Pointer

LastCP

stable log

...

begin (Ti)

...

begin (Tk)

...

write (..., Ti)

...

write (..., Tk)

...

write (..., Ti)

...

checkpoint
ActiveTrans: {Ti, Tk}

...

LastSeqNo´s

analysis pass

redo pass

undo pass

Redo Optimization 1: Light-Weight Checkpoints

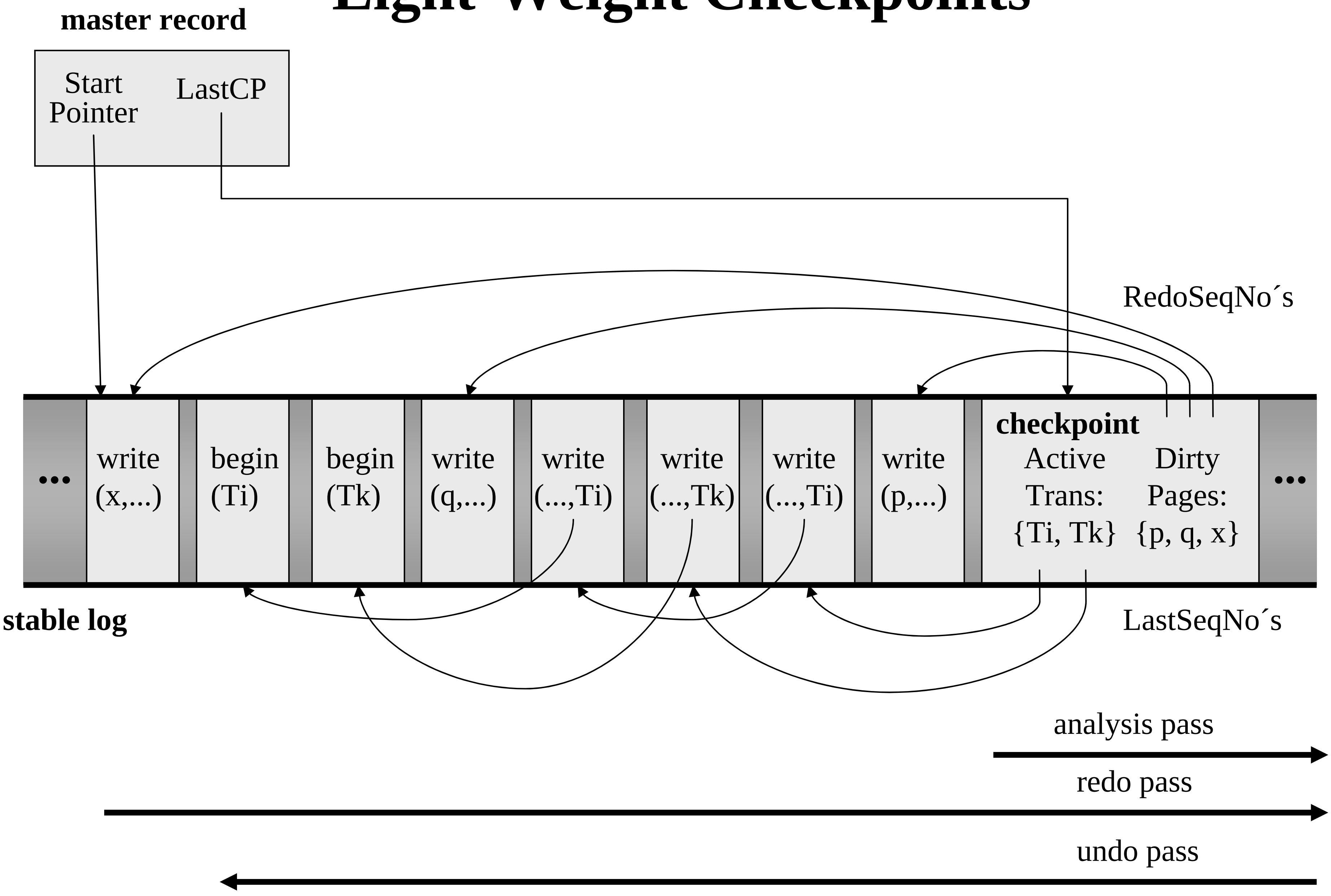
track dirty cache pages and periodically write **DirtyPageTable** (DPT) into **checkpoint log entry:**

```
   type DPTEntry: record of
      PageNo: id;
      RedoSeqNo: id; end;
   var DirtyPages: set[PageNo] of DPTEntry;
```

\+ add potentially dirty pages during analysis pass

\+ **flush-behind demon** flushes dirty pages in background

Light-Weight Checkpoints



master record

Start Pointer

LastCP

RedoSeqNo´s

...

write (x,...)

begin (Ti)

begin (Tk)

write (q,...)

write (...,Ti)

write (...,Tk)

write (...,Ti)

write (p,...)

checkpoint

Active Trans: {Ti, Tk}

Dirty Pages: {p, q, x}

...

stable log

LastSeqNo´s

analysis pass

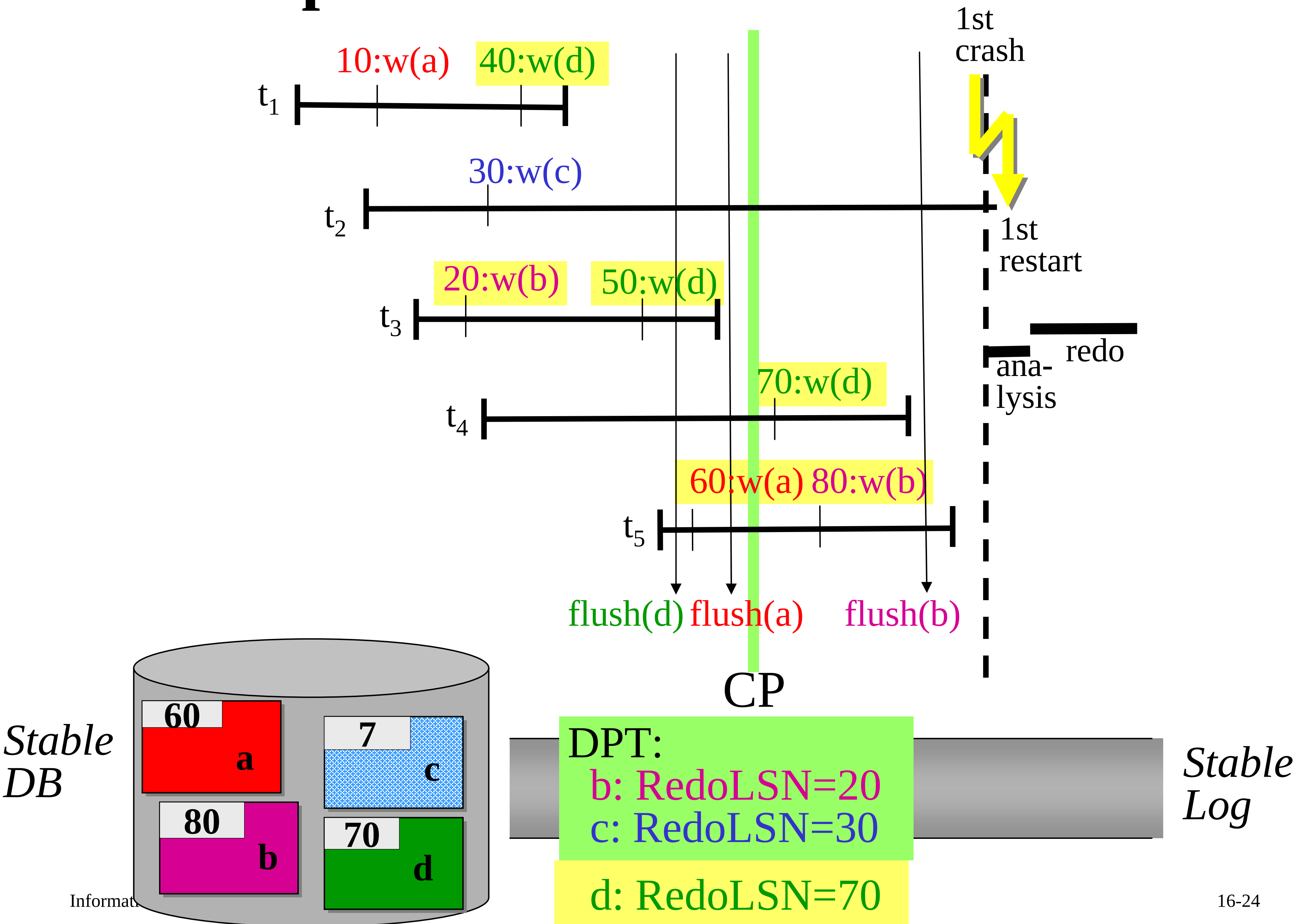
redo pass

undo pass

Redo Pass with CP and DPT

```
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; end; end;
```

Example for Redo with CP and DPT

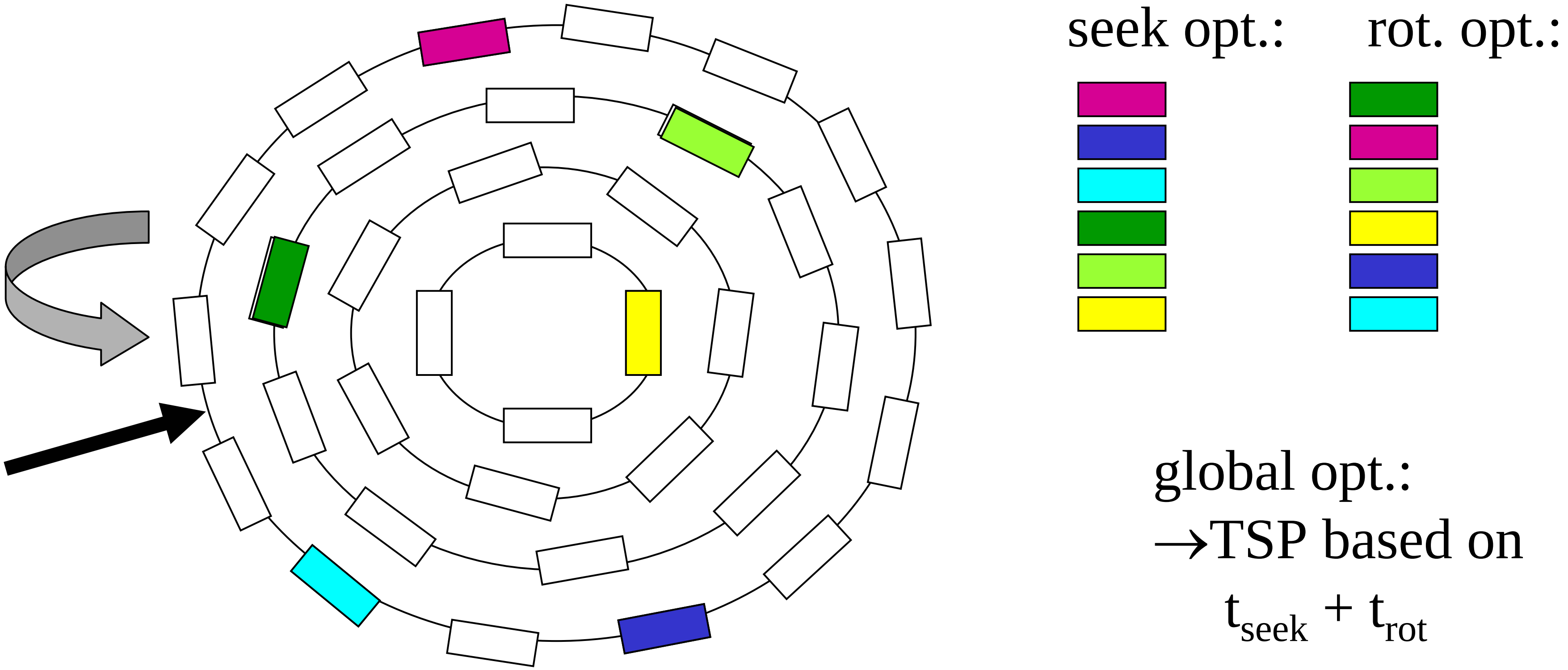


DPT:
b: RedoLSN=20
c: RedoLSN=30

d: RedoLSN=70

Benefits of Redo Optimization

- can save page fetches
- can plan prefetching schedule for dirty pages (order and timing of page fetches)
  - for minization of disk-arm seek time
  - and rotational latency



seek opt.:

rot. opt.:

global opt.:
→TSP based on
$t_{seek} + t_{rot}$

Redo Optimization 2: Flush Log Entries

during normal operation:

log flush actions (without forcing the log buffer)

during analysis pass of restart:

construct more recent DPT

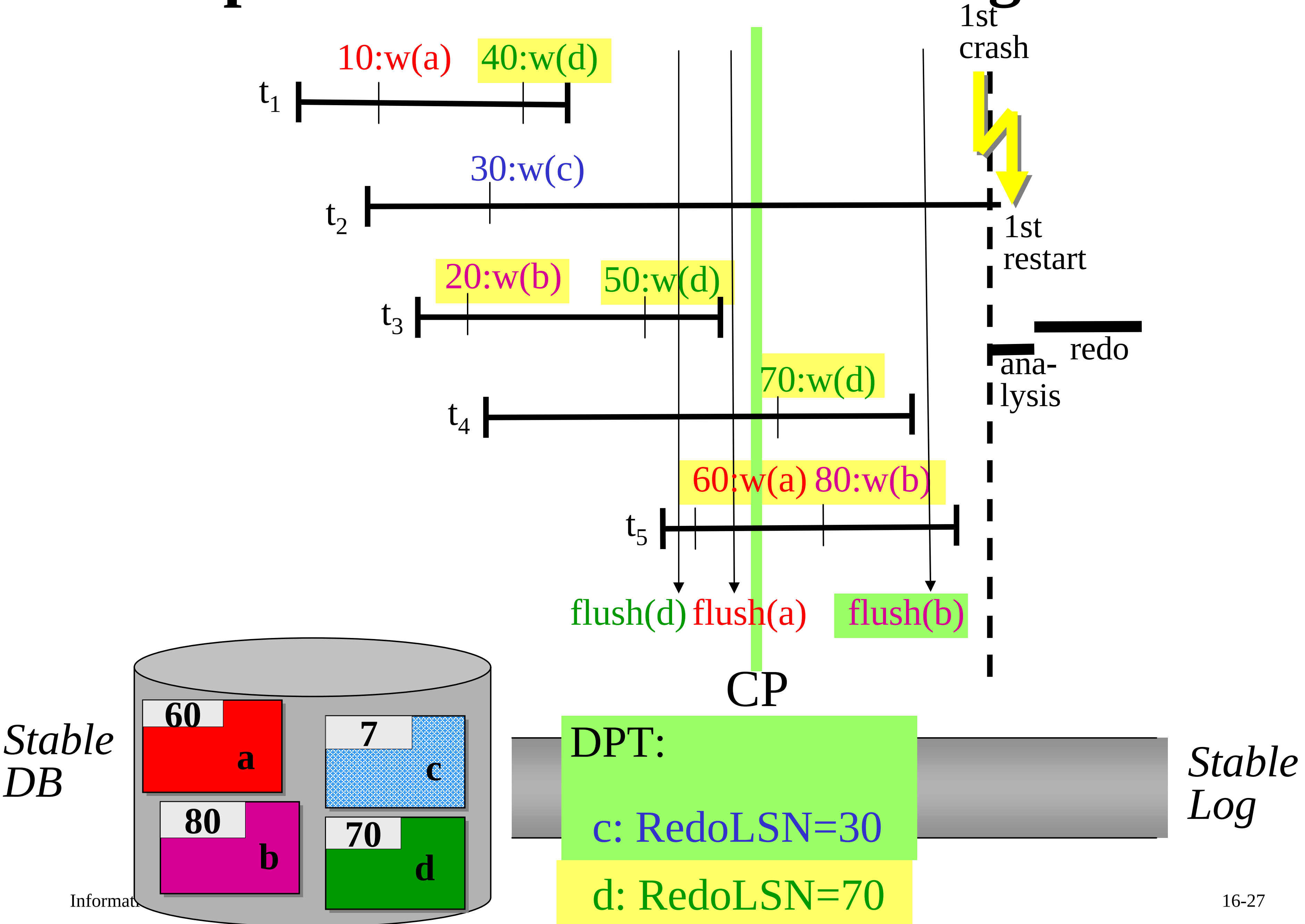
```
analysis pass ( ) returns losers, DirtyPages:
...
if StableLog[i].ActionType = write or full-write
      and StableLog[i].PageNo not in DirtyPages then
   DirtyPages += StableLog[i].PageNo;
   DirtyPages[StableLog[i].PageNo].RedoSeqNo := i; end;
if StableLog[i].ActionType = flush then
   DirtyPages -= StableLog[i].PageNo; end;
...
```

advantages:

→ can save many page fetches

→ allows better prefetch scheduling

Example for Redo with Flush Log Entries



DPT:

c: RedoLSN=30

d: RedoLSN=70

Redo Optimization 3: Full-write Log Entries

during normal operation:

„occasionally“ log **full after-image** of a page p (**absorbs** all previous writes to that page)

during analysis pass of restart:

construct **enhanced DPT**

DirtyPages[p].RedoSeqNo := LogSeqNo of full-write

during redo pass of restart:

can ignore all log entries of a page that precede its most recent full-write log entry

advantages:

→ can skip many log entries

→ can plan better schedule for page fetches

→ allows better log truncation

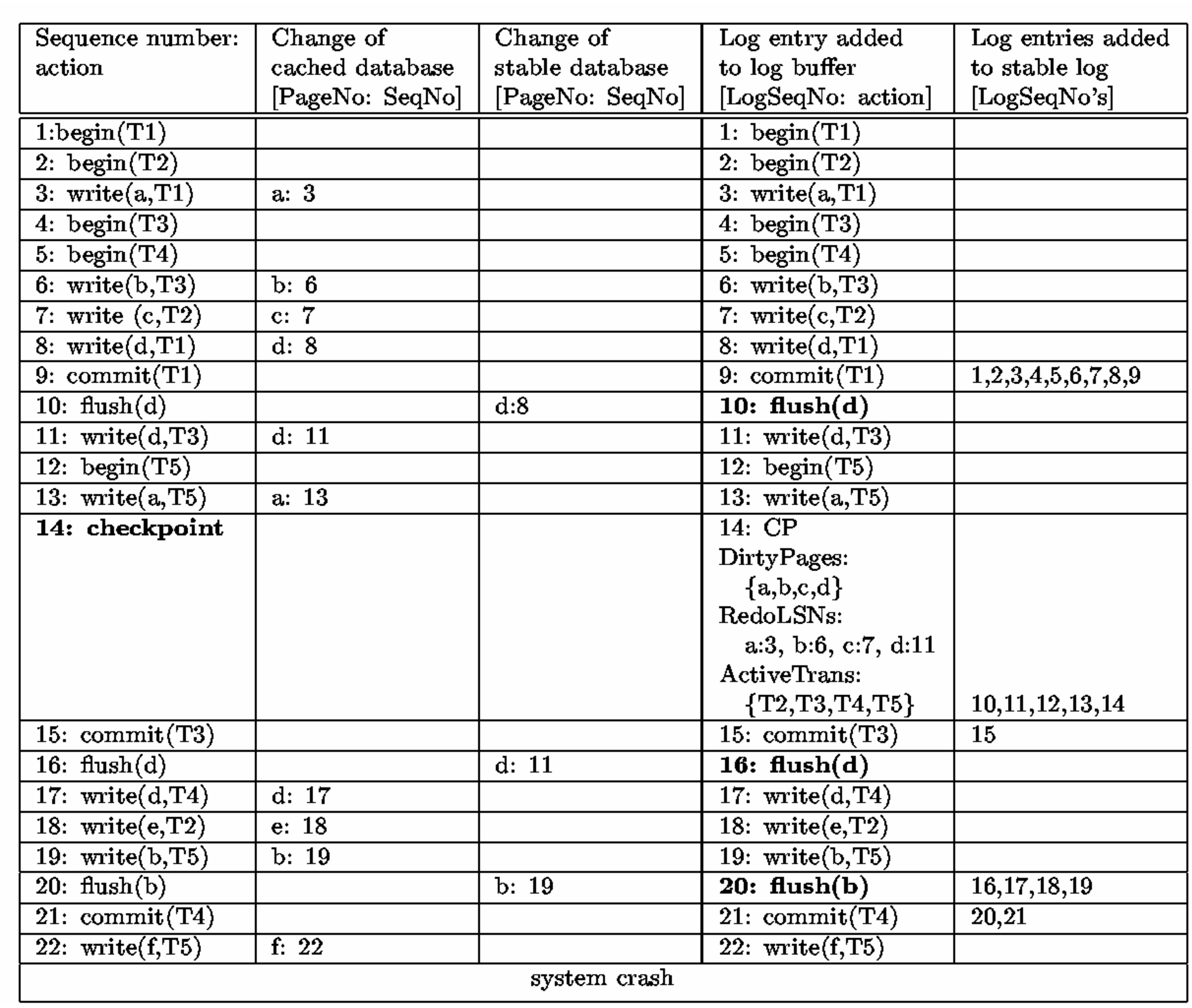
Correctness of Optimized Redo

Invariant during optimized redo pass:

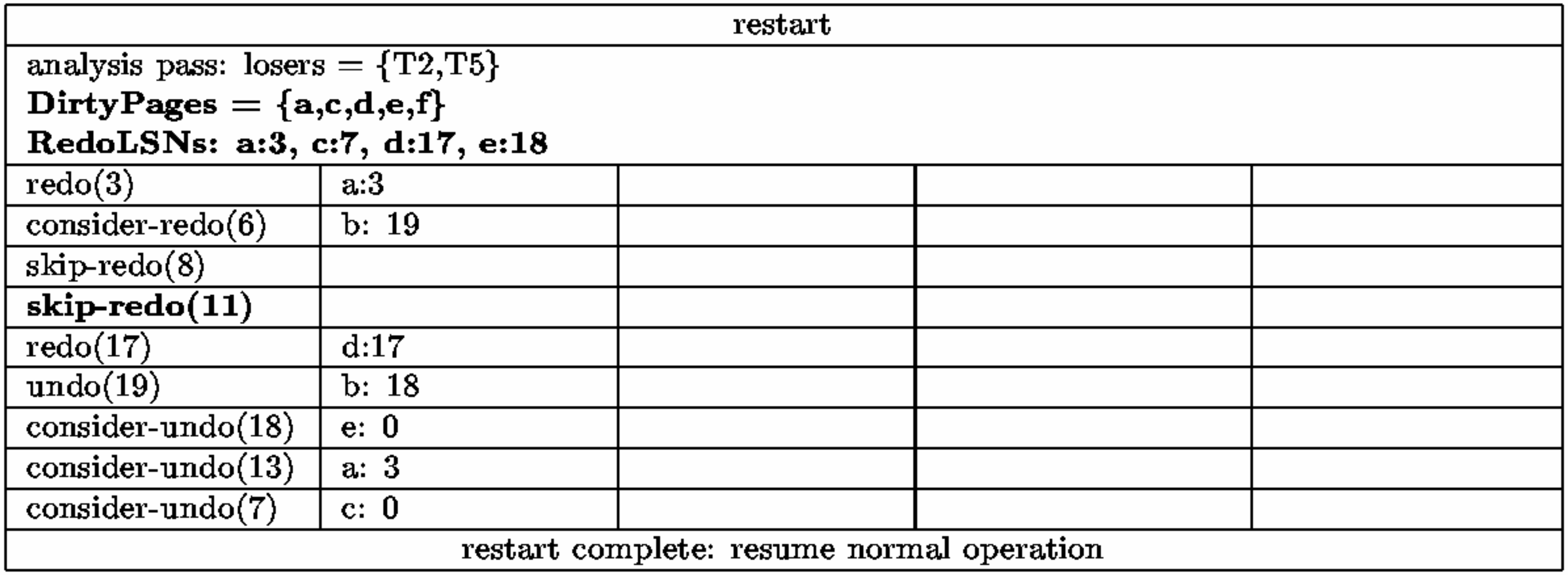
$$\begin{aligned}
&\forall pages\ \ p : \forall o \in stable\ log\ :\Big(\\
&\qquad \big(o.pageid\ = p\ \wedge \\
&\qquad \big(p \notin DirtyPages\ \vee o < DirtyPages\ [\ p\ ].RedoSeqNo\ \big)\big) \\
&\quad \Rightarrow o \in StableDB\ \Big)
\end{aligned}$$

builds on correct DPT construction during analysis pass

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				



<table>
<tr><th colspan="5">restart</th></tr>
<tr><td colspan="5">analysis pass: losers = {T2,T5}
DirtyPages = {a,c,d,e,f}
RedoLSNs: a:3, c:7, d:17, e:18</td></tr>
<tr><td>redo(3)</td><td>a:3</td><td></td><td></td><td></td></tr>
<tr><td>consider-redo(6)</td><td>b: 19</td><td></td><td></td><td></td></tr>
<tr><td>skip-redo(8)</td><td></td><td></td><td></td><td></td></tr>
<tr><td>**skip-redo(11)**</td><td></td><td></td><td></td><td></td></tr>
<tr><td>redo(17)</td><td>d:17</td><td></td><td></td><td></td></tr>
<tr><td>undo(19)</td><td>b: 18</td><td></td><td></td><td></td></tr>
<tr><td>consider-undo(18)</td><td>e: 0</td><td></td><td></td><td></td></tr>
<tr><td>consider-undo(13)</td><td>a: 3</td><td></td><td></td><td></td></tr>
<tr><td>consider-undo(7)</td><td>c: 0</td><td></td><td></td><td></td></tr>
<tr><td colspan="5">restart complete: resume normal operation</td></tr>
</table>

Korrektur: Ersetze consider-redo(6) durch skip-redo(6)!

Why is Page-based Redo Logging Good Anyway?

- testable state with very low overhead
- much faster than logical redo
  - → logical redo would require replaying high-level operations with many page fetches (random IO)
- can be applied to selective pages independently
  - → can exploit perfectly scalable parallelism for redo of large multi-disk db
  - → can efficiently reconstruct corrupted pages when disk tracks have errors (without full media recovery for entire db)

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:

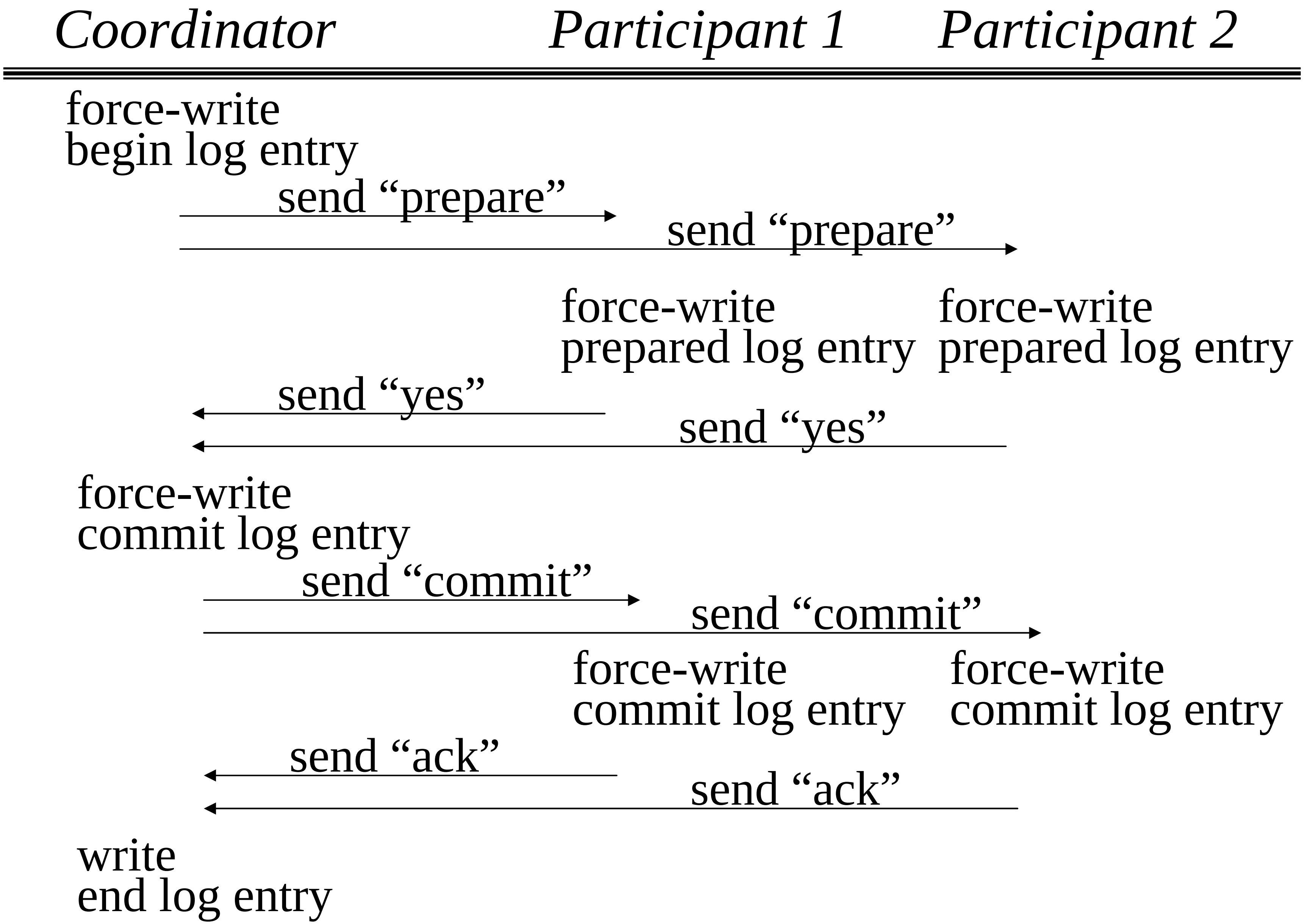
- Distributed handshake protocol known as **two-phase commit (2PC)**
- with a **coordinator** taking responsibility for unanimous outcome
- Recovery considerations for in-doubt transactions

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 *acks*
- **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”

send “prepare”

force-write
prepared log entry

force-write
prepared log entry

send “yes”

send “yes”

force-write
commit log entry

send “commit”

send “commit”

force-write
commit log entry

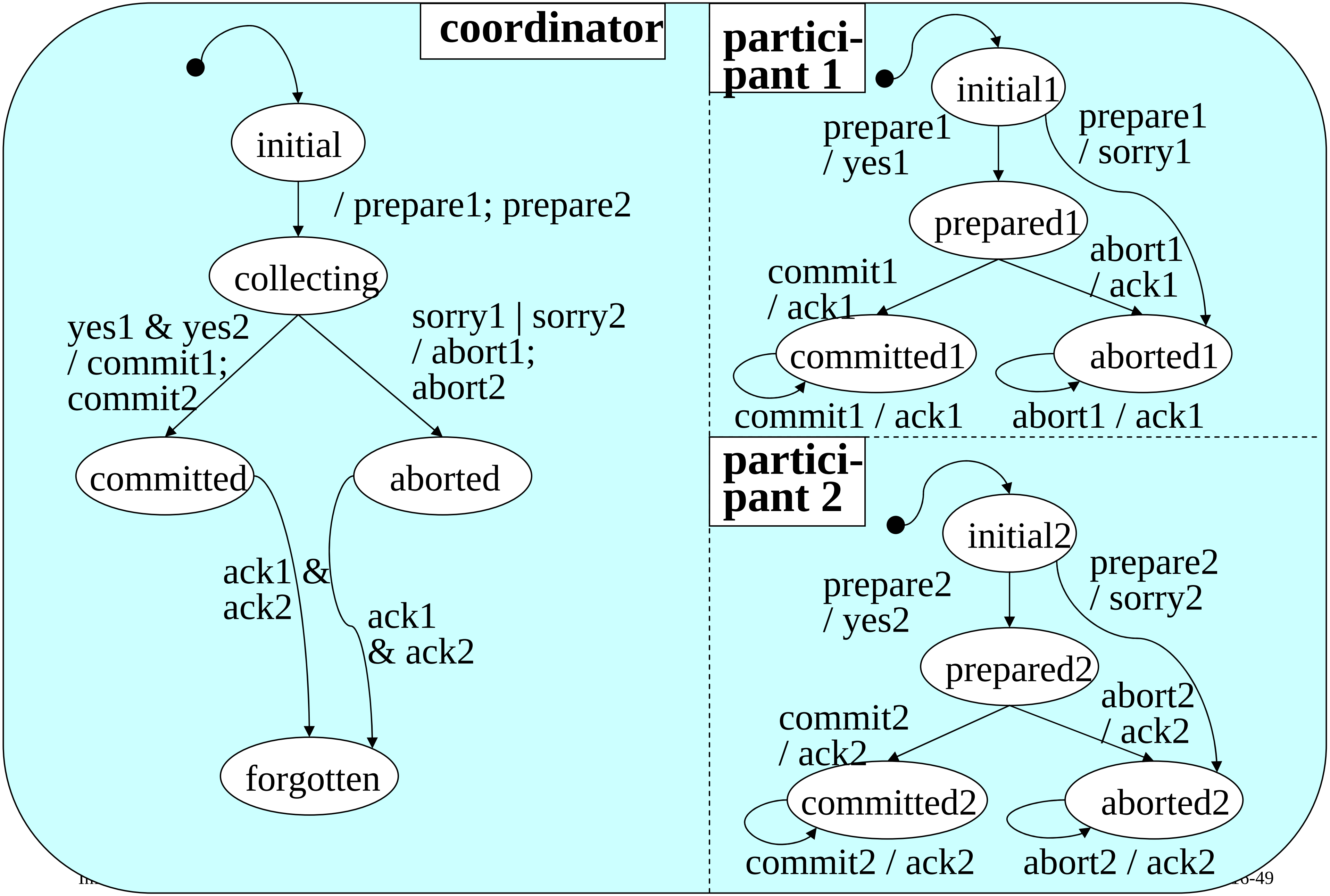
force-write
commit log entry

send “ack”

send “ack”

write
end log entry

Statechart for Basic 2PC



coordinator

initial

/ prepare1; prepare2

collecting

yes1 & yes2
/ commit1;
commit2

sorry1 | sorry2
/ abort1;
abort2

committed

aborted

ack1 &
ack2

ack1
& ack2

forgotten

**partici-
pant 1**

initial1

prepare1
/ yes1

prepare1
/ sorry1

prepared1

commit1
/ ack1

abort1
/ ack1

committed1

aborted1

commit1 / ack1

abort1 / ack1

**partici-
pant 2**

initial2

prepare2
/ yes2

prepare2
/ sorry2

prepared2

commit2
/ ack2

abort2
/ ack2

committed2

aborted2

commit2 / ack2

abort2 / ack2

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
  → Byzantine agreement
- no assumptions about network failure handling
  → can use datagrams or sessions for communication

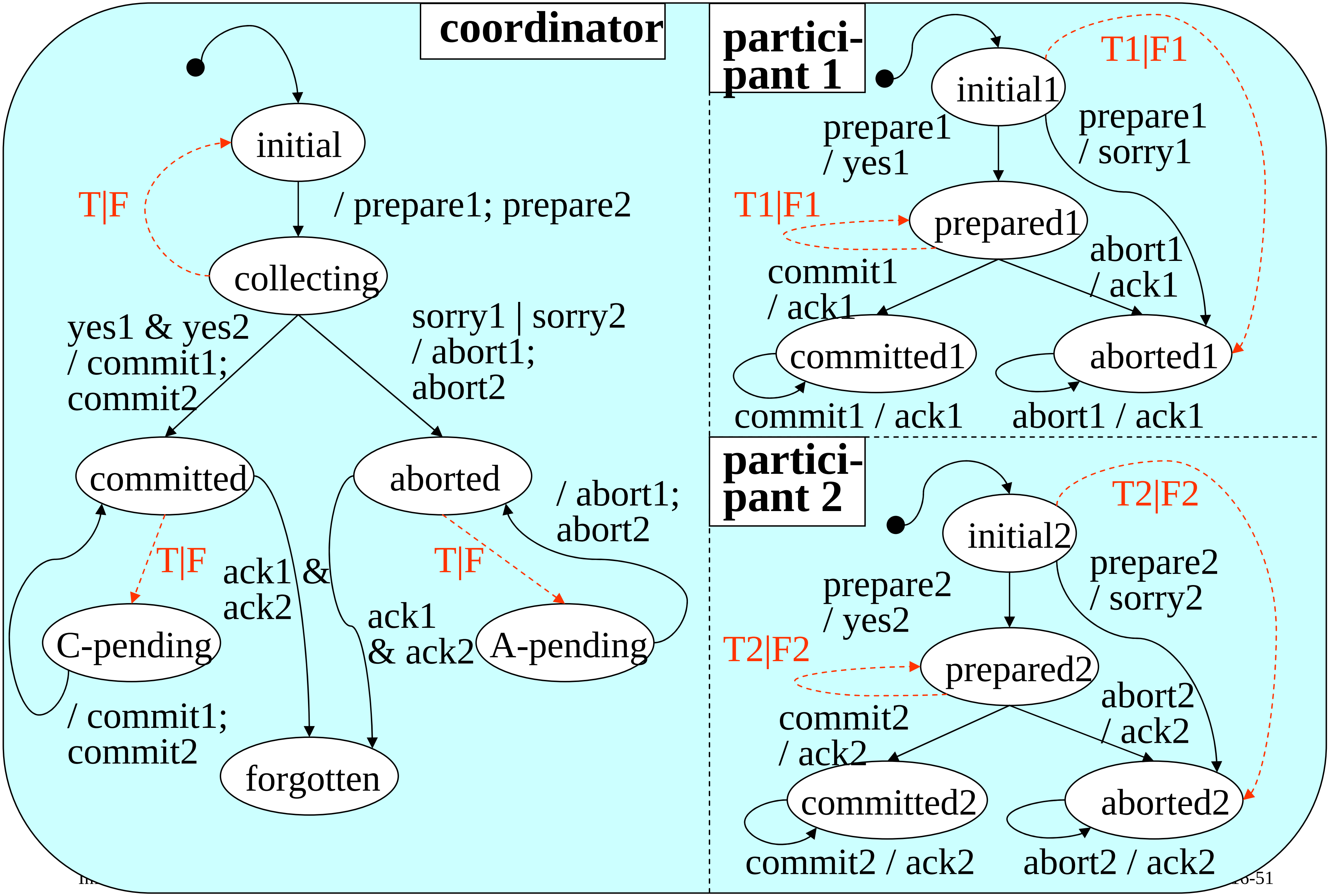
Restart protocol after failure (F transitions):

- 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 (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 (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).