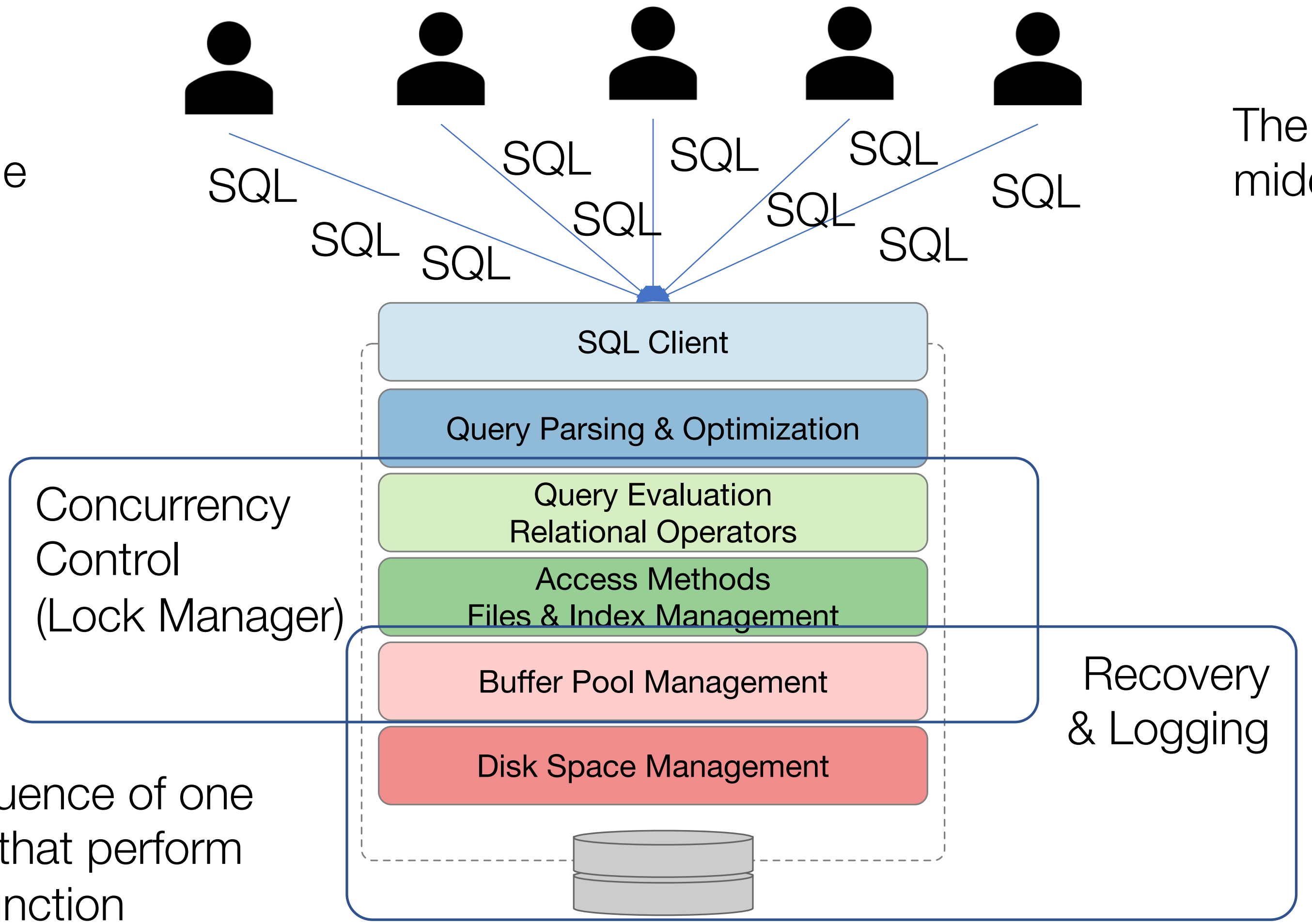
Recovery

Two users change the same record at the same time.

The power fails in the middle of your update



Transaction Manager

Transaction – a sequence of one or more operations that perform some higher-level function

Recovery Manager

Provides Atomicity & Durability

DBMS provide certain transaction guarantees (e.g. ACID) that make the lives of programmers easy 😎

ACID Transactions

Atomicity: All actions in a transaction happen, or none happen.

Consistency: If the DB starts out consistent, it ends up consistent at the end of the Xact! (The DBMS aborts transactions that violate any Integrity Constraints)

Isolation: Execution of each Xact is isolated from that of others

Durability: If a Xact commits, its effects persist.

Recovery Manager

- Ensures Atomicity & Durability
- Ensures Consistency by aborting/roll-backing transactions that violate integrity constraints

Why Do Transactions Abort?

User/Application explicitly aborts
Integrity constraint violated
Deadlock
System failure prior to successful commit

Why Do Databases Crash?

- Operator Error
  - Trip over the power cord
  - Type the wrong command
- Configuration Error
  - Insufficient resources: disk space
  - File permissions, etc.
- Software Failure
  - DBMS bugs, security flaws, OS bugs
- Hardware Failure
  - Media or Server

Atomicity, Durability, Recovery & The Buffer

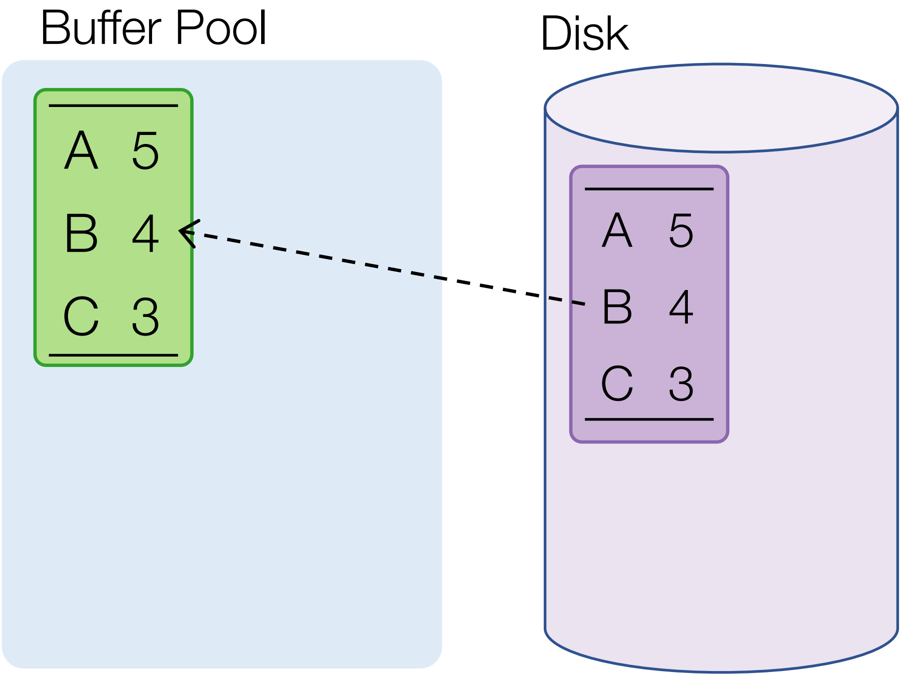
Atomicity, Durability & Recovery

Keep in mind: 

- A DBMS stores data on disk (non-volatile storage).
- *Durability* means that the effects of committed transaction persist even when you lose everything on volatile storage.
- We do not write directly to disk: we write to copies of disk pages in memory. Why?
    - Performance

Xact Schedule

T1	T2
BEGIN	
R(A)	

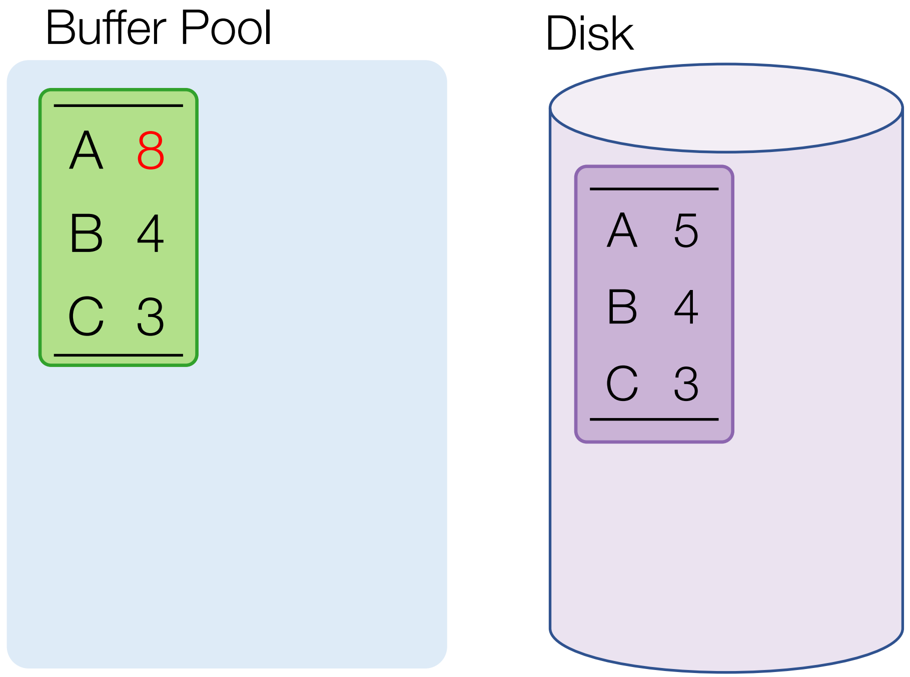


Keep in mind: 

- A DBMS stores data on disk (non-volatile storage).
- *Durability* means that the effects of committed transaction persist even when you lose everything on volatile storage.
- We do not write directly to disk: we write to copies of disk pages in memory. Why?
  - Performance

Xact Schedule

T1	T2
BEGIN	
R(A)	
A := 8	
W(A)	



Atomicity, Durability & Recovery

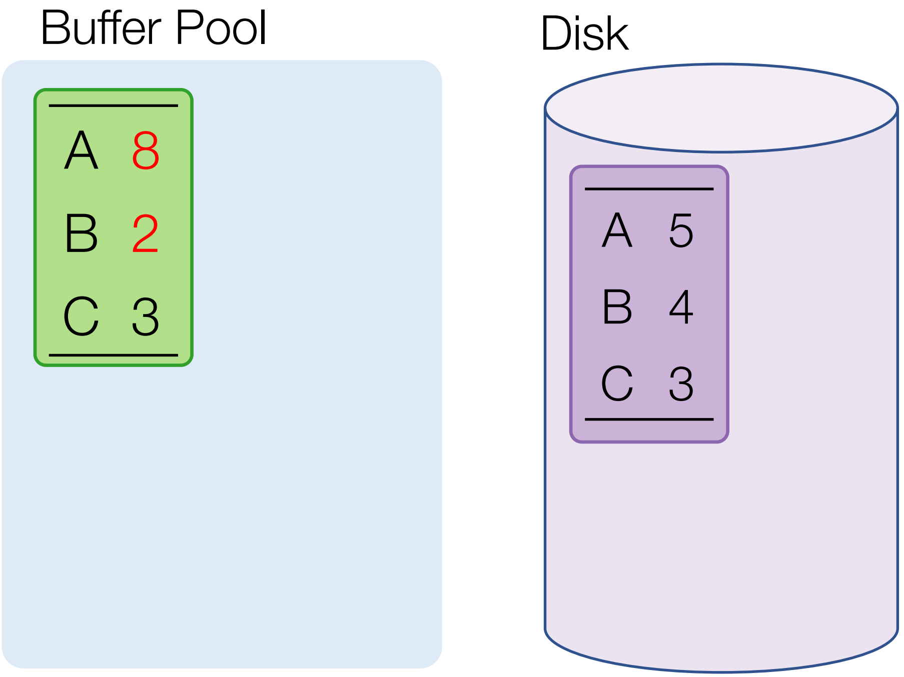
Atomicity, Durability & Recovery

Keep in mind:

- A DBMS stores data on disk (non-volatile storage).
- *Durability* means that the effects of committed transaction persist even when you lose everything on volatile storage.
- We do not write directly to disk: we write to copies of disk pages in memory. Why?
  - Performance

Xact Schedule

T1	T2
BEGIN	
R(A)	
A := 8	
W(A)	
	BEGIN
	R(B)
	B := 2
	W(B)

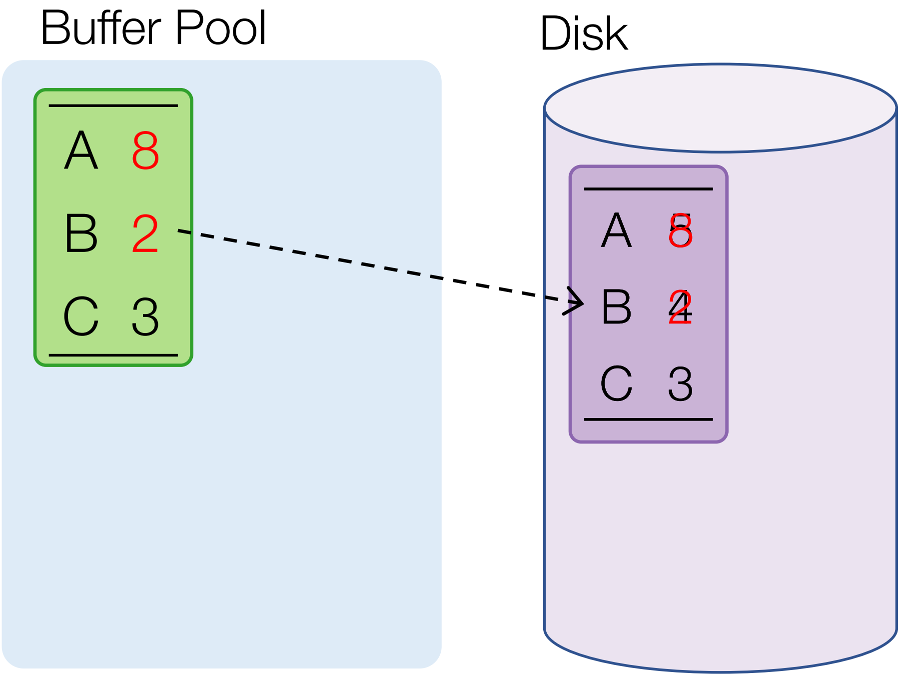


Keep in mind:

- A DBMS stores data on disk (non-volatile storage).
- *Durability* means that the effects of committed transaction persist even when you lose everything on volatile storage.
- We do not write directly to disk: we write to copies of disk pages in memory. Why?
  - Performance

Xact Schedule

T1	T2
BEGIN	
R(A)	
A := 8	
W(A)	
	BEGIN
	R(B)
	B := 2
	W(B)
	COMMIT



Do we "FORCE" the page to disk? — Durability

Do we persist the effects of T1 that has not committed? — Atomicity

Atomicity, Durability & Recovery

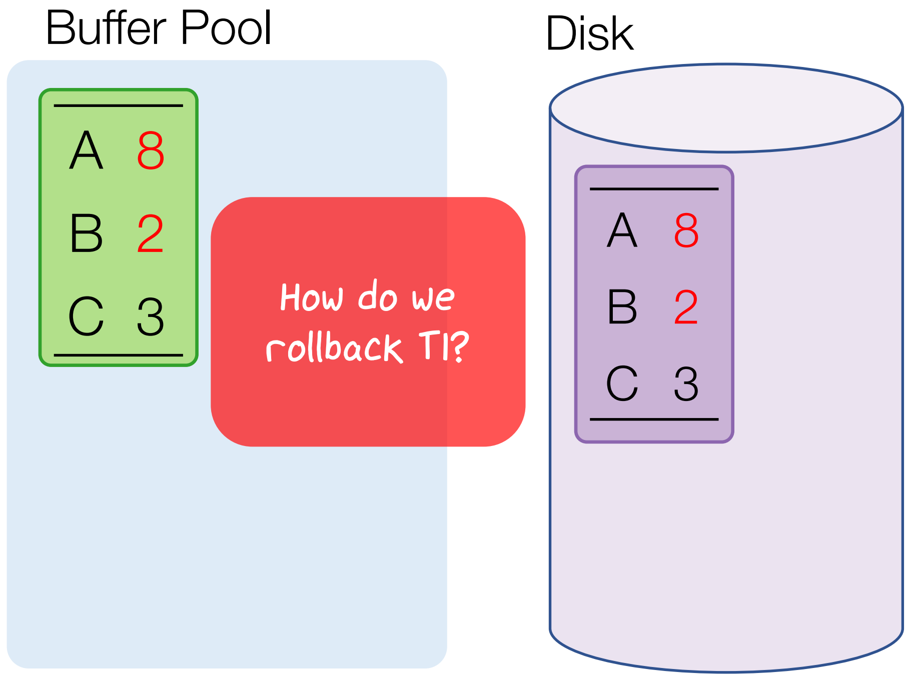
Keep in mind:

- A DBMS stores data on disk (non-volatile storage).
- *Durability* means that the effects of committed transaction persist even when you lose everything on volatile storage.
- We do not write directly to disk: we write to copies of disk pages in memory. Why?
    - Performance

Xact Schedule

T1	T2
BEGIN	
R(A)	
A := 8	
W(A)	
	BEGIN
	R(B)
	B := 2
	W(B)
	COMMIT
ABORT	

Crash



Do we "FORCE" the page to disk? — Durability

Do we persist the effects of T1 that has not committed? — Atomicity

Atomicity, Durability & Recovery

Building a WAL

Buffer Policy & Recovery

	NO-STEAL	STEAL
FORCE		
NO-FORCE		

Buffer Policy

STEAL

An uncommitted Xact ***can overwrite*** the most recent committed value of an object on disk.

Dirty pages can be "stolen" by page replacement policy

FORCE

All updates by a Xact are reflected on disk before the Xact can commit.

Recovery Operations

- *Undo*: Remove effects of an incomplete or aborted Xact
- *Redo*: Redo the effects of a committed Xact for durability.

Assumptions

- *Strict 2PL* in effect

	NO-STEAL	STEAL
FORCE	NO UNDO NO REDO	
NO-FORCE		

Simple Recovery

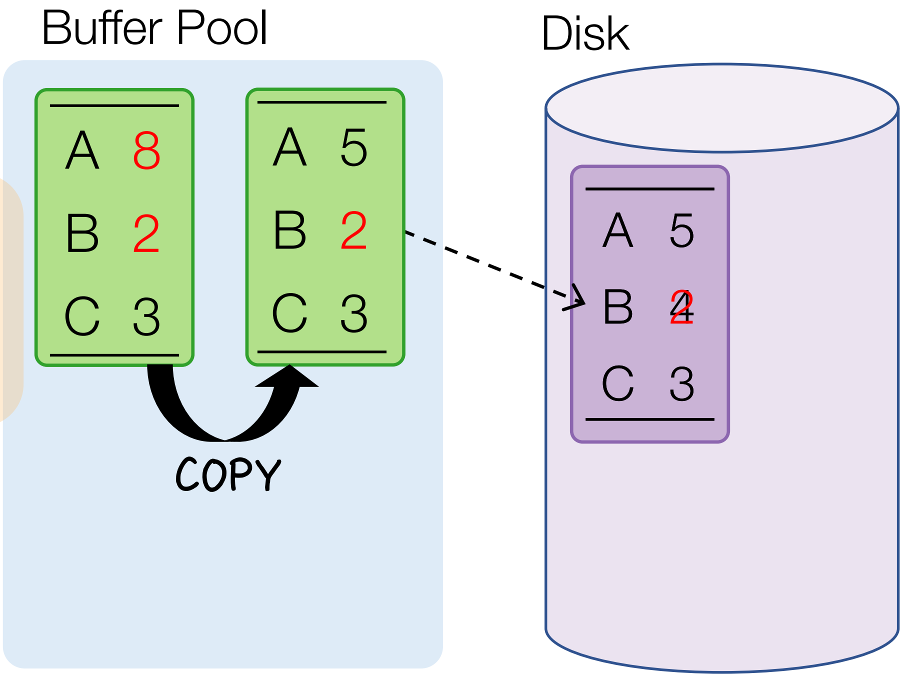
- *No* need to *undo* changes of an aborted Xact because the changes are not written to disk.
- *No* need to *redo* changes of a committed Xact because all the changes are guaranteed to be written to disk at commit time.

Xact Schedule

T1	T2
BEGIN	
R(A)	
A := 8	
W(A)	
	BEGIN
	R(B)
	B := 2
	W(B)
	COMMIT
ABORT	

"NO-STEAL" means we do not flush T1's uncommitted changes to disk

"FORCE" means we flush the changes of T2 to disk here!



What if Xact can't pin all its pages in the buffer?

Still need a way to atomically write:

- Atomic hardware writes
- Shadow paging

Poor performance!

Random IOs at every commit.

	NO-STEAL	STEAL
FORCE	NO UNDO NO REDO	UNDO NO REDO
NO-FORCE	NO UNDO REDO	**UNDO REDO**

NO FORCE

What if system crashes before dirty buffer page of a committed transaction is flushed to DB disk?

- Flush as little as possible, in a convenient place, prior to commit.
- You can use this to REDO modifications after the crash!

STEAL

What if a transaction that flushed updated pages to the DB disk aborts?

- Must retain old or before-update images of the flushed pages to UNDO any updates to them?

What if system crashes before Xact is finished?

- Consider these transactions as aborted! And you need to undo them.

Buffer Policy & Recovery

	LSN	Xid	type	object	Before	After
Log	…					
	101	1	BEGIN	-	-	-
	102	1	UPDATE	A	10	20
	103	2	BEGIN	-	-	-
	104	2	UPDATE	B	5	0
	105	1	COMMIT			
Log Tail Still in memory	106	2	ABORT	-	-	-
	107	3	BEGIN	-	-	-
	108	3	UPDATE	A	20	15
	…					
	150	5	UPDATE	C	100	150

Allows STEAL/NO-FORCE

LOG: An **ordered** list of log records to allow *REDO/UNDO* for every update

- Sequential writes to log (on a separate disk).
- Minimal info written to log: pack multiple updates in a single log page.

Good performance

Logging

Write-Ahead Logging (WAL)

flushedLSN
Pointer to last log record flushed to disk

Log

LSN	Xid	type	object	Before	After
…					
101	1	BEGIN	-	-	-
102	1	UPDATE	A	10	20
103	2	BEGIN	-	-	-
104	2	UPDATE	B	5	0
105	1	COMMIT			

Log Tail memory

LSN	Xid	type	object	Before	After
106	2	ABORT	-	-	-
107	3	BEGIN	-	-	-
108	3	UPDATE	A	20	15
…					
150	5	UPDATE	C	100	150

DB pages on disk

pageLSN: 105	
A	20
B	0
C	100

pageLSN: 090	
D	5
E	4
F	3

pageLSN pointer to log record of most recent update

Buffer Pool

pageLSN: 150	
A	15
B	5
C	150

pageLSN: 090	
D	9
E	9
F	9

Before page i is flushed to DB:
pageLSN(i) <= flushedLSN

1. Must force the log record for an update before the corresponding data page gets to the DB disk. — + UNDO gives *Atomicity*

2. Must force **all** log records for a Xact before commit. — + REDO gives *Durability*

ARIES

LSN	Xid	type	pageID	object	Before	After
000	1	BEGIN		-	-	-
001	1	UPDATE	12	X	109	108
…						
101	76	BEGIN		-	-	-
102	63	UPDATE	8	A	10	20
103	77	BEGIN		-	-	-
104	64	ABORT				
105	63	COMMIT				
106	77	UPDATE	10	D	-	-
107	78	BEGIN		-	-	-
108	76	UPDATE	8	A	20	15
…						
150	95	UPDATE	8	C	100	150

TIME

CRASH

NAÏVE RECOVERY

Start from an initial DB
Replay the log

The whole log! Can this be cheaper?
Initial Database! Can't we just identify only the pages that are dirty and recover those?

Now move backwards undo each transaction that did not commit!

How do we know which transactions to abort?
How do we find their instructions to rollback?

LSN	Xid	type	pageID	object	Before	After
000	1	BEGIN		-	-	-
001	1	UPDATE	12	X	109	108
…						
100	CHECKPOINT					
101	76	BEGIN		-	-	-
102	63	UPDATE	8	A	10	20
103	77	BEGIN		-	-	-
104	64	ABORT				
105	63	COMMIT				

MASTER RECORD
LAST CHECKPOINT LSN: 100

TIME

ARIES RECOVERY

ANALYSIS

CRASH

The whole log! Can this be cheaper?

Initial Database! Can't we just identify only the pages that are dirty and recover those?

YES, Start analyzing the log from the last checkpoint to identify loser transactions & dirty pages

How do we know which transactions to abort?
How do we find their instructions to rollback?

ARIES RECOVERY

LSN	Xid	type	pageID	object	Before	After
000	1	BEGIN		-	-	-
001	1	UPDATE	12	X	109	108
…						
100	CHECKPOINT					
101	76	BEGIN		-	-	-
102	63	UPDATE	8	A	10	20
103	77	BEGIN		-	-	-
104	64	ABORT				
105	63	COMMIT				

TIME

ANALYSIS

CRASH

MASTER RECORD
LAST CHECKPOINT LSN: 100

Transaction Table

Xid	status	lastLSN
63	running	87
64	running	99
63	commit	105
64	abort	104

Dirty Page Table

pageID	recLSN
9	89
8	102

YES, Start analyzing the log from the last checkpoint to identify loser transactions & dirty pages

How do we know which transactions to abort?
How do find their instructions to rollback?

ABORT "running" and "abort" Xacts in the Xact table

LSN	Xid	type	pageID	object	Before	After
000	1	BEGIN		-	-	-
001	1	UPDATE	12	X	109	108
089	58	UPDATE	9	G	150	120
…						
100	CHECKPOINT					
101	76	BEGIN		-	-	-
102	63	UPDATE	8	A	10	20
103	77	BEGIN		-	-	-
104	64	ABORT				
105	63	COMMIT				

TIME

CRASH

ARIES RECOVERY

YES, Start analyzing the log from the last checkpoint to identify loser transactions & dirty pages

ANALYSIS

REDO

Transaction Table

Xid	status	lastLSN
76	running	101
77	running	103

Repeat History --- all of it! Why?
It is too complex to do otherwise!

Start at the smallest recovery LSN.
Why?
First record of an update that may not have been flushed to disk!

How do we know which transactions to abort?
How do find their instructions to rollback?

ABORT "running" and "abort" Xacts in the Xact table

MASTER RECORD
LAST CHECKPOINT LSN: 100

LSN	prev LSN	Xid	type	pageID	object	Before	After	undo Next
000	-	1	BEGIN		-	-	-	-
001	000	1	UPDATE	12	X	109	108	108
089	085	58	UPDATE	9	G	150	120	
…	…							
100			CHECKPOINT					
101	-	76	BEGIN		-	-	-	-
102	94	63	UPDATE	8	A	10	20	20
103	-	77	BEGIN		-	-	-	-
104	96	64	ABORT					
105	102	63	COMMIT					
106	104	CLR; UNDO T64		10				96

MASTER RECORD
LAST CHECKPOINT LSN: 100

Log CLRs for every rollback and undoNext in CLR. Why? To avoid repeating undos!

ARIES RECOVERY

ABORT "running" and "abort" Xacts in the Xact table

ANALYSIS

REDO

UNDO

Transaction Table

Xid	status	lastLSN
76	running	101
77	running	103
64	abort	104

Dirty Page Table

pageID	recLSN
9	99
8	102
10	106

For each loser, perform simple transaction abort, following prevLSN chains in the Log to rollback with before images.

How do find their instructions to rollback?