Logging & Recovery

April 18, 2017

Announcements

- CSE-662 **Wait List** created
- I will **force reg** up to 10 students for CSE-662
  - Required: B+ in 562
  - If >10 eligible, selection will be based on weighted avg of project/exam grades.
- **In-Class Final Exam**: May 11
  - If this is a problem, contact me directly.

What does it mean for a transaction to be committed?

If commit <u>returns</u> <u>successfully</u>, the transaction…

- … is recorded completely (atomicity)
- … left the database in a stable state (consistency)
- …'s effects are independent of other xacts (isolation)
- … will survive failures (durability)

commit
returns
successfully

=

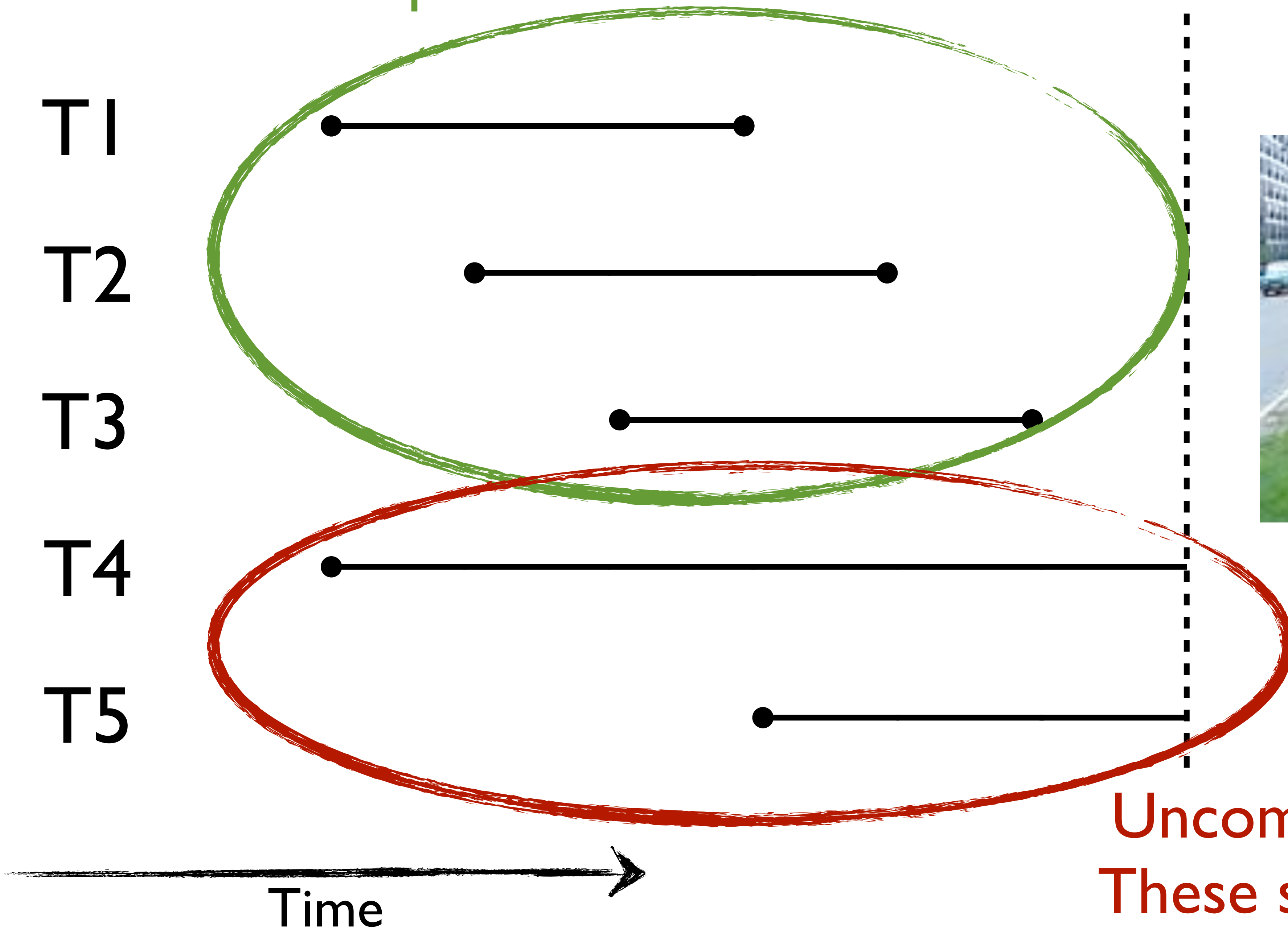
the xact's
effects
are visible
<u>forever</u>

commit returns successfully = the xact's effects are visible <u>forever</u>

commit called but doesn't return = the xact's effects <u>may</u> be visible

Motivation

Committed Transactions.
These should be present when the DB restarts.



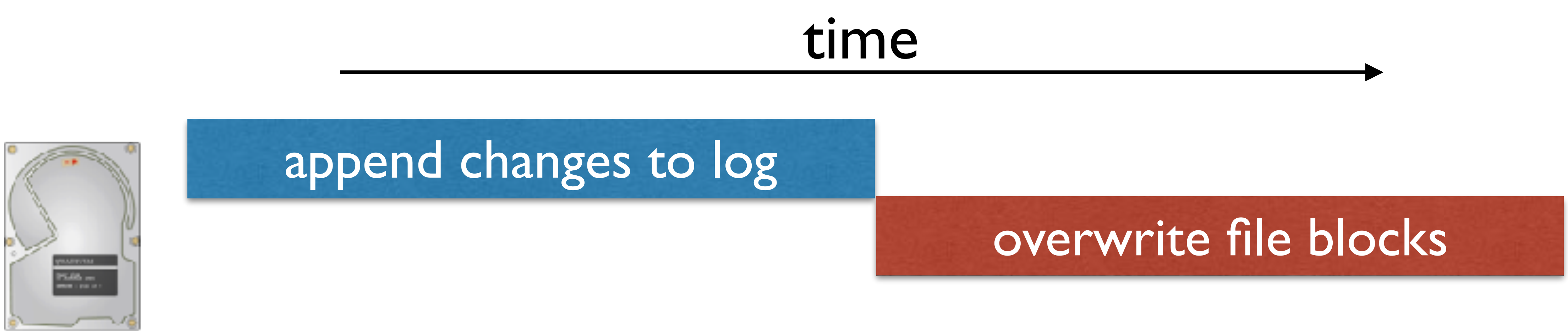
Uncommitted Transactions.
These should leave no trace

ACID

- **Isolation**: Already addressed.
- **Atomicity**: Need writes to get *flushed* in a single step.
  - IOs are only atomic at the page level.
- **Durability**: Need to *buffer* some writes until commit.
  - May need to free up memory for another xact.
- **Consistency**: Need to roll back incomplete xacts.
  - May have already paged back to disk.

Atomicity

- **Problem**: IOs are only atomic for 1 page.
    - What if we crash in between writes?
- **Solution**: Logging (e.g., Journaling Filesystem)
    - Log everything first before you do it.

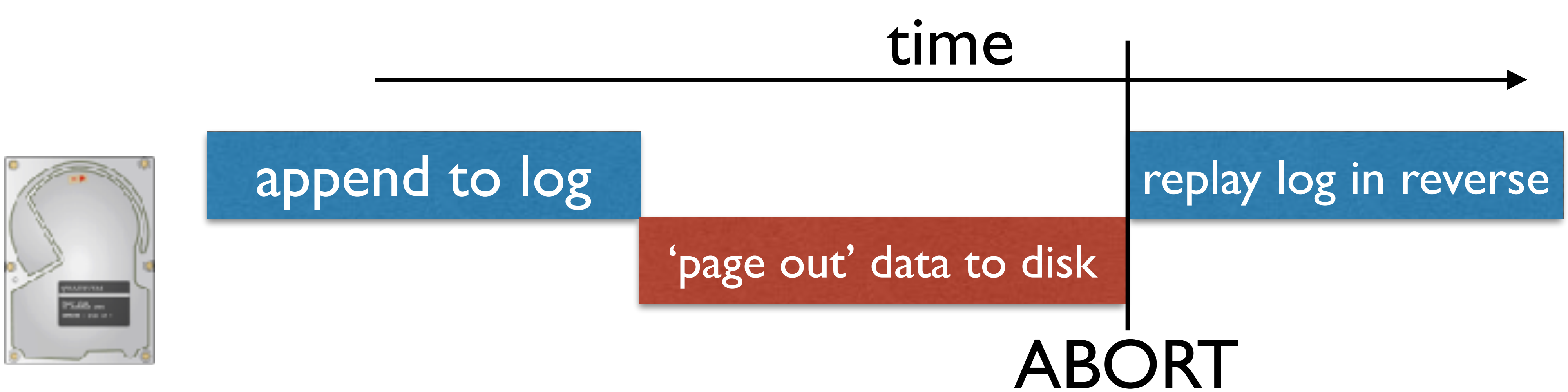


Durability / Consistency

- **Problem**: Buffer memory is limited
    - What if we need to 'page out' some data?

- **Solution**: Use log (or similar) to recover buffer
    - *Problem*: Commits more expensive

- **Solution**: Modify DB in place, use log to 'undo' on abort
    - *Problem*: Aborts more expensive



Problem 1: Providing durability under failures.

Simplified Model

When a write succeeds, the data is completely written

Problems

- A crash occurs part-way through the write.

- A crash occurs before buffered data is written.

Write-Ahead Logging

Before writing to the database, first write what you plan to write to a log file…

Log

`W(A:10)`

A	8
B	12
C	5
D	18
E	16

Write-Ahead Logging

Once the log is safely on disk
you can write the database

A	~~8~~ 10
B	12
C	5
D	18
E	16

Log

`W(A:10)`

Write-Ahead Logging

Log is append-only,
so writes are always
efficient

Log

```
W(A:10)
W(C:8)
W(E:9)
```

A	~~8~~ 10
B	12
C	5
D	18
E	16

Write-Ahead Logging

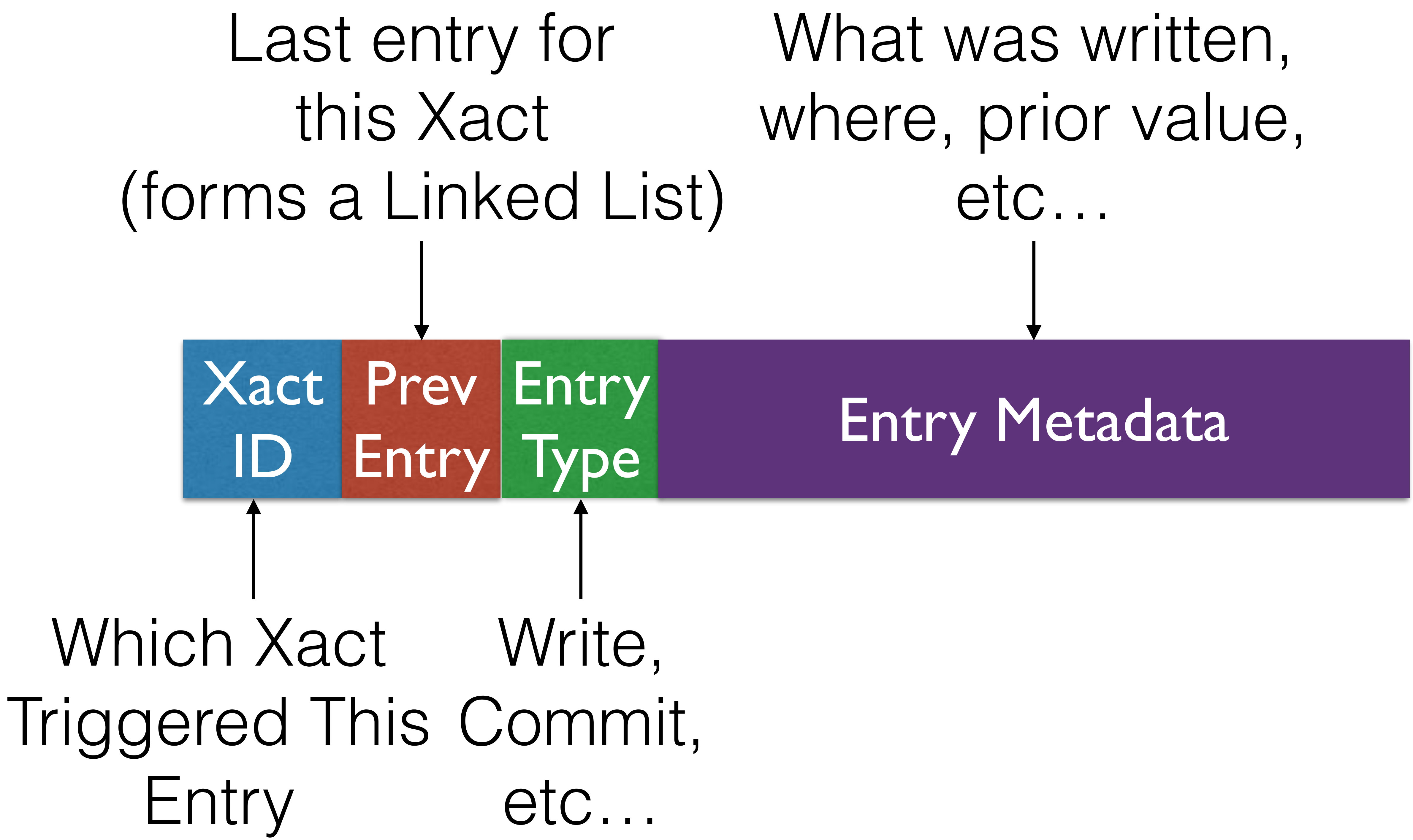
...allowing random writes
to be safely batched

Log

```
W(A:10)
W(C:8)
W(E:9)
```

A	~~3~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Anatomy of a log entry



Problem 2: Providing rollback.

Single DB Model

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

A	8
B	12
C	5
D	18
E	16

Single DB Model

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

A	~~8~~ 20
B	12
C	5
D	18
E	16

Single DB Model

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

A	~~8~~ 20
B	~~12~~
C	5
D	18
E	~~16~~ 19

Single DB Model

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

A	~~8~~ 20
B	~~12~~ 14
C	5
D	18
E	~~16~~ 19

Single DB Model

A	~~8~~ 20
B	~~12~~ ~~14~~ 15
C	5
D	18
E	~~16~~ 19

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

Staged DB Model

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

A	8
B	12
C	5
D	18
E	16

A	8
B	12
C	5
D	18
E	16

Staged DB Model

Txn 1

```
A = 20
B = 14
COMMIT
```

Txn 2

```
E = 19
B = 15
ABORT
```

A	~~8~~ 20
B	~~12~~ 14
C	5
D	18
E	16

A	8
B	~~12~~ 15
C	5
D	18
E	~~16~~ 19

Staged DB Model

<u>Txn 1</u>

```
A = 20
B = 14
COMMIT
```

<u>Txn 2</u>

```
E = 19
B = 15
ABORT
```

A	~~8~~ 20
B	~~12~~ 14
C	5
D	18
E	16

Is staging always possible?

- Staging takes up more memory.

- Merging after-the-fact can be harder.

- Merging after-the-fact introduces more latency!

for the single database model

Problem 2: Providing rollback.

^

UNDO Logging

Store both the "old" and the "new" values of the record being replaced

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Log

```
W(A:8→10)
W(C:5→8)
W(E:16→9)
```

UNDO Logging

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Active Xacts

Xact:1, Log: 45

Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

UNDO Logging

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

Active Xacts

Xact:1, Log: 45
ABORT
Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

UNDO Logging

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	16

Active Xacts

Xact:1, Log: 45 **ABORT**

Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

UNDO Logging

A	~~8~~ 10
B	12
C	5
D	18
E	16

Active Xacts

ABORT

Xact:1, Log: 45

Xact:2, Log: 32

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

UNDO Logging

A	8
B	12
C	5
D	18
E	16

Active Xacts

ABORT

Xact:1, Log: 45

Xact:2, Log: 32

Log

```
➡ 43: W(A:8→10)
  44: W(C:5→8)
  45: W(E:16→9)
```

Problem 3: Providing atomicity.

Goal: Be able to reconstruct all state at the time of the DB's crash (minus all running xacts)

Transaction Table

Transaction	Status	Last Log Entry
Transaction 24	VALIDATING	99
Transaction 38	COMMITTING	85
Transaction 42	ABORTING	87
Transaction 56	ACTIVE	100

Buffer Manager

Page	Status	First Log Entry	Data
24	DIRTY	47	01011010...
30	CLEAN	n/a	11001101...
52	DIRTY	107	10100010...
57	DIRTY	87	01001101...
66	CLEAN	n/a	01001011...

DB State

On-Disk
(or rebuildable)

A	~~8~~ 10
B	12
C	~~5~~ 8
D	18
E	~~16~~ 9

In-Memory Only!

Active Xacts

Xact:1, Log: 45

Xact:2, Log: 32

On-Disk

Log

```
43: W(A:8→10)
44: W(C:5→8)
45: W(E:16→9)
```

ARIES Recovery

1. Rebuild Transaction Table
2. Rebuild Buffer Manager State
3. ABORT Crashed Transactions

Transaction Table

Step 1: Rebuild Transaction Table

- Log all state changes
- Replay state change log entries

Required Log Entries

Log every COMMIT
(replay triggers commit process)

Log every ABORT
(replay triggers abort process)

New message: END
(replay removes Xact from Xact Table)

What about BEGIN?
(when does an Xact get added to the Table?)

Transaction Commit

- Write **Commit** Record to Log
- All Log records up to the transaction's LastLSN are flushed.
    - Note that Log Flushes are Sequential, Synchronous Writes to Disk
- Commit() returns.
- Write **End** record to log.

Speeding Up Recovery

- **Problem**: We might need to scan to the very beginning of the log to recover the full state of the Xact table (& Buffer Manager)
- **Solution**: Periodically save (checkpoint) the Xact table to the log.
  - Only need to scan the log up to the last (successful) checkpoint.

Checkpointing

- **begin_checkpoint** record indicates when the checkpoint began.
    - Checkpoint covers all log entries before this entry.
- **end_checkpoint** record contains the current transaction table and the dirty page table.
    - Signifies that the checkpoint is now stable.

Buffer Manager

Step 2: Recover Buffered Data

- Where do we get the buffered data from?

Save Dirty Page Table with Checkpoint

Consistency

Step 3: Undo incomplete xacts

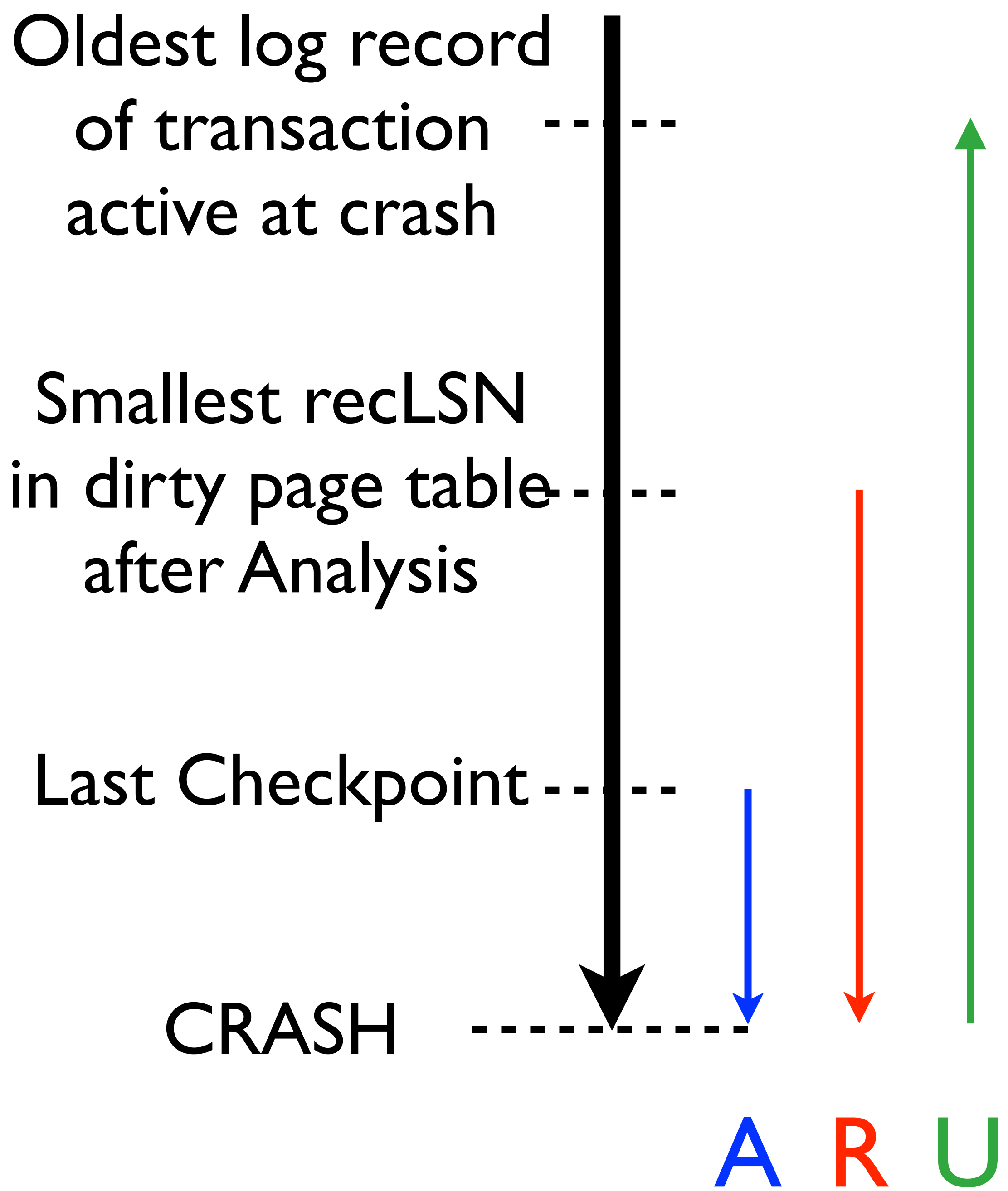
- Record *previous values* with log entries
- Replay log in reverse (linked list of entries)
  - Which Xacts do we undo?
  - Which log entries do we undo?
  - How far in the log do we need to go?

Compensation Log Records

- **Problem**: Step 3 is expensive!
  - What if we crash during step 3?
- **Optimization**: Log undos as writes as they are performed (CLRs).
  - Less repeat computation if we crash during recovery
  - Shifts effort to step 2 (replay)
  - CLRs don’t need to be undone!

ARIES Crash Recovery

- Start from checkpoint stored in master record.
- Analysis: Rebuild the Xact Table
- Redo: Replay operations from all live Xacts (even uncommitted ones).
- Undo: Revert operations from all uncommitted/aborted Xacts.



Recovery Example

LSN	Log
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update: T2 writes P3
30	T1 Abort
40	CLR Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5

PrevLSNs

CRASH! Restart!

Analysis

LSN	Log
⇒ 00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update: T2 writes P3
30	T1 Abort
40	CLR Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5

Xact Table

T1; 10

T2; 60

T3; 50

DP Table

P5; 10

P3; 20

P1; 50

Redo

LSN	Log
00	begin_checkpoint
05	end_checkpoint
10	update: T1 writes P5
20	update: T2 writes P3
30	T1 Abort
40	CLR Undo T1 LSN 10
45	T1 End
50	update: T3 writes P1
60	update: T2 writes P5

Xact Table

T2; 60

T3; 50

DP Table

P5; 10

P3; 20

P1; 50

Undo

LSN	Log
00,05	begin_checkpoint, end_checkpoint
10	update: T1 writes P5
20	update: T2 writes P3
30	T1 Abort
40, 45	CLR Undo T1 LSN 10; T1 End
50	update: T3 writes P1
60	update: T2 writes P5
70	CRASH
80	CLR: Undo T2, LSN 60
90,95	CLR: Undo T3, LSN 50; T3 End
100	CRASH
110	CLR: Undo T2, LSN 20; T2 End

CRASH!

Xact Table

T2; 60

T3; 50

ToUndo

60

50

20