# Solutions to problems for CSE 444 section 4 (April 23, 2009)

## Slide 2: Why do we need to recover a DB?

System failures can cause the DBMS to crash:

- OS crashes
- Viruses and malware
- Power failure

When the DBMS comes back up, the database must be returned to a state consistent with the atomic transaction concept before we can accept queries again – that is, every transaction must be either complete or have their effects on the database canceled.

## Slide 3: How can logging make recovery easier/better?

- **Atomicity of transactions:** after recovery, transactions that were incomplete in the crash have their effects erased, either by changing back the data they changed (undo logging) or replaying the old values set by complete transactions (redo logging).
- **Durability of transactions:** after recovery, transactions that completed in the crash have their effects restored, so these transactions continue to persist.

But logging-based recovery makes the assumption that neither the log nor the data already on disk were corrupted by the crash – logging does not help against disk crashes.

## Slide 5: An undo logging problem

- A can be written after log entry 2
- B can be written after log entry 3
- C must be written after log entry 5 and before log entry 12
- D can be written after log entry 7
- E must be written after log entry 8 and before log entry 12
- F can be written after log entry 10
- G can be written after log entry 11

General rule: When using an undo log, we can only write each datum to disk after the log entry for that item's change and before the COMMIT for the transaction.

## Slide 6: Undo logging problem, continued

The recovery proceeds as follows:

1. Find uncommitted transactions (T1, T3, and T4), possibly concurrently with the data changes made in step 2.
2. Make the following changes to the various data, in that order (latest to earliest):
    a. G=g
    b. F=f
    c. D=d
    d. B=b
    e. A=a
3. Write <ABORT T1>, <ABORT T3>, and <ABORT T4> to the log on disk.

Questions to ponder:

- **Why do we ignore committed transactions?** Because COMMIT must come after all writing to disk, we know all committed transactions have written everything to disk, so their effects are already "safe".
- **Why do we run backward?** To make sure that after we have undone all the incomplete transactions, then the values of the database elements will be the most recent values *from committed transactions*. In this example this is not a problem, because no two transactions write the same database element. However, if we change log entry 11 from <T3, G, g> to <T3, F, $f_{new}$> (where $f_{new}$ is the value of F that replaced f in the action of log entry 10), we see that doing the undoes from earliest to latest would leave value $f_{new}$ in F, even though $f_{new}$ was written by T3 which did not commit.
- **Why do we need to write ABORT records for incomplete transactions?** To preserve the atomicity constraint – that, in both the data blocks and log on disk, transactions either completed fully (either committing or aborting) or never started.

### Slide 7: Undo logging problem, part 3

The recovery proceeds the same way as in slide 6, except that we do not change F and G, do change C=c and E=e, and add <ABORT T2>. That is, we do the following:

1. Find uncommitted transactions, possibly concurrently with the data changes made in step 2.
2. Make the following changes to the various data, in that order (latest to earliest):
    a. E=e
    b. D=d
    c. C=c
    d. B=b
    e. A=a
3. Write <ABORT T1>, <ABORT T3>, and <ABORT T4> to the log on disk.

### Slide 8: What if it was a redo log?

- Everything except C and E can never be written.
- C and E can be written after log entry 12.

General rule: When using a redo log, we can write a datum to disk only after its update log entry, *and* the COMMIT entry for its transaction, have been written to disk.

### Slide 9: Redo log problem, continued

The recovery proceeds as follows:

1. Find committed transactions (only T2); this cannot be done concurrently with the data changes of step 2.
2. Make the following changes to the various data, in that order (latest to earliest):
    a. C=c
    b. E=e
3. Write <ABORT T1>, <ABORT T3>, and <ABORT T4> to the log on disk.

Questions to ponder:

- **Why do we only redo committed transactions?** Because COMMIT must precede any writing to disk, we know that uncommitted transactions have not written anything, so they have no effects and can be ignored.
- **Why do we go forward?** To ensure that when we are done, we have redone the most recent updates.

### Slide 10: Log checkpointing

We add checkpoints so we don't have to read the full log to do recovery. With checkpoints, we only need to read from or to a point not far from the start of the most recent checkpoint (or the immediately previous one in some cases).

### Slide 11: Garcia-Molina, problem 17.2.7 (i)

- Entry 2: END CKPT allowed after entry 3
- 5: after 14
- 7: after 14
- 10: after 16
- 13: after 16

General rule: when checkpointing an undo log, we need to wait for all transactions active at the start of the checkpoint to complete before ending the checkpoint.

### Slide 12: Checkpoints look different in undo and redo logs!

The left-hand log can only be a redo log, while the right-hand log can be either an undo or a redo log. This is because the left-hand log has END CKPT before T2 has committed, which is legal only in redo logs, while the right-hand log has END CKPT after both T2 and T3, which is allowed in both undo and redo logs.

General rules:
Recall that in an undo log, once we start a checkpoint we need to wait until all transactions that were incomplete at the START CKPT commit or abort.. Then, on recovery, we know that all incomplete transactions (the ones to be undone) occurred after the checkpoint started, so we only need to read the log to the start of the most recent checkpoint. (If there was a crash during the checkpoint, then we see START CKPT before END CKPT, and we also need to read back until the start of TX in the checkpoint that did not complete during the checkpoint.)

In a redo log, we just need to flush all committed transactions to disk before ending the checkpoint, so we don't wait for all incomplete transactions. Then, we know to recover only transactions still active at the most recent START CKPT or which became active after that, since those transactions are the only ones not guaranteed to have their effects written out to disk when the checkpoint ends. (If there was a crash during the checkpoint, we also need to find the previous checkpoint and redo transactions that were either active during that checkpoint or had started during that one.)

In both cases, we need to mark the start of transactions explicitly; otherwise we may not know where to stop looking for the effects of certain transactions (where that rather than the start of a checkpoint is the sign for us to stop working).

### Slide 13: Undo-log recovery with checkpoints

Answers to each question are given in the order that the questions were asked.

1. We read backward until we see the START CKPT (log entry 9). This is because we previously saw an END CKPT (log entry 16), so we know that all the transactions active when the checkpoint started have already completed. Thus, neither those transactions nor the ones that completed prior to the START CKPT need to be examined for possible undo.
2. We find that we don't actually need to change any data because all the transactions committed before the crash, so there are no uncommitted transactions that need to be undone.

### Slide 14: Redo-log recovery with checkpoints

Answers to each question are given in the order that the questions were asked.

1. We read backward until we see the START CKPT (log entry 9), to identify which transactions must be redone from the START CKPT and any START entries that follow. Once we have identified the transactions to redo, we continue reading backward until we find the earliest of the START records for those transactions. Finally, we redo all the transactions by reading forward from that START to the end of the log. Because we previously saw an END CKPT (log entry 16), we know the crash didn't occur during a checkpoint, so it's not necessary to search for a previous checkpoint and redo transactions that were active during or after *that* checkpoint.
2. The transactions redone are T2 and T3 (in the START CKPT header, so they were active when the checkpoint started) and T4 (started during the checkpoint). All these transactions committed, but they must be redone nonetheless because we don't know whether they were flushed to disk or not. This is because a redo log checkpoint only guarantees that transactions committed *before* the start of the checkpoint are flushed to disk.
3. Reading from log entry 4 (START T2), we make the following changes, in this order:
   - C=c
   - D=d
   - E=e
   - F=f
   - G=g

### Slide 15: A slightly harder problem

There are 3 possible schedules:

a. <START T>; <T, A, 10>; <T, B, 20>; <OUTPUT A>; <OUTPUT B>; <COMMIT T>
b. Same as (a), but swap <OUTPUT A> and <OUTPUT B>
c. <START T>; <T, A, 10>; <OUTPUT A>; <T, B, 20>;<OUTPUT B>; <COMMIT T>

As discussed previously (slide 5), the general rule is that any schedule is acceptable if the event <OUTPUT Ai> follows <T, Ai, ai> and precedes <COMMIT T>.

## Slide 16: A slightly harder problem, continued

For the example, there are 15 possible schedules, which can be listed, albeit tediously.

General rule:
Remember that all schedules are acceptable if every OUTPUT event follows its update log entry and precedes the COMMIT.

Formally, schedules are acceptable if they are of form:
<START T>; <T, A1, a1>; ... <T, An, an>; <COMMIT T>
where
<OUTPUT An>: can be placed in only 1 interval,
namely between <T,An,an> and <COMMIT T>

<OUTPUT An-1>: can be placed in 3 intervals,
defined by the points <T,An-1>, <T,An> <Out An> <Commit>

<OUTPUT An-2>: can be placed 5 intervals, defined by the points:
<T, An-2>, <T, An-1>, <T,An>, <Commit> and the points
<OUTPUT An-1>, <OUTPUT An>

And so on... giving 1*3*5* . . . * (2n-1) possible orderings.