Databases
Recovery: on the fly checkpointing

Ad Feelders & Hans Philippi

Universiteit Utrecht

March 13, 2024

Logging

Information related to each transaction must be written to disk in the following order:

- UNDO (order requirement of (1) – (2) is per element)
  1. The log records indicating changed database elements
  2. The changed database elements (old values)
  3. The `COMMIT` log record
- REDO
  1. The log records indicating changed database elements
  2. The `COMMIT` log record
  3. The changed database elements (new values)
- UNDO/REDO (order requirement of (1) – (2) is per element)
  1. The log records indicating changed database elements (old and new values)
  2. The changed database elements

  The <`COMMIT T`> log record can precede or follow any of the changes to the database elements on disk

Implications for recovery

- UNDO
  - Committed transactions can be ignored, because all their changes have been written to disk
  - Uncommitted transactions must be undone, because some changes may have reached the disk, leaving the database in an inconsistent state
- REDO
  - Committed transactions must be redone, because some of their changes may *not* have reached the disk, leaving the database in an inconsistent state
  - Uncommitted transactions can be ignored, because none of their changes has been written to disk
- UNDO/REDO
  - Committed transactions must be redone
  - Uncommitted transactions must be undone

Checkpoints: non-quiescent

- UNDO
  1. `T1,...,Tk` are the active transactions when starting checkpointing
  2. Write a log record `<START CKPT(T1,...,Tk)>` and flush the log, and continue processing
  3. Wait until all of `T1,...,Tk` commit or abort ...
  4. ... which implicitly means that all data of committed transactions in `(T1,...,Tk)` has been flushed
  5. Write `<END CKPT>` and flush the log

Example UNDO logging

```
<START T1>
<T1, B, 10>
<COMMIT T1>
<START T3>
<START T2>
<T2, A, 5>
<T3, C, 10>
<START CKPT (T2, T3)>
<T2, D, 15>
<START T4>
<START T5>
<T4, F, 25>
<T5, H, 12>
<COMMIT T2>
```

Consider a crash occurring after `<COMMIT T2>`

Crash after <COMMIT T2>

- Scanning the log backwards from the last entry, the first checkpoint entry we encounter is <START CKPT (T2, T3)>
- This means that it is possible that there are updates by T3 before this <START CKPT (T2, T3)> that need to be undone
- In this case, that means that we have to scan backwards further to <START T3>
- We have to UNDO all updates by T3, T4, T5 since they did not commit before the crash
- REDO? No, we don't!

Example UNDO logging

```
<START T1>
<T1, B, 10>
<COMMIT T1>
<START T3>
<START T2>
<T2, A, 5>
<T3, C, 10>
<START CKPT (T2, T3)>
<T2, D, 15>
<START T4>
<START T5>
<T4, F, 25>
<T5, H, 12>
<COMMIT T2>
<COMMIT T3>
<END CKPT>
<COMMIT T4>
<T5, G, 30>
```

Consider a crash occurring after <T5, G, 30>

Crash after <T5, G, 30>

- Scanning the log backwards from the last entry, the first checkpoint entry we encounter is <END CKPT>; therefore we have to scan back to <START CKPT (T2, T3)>
- T4 started after the checkpoint, but has already committed
- We have to UNDO updates by T5 since it did not commit before the crash
- T5 started after the checkpoint, so there are no things to undo for T5 before the checkpoint
- REDO? No, we don't!

Checkpoints: non-quiescent

- REDO
  1. `T1,...,Tk` are the active transactions when starting checkpointing
  2. Write a log record `<START CKPT(T1,...,Tk)>` and flush the log, and continue processing
  3. Flush "dirty buffers" of transactions that committed before `<START CKPT>`
  4. Write `<END CKPT>` and flush the log

Example REDO logging

```
<START T1>
<T1, B, 10>
<COMMIT T1>
<START T3>
<START T2>
<T2, A, 5>
<T3, C, 10>
<START CKPT (T2, T3)>
<T2, D, 15>
<START T4>
<START T5>
<T4, F, 25>
<T5, H, 12>
<COMMIT T2>
```

Consider a crash occurring after `<COMMIT T2>`

Crash after <COMMIT T2>

- Scanning the log backwards from the last entry, the first checkpoint entry we encounter is <START CKPT (T2, T3)>; therefore we have to scan all the way back to the beginning of the log unless we encounter an <END CKPT> first
- Since in this case we don't, we scan the log backwards all the way to the beginning
- We have to redo updates of T1 and T2, because they have been committed
- UNDO? No, we don't!

Example REDO logging

```
<START T1>
<T1, B, 10>
<COMMIT T1>
<START T3>
<START T2>
<T2, A, 5>
<T3, C, 10>
<START CKPT (T2, T3)>
<T2, D, 15>
<START T4>
<START T5>
<T4, F, 25>
<T5, H, 12>
<COMMIT T2>
<COMMIT T3>
<END CKPT>
<COMMIT T4>
<T5, G, 30>
```

Consider a crash occurring after <T5, G, 30>

Crash after <T5, G, 30>

- Scanning the log backwards from the last entry, the first checkpoint entry we encounter is <END CKPT>; therefore we have to scan all the way back to <START CKPT (T2, T3)>
- On the way, we see that T2, T3 and T4 have been committed
- T5 is not committed, so it needs no redoing
- <START CKPT (T2, T3)> in combination with <END CKPT> guarantees that all data of all committed transactions before <START CKPT (T2, T3)> have been flushed (i.e. T1)
- But we need to redo all updates of T2 and T3 ...
- ... so we have to scan back further to <START T3>
- REDO actions for T2, T3 and T4
- UNDO? No, we don't!

Checkpoints: non-quiescent

- UNDO/REDO
  1. `T1,...,Tk` are the active transactions when starting checkpointing
  2. Write a log record `<START CKPT(T1,...,Tk)>` and flush the log, and continue processing
  3. Flush *all* buffers created before the START CKPT
  4. Write `<END CKPT>` and flush the log

UNDO/REDO recovery policy

1. Redo all committed transactions in order earliest-first
2. Undo all incomplete transactions in order latest-first

Example

Combined UNDO/REDO logging

```
<START T1>
<T1, B, 10, 11>
<COMMIT T1>
<START T3>
<START T2>
<T2, A, 5, 6>
<T3, C, 10, 11>
<START CKPT (T2, T3)>
<T2, D, 15, 16>
<START T4>
<START T5>
<T4, F, 25, 26>
<T5, H, 12, 13>
<COMMIT T2>
```

Consider a crash occurring after `<COMMIT T2>`

Crash after <COMMIT T2>

- Since we use UNDO/REDO logging, committed transactions should be REDONE, and uncommitted (i.e. active) transactions should be UNDONE
- Scanning the log backwards from the last entry, the first checkpoint entry we encounter is <START CKPT (T2, T3)>; therefore we have to scan all the way back to the beginning of the log unless we encounter an <END CKPT> first
- Since in this case we don't, we scan the log backwards all the way to the beginning
- We have to REDO updates by (T1, T2) because they committed before the crash
- We have to UNDO updates by (T3, T4, T5) since they did not commit before the crash

Example

Combined UNDO/REDO logging

```
<START T1>
<T1, B, 10, 11>
<COMMIT T1>
<START T3>
<START T2>
<T2, A, 5, 6>
<T3, C, 10, 11>
<START CKPT (T2, T3)>
<T2, D, 15, 16>
<START T4>
<START T5>
<T4, F, 25, 26>
<T5, H, 12, 13>
<COMMIT T2>
<COMMIT T3>
<END CKPT>
<COMMIT T4>
<T5, G, 30, 31>
```

Consider a crash occurring after `<T5, G, 30, 31>`

Crash after <T5, G, 30, 31>

- Scanning the log backwards from the last entry, the first checkpoint entry we encounter is <END CKPT>; therefore we can be sure that all buffers before <START CKPT (T2, T3)> have been written to disk
- We have to UNDO updates of T5 since it did not commit
- We have to REDO updates of T4 and T2 because they committed before the crash, but not before <START CKPT>, and they made changes after <START CKPT>
- We don't have to REDO updates of T1 because it committed before <START CKPT>
- We don't have to REDO updates of T3 because it didn't make any changes after <START CKPT>
- We scan the log backwards until we have seen <START CKPT (T2, T3)>; the dirty pages of T2 and T3 written before <START CKPT> have been flushed