# IN3020/4020 – Database Systems
# Spring 2021, Week 7.2

# Transaction Management: ACID Characteristics and Logging

Dr. M. Naci Akkøk, Chief Architect, Oracle Nordics

Based upon slides by E. Thorstensen from Spring 2019

## Transactions

We want to be able to perform operations where we guarantee the following ACID characteristics:

- **Atomicity:** The operation is performed correctly or not at all
- **Consistency:** The database should always remain consistent, or be consistent
- **Isolation:** Users should not have to coordinate operations in between
- **Durability:** Performed operations should remain performed

The operations are transactions - collections of queries.

To guarantee ACID, we must have some mechanisms.

## The challenges

- Interruptions:
  - Machine crash, process failure, disk failure.
- Conflicts:
  - Between two transactions
  - Between a transaction and the integrity rules
- In addition, we may wish to manually cancel a transaction.

## A simple example

```
UPDATE accounts SET balance = balance - 100.00
  WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
  WHERE name = (SELECT branch_name FROM accounts
    WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
  WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
  WHERE name = (SELECT branch_name FROM accounts
    WHERE name = 'Bob');
```

Nice if this is not executed halfway.
Also, nice that another transaction does not read incorrect data.

## Parallel execution (concurrency)

- We can easily guarantee isolation: Only one transaction gets access at a time.
- The price is that everyone must wait a “longer” time. Not acceptable.
- If several people are reading and writing at the same time, we must be able to say something about what the outcome will be when everyone finishes (at the end of all concurrent transactions).
- If it is not possible to get a good result, then we have a conflict. One of the transactions must then be rolled back.

## Conflict example

```
-- Transaction 1 BEGIN;
UPDATE accounts SET balance = balance - 100.00
  WHERE acctnum = 12345;
UPDATE accounts SET balance = balance + 100.00
  WHERE acctnum = 7534;
COMMIT;

-- Transaction 2 BEGIN;
UPDATE accounts SET balance = balance - 100.00
  WHERE acctnum = 12345;
UPDATE accounts SET balance = balance + 100.00
  WHERE acctnum = 3456;
COMMIT;
```

If both read initial balance (say 200), then both will reduce balance by 100, but end up paying 100 for each of the two other accounts! Reduced 100, paid 200. Very good deal for someone, but not for the bank, not for the DBMS.

## Interaction between transactions

- So, we do want as much parallelism as possible.
- At the same time, we want to be able to demand that the result be as if there was no parallelism (serializable execution). As if each transaction lived its life totally independently.
- This will cost us. Thus, it is also nice to be able to demand something less demanding.
- Several requirements result in more expensive execution.
- Whatever we demand, there must be mechanisms to guarantee it.

## We want to implement ACID constraints

For that, we need several mechanisms,
like **Concurrency Control**

## Two main types of concurrency control

- Optimistic and pessimistic.
- **Optimistic:** Drive on! If we get a prohibited interaction, we simply clean up (via rollback).
- **Pessimistic:** We guarantee that no prohibited interactions can occur. Implies limited parallelism.
- Benefit of optimistic concurrency: If things rarely go wrong, we won't have to do extra work.
- Benefit of pessimistic concurrency: If things do often go wrong, we don't have loads of rollbacks.

## Wishlist

- We see that violations of A, C, and D can occur both from interruptions and from problems around parallel execution.
- In the least, we need a rollback mechanism (to be able to  undo)
- Also great if we can continue after a crash and complete a started transaction (allows us to write to disk in chunks).

## Typical use-case

- The server dies. Some transactions were canceled halfway. They have managed to write only some of their updates to disk.
- We are now in an inconsistent state (if not, it is by pure luck).
- Need to restore consistent state: The incomplete transactions that cannot be completed must be rolled back (their changes must be undone).

- Or… are there other alternatives? Discuss!

## Logging

- Main idea: We have a log file where we write information that will allow us to perform rollback / recovery.
- In other words: Before something is done with data on disk, what is to be done is written into the log-file on disk (Write-Ahead Logging, WAL).

Exam question ☺

- The log will become big quickly, but it can be managed.
- The log is sequential, so it is cheap to write to it.

## Log types

- Two intuitive approaches:

  - UNDO: The log contains the old value when updating
  - REDO: The log contains the new value.

- We can combine UNDO / REDO. They have their advantages and disadvantages.

## Log entry types

- Let's use a simplified model with the following log entry types:
  - `START T`
  - `OLDV (T, X, v)`
    T changed X (a "data element"), and v was the old value.
  - `NEWV (T, X, v)`
    T changed X, and v is the new value.
  - `UPD (T, X, o, n)`
    T changed X from o to n (from old to new)
  - `ABORT T`
  - `COMMIT T`

Whatever that is. Could be a tuple, a record an attribute value.

## UNDO-Logging

- We write `OLDV (T, X, v)` entries in the log, as well as `START T` and `COMMIT / ABORT`
- For this to work, the following rules must be observed:
  - `START T` must be logged onto disk before anything else about T.
  - An `OLDV` record in the log must be on disk before the update gets there.
  - `COMMIT T` should not be on disk until all data updates are on disk.
- Otherwise, the order of the individual log entries / data updates do not matter.

## Why/how does this work?

- Remember that we are in recovery mode: We only have data on disk and data in log (we´re not dealing with the external world).
- We read the undo log backwards (from the last entry). If it says `COMMIT T`, we know that data is on disk. All is fine!
- If there is no no `COMMIT` / `ABORT` (but START T somewhere), we know that T may have made changes to disk, but <u>only</u> where we have `OLDV` records!
- We then change those disk entries to their old values.
- If multiple transactions have changed the same X and were canceled, we know that the order of the log entries OLDV $(T_1, X, v_1)$ and OLDV $(T_2, X, v_2)$ are correct and reflect the disk - so we can change them in the right order.

## Checkpointing

- As we pointed out, the log can be very long. We can't simply cut it anywhere randomly:

  ...
  OLDV $(T_5,$ X, v)
  OLDV $(T_7,$ Y, w)
  **COMMIT $T_4$**
  ...

- Before `COMMIT` $T_4$, there may be a long list of records we need.
- Checkpointing is a technique for "cutting" the log up.

## Checkpointing – UNDO log

- Simple idea: Stop accepting new transactions, wait for all the active transactions to end, and enter a special log entry CKPT.

- Big disadvantage: In practice, the system goes down for a period of time (we stop further transactions).

- Can delete the entire log before CKPT.

## Nonquiescent checkpointing

- Better idea: Include active transactions!

- Write `START CKPT (`$T_1$`,.., `$T_k$`)` to disk.
- Continue as usual until all $T_i$ are finished.
- Write `END CKPT` to disk.

## Recovery with checkpoint

- If we read an END CKPT, we know that all transactions that may have been interrupted started after `START CKPT (...)`.
- In other words, after `END CKPT` is written to disk, the log before `START CKPT` can be deleted.

- (Does one delete a log?)

## Interruptions during checkpoint

- If we have a `START CKPT (...)` but no `END CKPT`, then we were interrupted during checkpointing.

```
START CKPT (...) ...
   START T5
   ...
   OLDV (T5, Y, w) ...
END CKPT
   ...
   OLDV (T5, X, v) ...
START CKPT (...) ...
```

- Everything that started before this did complete!

## Summary of UNDO

- We write old values in log to disk before we actually write them into the tables on disk.
- We write COMMIT in log on disk after everything is on disk.
- This can lead to some random writes, and possible data loss during interruptions.
- Secures C and A, a bit questionable on D.

- By the way, what happens if <u>recovery</u> is interrupted?

## REDO logging

- Also known as roll-forward recovery.
- We write `NEWV (T, X, v)` entries in the log, as well as `START / COMMIT / ABORT`.
- New rule: Before an update `(X, v)` can end up on disk, both `NEWV (T, X, v)` and `COMMIT T` must end up in log and on disk.
- In other words, none of the transaction updates can be flushed until the entire log for this transaction is flushed!

## What is the advantage of this rule?

- We can postpone the flush of data for as long as we want.
- Everything is in the log anyway!
- Rollback is free. Enter `ABORT T` in the log.

## Why does this work?

- We are in recovery. We have data only on disk and in the log.
- We read the log from scratch. If we have an interrupted transaction (`START` with no `COMMIT / ABORT`), we know it is not on disk! No recovery needed.
- Transactions that *have* `COMMIT T`, on the other hand, may have been interrupted while writing to disk.
- But, we know what they were meant to do, so we write the new values to disk.

## Checkpointing, REDO

- Enter `START CKPT` $(T_1,\ldots, T_k)$ to disk
- Write to disk everything that has to do with committed transactions (eventually wait for this “committing” to happen).
- Write `END CKPT` to disk.

- If we see an `END CKPT`, we know that everything that has `COMMIT T` before this point is on disk.
- If we see a `START CKPT` without an `END CKPT`, we must find the previous `END CKPT`.

## REDO checkpointing – deleting the log

```
START T5
START CKPT(T5, . . .) ...
...
NEWV(T5, Y, w) ...
END CKPT
...
START CKPT(T5, . . .) ...
END CKPT
START CKPT(T5, . . .) ...
```

- Must find `START` on all active transactions in the latest `CKPT`. Can delete things before the first such `START`. DISCUSS!

## REDO summary

- Write entire transactions to log on disk before they are written to the tables on disk.
- Allows us to collect writes, as well as recovery from backup.
- Secures A, C, <u>and</u> D.
- Disadvantage: Entire transactions must remain in the memory until `COMMIT` (can't write when we want to).

## UNDO / REDO

- We can combine these two. Avoids the problems of the two, but requires double logging
- `UPD (T, X, o, n)` entries. Includes both old and new value.
- New rule: `UPD (T, X, o, n)` must be on disk (in the log) before the corresponding table change.
- No interaction with `COMMIT`!
  We can postpone writing if we want to!

## Recovery under UNDO / REDO

- Must do both UNDO for interrupted transactions (those without `COMMIT` in log), …
- and `REDO` for those with `COMMIT` in log.

- We know nothing about the state of the disk:
  - There may be missing data that should be there,
  - And there may be data that should not be there.

# Checkpointing with UNDO / REDO

- Enter `START CKPT (`$T_1$`,..,` $T_k$`)` to disk
- Write all changed blocks in memory to disk.
- Write `END CKPT` to disk.

- We are allowed to write unfinished transactions.
- REDO must only be done from `END CKPT`.
- UNDO can span several CKPTs.

## UNDO / REDO summarized

- Benefits from both UNDO and REDO - flexible writing, easy checkpointing.
- Disadvantage: Double as much log.
- Postgres uses REDO.

- What does MySQL and Oracle use?

## Recovery from backup

- REDO logs allow us flexible backup.
- Take a data-dump once a day. Delete log with respec to checkpoint before the dump.
- If the data disk dies and the log is on another disk, we can run REDO from log and backup.

## Logging in Postgres

- Postgres allows unlogged tables. Ouch! Use with care!
- Postgres also allows asynchronous commit. The log records are then sent to disk before data, but not flushed.

  https://www.postgresql.org/docs/9.2/static/wal-async-commit.html
- Checkpointing (and thus flushing of data) is an expensive operation and can be configured so as not to overload the disk.