DIS Report: Assignment 4

Can Liu, Moritz Lahann

30.05.2021

a)

The program first checks if a previous log is present. If so, it initiates recovery, otherwise it runs the demo. In the demo, five client threads are created, which randomly write to some pages and commit transactions (repeating this multiple times) through a persistence manager. This is implemented as a singleton, so all threads speak to one PM. The persistence manager has an internal state which stores a buffer of operations and a set of all uncommitted transactions. Beginning, writing and committing are done through the PM. On a write, the PM keeps count of the log state with a monotonously increasing LSN (implemented as a synchronized counter). The PM saves the write action in it's internal state in a Hash Table, with page numbers as keys. If the same page is written to again, the value in the hash table can be overwritten, saving space (we don't need the actions to be atomic according to the specification). The PM (in the same write method still) saves the transaction in its state (if not present yet) or adds a new page to the transaction if required (if transaction already present). This is done with a hash table as well (transaction ID as key). The data objects are implemented as classes. The PM then writes a new line to the log file containing LSN, TransactionId, pageId, and data. It then checks its internal buffer to see if there are more than 5 pages present.

If so, the PM checks each transaction to see if it has been committed. If so, it gets the page numbers this transaction has written to, looks them up in the buffer, and writes them to text files. Committed transactions and persisted pages are removed from the internal state of the PM.

When a client commits a transaction, the PM logs this and sets a flag on the transaction in its internal state, marking it as committed (for the buffer check above).

This accomplishes recovery because committed but not persisted data can be recovered from the log file and persisted in the pages.

The recovery works as follows. First, all transactions from the log that were committed (EOT in log file with transaction id) are gathered. Only committed transactions are to be recovered. Persisting uncommitted changes would violate consistency. The log records are then sequentially ran through again. If a write belongs to a winner transaction, the page is read from the disk (if it exists).

If the LSN of the page is lower than the LSN of the log entry for the page (i.e. there were committed changes to the page that weren't persisted yet, it is stale) the new data and a new LSN (monotonously increasing) is saved for the page. The pages that are updated in recovery are saved in a hash table. If this was written to the file right away, only the first write on a page would ever go through because of the page LSN condition. When all pages to be recovered are gathered (with their most recent data from the log and new LSNs), they are then persisted on the disk. Now, the last committed state has been successfully recovered.

b)

i)

The program was manually stopped after a couple of seconds (exit code -1). In the logs, we can see that for example transaction 6 was writing to page 29, and earlier 21. This transaction is uncommitted. In the pages folder, there are no pages 29 or 21. Thus, the uncommitted write data has not been persisted. See Figure 1.

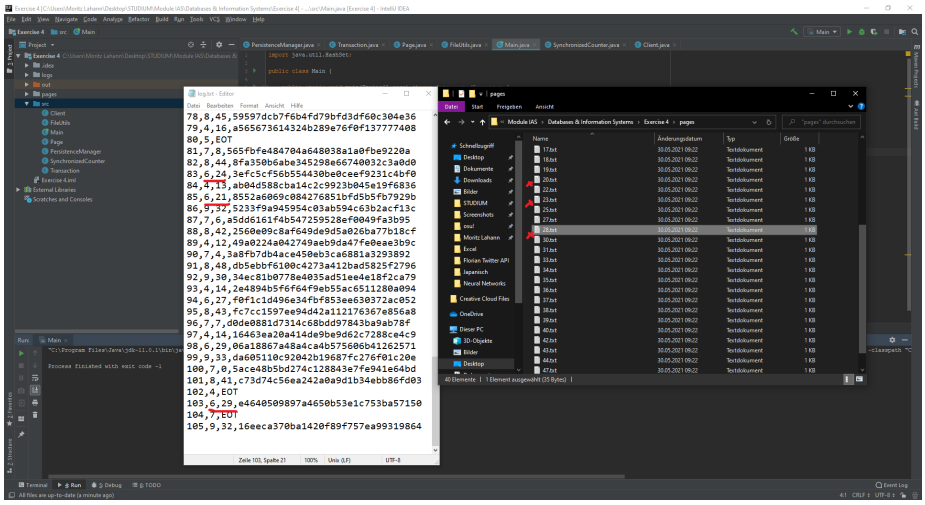

Figure 1: Uncommitted data is not persisted.

c)

i)

The program was manually stopped after a couple of seconds. Transaction 2 was committed, so two pages from it were modified: Page 31 was deleted, page 38 was modified without changing the LSN. The recovery was then started. After the recovery was finished, the program exited. Page 31 was restored with the previous user data (with a new LSN). Page 38 was not affected by the recovery, and was left in the changed state. This is because the LSN of page 38 was not changed, so to the recovery, the page was up-to-date. The edit of page 38 was done outside of a persistence management system. That violates the assumptions of the persistence manager. So the recovery was successful within the expectations, but it did not manage to fully restore the state. See Figure 2.

ii)

The buffer size was changed to 1000. The program was started and manually stopped after a while. The log had committed transactions, for example transaction 304 had written "78ed692bf0c244449db149e92f696453´´ to page 18. However, because the buffer wasn't full yet (we can reach max. 50 datasets with 5 clients with 10 pages each), the page wasn't persisted yet (pages folder wasn't created). After running the recovery, the page was now persisted with the correct value (the last write to this page in a committed transaction). The recovery was successful. See Figure 3.

iii)

With a buffer of 1000, the program was stopped after a while. Transaction 17 was committed but had not been persisted, and it had last written to page 0. This file was not persisted, not present. The file was manually created with a LSN of 145 (higher than the last LSN during runtime). Recovery was started. After recovery, the rest of the pages from committed transactions were persisted. The value of page 0 however was not recovered. This is because the LSN is higher than what was recorded in the log. A recovery should not overwrite changes that were later committed successfully (that would lose data and force another recovery from whatever wrote to that page with that LSN). See Figure 4 for screenshots.

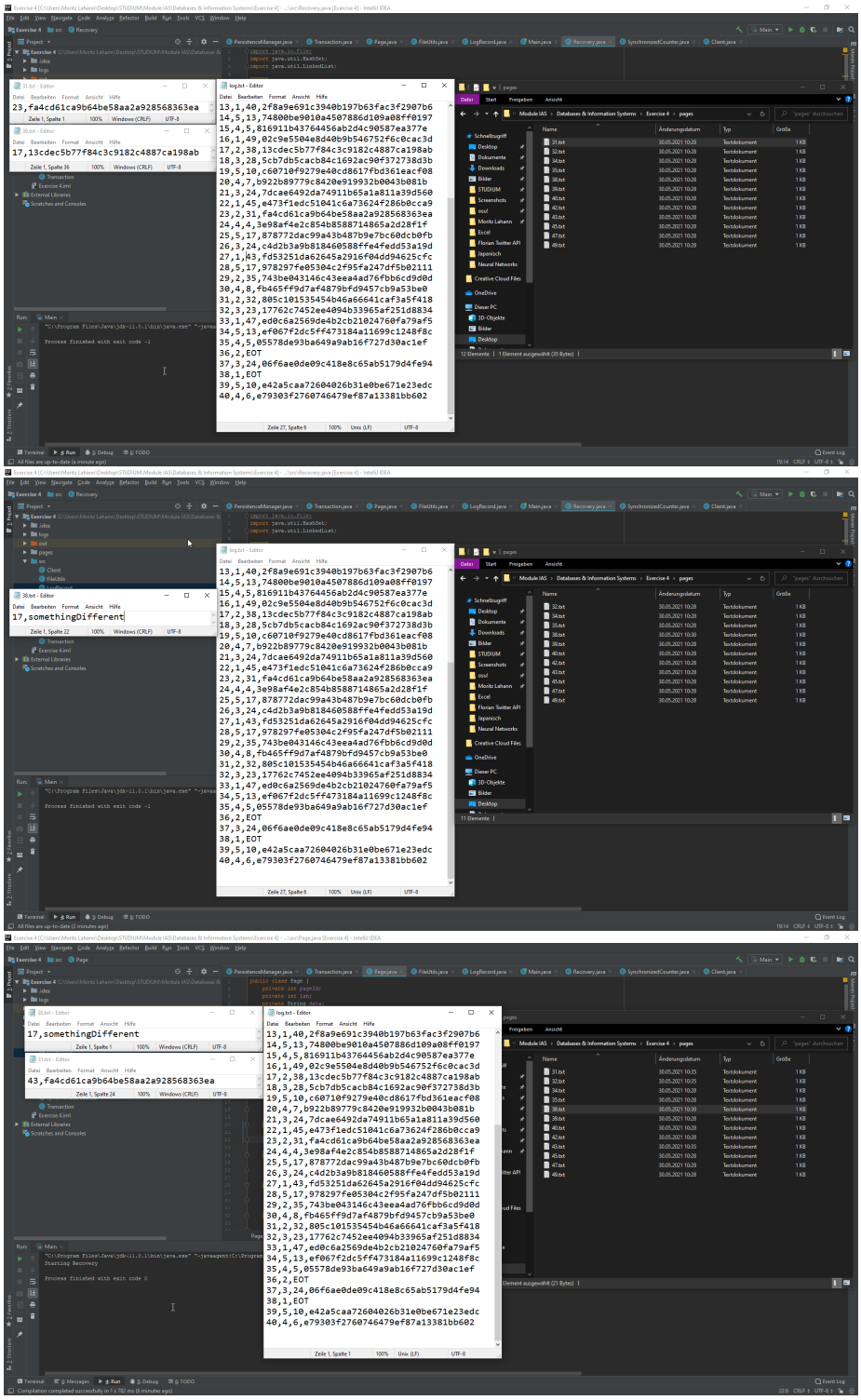

Figure 2: Recovery Process

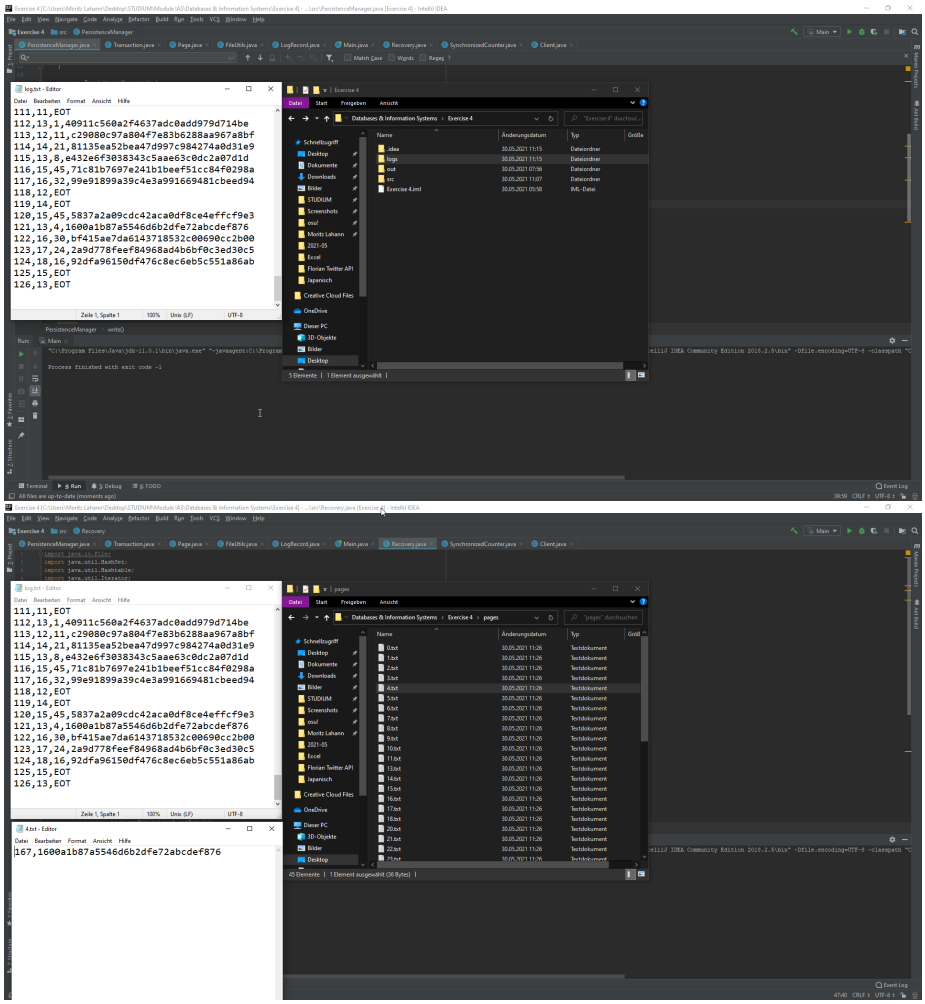

Figure 3: Recovery of non-persisted data

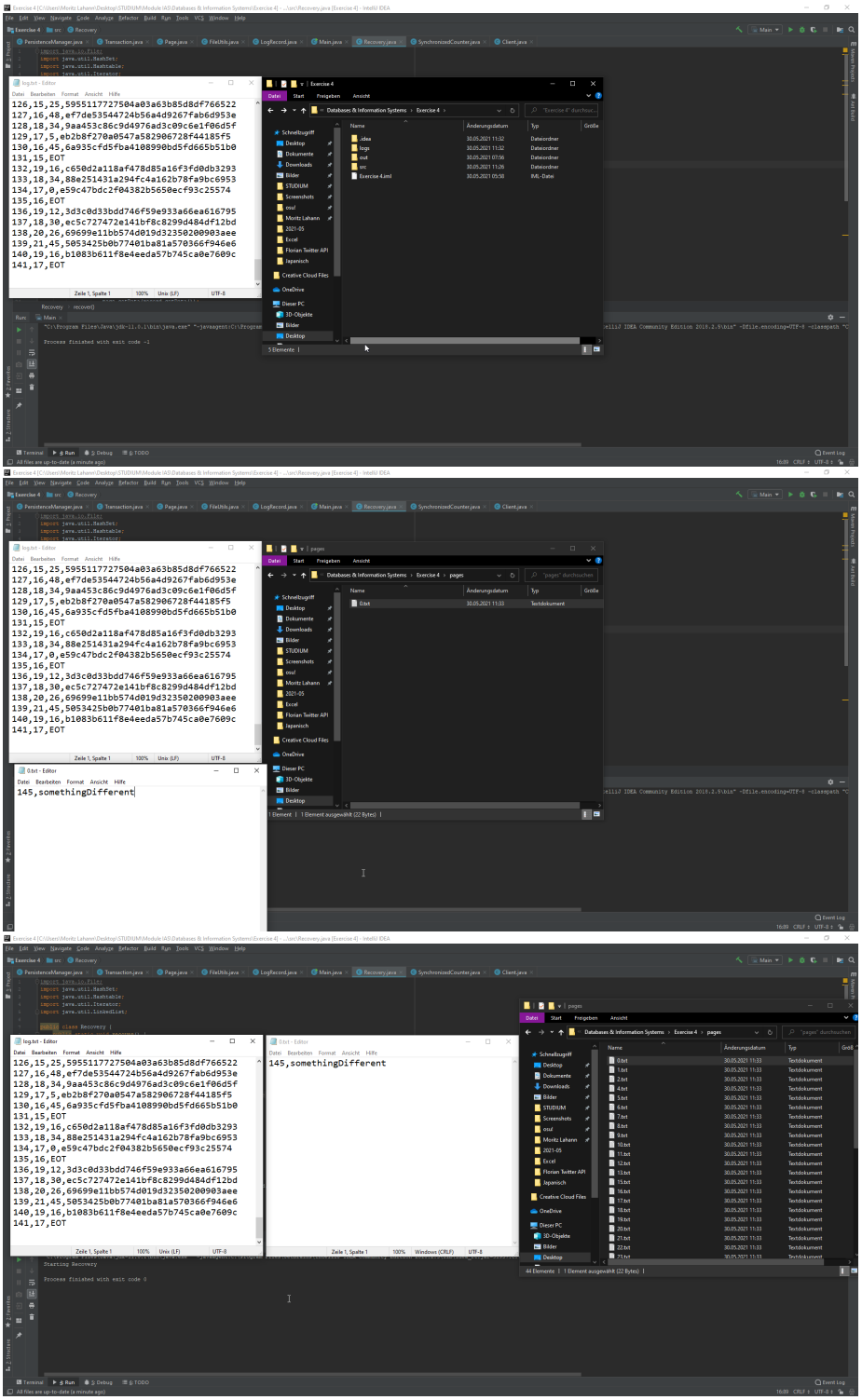

Figure 4: Recovery of non-persisted data with greater LSN