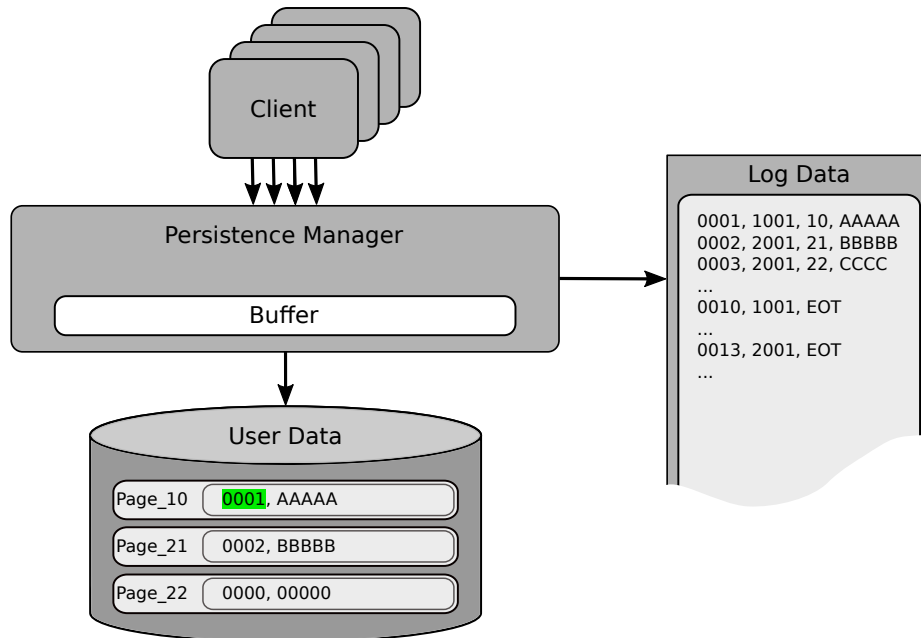
	Course	Databases and Information Systems 2020		
	Exercise Sheet	4		
	Points	–		
	Release Date	May 19 th 2020	Due Date	June 09 th 2020

4 Logging and Recovery




Note

- You can use a language of your own choice but be aware, that we might not be able to help you with your code if you use anything but Java or Python.
- Persistence manager and clients can be implemented as components of the same application, so that clients can be started as single threads accessing the only persistence manager instance (Singleton pattern).
- Use the following procedure to save user and log data: store every page in a single (text) file that is accessed by the persistence manager (e.g. via `FileReader` and `FileWriter` if you use Java). The name of a file corresponds to the page ID and every file contains a single line of text with all the information, for example separated by commas.
- The buffer can be implemented with a `Hashtable`. Besides the user data, the correspondence between transactions and datasets has to be administered and an overview over ongoing and already completed transactions has to be maintained.
- Creating checkpoints is not necessary.
- The implementation of a graphical user interface is neither required nor desired.

4.1 Persistence Administration

Realise the above architecture for a simplified persistence manager that allows concurrent requests from clients to user data, performs deferred writes for modified data and takes the necessary precautions for recovery after a system failure.

Besides the user data, the persistence manager also administers log data to keep track of modifications that have not yet been persisted. User data and log data have to be saved to a persistent storage.

	Course	Databases and Information Systems 2020		
	Exercise Sheet	4		
	Points	–		
	Release Date	May 19 th 2020	Due Date	June 09 th 2020

User data are stored as follows: one text file per page containing the log sequence number (LSN) and user data. The file name contains the page ID.

The log data are stored in the following way: One text file containing one line for each log entry. A log entry either consists of LSN, transaction ID, page ID and user data or of LSN, transaction ID and EOT (end of transaction).

Initially, user data that have been modified by a write operation are stored to the internal buffer of the persistence manager. Already existing versions can be overwritten directly in the buffer without having been written to the persistent storage.

The buffer may contain an arbitrary number of operations but if the buffer contains more than five datasets after a write operation, the persistence manager checks whether there are operations related to already committed transactions. If there are such operations, then these operations are written directly to the persistent storage (non-atomic). Thus, outdated datasets can reside in the persistent storage and datasets of committed transactions that have not been written to the persistent storage can reside in the buffer, but “dirty” datasets of uncommitted transactions cannot exist in the persistent storage (no force, no steal).

Log information has to be written to the persistent storage immediately and before the completion of a transaction, in particular, so that a recovery can be performed in the event of a system failure.

There's only one persistence manager that is accessed concurrently by several clients (remember the Singleton pattern and thread safety during concurrent access). The persistence manager should offer at least the following operations:

- `beginTransaction()`: starts a new transaction. The persistence manager creates a unique transaction ID and returns it to the client.
- `commit(int taid)`: commits the transaction specified by the given transaction ID.
- `write(int taid, int pageid, String data)`: writes the given data with the given page ID on behalf of the given transaction to the buffer. If the given page already exists, its content is replaced completely by the given data.

4.2 Clients


Implement a client class that can be started in several instances in parallel, so that all instances access the persistence manager concurrently (Singleton pattern).

The clients repeatedly execute transactions on the persistence manager according to the following scheme:

```
beginTransaction() write() write() ... commit()
```

For more realistic behaviour, every operation is followed by a brief pause. The number of writes in a transaction may vary.

In order to get along without locks, the clients do not access the same pages, i.e. Client 1 accesses pages 10..19, Client 2 accesses pages 20..29 and so forth. More than one page may be modified by one transaction. The user data may consist of simple strings.

	Course	<i>Databases and Information Systems 2020</i>		
	Exercise Sheet	4		
	Points	–		
	Release Date	May 19 th 2020	Due Date	June 09 th 2020

4.3 Recovery Tools

Due to the noforce, nosteal, non-atomic implementation of the persistence manager, no undo recovery is required after a system failure, but a redo recovery is. Implement a recovery tool that performs the analysis and redo phase of the crash recovery as discussed in the lecture. First, the so-called winner transactions have to be determined from the log data. After that, the pending write operations have to be executed. Remember updating the LSNs in the user data.

Content of your Report

The following questions and tasks should be answered by your report. Use screen shots for illustration and explanation purposes as well as to show your results. In case you did not implement a Java application, please provide a short introduction of how to install and use your program.

- a) Describe in your own words how the program works and how the persistence manager accomplishes logging and recovery.
- b) Test your logging and buffering by performing the following tasks. Make sure all pages and the log file are empty before every task. (Document your results using screen shots!):
 - i) Start your program and stop it after a while. If you can find operations related to uncommitted transactions in the log file you can proceed. Otherwise, start this task again. It may help to increase the number of operations per transaction.
Have operations related to the uncommitted transactions been persisted by your program?
- c) Test your recovery by performing the following tasks. Make sure all pages and the log file are empty before every task. (Document your results using screen shots!):
 - i) Run your program until it finishes (if it does not, stop it manually). Choose one page that has been written by a committed transaction and delete it. Choose another page that also has been written by a committed transaction and change the data of the page (Caution: Do not change the LSN!). Now, restart your program and stop at the end of the recovery. Compare current values of the two pages you chose with the values previous to your manipulation. Was your recovery successful?
 - ii) Change the size of your buffer from 5 to 1000. Start your program and stop it after a while. Find the latest write operation that belongs to a committed transaction but has not been persisted yet (If you cannot find one, increase the buffer size further, reduce the operations within the transactions or stop the program earlier).
What is the current state of the page accessed by the operation? Restart your program and stop at the end of the recovery. What is the state of the page now? Was the recovery successful?
 - iii) Change the size of your buffer from 5 to 1000. Start your program and stop it after a while. Find the latest write operation that belongs to a committed transaction but has not been persisted yet (If you cannot find one, increase the buffer size further, reduce the operations within the transactions or stop the program earlier).
Change the LSN of the corresponding page to a value higher than the last LSN in the log file. Restart your program and stop at the end of the recovery. Compare the current value of the page with its previous value. What do you observe? Briefly explain.