Team 17

107060008張承勛
107062123鄭宇軒
107062315黃偉哲

Implementation

- Modify LockTable
  We first modified the class `LockTable` to follow the compatibility among different kinds of locks provided in the spec of this assignment.

- Create per-transaction private work space
  Whenever a record is modified, `RecordPage.setVal()` will be called to set the modification into the buffer and therefore other transactions can see the modification. However, according to the specification, the modifications are invisible to other transactions, but are visible to the transaction making the modifications.

  Our method to accomplish this task is to create **private** List<Integer> offset; **private** List<RecordId> recordId; **private** List<Constant> modified; as the private work space. Any modification will be recorded into these arrays in `RecordPage.setVal()` instead of recording into the buffer directly.

  Furthermore, a transaction must be aware of what modification it has already made, so I also modified the implementation of `RecordPage.getVal()`. When it is called, it would search the desired data in the private work space first. If the data is not in the private work space, which means the field of a certain record is never modified, it draws the data from the buffer then.

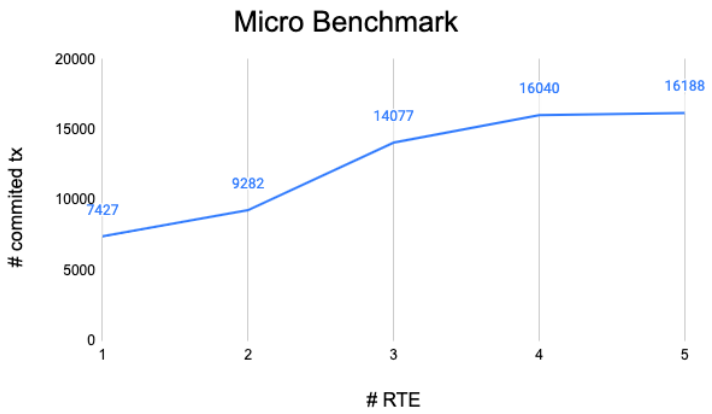
- In-place update the modified records
  According to the spec, before a transaction makes its modifications visible, it must convert all exclusive locks to cetify locks first. So I created a new method inside `concurMgr`, which is used to put a cetify lock on a certain record.

  Once a transaction commits, it must go through the private work space arrays and set these data into the buffer. Basically, we already have all the information we need in order to flush the private data into the buffer. We can obtain the `BlockId` from `RecordId.block()`, and then we can obtain the buffer from `bufferMgr().pin(BlockId)`.

What is worth-mentioning is that we must notice whether the block current record it belongs to is equal to the next record it belongs to. If not, we need to change the buffer that we are going to write data into.

Finally, clear all the private work space arrays in case they overflow.

Experiment

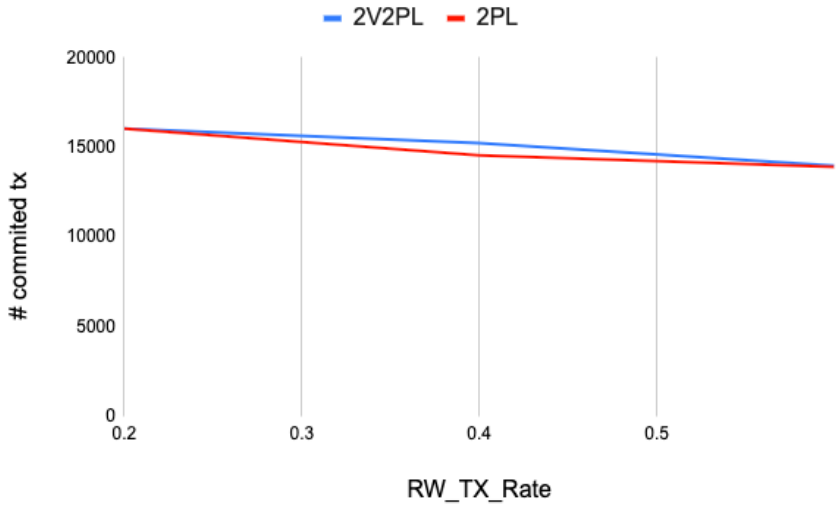


According to the results after only tuning # RTE, I use 4 RTE for all the following experiments. And I believe the results are due to my 4-core CPU such that the optimal performance is almost reached when every RTE binds to one core.

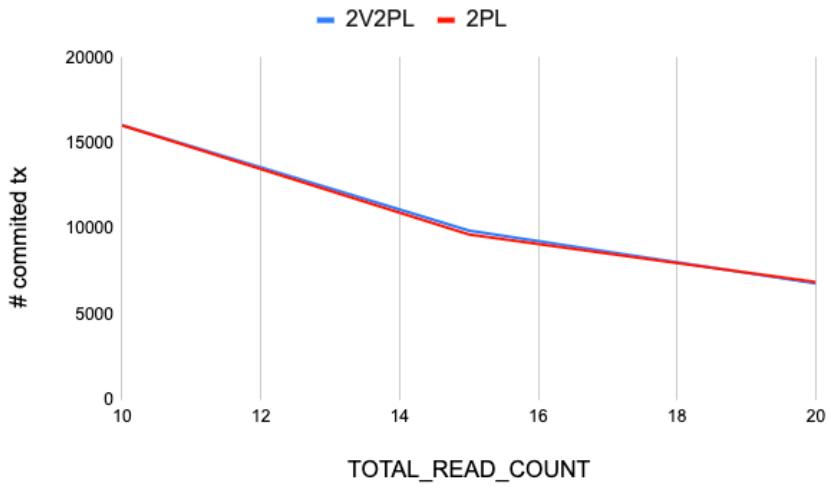
- Environment

Local Machine	Macbook Air 11-inch, Early 2015
Memory	8GB 1600MHz DDR3
CPU	Intel Core i5-5250U 1.6GHz
Disk	APPLE SSD AP0256H Media
OS	macOS Big Sur
# RTE	4

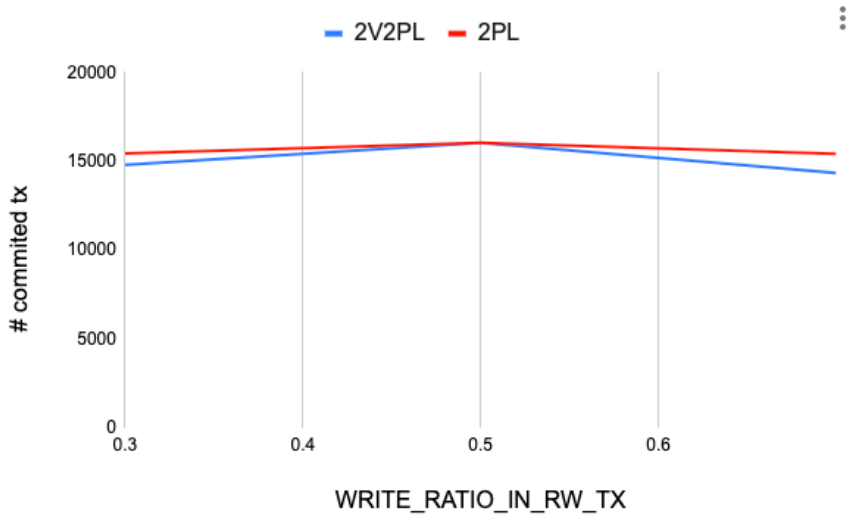
- Micro Benchmark



It is not a surprise that the performance decreases with the increasing RW_TX_Rate. Higher probability of encountering a deadlock or more time waiting for a lock to release comes after the higher probability of generating a write transaction. However, I suppose the performance of 2V2PL can exceed 2PL if RW_TX_Rate increases since the purpose of 2V2PL is to avoid writing blocks reading, which is not the case according to the experiment. I think this is because even when writing doesn't block reading, the time spent on waiting c_lock is still considerable.



The performance of 2V2PL and 2PL are still very close in this experiment. I believe this is because when doing this experiment, I set RW_TX_Rate to be 0.2, so there are not too many writing-correlating transactions in the benchmarking process, and therefore the advantage of 2V2PL is not significant. Besides, it is quite reasonable that the performance decreases with the increasing TOTAL_READ_COUNT, since every transaction must spend more time on reading data from either buffer or private work space.



In the figure we can see that both 2V2PL and 2PL reach their best performance when WRITE_RATIO_IN_RW_TX is set to 0.5, and 2PL performs better than 2V2PL everywhere. This might be caused by my 2V2PL implementation. The purpose of 2V2PL is to avoid writing blocks reading between transactions, but reading and writing are still sequential inside the same transaction. Furthermore, in my implementation, when a transaction is seeking for some data, it would search the data in the private work space first; if not found, search the buffer afterward. This would make transactions have to do extra work when seeking for some certain data.