Database Management Systems [236510]

HW2
ROMV scheduler

Submitted by:

Elad Nachmias – 301727319 – eladn@campus.technion.ac.il
Tal Elfand – 203934450 – talhe@campus.technion.ac.il

Table of contents:

The main driver:

The main driver is responsible to run and test workloads. It is located in the file “main.py”.

It has a few options for testing and logging:

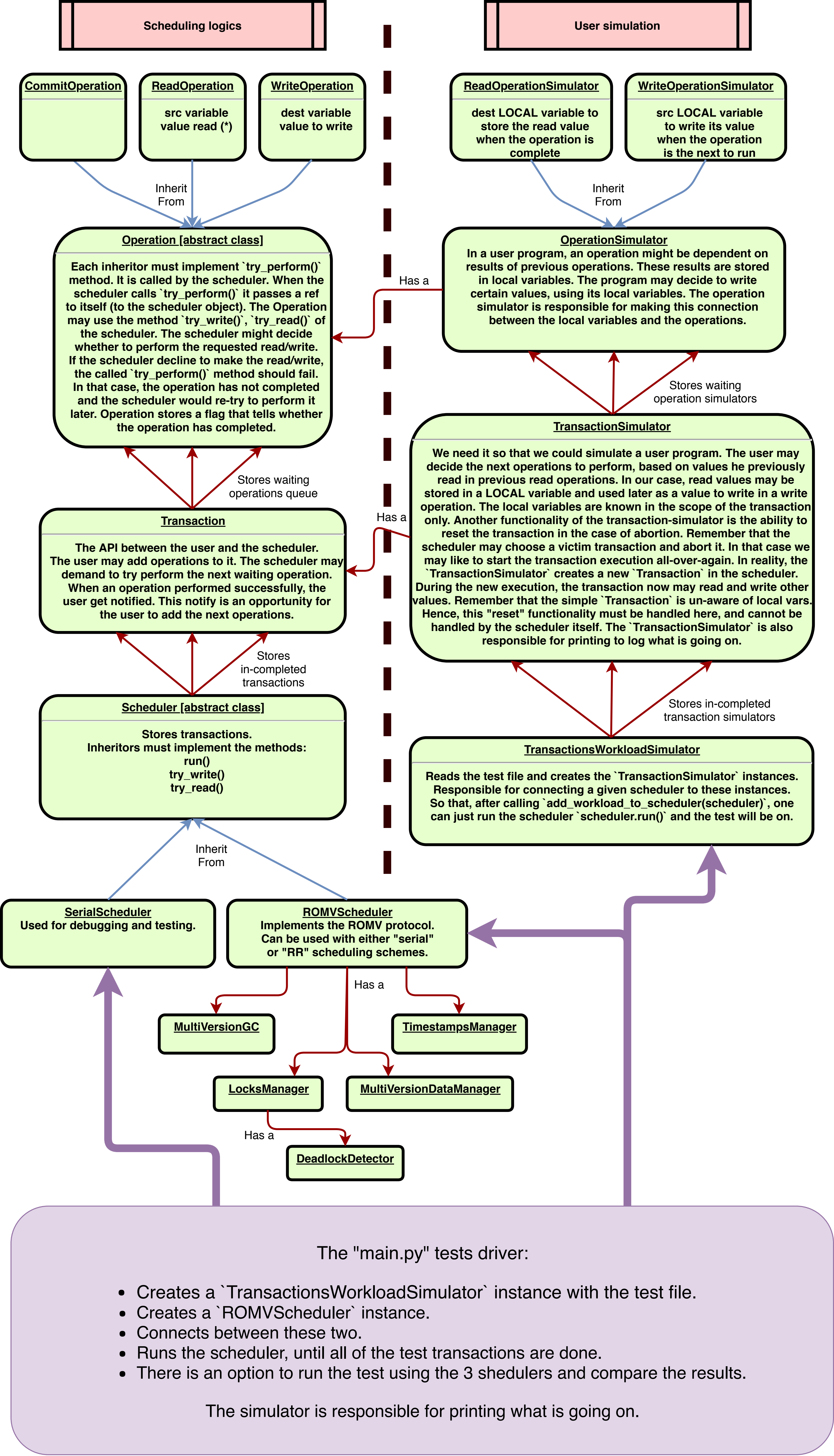
- Using certain test files. [--tests test_file1 [test_file2 [...]]]
- Forcing using a certain scheduler:
    - --sched=by-test [the default option]
    - --sched=romv-rr
    - --sched=romv-serial
    - --sched=simple-serial
    - --sched=compare-all
      Run the chosen tests using all 3 schedulers and compare their intermediate and final results. Firstly run the ROMV scheduler using RR scheduling scheme. Then run the ROMV scheduler using serial scheduling scheme and the simple serial scheduler, both using the same transaction order as the serialization order as received by the first run (by the ROMV RR scheduling type). Then verify that the values being read into local variables and values being written to db-variables by each operation in each transaction are the same for the each one of the 3 executions. Then also verify that the latest versions of variables in all executions have the same values.

Options for logging:

- --log-variables: Print the variables values after each change.
- --log-locks: Print the locks table whenever a lock is acquired or released.
- --log-wait-for: Print enumeration of the the transactions that an operation waits for.
- --log-deadlock-cycle: Print a found deadlock cycle caused resetting the transaction.
- --log-gc: Print when the GC marks a version to be evicted and when actual eviction happens.
- --log-transaction-state: Print the transaction state before each attempt to perform an operation.
- --log-oded-style: Use Oded's style for printing the log lines.
- --log-sched-prefix: Print the scheduling type in the right side of each printed run-log line.
- --log-serialization-point: Print the serialization point of each transaction.
- Some of these are set by default. Use the `--help` to see which.
- Each such option --log-* has also a --no-log-* version to explicitly turn off this kind of logging.

Please run `main.py --help` to see a full description of all of the available options.

For running with the logging settings that we were asked to conform to use:
`python3 main.py --log-oded-style --log-none`
However, the default printing settings are quite informative.


Scheduling logics
User simulation
CommitOperation
ReadOperation
src variable value read (*)
WriteOperation
dest variable value to write
Inherit From
Operation [abstract class]
Stores waiting operations queue
Transaction
Stores in-completed transactions
Scheduler [abstract class]
Inherit From
SerialScheduler
Used for debugging and testing.
ROMVScheduler
Implements the ROMV protocol.
Can be used with either "serial" or "RR" scheduling schemes.
Has a
MultiVersionGC
TimestampsManager
LocksManager
MultiVersionDataManager
Has a
DeadlockDetector
ReadOperationSimulator
dest LOCAL variable to store the read value when the operation is complete
WriteOperationSimulator
src LOCAL variable to write its value when the operation is the next to run
Inherit From
OperationSimulator
Has a
Stores waiting operation simulators
TransactionSimulator
Has a
Stores in-completed transaction simulators
TransactionsWorkloadSimulator

Operation [abstract class]

Each inheritor must implement `try_perform()` method. It is called by the scheduler. When the scheduler calls `try_perform()` it passes a ref to itself (to the scheduler object). The Operation may use the method `try_write()`, `try_read()` of the scheduler. The scheduler might decide whether to perform the requested read/write. If the scheduler decline to make the read/write, the called `try_perform()` method should fail. In that case, the operation has not completed and the scheduler would re-try to perform it later. Operation stores a flag that tells whether the operation has completed.

Transaction

The API between the user and the scheduler. The user may add operations to it. The scheduler may demand to try perform the next waiting operation. When an operation performed successfully, the user get notified. This notify is an opportunity for the user to add the next operations.

Scheduler [abstract class]

Stores transactions.
Inheritors must implement the methods:
run()
try_write()
try_read()

OperationSimulator

In a user program, an operation might be dependent on results of previous operations. These results are stored in local variables. The program may decide to write certain values, using its local variables. The operation simulator is responsible for making this connection between the local variables and the operations.

TransactionSimulator

We need it so that we could simulate a user program. The user may decide the next operations to perform, based on values he previously read in previous read operations. In our case, read values may be stored in a LOCAL variable and used later as a value to write in a write operation. The local variables are known in the scope of the transaction only. Another functionality of the transaction-simulator is the ability to reset the transaction in the case of abortion. Remember that the scheduler may choose a victim transaction and abort it. In that case we may like to start the transaction execution all-over-again. In reality, the `TransactionSimulator` creates a new `Transaction` in the scheduler. During the new execution, the transaction now may read and write other values. Remember that the simple `Transaction` is un-aware of local vars. Hence, this "reset" functionality must be handled here, and cannot be handled by the scheduler itself. The `TransactionSimulator` is also responsible for printing to log what is going on.

TransactionsWorkloadSimulator

Reads the test file and creates the `TransactionSimulator` instances. Responsible for connecting a given scheduler to these instances. So that, after calling `add_workload_to_scheduler(scheduler)`, one can just run the scheduler `scheduler.run()` and the test will be on.

The "main.py" tests driver:

- Creates a `TransactionsWorkloadSimulator` instance with the test file.
- Creates a `ROMVScheduler` instance.
- Connects between these two.
- Runs the scheduler, until all of the test transactions are done.
- There is an option to run the test using the 3 shedulers and compare the results.

The simulator is responsible for printing what is going on.

Main scheduling classes:

Note that some of the following explanations are quite technical. We just want to allow others to understand our code in the future it needed. Note also that the code is well documented and self-explained. In fact, the following explanations are copied from the source-code.

SchedulerInterface

This is a pure-abstract class. It is inherited later by the `ROMVScheduler` and the `SerialScheduler`. The `SchedulerInterface` include basic generic methods for storing the transactions in lists and iterating over them. The `SchedulerInterface` is defined an interface, so that any actual scheduler that inherits from `SchedulerInterface` must conform with that interface. Any scheduler must implement the following methods:

- run(): It iterates over that transactions and tries to perform their first awaiting operation. The method `iterate_over_ongoing_transactions_and_safely_remove_marked_to_remove_transactions()` of the scheduler might be handy for the implementation of this method.
- try_write(transaction_id, variable, value): When the scheduler tries to perform a "write" operation (of a transaction), the operation receives a pointer to the scheduler. The method `operation.try_perform(scheduler)` might call the method `scheduler.try_write(..)`. This method might succeed of fail. If it fails it means it couldn't acquire the necessary locks for that transactions, and the operation should wait until these desired locks will be released. That mechanism allows the scheduler to enforce its own locking mechanism.
- try_read(transaction_id, variable): Similar to the explanation above. Except that here the scheduler might choose to handle differently read-only transactions.

Transaction

Transaction is the main API between the scheduler and the user. It allows the user to add operations to it, using `add_operation(..)` method.

The scheduler calls the method `try_perform_next_operation(..)` when it decides to. When the scheduler calls this method, the transaction tells the next operation to try perform itself, using the method `next_operation.try_perform(..)`. If the next operation successfully performed itself, the transaction would remove this operation from the `_waiting_operations_queue`.

After each time the scheduler tries to execute the next operation (using the above-mentioned method), a users' callback is called. If the operation has been successfully completed, the callback `on_operation_complete_callback(..)` is called. Otherwise, the callback `on_operation_failed_callback(..)` is called.

When `try_perform_next_operation(..)` is called (by the scheduler) but the queue `_waiting_operations_queue` is empty, the scheduler calls to the user callback `_ask_user_for_next_operation_callback(..)`. It gives the user an opportunity to add the next operation for that transaction. However, the user does not have to do so.

The user can also take advantage of the callback `on_operation_complete_callback(..)` in order to add the next operation to be performed.

All of these mentioned users' callbacks are set on the transaction creation.

Operation

This is a pure abstract class for an operation (one cannot make an instance of `Operation`). It is inherited later on by `WriteOperation`, `ReadOperation`, and `CommitOperation`. As can be seen later, each Transaction may contain operations.

Transaction-ID can be assigned once in a life of an operation object, and it should be done by the containing transaction (when adding the operation to the transaction).

Operation is born un-completed. It may become completed after calling `try_perform(..)`. The method `try_perform(..)` is implemented by the inheritor classes.

SerialScheduler

Simple serial scheduler. We implemented such one in order to perform tests on the `ROMVScheduler`. Note that the `ROMVScheduler` has also a serial scheduling scheme option. By passing the flag `--sched=compare-all` to `main.py`, we compare the results of our 3 schedulers.

ROMVScheduler

The `ROMVScheduler` inherits from the basic `SchedulerInterface` and implements the ROMV protocol. It mainly implements the methods: `run(..)`, `try_read(..)`, and `try_write(..)`.

We use the `LocksManager` for managing locks over the variables (which uses the `DeadlockDetector`). When the `ROMVScheduler` encounters a deadlock, it chooses a victim transaction and aborts it. For simplicity, the chosen victim is the first that closed the deadlock-cycle.

The updates made during the update-transaction are stored inside of a local mapping inside of the update transaction. Find additional details about it under `UMVTransaction` class in `romv_transaction.py`.

We use the `MultiVersionDataManager` for managing the versions and accessing the disk.

We use the `MultiVersionGC` for evicting unnecessary old versions.

We use the `TimestampsManager` for assigning timestamps for transactions.

UMVTransaction

The updates made during the update-transaction are stored inside of a local mapping. It can be stored in the RAM or on the disk or both (caching) - we did not explicitly referred that (in purpose).

The reason that it is ok is because no other transaction in the system can read these updated version until this update-transaction commits. On commit, these updates are written as new versions in the disk.

If a transaction updated a variable more than single time, only the final version would be stored on the disk. Again, this is ok because the protocol promises that no other transaction can access these intermediate updates. When this update transaction performs a read, we first check whether this variable has been written before by this transaction. If so, we load and return the value in this the local mapping. Otherwise we read from the disk.

Practically this can be done in real-life case by maintaining a "bloom-filter" in the memory that indicates which variables that transaction may have written to avoid this 2 disk accesses. For each variable that the transaction updates, we store the previous version of that variable. This data is needed for the GC mechanism. In order to know which versions to evict, in a case where there is no reader that is "responsible" for the previous version.

MultiVersionDataManager

The class is responsible for managing the versions of values of the variables in the system. We make sure to explicitly denote when we access the disk.

DeadlockDetector

This class is responsible for deadlock detecting using a "wait-for" dependency graph. For simplicity, we used the `networkx` python library to maintain the graph and to search for cycles in it. In real-life case we might find a more-efficient solution, maybe by using properties of our graph (for example: out degree = 1).

<u>**The user simulator**</u>:

These classes are responsible for parsing the workload test files and simulate the execution of the transactions (and their operations) in any given scheduler that conforms to the `SchedulerInterface`.

<u>TransactionsWorkloadSimulator</u>

Parse the input test file and add transactions and their operations to the given scheduler.

<u>TransactionSimulator</u>

Simulate the execution of a transaction. Stores all of the local variables that can be accessed by the operation-simulators.

After each operation completes, the `on_complete_callback` will be called (by the scheduler), and the next operation to perform would be added by the `TransactionSimulator`.

When the scheduler encounters a deadlock, it chooses a victim transaction and aborts it. In that case, we should actually "reset" a transaction. Means, if it is aborted we should try to execute it all over again.

Prima facie, we could think to add a transaction "reset" feature to the `SchedulerInterface` module itself. In reality it makes no sense to do so, because the read operations might read other values in two different executions. Hence, local-variables (which are not known by the scheduler but only known to the user), might get other realizations during the two executions. Hence, the operations that the user create and add to the transaction might be different in the two executions.

In conclusion, we understand that the "reset" feature must be supported by the `TransactionSimulator`, that simulates the user. It means that the `TransactionSimulator` would have to store all of the operation-simulators, so that they could be used again in a case of aborting a transaction.

<u>OperationSimulator</u>

Operation simulator is responsible for storing an operation to perform. A simple `Operation` is not familiar with the concept of "local variables". Sometimes the next operation to perform might read a value from a local variable, or write a value to it. The inheritors `ReadOperationSimulator` and `WriteOperationSimulator` handle accessing these local variables. All of the local variable are stored by the `TransactionSimulator` as can be seen later. The `TransactionSimulator` may contain instances of kind `OperationSimulator`.

The garbage-collecting mechanism

The class that responsible for maintaining the GC mechanism is `MultiVersionGC`.

The basic intuition is to evict the versions that no one would potentially need in the future. An update transaction may need only the latest version of any variable. But a read-only transaction may need the latest version since the transaction has born. We have to somehow keep track (efficiently) of old versions that may be needed in the future, so we could conservatively detect versions that no-one may need and evict it. Here we explain how the `MultiVersionGC` does it.

Each ongoing read-only transaction holds a set of old versions of variables. For the rest of this explanation we are going to name this set by "the responsibility set" of that read-only transaction. This naming would make sense soon.

Note that a version becomes potential for removal only when a new version exists for the same variable. Hence, we are going to track only such versions. We will not track latest versions. When an update-transaction commits a new version, it checks whether a previous version exists for that variable. This check is efficient because the `MultiVersionDataManager` stores a cache (in RAM) with the latest timestamp of each variable in the system. So no disk operation is needed here. We will see later on that no disk operation is needed at all for the versions tracking and "responsibility-passing" mechanism.

If there exists a previous version for that variable, from this moment and on, someone has to be responsible for this previous version. So, the just-committed-updater had now just become responsible for that old version, but this transaction has just committed and going to be removed from the system, so it is going to pass the responsibility into someone else's hands.

So, the just-committed-updater searches for the youngest read-only transaction that has been born after the timestamp of the previous transaction. If such RO transaction exists, any transaction that may read from this previous version must be that RO transaction, or an older one. That is because we took the ***younger*** transaction of the ones that can read this version. In that case, the just-committed-updater assigns this reader to be responsible for the previous version. We will see below how that reader would handle the eviction of that version.

If no such reader exists, it means that currently there is no ongoing reader in the system with timestamp between the timestamp of the previous version and the timestamp of the new version. RO transactions that are older from the previous version are not able to read from it and RO transactions that are younger from the just-committed-version will read its value. So, no reader would need the previous version.

In that case, the just-committed-updater would mark this version for eviction (we will later see how it is done and when the actual eviction happens). When a read-only transaction commits, it has to handle the versions under its responsibility. To do so, it goes over the versions that has been assigned under its "responsibility set" one-by-one and checks whether the youngest older read-only transaction may need it. If so, it passes the responsibility for that version to that youngest older reader. If not, it marks this version for eviction.

The GC is informed each time a transaction is committed. When the GC informed about a committed transaction, it checks about versions that can be evicted, as described above. When the GC encounters a version that has to be evicted, it does not actually perform the eviction. The eviction details are stored as a GC "job", in a dedicated gc-jobs- queue. Then, when the scheduler decides, it can offline call to `run_waiting_gc_jobs()` to perform the waiting previously registered evictions. The actual eviction application performs disk accesses. The idea is to avoid blocking the scheduler in favor of accessing the disk to perform GC evictions.

Notice that all of the described operations that take place whenever a transaction commits are efficient because of the data structures we maintain in the system: (1) list of RO transactions sorted by timestamps in `ROMVScheduler`, and (2) the timestamps of latest versions per variable in `MultiVersionDataManager` and in `UMVTransaction`.