Principles of Computer System Design (PCSD), Block 2, 2012 / 2013

Assignment 2

This assignment is due via Absalon on December 13, 23:55pm. This assignment is going to be evaluated pass-fail, as announced in the course description. This assignment should be solved individually. Cooperation on or discussion of the contents of the assignment with other students is strictly forbidden. The solution you provide should reflect your knowledge of the material alone.

A well-formed solution to this assignment should include a PDF file with answers to all exercises as well as questions posed in the programming part of the assignment. In addition, you must submit your code along with your written solution. Evaluation of the assignment will take both into consideration.

Do not get hung up in a single question. It is best to make an effort on every question and write down all your solutions than to get a perfect solution for only one or two of the questions.

Note that all homework assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Exercises

Question 1 (Serializability & Locking). Consider the following two transaction schedules with time increasing from left to right. *C* indicates commit.

Schedule 1

```
T1: R(X)                                  W(Y)  C
T2:       W(Z)  W(X)  C
T3:                       R(Z)  R(Y)  C
```

Schedule 2

```
T1: R(X)                  W(Y)  C
T2:           R(Z)                W(X)  W(Y)  C
T3:       W(Z)       C
```

Now answer the following questions:

(a) Draw the precedence graph for each schedule. Are the schedules conflict-serializable? Explain why or why not.
(b) Could the schedules have been generated by a scheduler using strict two-phase locking (strict 2PL)? If so, show it by injecting read/write lock operations in accordance to strict 2PL rules. If not, explain why.

Question 2 (Optimistic Concurrency Control). Consider the following scenarios, which illustrate the execution of three transactions under the Kung-Robinson optimistic concurrency control model. In each

scenario, transactions T1 and T2 have successfully committed, and the concurrency control mechanism needs to determine a decision for transaction T3. The read and write sets (RS and WS, respectively) for each transaction list the identifiers of the objects accessed by the transaction.

Scenario 1
T1: RS(T1) = {1, 2, 3}, WS(T1) = {3}, T1 completes before T3 starts.
T2: RS(T2) = {2, 3, 4}, WS(T2) = {4, 5}, T2 completes before T3 begins with its write phase.
T3: RS(T3) = {3, 4, 6}, WS(T3) = {3}, allow commit or roll back?

Scenario 2
T1: RS(T1) = {1, 2, 3}, WS(T1) = {3}, T1 completes before T3 begins with its write phase.
T2: RS(T2) = {5, 6, 7}, WS(T2) = {8}, T2 completes read phase before T3 does.
T3: RS(T3) = {3, 4, 5, 6, 7}, WS(T3) = {3}, allow commit or roll back?

Scenario 3
T1: RS(T1) = {2, 3, 4, 5}, WS(T1) = {4}, T1 completes before T3 begins with its write phase.
T2: RS(T2) = {6, 7, 8}, WS(T2) = {6}, T2 completes before T3 begins with its write phase.
T3: RS(T3) = {2, 3, 5, 7, 8}, WS(T3) = {7, 8}, allow commit or roll back?

For each scenario, predict whether T3 is allowed to commit. If T3 is allowed to commit, state which validation tests were necessary to reach that conclusion. If T3 must be rolled back, state all tests which T3 fails, along with the offending objects from the read and write sets of T1 and T2.

Question 3 (Recovery Concepts). Regarding recovery techniques, answer the following questions:

(a) In a system implementing force and no-steal, is it necessary to implement a scheme for redo? What about a scheme for undo? Explain why.
(b) What is the difference between nonvolatile and stable storage? What types of failures are survived by each type of storage?
(c) In a system that implements Write-Ahead Logging, which are the two situations in which the log tail must be forced to stable storage? Explain why log forces are necessary in these situations and argue why they are sufficient for durability.

Question 4 (ARIES). Consider the recovery scenario described in the following, in which we use the ARIES recovery algorithm. At the beginning of time, there are no transactions active in the system and no dirty pages. A checkpoint is taken. After that, three transactions, T1, T2, and T3, enter the system and perform various operations. The detailed log follows:

LOG

LSN	LAST_LSN	TRAN_ID	TYPE	PAGE_ID
1	-	-	begin CKPT	-
2	-	-	end CKPT	-
3	NULL	T1	update	P2
4	3	T1	update	P1
5	NULL	T2	update	P5
6	NULL	T3	update	P3
7	6	T3	commit	-
8	5	T2	update	P5

```
9     8          T2         update     P3
10    6          T3         end        -
```

At this point in the execution, i.e., after LSN 10, the system crashes.

Now answer the following questions:

(a) Apply the ARIES recovery algorithm to the scenario above. Show:
   1. the state of the transaction and dirty page tables after the analysis phase;
   2. the sets of winner and loser transactions;
   3. the values for the LSNs where the redo phase *starts* and where the undo phase *ends;*
   4. the set of log records that may cause pages to be rewritten during the redo phase;
   5. the set of log records undone during the undo phase;
   6. the contents of the log after the recovery procedure completes.

Programming Task

In this programming task, you will implement log-based durability in `KeyValueBase`. As we know, `KeyValueBase`'s interface includes a set of atomic update operations (`insert`, `update`, `delete`, and `bulkPut`). Log-based durability is achieved by performing *logical logging* of these operations. In other words, the log records every call to these operations, along with the parameters given to the call. The goal of this logging mechanism is to allow the service to recover from fail-stop failures.

In order to bound recovery time, periodically the recovery subsystem initiates an *operation-consistent checkpoint* (in a classic ACID database, the technique would be called a *transaction-consistent checkpoint*). In contrast to a fuzzy checkpoint, an operation-consistent checkpoint quiesces the system, waiting for all in-flight atomic operations to complete. The checkpoint process then forces all unwritten state to disk. During this period, any new operations are simply blocked and the system is temporarily unavailable. After all unwritten data is forced, normal processing of operations resumes. Note that the log can be truncated up to the LSN of the last consistent checkpoint.

So as to reduce dependencies with respect to your solution of Assignment 1, you will be allowed to hand in a solution to this programming task that makes use of a single-threaded, in-memory implementation of `KeyValueBase`. You may choose to produce this trivial implementation of the interface and extend it with log-based durability, or alternatively, implement log-based durability against your group's implementation of Assignment 1. You should implement this programming task either in Java or C#. Make sure to use the latest version of Oracle's Java VM or of Microsoft .NET for this programming task.

Interface

We extend `KeyValueBase` with two components: the `Logger` and the `Checkpointer`. The `Logger` interface allows for overlapping log I/O with normal processing of operations:

1) `logRequest(LogRecord record)` → `FutureLogAck`: This function accepts log requests and returns a *future* to its caller. The future is only populated after the corresponding log request has been written to stable storage. The calling code is responsible for guaranteeing atomicity by blocking on the future after executing the operation's logic.

Clearly, the `Logger` must execute actual log I/O in a separate thread. So the `Logger` is a `Runnable`; however, you must always respect write-ahead logging in your implementation.

The `Checkpointer` is also a `Runnable`; it executes in a separate thread and periodically requests `KeyValueBase` to quiesce. To achieve the latter, `KeyValueBase`'s implementation must be extended to offer two special functions:

1) `quiesce()`: This function quiesces the system, i.e., the function initiates blocking of new operations and then itself blocks until all in-flight operations are concluded. Once the function returns to its caller, the system state is guaranteed to not be accessed by operations until a resume operation is called.

2) `resume()`: This function allows new operations to proceed and update system state.

The two operations above should *not* be exposed to clients of `KeyValueBase` via RPC. They are internal functions that should be employed exclusively by the `Checkpointer`.

Implementation

Provide an implementation for both the `Logger` and the `Checkpointer` components. In order to get you started, we provide interfaces and skeleton classes corresponding to these components.

In addition to implementing these two components, modify your implementation of `KeyValueBase` so that the system will automatically recover its state when the constructor for your implementation is invoked after after a system restart. The recovery procedure should only be executed after the first time the system is initialized. In other words, the first time the system is instantiated, data is normally loaded via the `init` method. Log-based recovery happens in the constructor after this first time, however. In the latter case, calling the `init` method becomes unnecessary and should be explicitly disallowed by raising an appropriate exception.

Question 1:. *Briefly describe your implementation of the* `Logger` *and the* `Checkpointer` *in* `KeyValueBase`. *In particular, explain how you achieved overlapping of log I/O and how you implemented the necessary synchronization for quiescence and checkpointing. (2 paragraphs)*

Question 2: *Describe how you tested your implementation of both components and ensured that durability was actually achieved in face of fail-stop failures. (2 paragraphs)*

Experiments

Once you have implemented log-based durability, you should experimentally quantify the overhead of this new feature in `KeyValueBase`. Again, the two main performance aspects we will focus on are throughput and latency.

Question 4: *Explain how you can quantify the overhead of log-based durability in both throughput and latency. Design and describe your experimental setup to quantify this overhead. Recall that you must document all parameters that may influence the performance of the system (e.g., e.g., number of clients, hardware characteristics, size of dataset managed, mix of operations).*

Question 5: *Carry out the experiment you described in Question 4 and report your overhead results. Explain the effects you observe regarding throughput and latency of your service.*

Extensions

Group commit is a technique commonly applied to logging subsystems. The core idea of group commit is to accumulate log records for a constant number K of transactions in main memory, and force all of these log records together to disk. Given that forced log I/Os contain more log records, group commit increases logging throughput. However, as K operations must be accumulated (or alternatively a timeout reached), group commit may increase latency.

Extend your implementation of log-based durability in `KeyValueBase` to include group commit of atomic update operations.

Question 6*: Briefly describe how you implemented group commit in* `KeyValueBase`*. (1 paragraph)*