Project Two Report

Group 12:
Parker Beckett, James Rider, Matt Myers

Design Document

Our implementation of logging and recovery will both modify existing files and introduce new files to the project. We plan to create four main files: three manager objects that will help to ensure logging and recovery works correctly and a log object that will be used by these managers. The log object will be simple and consist of an operation code, the key being changed, the old value, the new value, and the timestamp of the operation. This information will be written to the log file without much alteration, but having an object record the information will assist us in separating the different manager classes. Depending on the efficiency of our program we may include explicit memory addresses in these objects in order to speed up our memory access. Our first manager class is a log manager object that handles I/O writes to the disk as well as keeping a count of the amount of operations processed before the checkpoint manager is invoked. The checkpoint manager will contain methods to write the state of the tree to the disk whenever this threshold is crossed. At the moment we have not decided on the most appropriate threshold beyond the granularity parameters suggested in the problem statement, but we plan to experiment with time, number of operations, and other parameters in order to find an ideal metric. The final manager class we will need is a recovery manager class, whose main role will be to reconstruct the tree from a combination of the latest checkpoint and the logs present in stable memory as well as validate it. This class will serve as the main protection against crashes, although we will need to experiment more with the crash tests in order to more fully understand when it should be invoked.

These programs will be communicating to one another frequently. The b-tree will construct a log object and calling the log manager to flush to the log file before any changes are made to the tree, the checkpoint manager will call methods from the b-tree in order to write the tree data to the disk as well as call the purge methods of the log manager when checkpoints are made, the log manager will call the checkpoint manager when needed by whatever metric has been decided upon, and the recovery manager will access the log files and checkpoint files as well as call methods from the b-tree class in order to reconstruct the tree. There are likely more interactions between different programs, but these are the most important ones.

The b-tree will need a few additional methods in order to facilitate logging, the most significant of which is the aforementioned log write before any changes are made. This will be a call from within the b-tree that references the global log manager and passes the relevant arguments. We will not be implementing a .reconstruct(<log>, <checkpoint>) in the b-tree class but will rather implement this functionality in the recovery manager. A .getstate() method could prove useful to streamline checkpointing, but we will experiment with whether or not we may simply implement this in the checkpoint manager. We have also considered a .verify(tree) method in order to quickly check whether or not two trees are equivalent. We anticipate changing many arguments throughout the program files in order to properly pass references to the manager classes. Beyond that we have not encountered many instances in which our arguments should change significantly, but we will be receptive to such a change if necessary.

While we have been provided with test cases, we will write our own if necessary in order to confirm the correctness of our methods. In particular, our logging should be tested by performing many operations on our tree, pausing operation, reconstructing a new tree from the log files, and comparing our trees. This will test our reconstruction method in our recovery

manager as well, although we will need additional tests in order to verify our recovery methods. At the moment our group is not aware of a good way to simulate a crash beyond an amateur division by zero or null reference, and we are unsure of how such an explicit error would be integrated into a test. However, once we come to a solution, we intend to crash our program at every important step and test how well our recovery works. We intend to try to crash when the log is written but the tree is not changed, before, after, and during checkpoints, during recovery, and many other critical spots in our process.

β^{ε}-Tree Summary

A β^{ε}-tree data structure is a write-optimized improvement upon the basic structure of the B+-tree. The B+-tree is itself an improvement upon the B-tree, storing data only at its leaf nodes in order to free space in the interior nodes and therefore reduce the depth of the tree. This structure has its advantages over the standard B-tree and can be considered a read-optimized data structure but itself struggles with a write heavy workload. Storing data in only the leaf nodes means that every operation ends up reaching these nodes. Whether a query for the information or the insertion of new data, the leaf nodes that most often occupy disk space rather than memory will be loaded into memory during each transaction.

That is undesirable, so the β^{ε}-tree attempts to solve this by instead partitioning the pivot nodes of the B+-tree into nodes that hold both pivot information and messages. The pivots allow for traversal and construction of the tree as expected, yet the message nodes can now hold records of queries as they happen. The nodes split when their pivot capacity is exceeded, and when the message buffer is exceeded the messages are flushed out the node to whichever of its children is the busiest. This amortization combined with keeping messages close to the root is why the β^{ε}-tree is so powerful when dealing with write-heavy instructions. However, writing to a β^{ε}-tree still suffers overhead from overwriting the same data multiple times as nodes receive new messages, so it is not as efficient as it might seem.

A smaller β means that there will be fewer pivots and more messages per node while a smaller ε means that there will be fewer messages and more pivots per node. More pivots means a shallower tree while less will result in a deeper tree; while more room in the nodes for messages allows for more operations on a node before it flushes its data to its child. The design choices around these parameters are at the heart of modern research on β^{ε}-trees and the real-time manipulation or correction of these parameters is an intriguing and important problem.

Implementation Report

The first major step in our implementation was including logging functionality. Our logging implementation delays any sort of tree change until the log granularity is exceeded; every instruction would instead simply be added to the backing vector of the logger class. Once the size of this backing array exceeded the allowed granularity, the logger iterated over each message and flushed each to the log file on disk. Once this was complete, the tree in memory was allowed to be changed, so the map and node operations present in the original upsert method were executed here with each element in the logger's backing array. Finally, as the log

file had been persisted and the in-memory tree was valid, the backing array of the logger was emptied in order to prevent duplicates.

Checkpointing was approached in a similar way, in which we waited until a certain threshold was crossed before performing a series of actions. Our checkpointer first flushes the rest of the log, then pushes the remaining nodes onto the disk. At this point we know the tree on disk is valid and up to date, so we can overwrite the log file with a single line that records the root node's id and version. Once this is recorded, we can attempt to destroy the old versions of the nodes still on the disk so we have only our valid tree on disk. We ensure that we do this atomically by verifying that all old versions were destroyed before noting in the log that a crash did not occur between when the deletions started and when they ended. The checkpointer also contains a method to reconstruct the tree from its root node by using the current ids and versions on the disk.

It should be noted that in cases where a crash occurs before deletion completes we note the lack of COMPLETE in the checkpoint log line and finish the checkpoint before recovering. This prevents instances where we would restore from the oldest files that are not actually the correct state of the tree.

The last step in our implementation was the recovery logic. While it is its own file and class, its functionality does not expand significantly beyond the logger and checkpointer. It simply combines the two in order to construct the tree from the on-disk information, using the methods of the checkpoint manager, and then uses the logger's functionality in order to walk through the existing log file and apply those changes to the tree. Once this is complete the recovery manager has done its job and the tree on-disk and in memory is valid; at this point we considered checkpointing but decided that in this instance the program is only experiencing a single crash and elected not to. In a real world application additional consideration would need to be given to decide if repeated crashes are expected in which case it would make sense to immediately checkpoint after recovery.

We disabled the automatic deletion of backing_store files and only handled deletion of them during the checkpointing process as well as the destructor of the backing_store. When checkpointing we verify that we only keep files that are needed, i.e. ones whose id is in the swap space. We don't have a way of knowing if there are unneeded files during recovery so they will unfortunately stick around until cleanup is done at the very end. In a real world application this would cause problems where the useless files accumulate with every crash. In such an instance additional consideration would need to be given such as walking the entire tree to ensure that ONLY needed files are kept during recovery. We are aware of this and decided it was beyond the scope of this assignment and went with the naive approach.

Our implementation guarantees that

Parameter Analysis

Both granularities determine how often the disk is accessed, so less granularity for either will result in a system with lower performance but higher reliability. The logging operation had less overhead than the checkpointing operation, which is still less expensive than the recovery process. The majority of this overhead comes from file I/O and random access on the disk, with the logger accessing the log file every log_granularity operations and the checkpoint accessing

the log file every checkpoint_granularity. The checkpoint also flushes each node of the tree in memory to disk every checkpoint_granularity, so they both have a significant effect on the runtime of each process.

The guarantees provided by our index are affected by these parameters. Because we only update the tree every log_granularity operations queries that come in between these times will return stale data, as such we only guarantee that the queries will be valid within log_granularity operations. A lower log_granularity would result in less stale queries but it would also slow down the runtime of the index as it would be writing to disk more often. Because queries are infrequent in the provided test_input file we did consider flushing the log every time a query came in. This would result in being able to guarantee valid queries every time at the cost of the queries taking longer each time. Additionally our index guarantees that in case of a crash at most log_granularity operations would be lost. Again, lower log_granularity means less lost information on crashes but slower operation as we write more often.

Group Contributions

James Rider: Checkpointing, Recovery, Troubleshooting, Integration
Parker Beckett: Design Document, Logging, Report
Matt Myers: Testing, Logging, Troubleshooting
Equal Contribution: Brainstorming, Code Discussion, Strategy, Crying