10/06/2023

Project #2: Logging and Recovery

Pranjal Patil, Semil Jain, Manisha Dodeja

1. Introduction

Bε-tree and how it differs from the standard B+-tree

A Bepsilon tree is a tree structure optimized for scenarios where write-heavy operations are predominant. The distinguishing feature of a Bepsilon tree is the presence of an "epsilon" value, which divides each node into two distinct parts. The first part of the Bepsilon tree, known as the Pivot Values section, operates similarly to a node in a B+ tree. It stores key-value pairs and facilitates efficient searching and retrieval operations. The second part, referred to as the Messages section, serves a unique purpose. It functions as a repository for all incoming messages, where each message corresponds to an operation performed on the tree. A message is typically composed of two components: the operation code (e.g., insert, delete, update) and the key associated with the operation. Additionally, the Messages section maintains a map that stores the key-value pairs of these operations.

Here's how the Bepsilon tree operates:

1. When an operation (e.g., insert) is initiated, it begins at the root node of the tree.

2. If the Messages section of the root node has available space, the operation is stored as a message within that node.

3. If the Messages section of the root node is full, the root node initiates a process to flush its messages down the tree. This flushing operation is carried out based on a specific algorithm and aims to create space for the new message.

4. During the process of flushing messages down the tree, certain scenarios may arise, such as node splits or merges. In response to these scenarios, pivot values are created within the nodes to manage these changes effectively.

5. For write-intensive operations, the Bepsilon tree's design proves to be optimal, as it efficiently manages incoming messages and ensures data consistency. However, for query operations, the Bepsilon tree may not be as efficient, as it necessitates traversing through the messages stored in each node along the path of the tree.

2. Implementation

We've introduced Logical Logging into our system, which involves several design modifications and the addition of new classes while making adjustments to existing ones. Logical Logging, in essence, entails a more streamlined approach to logging. Instead of storing extensive details in the logs, we primarily focus on recording the operations themselves and whether they were successful or not. The complex logic and processing are deferred to the recovery phase.
The reason behind adopting Logical Logging lies in our optimistic outlook regarding system crashes. We anticipate that system failures will be infrequent, allowing us to allocate additional time during the recovery process. By simplifying the logging phase and offloading complexity

to recovery, we aim to strike a balance between efficient real-time operations and robust system recovery.

a. New Class Additions

1. **Class LogEntry** (logging.cpp)

   Class LogEntry represents the attributes of each log entry that is going to be residing in the log file. It has a constructor that initializes its class variables with values provided while creating a log during transaction.

   Variables:
   - uint64_t type; //OpCodes- INSERT(0), DELETE(1), UPDATE(2), COMMIT (3), CHECKPOINT(4)
   - uint64_t tid; // transaction ID
   - uint64_t key; // Key
   - string val; // Value

   Example Log entries:

   1 0 1 78398 78398:
   2 3 1 0

   Breakdown of Log -

1 - LSN	2 - LSN
0 - type (Insert)	3 - type (Commit)
1 - Transaction ID	1 - Transaction ID
78398 - Key	0 - Key
78398: - Value	<Empty> - Value

2. **Class LogManager** (logging.cpp)

   The primary role of the LogManager is to manage and retain logs in memory. However, it also plays an indirect role in persisting logs onto disk. This disk persistence is achieved through collaboration with another essential component known as the Write Ahead Logging (WAL) system.

   Variables:

   - vector<LogEntry> logs (In Memory WAL)

   Methods:

   LogEntry create_log(uint64_t type, uint64_t tid, uint64_t key, string val): This function creates a LogEntry object with the given parameters (type, transaction ID, key, and value) and returns it. It serves the purpose of generating a new log entry.

   uint64_t getlsn(Wal* wal): Retrieves the Log Sequence Number (LSN) from a specified Wal object and returns it. The LSN represents the latest LSN in the log file.

bool append_log(uint64_t type, uint64_t tid = 0, uint64_t key = 0, string val = ""): Adds a new log entry to the logs vector. It creates a log entry using the create_log function and appends it to the collection. Returns true upon success.
void flushWALtoDisk(Wal* wal): Flushes the log entries stored in the logs vector to a Write-Ahead Log (WAL) represented by the provided Wal object. It iterates through the log entries, converts them to a string format, and writes them to the WAL. After flushing, it clears the logs vector and updates the master record with the latest LSN.

void flushRootToDisk(int target, int version, Wal* wal): Records the target id of the root node in the master record. This information is later used during recovery to retrieve the root node by accessing its target id from the master record.

3. **Class Wal** (buffer.cpp)

This class serves as a Write-Ahead Log (WAL) system, responsible for logging changes to disk before they are committed to the primary data storage. It ensures that modifications are reliably recorded and can be replayed for recovery if needed. The class also manages the latest Log Sequence Number (LSN) and provides methods to update and retrieve information from the master record.

Variables:

- uint64_t latest_lsn; // representing the latest Log Sequence Number (LSN)
- fstream fp; // An std::fstream object used to write log entries to a log file
- string log_file_path; / /A string containing the file path to the log file
- string master_record_path; // A string containing the file path to the master record file

Constructor:

Wal(string log_file_path, string master_record_path) // This constructor for the Wal class initializes an instance of the class with specified paths for the log file and master record file. It opens the log file for appending, reads the latest Log Sequence Number (LSN) from the master record, and assigns it to the latest_lsn member variable.

Methods:

uint64_t get_lsn(): This function returns the latest Log Sequence Number (LSN) stored in the latest_lsn member variable.

uint64_t write_log_to_disk(std::string c): Writes a log entry, represented by the input string c, to the log file. It appends the LSN and the log entry content to the log file, separated by a space, and increments the latest_lsn. It returns the new LSN.

std::string getMasterRecordValue(const std::string& key): Reads the master record file and retrieves the value associated with a specified key. It searches for the key in the master record and returns the corresponding value.

std::unordered_map<uint64_t, uint64_t> getObjectsToRecover(): The master record contains the IDs and versions of objects that persist on the disk after a checkpoint. This

method reads the master record file and retrieves a mapping of object IDs to versions into an unordered map, then returns the map.

std::vector<string> getRecoveryLogs(): Reads the log file and returns a vector of log entries that need to be replayed for recovery. It looks for log entries after a specified checkpoint LSN and collects them.

bool updateMasterRecordValue(const std::string& key, const std::string& value): Updates the value associated with a specified key in the master record file. It opens both the master record file and a temporary file, updates the value, and then replaces the old master record file with the updated one.

4. **Class recovery** (recovery.cpp)

   This class is responsible for executing recovery operations following a system failure. It utilizes the information stored in the Write-Ahead Log (WAL) and the master record to carry out these recovery tasks.

   Variables:

   - unordered_map<uint64_t, uint64_t> id_version_map: An unordered map that stores the mapping of most stable IDs to version numbers of objects that persist on the disk since the last checkpoint.
   - uint64_t root_target_id: An integer that stores the target ID of the stable root object. It is retrieved from the master record.
   - uint64_t last_checkpoint_lsn: An integer representing the LSN of the last checkpoint. It is retrieved from the master record.
   - swap_space *ss: A pointer to an object of the swap_space class.
     betree<uint64_t, string> *bet: A pointer to an object of the betree class
   - Wal *wal: A pointer to an object of the Wal class, representing the Write-Ahead Log.

   Constructor:

   recovery(swap_space *ssk, betree<uint64_t, string> *betk, Wal *walk):
   In the constructor, the recovery object is initialized with pointers to a swap_space, betree, and Wal object. It performs several important tasks during initialization:
   It retrieves and stores the root target ID from the WAL.
   It retrieves a mapping of the most stable IDs to version numbers of objects that persist on the disk since the last checkpoint from the master record through the WAL and initializes the id_version_map.
   It initializes the last_checkpoint_lsn from the master record.
   These steps ensure that the recovery object has the necessary information to perform recovery operations effectively.

   Methods:

   initialize_ssObjects(): This function initializes the swap_space objects map with keys as the most stable target IDs of objects that persist on the disk since the last checkpoint. For each entry in the mapping (key-value pair), it does the following:
   Creates a new object and sets its attributes:
   o->id is set to the key (target ID).

o->version is set to the value (version number).
o->pincount is initialized to 0.
o->refcount is initialized to 1.
o->is_leaf is set to false.
o->last_access is set to 0.
o->pagelsn is set to 0.
o->target_is_dirty is set to false.
Adds the object to the objects map with the target ID as the key.
Additionally, this function finds the maximum next_id value that should be used henceforth by finding the maximum ID among all stable target IDs.

initialize_betree(): This function initializes the betree with the root target ID retrieved from the master record.

start_operations(): This function performs recovery operations based on the log entries obtained from the Write-Ahead Log (wal). It receives log entries with LSN (Log Sequence Number) greater than the last checkpoint LSN from wal->getRecoveryLogs(). These log entries represent operations that need to be replayed to recover data. The function iterates through the log entries and calls the upsert method to replay the operations.

checkpoint_recovery(): This function performs a general checkpointing operation once after the recovery process.

b. Existing Class Modifications

In betree.hpp:

Additional Class Variables:

- uint64_t rec: This variable is introduced to determine whether the system is in the recovery phase (set to 1) or not (set to 0).
- uint64_t txnid: This variable is introduced to keep track of the transaction ID.

Additional Operations in Constructor:

rec is set to 1 when the system is in the recovery phase; otherwise, it is set to 0.
txnid is initialized to 1.
A LogManager object is initialized.
The lgmgr attribute of the ss object (of type swap_space and a class variable of betree) is initialized to the LogManager object.
The root is allocated with a new node only if rec is 0, indicating that the system is not in the recovery phase.

Additional Method Definition:

void initializeRootwithId(uint64_t root_target_id): This method is inherently called by the recovery class's inialize_betree() method to initialize the root target with the value stored in the master record. This occurs during the recovery phase, where a new node for the root should not be initialized.

Additional Method Calls/Operations During Upserts:

To implement the logging functionality, whenever an upsert operation is performed in the betree, the operation is logged by calling ss->add_log(opcode, txnid, (uint64_t)k, (string)v). The txnid is incremented by 1 for each new transaction. Note that logging doesn't happen when rec is set to 1 (during recovery).
In addition to the log, a commit log is stored for each transaction with an opcode of 3. This helps decide whether a transaction needs to be replayed during the recovery phase.
The current stable root's target ID is also continuously written to the master record by calling ss->flushRoottoDisk(root.target).

In swap_space.hpp:

Additional Class Variables:

- logcounter: This variable stores the number of logs in memory at present (log persistence).
- checkcounter: This variable is used to call checkpoint after a certain number of transactions (checkpoint persistence).
- lastlsnpushed: This variable stores the values of the last LSN that was pushed to disk.
- LogManager object lgmgr.
- Wal object wal.

Variables in Object Class:

Pagelsn: This variable stores the LSN of the most recent transaction that modified the object.

Additional Method Definition:

add_logs(): This method adds operations to our log. It also handles checkpointing when required and flushes logs to disk.

post_checkpoint(): This method is used to update the master record's checkpoint LSN to the latest value.

In swap_space.cpp:

Additional Methods:

initialize_objects(): This method takes a map of object IDs and their versions and stores it into the objects variable of the swap_space.

checkpoint(): This method is used to flush all in-memory objects to disk. Before this, the logs are also flushed to disk.

c. Implementation Explanation

1. Here we have ensured that all the operations are logged first and then implemented. The frequency with which we are inserting the logs into disk varies the accuracy (lost

transactions/operations). To ensure that there are no abnormalities when reconstructing the tree we have ensured that dirty pages are only after their respective pagelsn(as discussed above) log has been pushed to disk.

2. During checkpointing we have ensured that all obj_id and their versions are stored in the master record. It stores all the object id versions of the tree which are consistent on the disk. This helps us in constructing a tree from its last stable version (last checkpoint).
3. Next to construct the tree during recovery we need a root node. Hence during checkpointing we also store the root node of the consistent version of the tree. Using this we can set the root target of a betree object to this value and start the operations again.
4. We also store the latest lsn in the log.txt file in the master record. This is used in recovery to check if recovery is required or not. For example the checkpoint_lsn is 150 and latest_lsn is 150 in the master record. This indicates that we do not need to recover as the last log in text is the checkpoint itself.
5. We have also considered the case where we crash during recovery. For that we have called the checkpointing after recovery is done and also ensured the correctness by changing the master record values in the end once everything is done.
6. There is a rec variable introduced in betree. This variable if set to 1 will ensure that logs are not added. It is set to 1 if we are doing recovery.
7. During recovery we will be reconstructing from the latest stable version(checkpoint lsn). So we will take root and object ids and versions from master and and initialize swap space and betree accordingly. This is possible if there has been at least 1 checkpoint. Now in case that we are running a program and it crashes and there was no checkpoint done before. For this now we can't implement regular recovery as discussed before. For this case we first create a betree and call allocate of swap_space to initialize root. Then when we start recovery we have set the rec variable in betree to 1, indicating not to log these operations. In the end it changes it back to 0.

3. Testing, Tunable parameters, and Observations

In our TestScript.sh modifications, we've adapted the script to test the functionality of our logging and recovery classes, as well as to evaluate the system's performance under various conditions. Here's a summary of the changes made:

Logging and Recovery Integration: Modified the script to work with your custom logging and recovery classes. We've added code to initiate and control recovery procedures as needed.

Checkpoint and Persistence Granularity Testing: Adjusted checkpoint and persistence granularity settings. Testing with different granularities helps evaluate the system's behavior under various configurations.

Timing and Speed Evaluation: Timed the recovery process to measure how quickly it can restore the system's state. Measured the time it takes for the remaining inserts after a recovery. These timing tests provide insights into system performance.

Checkpoint Edge Cases: Varied the "kill time" in the script to simulate different checkpoint edge cases. This allows us to assess how the system behaves when checkpoints and recovery occur at different points in time, including just before a crash.

Observations

- **Frequent logging and checkpointing - Slower Execution**

  There's an additional overhead for logging and checkpointing. This overhead primarily arises from writing data to a persistent storage medium, such as a disk, which can be much slower compared to in-memory operations. Disk I/O takes a large amount of time reducing the system's performance. Therefore logging and checkpointing at smaller intervals results in an increase in execution time.

- **Frequent logging and checkpointing - Lesser probability of discrepancies**

  However, the key benefit of implementing frequent checkpointing is the substantial reduction in the risk of data loss or inconsistencies. This reduction occurs because the system diligently saves its state, including logs, at regular intervals. By doing so, the chances of losing critical log entries before they are permanently recorded on disk are significantly minimized. This in turn results in a decrease in incorrect result percentage.

- **Infrequent logging and checkpointing - Faster Execution**

  Opting for infrequent logging and checkpointing involves the practice of writing data to disk at more extended intervals, thereby resulting in minimal overhead. This approach leads to a relatively modest increase in execution time compared to systems with more frequent logging and checkpointing. By spacing out disk writes, the system can optimize its overall performance, as it spends less time on I/O operations (writing data to a persistent storage medium, such as a disk) and more time on executing the primary tasks. In essence, infrequent logging and checkpointing strike a balance between achieving data durability and mitigating the impact on execution time, making it an attractive option for scenarios where minimizing performance overhead is a priority.

- **Infrequent logging and checkpointing - greater probability of discrepancies**

  Selecting infrequent logging and checkpointing, with their longer intervals for saving data to disk, can result in a higher chance of encountering a rise in the incorrect result percentage during system recovery. When these processes happen less often, the system becomes more vulnerable to losing log records, especially if a system crash occurs before the logs get safely saved to durable storage. This increased vulnerability affects how well the system can recover. As a result, after recovery, there's a greater chance of finding differences between the actual data and what was expected, assuming everything ran smoothly without crashes. This increase in the incorrect result percentage highlights the importance of balancing the frequency of logging and checkpointing based on the specific needs and reliability goals of the system.

4. Results and Observations

a. We ran our test script for various persistence and checkpoint granularity, and also for various kill times. Below are results from some of our runs:

   For t=10400, p= 40, c =100

INCORRECT QUERY RESULTS: 0/400 (0%) INCORRECT

For t=10400, p= 100, c =1000

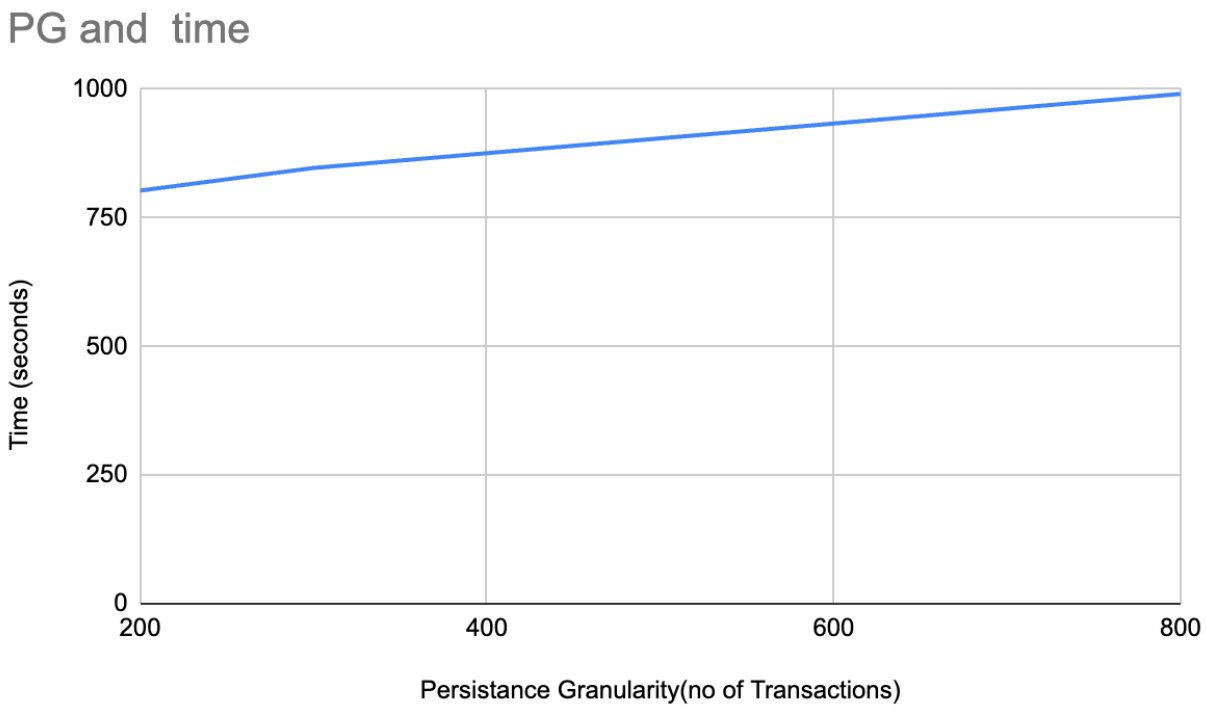
INCORRECT QUERY RESULTS: 0/400 (0%) INCORRECT

For t=10400, p= 4000, c =5000

INCORRECT QUERY RESULTS: 22/400 (5.5%) INCORRECT

Observations: Initially when persistence and checkpoint granularity are less, we get results to all of our queries, As we increase the values for p and c by a lot, we get some missing queries. This might be due to the fact that, when system crashes in the latter cases a lot of logs in memory, not yet written to disk are gone and the queries for those elements are missing.

b. Below graph shows different persistence granularities and the corresponding execution time it took to do the logging , checkpointing and recovery. **Please note: The time includes both the recovery time as well as the time for remaining operations and 400 queries.**

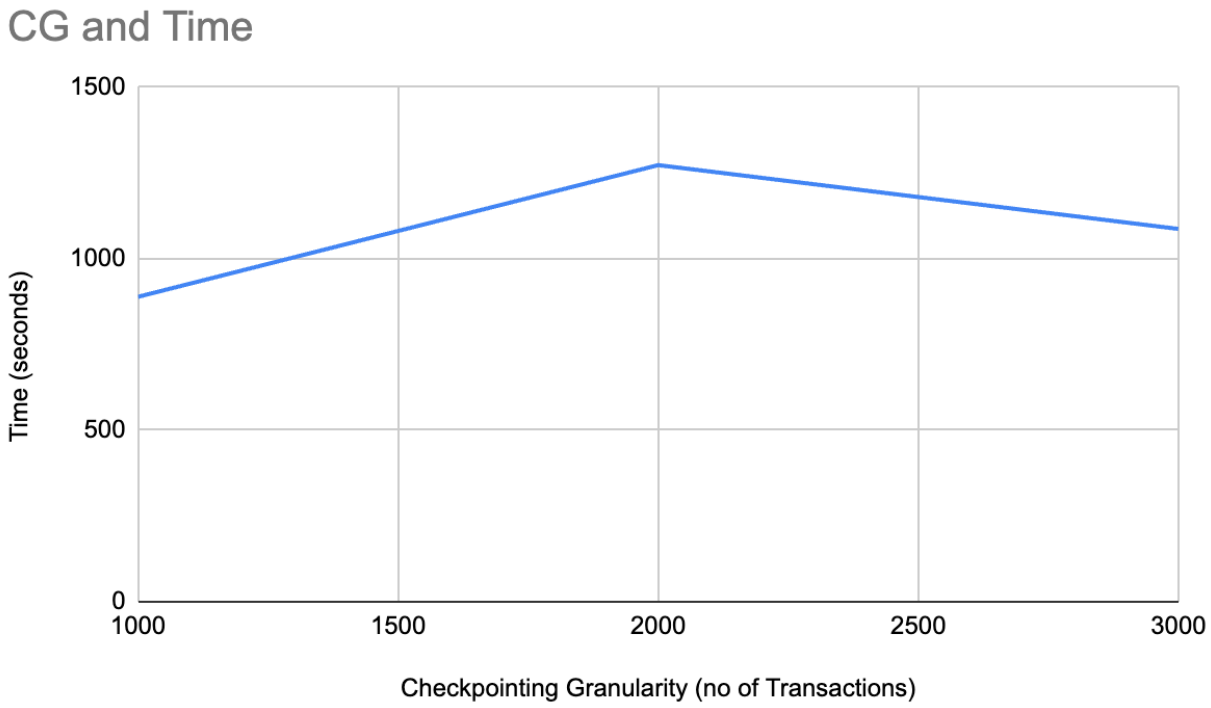


Observations: When we increase the persistence granularity (the number of transactions or operations after which logs are persisted to disk), we're essentially batching more operations before writing them to disk. This can have the following effects:

Increased Recovery Time: When a crash occurs, there are more uncommitted operations in the logs waiting to be replayed during recovery. This leads to a longer recovery time because the system needs to process and apply a larger number of operations.

Reduced Disk I/O: On the flip side, increasing persistence granularity can reduce the frequency of disk writes, which can be more efficient for the normal operation of the system. Frequent disk writes can be a performance bottleneck.
The choice of persistence granularity involves a trade-off between recovery time and normal operation efficiency. Smaller granularity (more frequent disk writes) reduces recovery time but may impact performance due to more I/O operations. Larger granularity (less frequent disk writes) improves normal operation efficiency but increases recovery time.

c. Below graph shows different checkpointing granularities and the corresponding execution time it took to do the logging ,checkpointing and recovery. **Please note: The time includes both the recovery time as well as the time for remaining operations and 400 queries.**



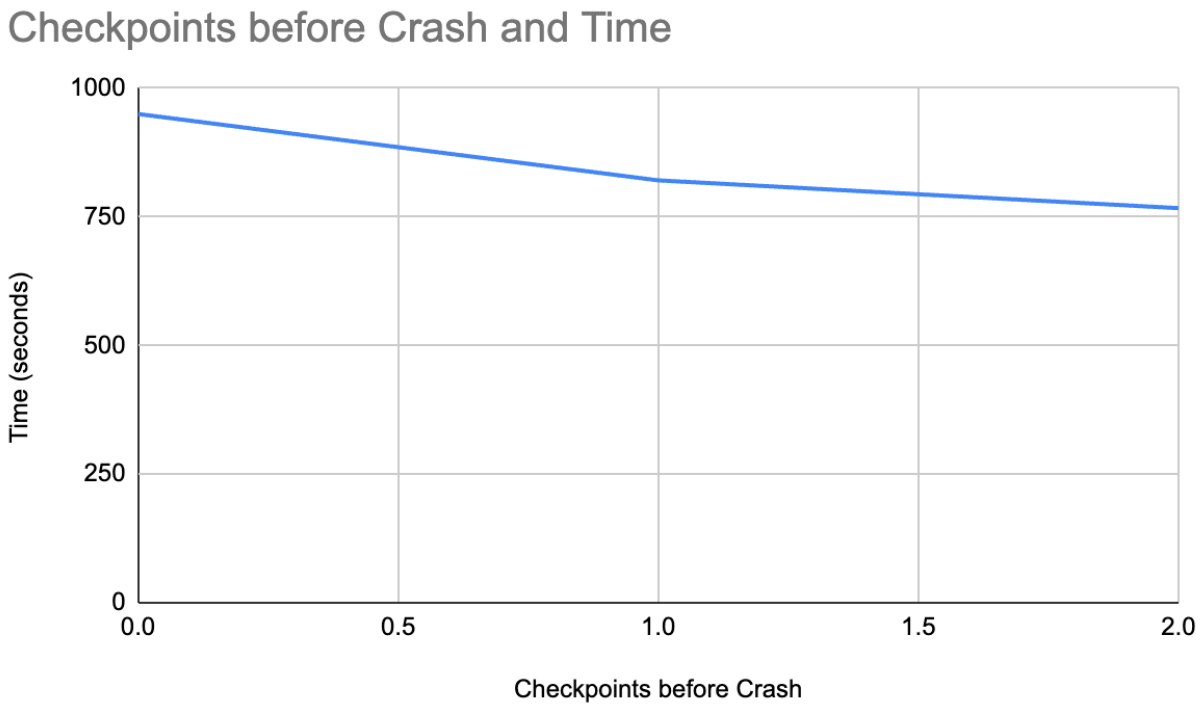
Observations: Our observations here mirror those discussed previously. As we increase the granularity, we observe a similar trend where the graph line initially increases. However, a noteworthy addition to these observations is that, after reaching a certain point, the line tends to become somewhat stagnant. We attribute this behavior to a delicate balance between the advantages and disadvantages of adjusting granularity.

One significant factor at play is the trade-off between reducing disk I/O through less frequent checkpointing and the time required for recovery. Beyond a certain point, the benefits gained from saving disk I/O by reducing checkpoint frequency may be offset by the additional time needed for recovery. This phenomenon raises a fundamental

question: what is the optimal granularity for both persistence and checkpointing? - Depends on how you think of your system as reliable and less frequent to crash.

We also tuned the kill time and observed the time and incorrect percentage at following scenarios

d. Graph for varying kill time to alter number of checkpoints that happen before a crash



Observations:

As we do more checkpoints, we have less logs to recover and hence time decreases as checkpoints increases.

System crashed before the first checkpoint
Here the correctness of recovery solely depends on the number of logs that were pushed onto the disk before the system crashed.
Execution time is more if flushing logs to disk is performed at smaller intervals.
Execution time isn't very large if logs are flushed at large intervals. Although this may result in a situation where recovery becomes impossible if the system crashes even before logs are persisted on disk.
Here, we made sure we replay the logs from the beginning and not log any operations as they are a part of recovery phase and not normal execution

System crashed after 1 checkpoint
We also recorded the time it took when crashed happened after the first checkpoint. Fewer logs to replay as we checkpointed once, decreasing time.

System crashed after 2 checkpoints
Even Fewer logs to replay as we checkpointed twice, decreasing time further.

5. Contributions

All of us were involved in all the phases of the project, Hence “equal contribution”. We were all part of the brainstorming process and then collaborated using github, working on parts and integrating later.

6. Acknowledgements and References

We extend our heartfelt gratitude to Professor Dr. Prashant Pandey for his comprehensive teaching and guidance throughout this course. His in-depth discussions on logging, checkpointing, and recovery were instrumental in enhancing our understanding of these fundamental concepts. We are thankful for his unwavering support and for patiently addressing our conceptual doubts during office hours.

We would also like to express our appreciation to our Teaching Assistants, Hunter McCoy and James McMohan, for their unwavering assistance and contributions to this project. Hunter played a pivotal role in helping us brainstorm edge cases, resolve code errors, and provided valuable insights into effective strategies. Furthermore, we are indebted to James McMahon, another Teaching Assistant, for his insightful discussions and support in shaping our approach towards logical logging. His contributions were instrumental in refining our project's design and methodology.

References:

An Introduction to Bε -trees and Write-Optimization:
http://supertech.csail.mit.edu/papers/BenderFaJa15.pdf

ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging:
https://web.stanford.edu/class/cs345d-01/rl/aries.pdf

Write-ahead logging and the ARIES crash recovery algorithm:
https://sookocheff.com/post/databases/write-ahead-logging/