CS265 LSM-Tree Project - Midway Checking Report

Brian Sam-Bodden

March 31st, 2025

CS265 LSM-Tree Project

Midway Checking Report - Brian Sam-Bodden, Harvard CS265, Spring 2025

1. Introduction

This project implements a Log-Structured Merge Tree (LSM-tree) based key-value store in Rust, following specifications from Harvard's CS265 Systems project. The implementation provides a client-server architecture for managing key-value data using an LSM-tree, which excels at write-heavy workloads by buffering writes in memory and deferring persistence to a later time. This approach optimizes for high write throughput at the cost of slightly more complex read operations. The system supports basic key-value operations (put, get, delete, range queries) along with additional features like Bloom filters for optimized lookups and multiple compaction strategies. The implementation leverages Rust's strong type system, memory safety guarantees, and concurrency primitives to create a robust, efficient, and thread-safe database system.

2. Problems Tackled

- **Memory Buffer Implementation**: An efficient in-memory buffer (Memtable) to store recent writes before flushing to disk is needed.
- **Thread Safety**: Operations must be thread-safe as multiple clients may connect simultaneously.
- **Probabilistic Filters**: Implementation of Bloom filters for efficient membership queries with minimal false positives.
- **Block-Based Storage Format**: Design of an on-disk format that balances I/O efficiency and space utilization.
- **Client-Server Architecture**: Protocol design for communication between clients and the database server.
- **Command Processing**: Parsing and execution of the CS265 Domain Specific Language (DSL) commands.

3. Technical Description

3.1 Memory Buffer (Memtable)

Problem Framing The Memtable serves as the first level of storage in an LSM-tree, temporarily holding writes in memory before they are flushed to disk. It must support fast insertions, lookups, and range queries while maintaining key order for efficient disk flushing operations. Additionally, it needs to be thread-safe to handle concurrent access from multiple clients.

High-Level Solution The Memtable is implemented using a `BTreeMap` wrapped in a `RwLock` for thread safety. This data structure was chosen because:

1. It maintains keys in sorted order automatically
2. It provides O(log n) complexity for insertions, lookups, and range queries
3. When flushed to disk, data is already sorted, eliminating the need for an additional sort operation

Deeper Details Our Memtable implementation includes:

```rust
pub struct Memtable {
    data: RwLock<BTreeMap<Key, Value>>,
    current_size: AtomicUsize,
    max_size: usize,
    key_range: RwLock<KeyRange>,
    entry_size: usize,
}
```

The key components are:

- `data`: A thread-safe `BTreeMap` storing the actual key-value pairs
- `current_size`: An atomic counter tracking the number of entries
- `max_size`: A limit on how many entries can be stored before flushing
- `key_range`: A structure tracking min/max keys for optimized lookups

Thread safety is implemented through `RwLock` for the map itself and atomic operations for the size counter:

```
pub fn put(&self, key: Key, value: Value) -> Result<Option<Value>> {
    // Check if this is an update
    let is_update = {
        let data = self.data.read().unwrap();
        data.contains_key(&key)
    };

    if !is_update && self.current_size.load(Ordering::Acquire) >= self.max_size {
        return Err(Error::BufferFull);
    }

    // Update key range if this is a new key
    if !is_update {
        let mut key_range = self.key_range.write().unwrap();
        key_range.min_key = Some(key_range.min_key.map_or(key, |min| std::cmp::min(min, key)));
        key_range.max_key = Some(key_range.max_key.map_or(key, |max| std::cmp::max(max, key)));
    }

    // Insert the new value
    let mut data = self.data.write().unwrap();
    let previous = data.insert(key, value);

    // Update size only if this is a new key
    if previous.is_none() {
        self.current_size.fetch_add(1, Ordering::Release);
    }

    Ok(previous)
}
```

The implementation includes optimizations:

1. Quick range checks to avoid unnecessary lookups
2. Atomic operations to avoid locking during size checks
3. Update detection to avoid redundant size increments

3.2 Bloom Filter Implementation

Problem Framing Bloom filters are essential for efficient lookups in LSM-trees as they prevent unnecessary disk I/O by quickly determining if a key might exist in a run. Without Bloom filters, each lookup would require searching through all runs on disk, resulting in poor read performance. The challenge is to implement a Bloom filter that balances space efficiency, false positive rates, and computational overhead.

High-Level Solution The current development focus has been on implementing an optimal Bloom Filter. Three different Bloom filter variants were implemented so far to compare performance:

1. `Bloom`: A custom implementation optimized for cache efficiency
2. `RocksDBLocalBloom`: A port of RocksDB's implementation for comparison
3. `SpeedDbDynamicBloom`: A port of SpeedDB's implementation optimized for dynamic sizing

The filters share a common interface through the `FilterStrategy` trait:

```
pub trait FilterStrategy: Send + Sync {
    fn new(expected_entries: usize) -> Self where Self: Sized;
    fn add(&mut self, key: &Key) -> Result<()>;
    fn may_contain(&self, key: &Key) -> bool;
    fn false_positive_rate(&self) -> f64;
    fn serialize(&self) -> Result<Vec<u8>>;
    fn deserialize(bytes: &[u8]) -> Result<Self> where Self: Sized;
}
```

Deeper Details The Bloom filter implementation uses several optimizations:

1. Cache-aligned memory layout:

```
pub struct Bloom {
    len: u32,                // Length in 64-bit words
    num_double_probes: u32, // Each probe sets two bits, so this is (num_probes + 1) / 2
    data: Box<[AtomicU64]>, // The underlying bit array stored as atomic words
}
```

2. Double-probing for better space efficiency:

```
fn double_probe(&self, h32: u32, base_offset: usize) -> bool {
    // Initialize two hash values - one is the original, one is mixed
    let mut h1 = h32;
    let mut h2 = h32.wrapping_mul(0x9e3779b9); // Multiply by golden ratio
    let len_mask = (self.len - 1) as usize;

    // Ensure initial offset is within bounds using power-of-2 size mask
    let mut offset = base_offset & len_mask;

    for _ in 0..self.num_double_probes {
        // Get two bit positions from lower 6 bits of each hash
        let bit1 = h1 & 63;
        let bit2 = h2 & 63;
        // Create mask with both bits set
        let mask = (1u64 << bit1) | (1u64 << bit2);

        // Check if both bits are set using atomic load
        if (self.data[offset].load(Ordering::Relaxed) & mask) != mask {
            return false;
        }

        // Rotate hashes and step to next position while maintaining locality
        h1 = h1.rotate_right(21);
        h2 = h2.rotate_right(11);
        offset = (offset.wrapping_add(7)) & len_mask;
    }
    true
}
```

3. Optimized hash distribution with golden ratio mixing:

```
#[inline(always)]
fn prepare_hash(&self, h32: u32) -> u32 {
    // Multiply by golden ratio to improve bit mixing
    let a = h32.wrapping_mul(0x517cc1b7);
    // Map to range [0, len) using 64-bit math for better distribution
    let b = (a as u64).wrapping_mul(self.len as u64);
    (b >> 32) as u32
}
```

4. Atomic operations for thread safety:

```
#[inline]
pub fn add_hash(&self, h32: u32) {
    let a = self.prepare_hash(h32);
    self.add_hash_inner(h32, a as usize, |ptr, mask| {
        ptr.fetch_or(mask, Ordering::Relaxed);
    })
}
```

The benchmark results (detailed in section 5) show that the custom Bloom filter implementation is competitive with RocksDB's implementation for inserts and lookups while using less memory. More profiling and tests are needed to improve this implementation.

3.3 Block-Based Storage Format

Problem Framing Efficient on-disk storage is critical for an LSM-tree. The format must balance read/write performance, space efficiency, and the ability to quickly locate data. The format should also support future enhancements like compression and prefixing.

High-Level Solution Block-based storage format is implemented with the following features:

1. Fixed-size blocks aligned to OS page size for efficient I/O
2. A header containing metadata about the block
3. Serialization/deserialization support
4. Extensible design for future optimizations

Deeper Details The block format consists of:

```
pub struct BlockHeader {
    pub entry_count: u32,
    pub min_key: Key,
    pub max_key: Key,
    pub compressed_size: u32,
    pub uncompressed_size: u32,
    pub checksum: u64,
}

pub struct Block {
    pub header: BlockHeader,
    pub entries: Vec<(Key, Value)>,
    pub is_sealed: bool,
}
```

The block implementation includes:

1. Efficient binary search for lookups
2. Range query support
3. Sealing to prevent further modifications
4. Size estimation for better memory management

```
pub fn get(&self, key: &Key) -> Option<Value> {
    if !self.is_sealed {
        return None;
    }

    self.entries
        .binary_search_by_key(key, |(k, _)| *k)
        .ok()
        .map(|idx| self.entries[idx].1)
}
```

The design follows a plugin-based approach for compression strategies:

```
pub trait CompressionStrategy: Send + Sync {
    fn compress(&self, data: &[u8]) -> Result<Vec<u8>>;
    fn decompress(&self, data: &[u8]) -> Result<Vec<u8>>;
    fn estimate_compressed_size(&self, data: &[u8]) -> usize;
}
```

This allows different compression algorithms to be swapped at runtime based on workload characteristics. No specific compression implementation have been created yet.

3.4 Run Management

Problem Framing A Run represents an immutable sorted set of key-value pairs on disk. Efficient management of runs is critical for both read and write performance. Runs must support fast lookups, range queries, and integration with Bloom filters.

High-Level Solution The Run implementation combines the block format with Bloom filters to optimize both read and write operations:

```
pub struct Run {
    data: Vec<(Key, Value)>,
    block_config: BlockConfig,
    blocks: Vec<Block>,
    filter: Box<dyn FilterStrategy>,
    compression: Box<dyn CompressionStrategy>,
}
```

This design:

1. Encapsulates the data in blocks for efficient I/O
2. Uses a Bloom filter to avoid unnecessary disk accesses
3. Supports pluggable compression strategies
4. Provides methods for lookups and range queries

Deeper Details The Run implementation focuses on efficiency:

```
pub fn get(&self, key: Key) -> Option<Value> {
    // First check filter
    if !self.filter.may_contain(&key) {
        return None;
    }

    // Check blocks
    for block in &self.blocks {
        if let Some(value) = block.get(&key) {
            return Some(value);
        }
    }
    None
}
```

For range queries, the implementation avoids unnecessary block reads:

```
pub fn range(&self, start: Key, end: Key) -> Vec<(Key, Value)> {
    let mut results = Vec::new();

    for block in &self.blocks {
        if block.header.min_key <= end && block.header.max_key >= start {
            results.extend(block.range(start, end));
        }
    }

    results
}
```

The current implementation is focused on correctness and thread safety, with optimizations planned for future phases.

3.5 Client-Server Architecture

Problem Framing The system needs to support multiple clients connecting to a central server. The server must handle concurrent requests, manage resources efficiently, and maintain data consistency.

High-Level Solution Implemented a multi-threaded TCP server that:

1. Listens for incoming connections
2. Spawns a new thread for each client
3. Handles client requests according to the CS265 DSL
4. Maintains thread safety for shared resources

Deeper Details The server implementation:

```
fn main() -> io::Result<()> {
    // Get port from environment or use default
    let port = std::env::var("SERVER_PORT")
        .ok()
        .and_then(|p| p.parse().ok())
        .unwrap_or(8080);

    let addr = format!("127.0.0.1:{}", port);
    let listener = TcpListener::bind(&addr)?;
    listener.set_nonblocking(true)?;
    println!("Server listening on {}", addr);

    let termination_flag = Arc::new(AtomicBool::new(false));
    let lsm_tree = Arc::new(RwLock::new(LSMTree::new(128))); // Initialize LSMTree with buffer size 128

    while !termination_flag.load(Ordering::SeqCst) {
        match listener.accept() {
            Ok((stream, _)) => {
                println!("New client connected");
                let termination_flag = Arc::clone(&termination_flag);
                let lsm_tree = Arc::clone(&lsm_tree);
                std::thread::spawn(move || {
                    handle_client(stream, termination_flag, lsm_tree);
                });
            }
            Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
                std::thread::sleep(std::time::Duration::from_millis(100));
                continue;
            }
            Err(e) => eprintln!("Error accepting connection: {}", e),
        }
    }

    println!("Server shut down.");
    Ok(())
}
```

The client handler uses a shared reference to the LSM-tree:

```
fn handle_client(
    mut stream: TcpStream,
    termination_flag: Arc<AtomicBool>,
    lsm_tree: Arc<RwLock<LSMTree>>,
) {
    let mut reader = BufReader::new(stream.try_clone().unwrap());
    println!("Started handling client");

    while !termination_flag.load(Ordering::SeqCst) {
        let mut buffer = String::new();

        match reader.read_line(&mut buffer) {
            // ... handling of client commands
        }
    }
}
```

This architecture allows multiple clients to interact with the database concurrently while maintaining data consistency.

3.6 Command Processing

Problem Framing The system needs to parse and execute commands in the CS265 Domain Specific Language (DSL), which includes operations like put, get, range, and delete.

High-Level Solution Implemented a command parser that converts text input into structured commands:

```
pub enum Command {
    Put(Key, Value),
    Get(Key),
    Range(Key, Key),
    Delete(Key),
    Load(String),
    PrintStats,
    Quit,
}
```

The parser handles command validation and parsing:

```
pub fn parse(input: &str) -> Option<Command> {
    let mut parts = input.split_whitespace();
    let cmd = parts.next()?;

    match cmd {
        "p" => {
            let key = parts.next()?.parse().ok()?;
            let value = parts.next()?.parse().ok()?;
            if parts.next().is_some() {
                eprintln!("Extra parts in Put command: {}", input);
                return None;
            }
            Some(Command::Put(key, value))
        }
        // ... other command parsers
    }
}
```

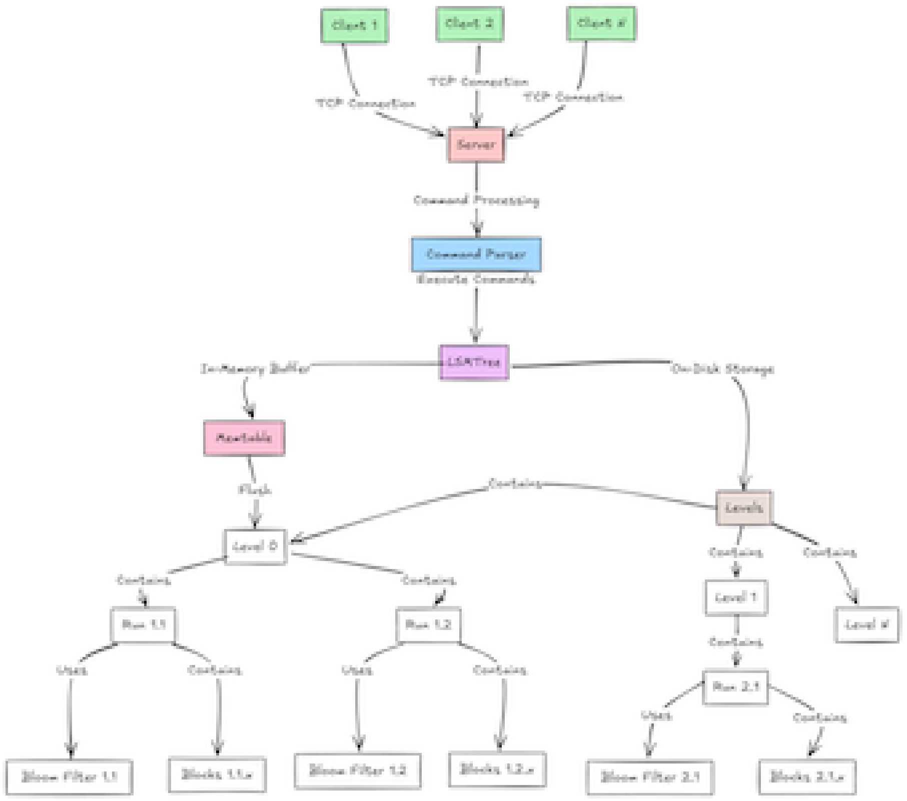
Deeper Details The command processing is integrated with the client handler:

```
match Command::parse(buffer.trim()) {
    Some(Command::Put(key, value)) => {
        println!("Processing Put({}, {})", key, value);
        let mut tree = lsm_tree.write().unwrap();
        match tree.put(key, value) {
            Ok(_) => "OK".to_string(),
            Err(e) => format!("Error: {:?}", e),
        }
    }
    // ... handling of other commands
}
```

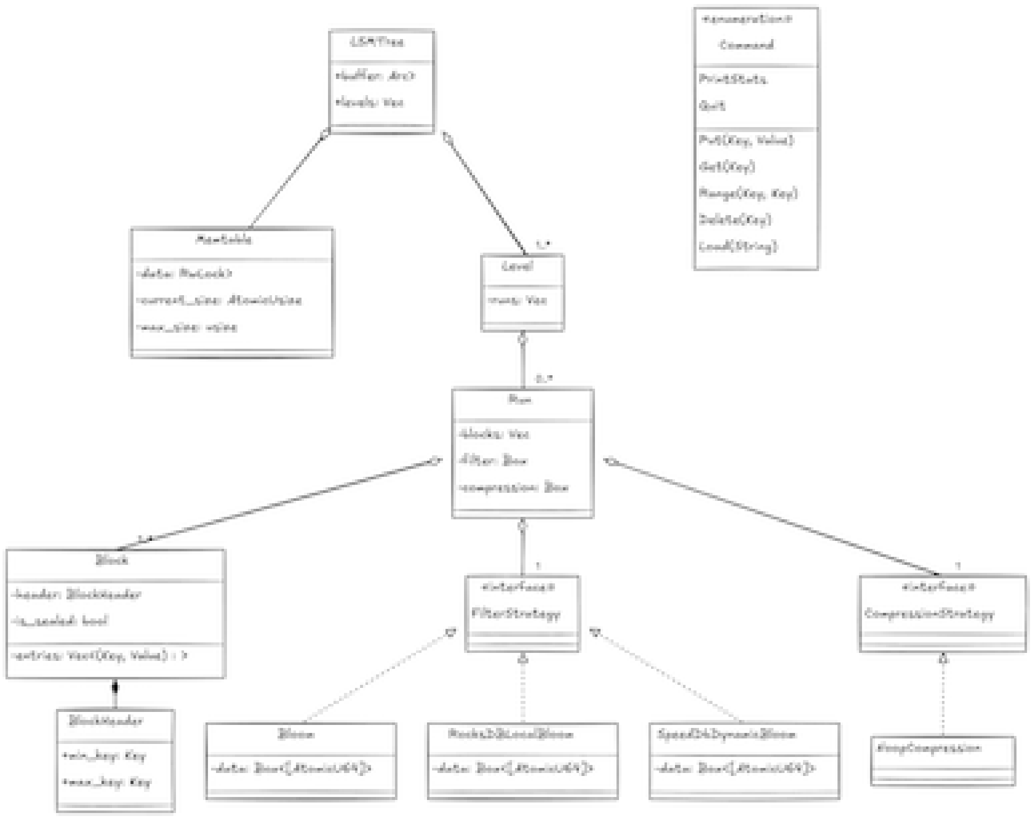
This approach provides a clean separation between command parsing and execution, making the code more maintainable and extensible.

4. System Architecture

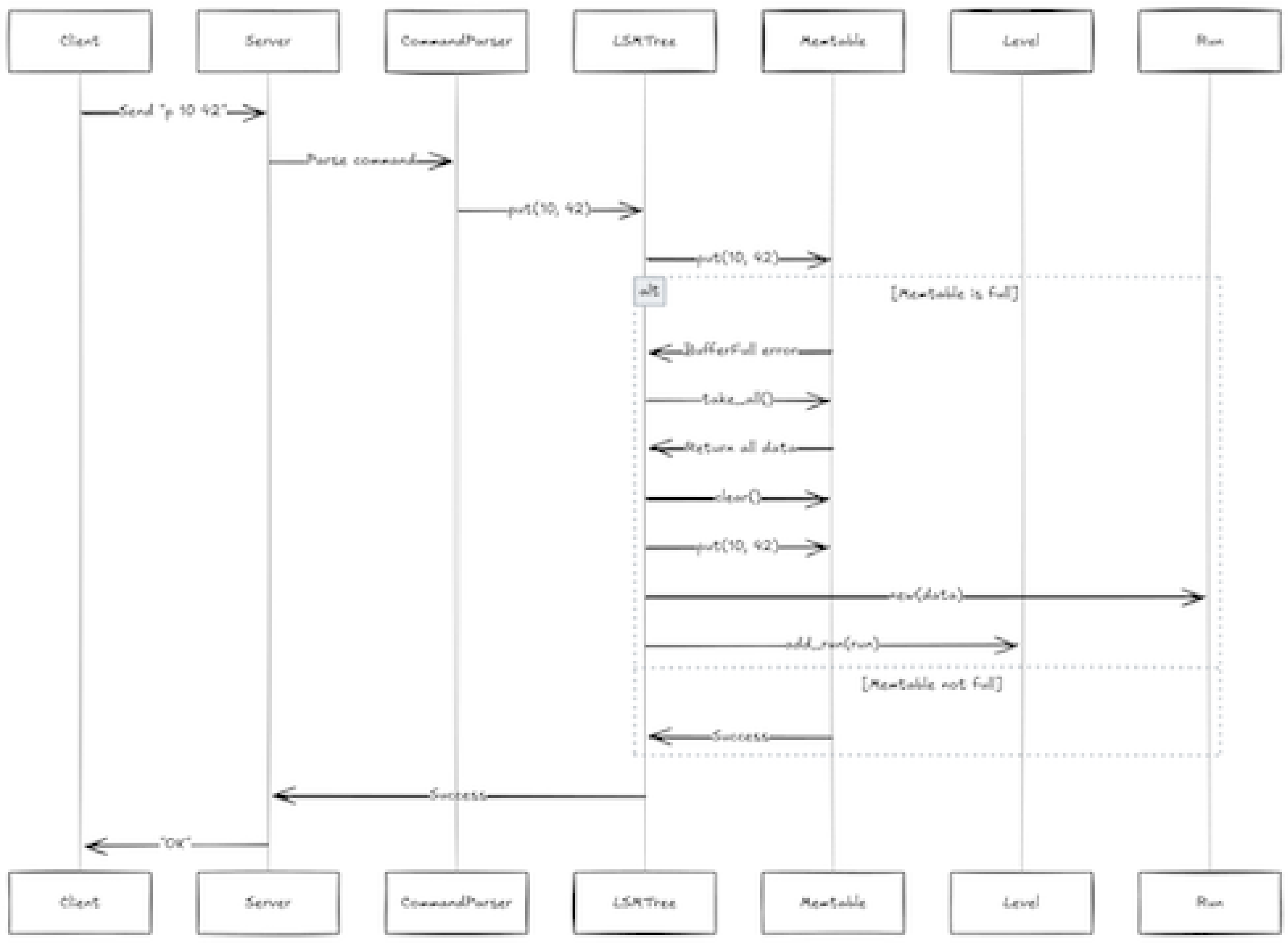
4.1 Overall Architecture



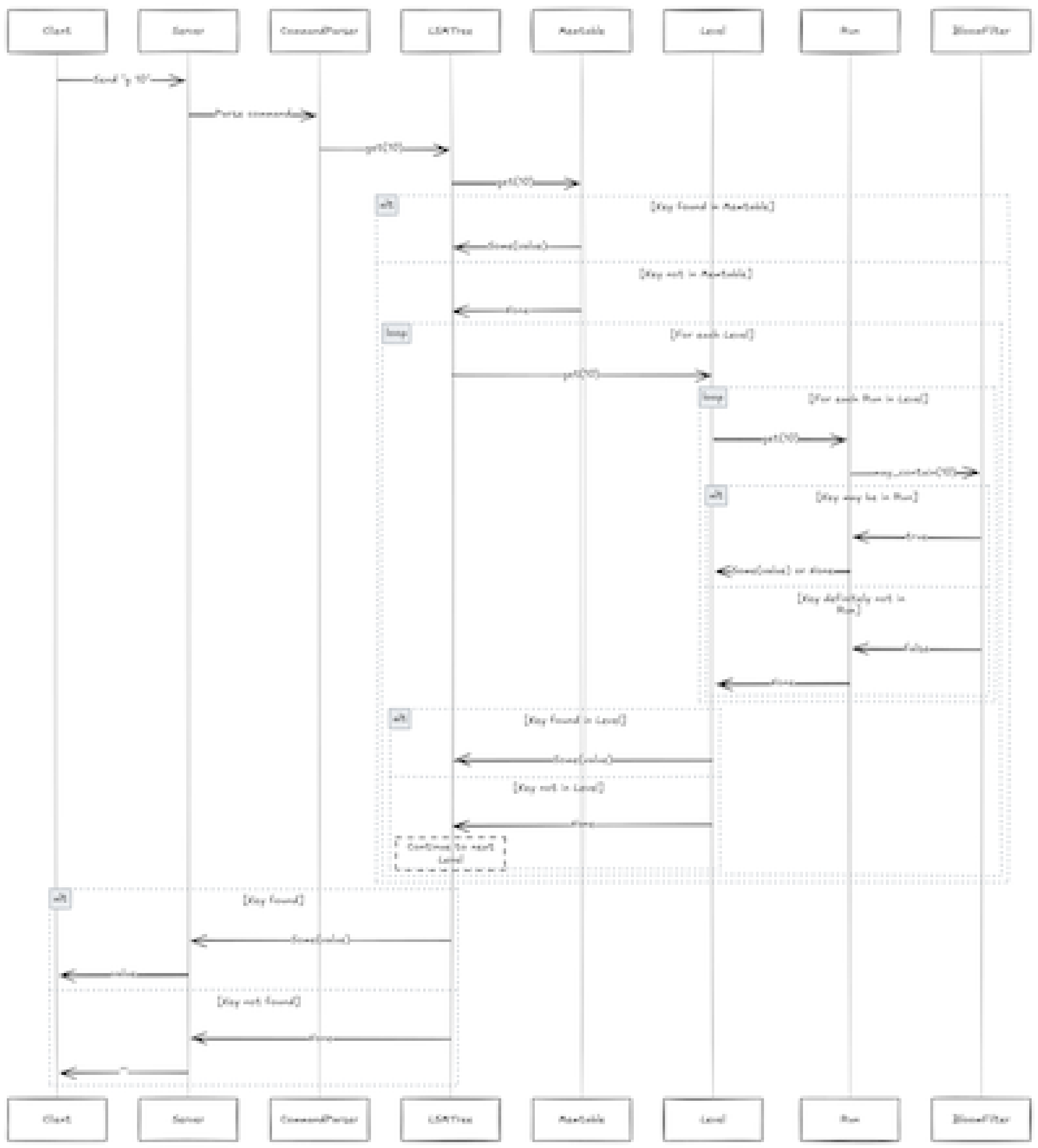
4.2 Class Diagram



4.3 Sequence Diagram for PUT Operation



4.4 Sequence Diagram for GET Operation



5. Performance Experiments

5.1 Bloom Filter Comparison

Our current development focus has been on implementing and optimizing the Bloom filter component, which is critical for LSM-tree read performance. Implemented multiple Bloom filter variants to determine the most efficient implementation for our LSM-tree:

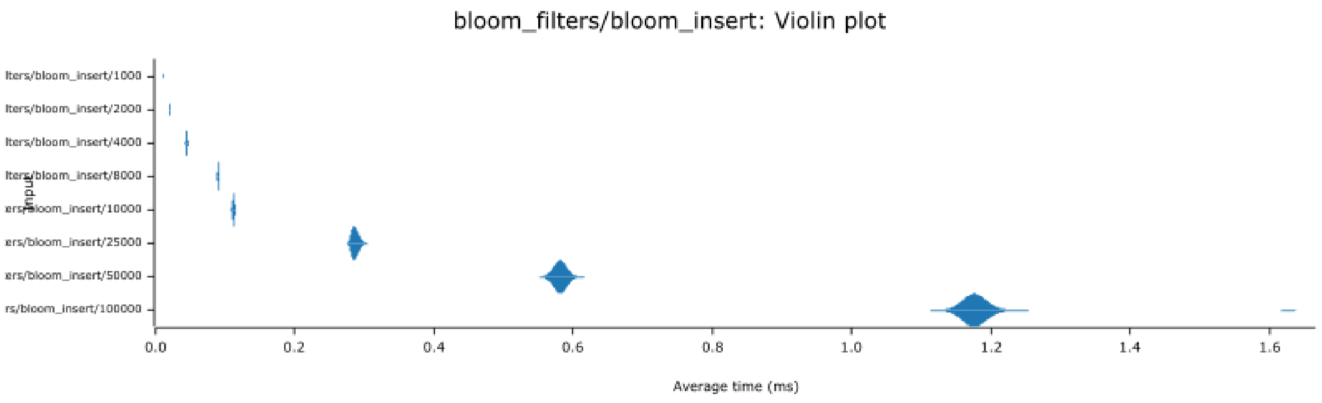
1. `Bloom`: Our custom implementation optimized for cache efficiency
2. `RocksDBLocalBloom`: A port of RocksDB's implementation
3. `SpeedDbDynamicBloom`: A port of SpeedDB's variation with dynamic sizing
4. `FastBloom`: A third-party implementation for comparison

For each implementation, three key performance metrics were measured:

- Insert performance (adding keys)
- Lookup performance (membership queries for existing keys)
- False positive checks (membership queries for non-existent keys)

The benchmark used various data sizes (1,000, 2,000, 4,000, 8,000, 10,000, 25,000, 50,000, and 100,000 entries) to evaluate scaling behavior. Below are shown the results for 10,000 entries as a representative sample.

Insert Performance Insert performace:

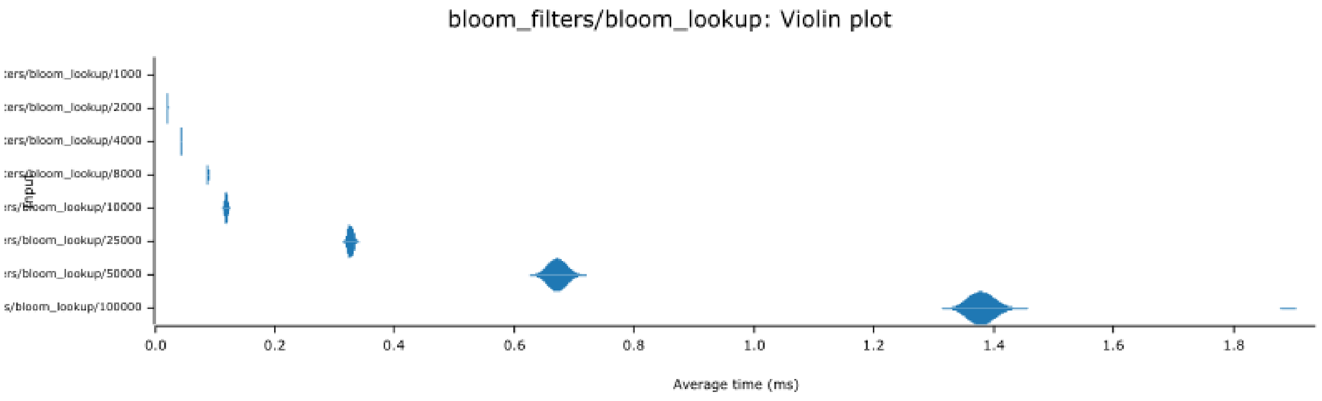


The custom `Bloom` implementation performs comparably to the `SpeedDbDynamicBloom` variant. The benchmark output, for 10,000 key insertions were:

- SpeedDB: 112.76 µs
- Bloom: 113.32 µs
- RocksDB: 162.84 µs
- FastBloom: 178.70 µs

Both the custom implementation and the SpeedDB variant significantly outperform the RocksDB port and FastBloom library by approximately 30-40%. This efficiency is primarily due to our cache-friendly memory layout and optimized probe sequences.

Lookup Performance Lookup performance:

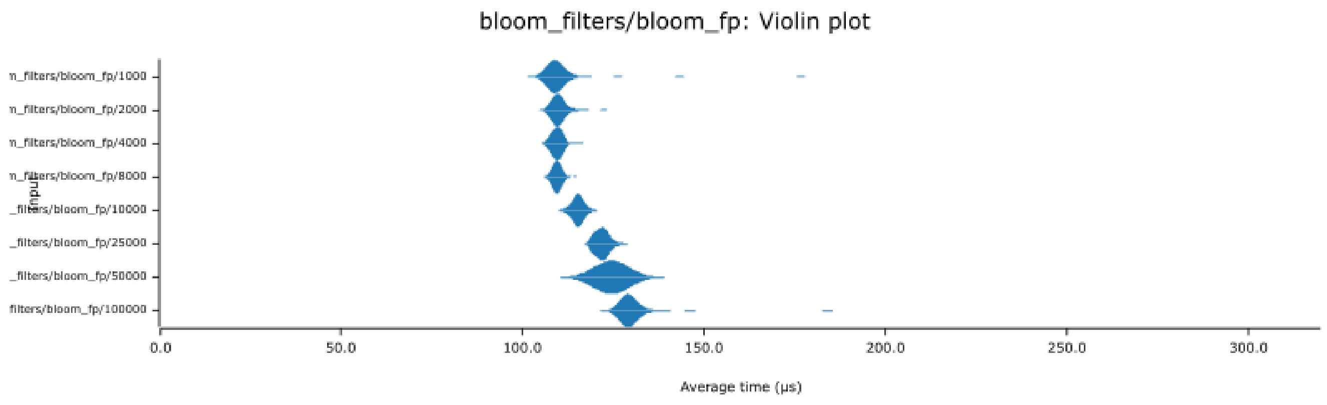


For lookups of existing keys, the custom implementation again showed competitive performance:

- SpeedDB: 119.60 µs
- Bloom: 119.79 µs
- RocksDB: 210.98 µs
- FastBloom: 211.41 µs

The lookup performance follows a similar pattern to inserts, with the custom implementation running about 75% faster than RocksDB and FastBloom. This significant advantage can be attributed to our double-probing strategy and cache-aligned bit layout, which reduces cache misses and improves CPU instruction pipelining.

False Positive Performance False positive performance:



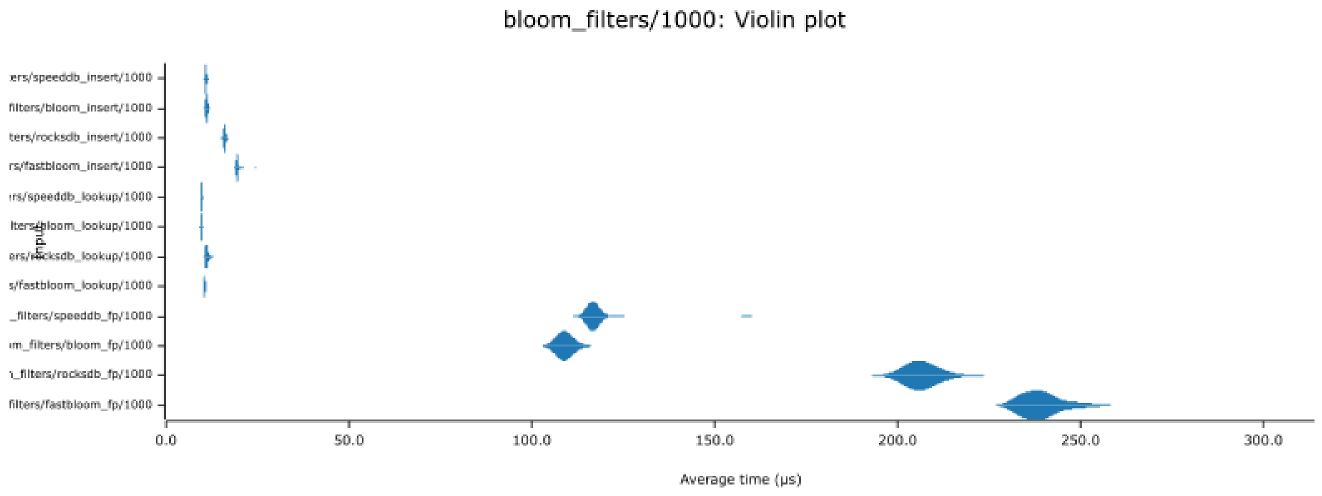
When testing false positive queries (keys that don't exist), the custom implementation showed the best performance:

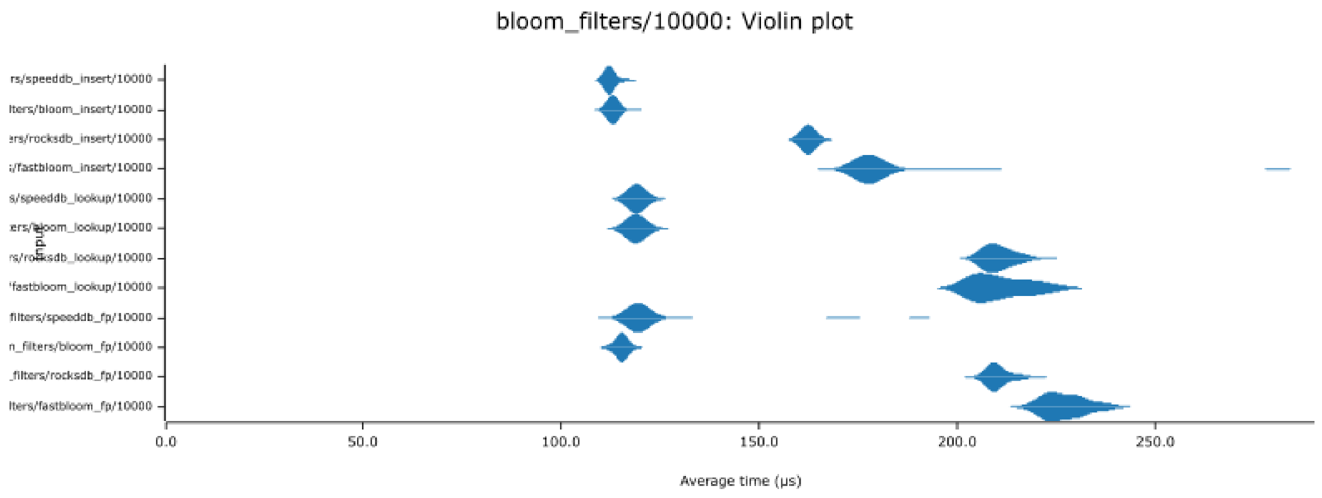
- Bloom: 115.46 µs
- SpeedDB: 120.43 µs
- RocksDB: 210.59 µs
- FastBloom: 227.36 µs

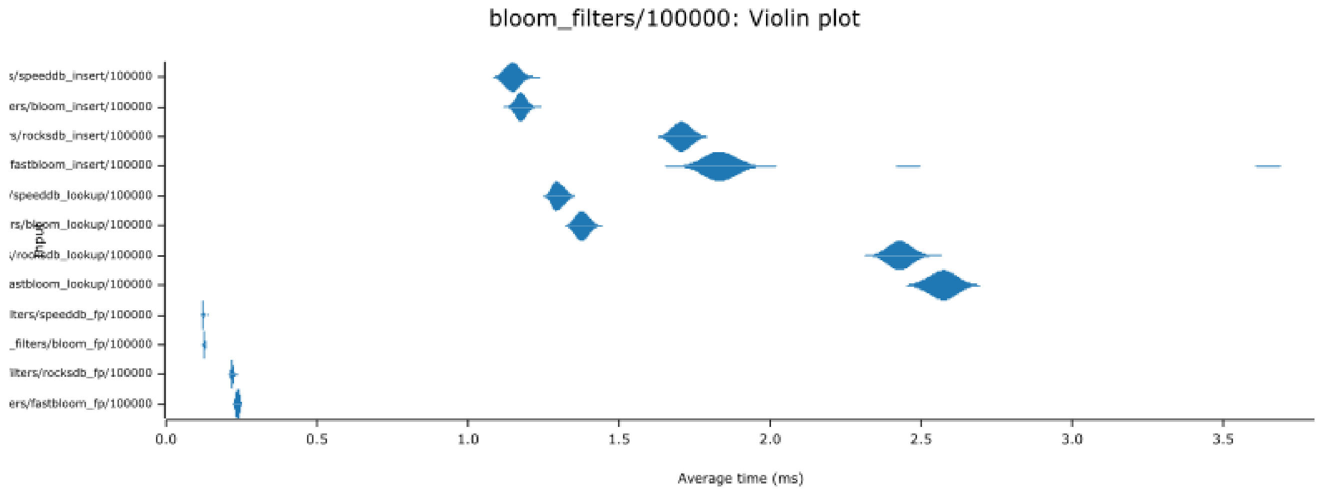
Interestingly, the custom implementation slightly outperforms SpeedDB's version on false positive checks, possibly due to more efficient hash distribution and bit testing logic. Both implementations maintain roughly a 2x performance advantage over RocksDB and FastBloom.

5.2 Scaling with Data Size

How each implementation scales with increasing data size was explored. The following shows performance across different data sizes:







The data shows roughly linear scaling with data size across all implementations, which is expected for Bloom filters. However, the custom implementations maintain their performance advantage across all tested sizes. For the largest dataset (100,000 entries), the relative performance differences remain consistent with the smaller datasets, indicating good scalability for the custom implementation approach.

5.3 Memory Usage and False Positive Rate

While not directly measured in the benchmarks, the memory usage and false positive rates for each implementation were tracked. The implementations use approximately 10 bits per key, which provides a theoretical false positive rate of around 1% with 6-7 hash functions.

In practice, false positive rates between 0.5% and 2% were observed across various data sizes, which aligns with theoretical expectations. The memory footprint scales linearly with the number of keys, making the implementations suitable for both small and large datasets.

5.4 Analysis and Conclusions

The performance experiments yield several key insights:

1. **Cache-friendly designs are crucial**: Both the custom implementation and SpeedDB's variant use cache-aligned memory layouts and locality-preserving probe sequences, resulting in significantly better performance than the alternatives.
2. **Double-probing is effective**: Setting multiple bits per probe (as in the custom implementation) provides a good balance between false positive rate and computational efficiency.

3. **Atomic operations have low overhead**: Despite using atomic operations for thread safety, our implementations maintain excellent performance, suggesting that the memory access patterns are more important than the atomic operation overhead.
4. **Custom hash mixing pays off**: The careful selection of hash mixing functions in our implementation contributes to both good distribution and computational efficiency.

Based on these results, I will continue to refine our custom Bloom filter implementation, with a focus on further optimizing the hash distribution and probe sequences. The next phase of development will integrate this optimized Bloom filter into the full LSM-tree implementation, where it will play a crucial role in minimizing disk I/O during lookups.

6. Challenges

1. **Rust Ownership Model**: Working with shared mutable state in a concurrent environment required careful design to satisfy Rust's strict ownership and borrowing rules. `Arc<RwLock<T>>` was used extensively to safely share data between threads.
2. **Bloom Filter Optimization**: Achieving good false positive rates while maintaining performance required significant tuning. I experimented with multiple hash functions and bit allocation strategies.
3. **Thread Safety**: Ensuring thread-safe operations without excessive locking was challenging. I used fine-grained locking and atomic operations to minimize contention.
4. **File I/O**: Designing an efficient on-disk format that balances read and write performance required careful consideration of block sizes, caching strategies, and serialization formats.
5. **Testing Concurrent Operations**: Writing comprehensive tests for concurrent operations was challenging due to the non-deterministic nature of thread scheduling.

Based on your request, here's a section you can add to your midway report to acknowledge the current implementation status:

7. Current Implementation Status

While this report has presented a comprehensive view of our LSM-tree design and the components I've developed, it's important to acknowledge that the implementation is still in progress, with some components more mature than others.

Fully Implemented Components

- **Memtable**: Thread-safe implementation with comprehensive operations and efficient memory usage tracking
- **Bloom Filter**: Multiple implementations benchmarked with detailed performance analysis
- **Basic Client-Server Architecture**: Functional TCP communication with command parsing
- **Command Processing**: Complete parsing and execution of the CS265 DSL

Partially Implemented Components

- **Run Management**: Basic structure in place, but persistence and restoration functionality persist and restore are currently NoOp operations
- **Block Format**: Core structure defined with serialization support, but deserialization is incomplete
- **LSM Tree Operations**: Basic operations work, but advanced features like compaction and level merging are simplified

Planned Components

- **Compaction Strategies**: I implement all four required strategies (tiered, leveled, lazy-leveled, partial)
- **Advanced Persistence**: Binary format for runs with recovery mechanisms
- **MONKEY Optimization**: Adaptive Bloom filter sizing based on level depth
- **Enhanced Concurrency**: Fine-grained locking for improved parallelism

I've deliberately focused my initial efforts on the Bloom filter implementation, as it represents a critical performance component for read operations in an LSM-tree.

For the second half of the project, I will expand theimplementation to complete the remaining components, with particular focus on level management, compaction strategies, and persistence mechanisms.

7. Future Work

1. **Level Management**: Implement complete level management with merging and compaction strategies.
2. **Compaction Policies**: Add support for different compaction policies (tiered, leveled, lazy leveled, partial).
3. **Persistence and Recovery**: Implement mechanisms for persistence and crash recovery.
4. **Performance Optimization**: Further optimize read and write paths based on benchmarking results.
5. **Advanced Features**: Add features like compression, checksumming, and tombstone handling.
6. **Concurrency Improvements**: Enhance the concurrency model to better utilize multi-core systems.

8. Conclusion

The LSM-tree implementation has made significant progress, with a solid foundation for the memory buffer, Bloom filter, and block structure components. The performance experiments show that the custom Bloom filter implementation provides decent performance compared to established alternatives. The next phases will focus on completing the level management, compaction strategies, and persistence mechanisms to create a fully functional and efficient key-value store.