BLINK DB - Part A: Storage Engine Design Document

24CS60R77 Ishan Rai

1. Workload Optimization Target

Selected Workload: Balanced (40% GET, 40% SET, 20% DEL)

My implementation is optimized for a **balanced workload** with approximately equal read and write operations. This choice was made for the following reasons:

- **Versatility**: A balanced approach ensures reasonable performance across various application scenarios without sacrificing either read or write efficiency.
- **Content Management Systems**: The project specification mentions content management systems as a target application, which typically exhibit balanced read/write patterns.
- **Future Extensibility**: This approach provides a solid foundation for Part B's network implementation, where request patterns may vary.

Benchmark Results and Analysis

Benchmark was conduted across three different workload patterns to validate design decisions. Each benchmark consisted of approximately 100,000 operations with the distribution matching my target workload profiles.

Benchmark Summary

Workload Type	Operations/ second	SET Latency (avg/p95)	GET Latency (avg/p95)	DEL Latency (avg/p95)
Balanced	255,770.41	0.004ms / 0.006ms	0.003ms / 0.005ms	0.003ms / 0.005ms
Read-heavy	229,719.66	0.005ms / 0.009ms	0.004ms / 0.006ms	0.004ms / 0.006ms
Write-heavy	130,594.68	0.007ms / 0.016ms	0.007ms / 0.016ms	0.007ms / 0.016ms

Analysis

The benchmark results strongly validate the choice to optimize for a balanced workload:

1. **Superior Throughput**: The balanced workload achieves the highest throughput at 255,770 operations per second, approximately 11% faster than read-heavy and 96% faster than write-heavy workloads.
2. **Sub-Microsecond Latencies**: All operations demonstrate excellent performance with average latencies between 3-4 microseconds and p95 latencies between 5-6 microseconds for the balanced workload.

3. **Consistent Performance**: The balanced workload shows the most consistent performance across all three operation types, with minimal variance between SET, GET, and DEL operations.

4. **Write-Heavy Limitations**: The significant performance drop in the write-heavy scenario (130,594 ops/sec) confirms that our hash table implementation, while excellent for balanced workloads, experiences expected performance degradation under write-intensive conditions due to more frequent collisions and potential rehashing operations.

5. **Read-Heavy Competitiveness**: The read-heavy workload performs reasonably well (229,719 ops/sec), demonstrating that our implementation can efficiently handle read-biased scenarios, though it still performs best with a balanced distribution.

2.Data Structure Selection

Primary Data Structure: Custom Hash Table with Separate Chaining

I implemented a custom hash table with separate chaining for collision resolution rather than using alternative structures like B-Trees or LSM trees.

Justification:

- **Average O(1) Time Complexity**: Hash tables provide constant-time operations for SET, GET, and DEL in the average case, making them ideal for balanced workload.
- **Memory Efficiency**: Implementation maintains reasonable memory overhead while providing fast access.
- **Dynamic Resizing**: The table automatically resizes when the load factor exceeds 0.7 (growth) or falls below 0.2 (shrinkage), balancing memory usage with performance.

Trade-offs Considered:

Data Structure	Advantages	Disadvantages	Suitability
Hash Table	O(1) average operations, simple implementation	No inherent ordering, potential for collisions	**High** for balanced workload
B-Tree	Ordered data, good range queries	O(log n) operations, more complex	Medium for read-heavy workload
LSM Tree	Excellent write performance	Read amplification, complex implementation	Low for balanced workload

3. Memory Management Strategy

LRU Eviction Policy with TTL Support

To address the requirement of periodically flushing outdated data while efficiently restoring older data when accessed, I implemented a hybrid approach:

LRU (Least Recently Used) Mechanism:

- Implementation: Maintained via MemoryManager class using a doubly-linked list (std::list) to track access order.
- Operation: Keys move to the front of the list when accessed, with least recently used keys at the back.
- Eviction: When memory limits are reached, items are removed from the back of the list.
- Time Complexity: O(1) for eviction, O(n) for key lookup (mitigated by infrequent evictions).

TTL (Time-To-Live) Expiration:

- Implementation: Each entry stores a TTL value and last accessed timestamp.
- Background Thread: A dedicated thread periodically scans for and removes expired entries.
- Command Support: The SET command supports an optional EX parameter for TTL specification.

Memory Limit Enforcement:

- Configurable Limit: Default maximum memory is set to 1GB.
- Tracking: Current memory usage is updated on each operation.
- Enforcement: When memory exceeds the limit, LRU eviction is triggered.

4. Implementation Details

1. StorageEngine

- Purpose: Provides the core CRUD operations (SET, GET, DEL).
- Thread Safety: All operations are protected by mutex locks.
- Memory Management: Tracks memory usage and enforces limits.
- TTL Handling: Background thread for expiration checks.

2. HashTable

- Design: Template class supporting any key-value types.
- Collision Resolution: Separate chaining with linked lists.
- Dynamic Resizing: Grows/shrinks based on load factor.
- Performance: O(1) average case for all operations.

3. MemoryManager

- Purpose: Implements LRU tracking and eviction.
- Data Structure: Doubly-linked list for O(1) eviction.
- Integration: Called by StorageEngine during operations.

4. REPL Interface

- Purpose: Command-line interface for user interaction.
- Features: Handles quoted values, command parsing, error reporting.
- Separation: Decoupled from StorageEngine for Part B integration.

Key Algorithms

Memory Limit Enforcement Algorithm:

```
PROCEDURE enforce_memory_limits():
    WHILE current_memory > max_memory AND LRU queue not empty:
        key = Remove least recently used key from LRU queue
        IF key exists in store:
            Decrease current_memory by (key size + value size)
            Remove key from store
```

TTL Expiration Algorithm:

```
PROCEDURE evict_expired():
    current_time = now()
    keys_to_remove = empty list

    FOR EACH bucket in hash table:
      FOR EACH entry in bucket:
        IF entry has TTL AND (current_time - entry.last_accessed > entry.TTL):
          Add entry.key to keys_to_remove

    FOR EACH key in keys_to_remove:
        Remove key from store
        Remove key from LRU queue
```

5. Performance Characteristics

Time Complexity

Operation	Average Case	Worst Case
SET	O(1)	O(n)
GET	O(1)	O(n)
DEL	O(1)	O(n)
LRU Update	O(n)	O(n)
TTL Check	O(n)	O(n)

Space Complexity

- Hash Table: O(n) where n is the number of entries
- LRU Queue: O(n) for tracking all keys
- Entry Overhead: Each entry requires additional space for TTL and timestamp

Optimizations

1. Batch TTL Expiration: Expired keys are collected first, then removed in batch to reduce lock contention.
2. Dynamic Resizing: Hash table capacity adjusts automatically to maintain optimal load factor.
3. Memory Tracking: Accurate tracking of memory usage to prevent overflow.

6. Trade-offs and Considerations

Performance vs. Memory Usage

- Trade-off: Higher memory usage for faster access times.
- Mitigation: LRU eviction ensures memory stays within limits.

LRU Implementation Complexity

- Trade-off: O(n) complexity for key lookup in LRU queue.
- Consideration: For very large datasets, a more complex LRU implementation using a hash map of iterators could reduce this to O(1).

TTL Scanning Overhead

- Trade-off: O(n) periodic scans for expired entries.

- Consideration: For production use, a priority queue ordered by expiration time would be more efficient.

7. Conclusion

Our storage engine implementation successfully meets the requirements specified in the project document. By choosing a hash table with separate chaining and implementing a hybrid LRU-TTL eviction policy, we've created a balanced solution that performs well for mixed workloads while efficiently managing memory constraints. The design maintains a clear separation between the storage engine and REPL interface, facilitating the integration with the network layer in Part B.
