BLINK DB: LSM Tree Storage Engine Design Document

1. Introduction

This document describes the design decisions and engineering optimizations for the BLINK DB key-value storage engine. The implementation uses a Log-Structured Merge (LSM) Tree architecture to optimize for write-heavy workloads while still providing efficient reads and range queries.

2. Workload Optimization

2.1 Target Workload Analysis

Among the three typical database workloads (read-heavy, write-heavy, and balanced), **BLINK DB is primarily optimized for write-heavy workloads**. This optimization is achieved through the LSM Tree architecture which provides the following benefits for write-intensive operations:

- **In-memory buffering**: Writes go to an in-memory structure (MemTable) first, avoiding immediate disk I/O
- **Sequential writes**: When flushing to disk, writes are performed sequentially rather than randomly
- **Batched operations**: Multiple writes are batched together when creating SSTables, improving throughput
- **Reduced write amplification**: The level-based structure prevents excessive rewrites of data

While optimizing for writes, the design still maintains reasonable read performance through:

- In-memory caching via LRU mechanism
- Sorted data structures for efficient binary searches
- Multi-level organization to limit the number of files that need to be checked

3. Data Structure Selection

3.1 LSM Tree vs. B-Tree Comparison

Criteria	LSM Tree	B-Tree	Reason for Selection
Write Performance	High	Medium	LSM Trees excel at ingestion with minimal random I/O
Read Performance	Medium	High	B-Trees have lower read amplification
Space Efficiency	High	Medium	LSM Trees have better compression opportunities
Range Query	Good	Excellent	Both support efficient range queries
Implementation Complexity	Medium	Medium	Both have similar complexity

Decision: LSM Tree was selected due to its superior write performance and space efficiency, which are critical for write-heavy workloads.

3.2 Core Data Structures

3.2.1 MemTable

```
struct MemTable {
   vector<KeyValue> entries;    // Vector of key-value entries
   size_t size_bytes;           // Current size in bytes
}
```

- Uses a sorted vector of key-value pairs
- Maintains entries in key order for efficient searches
- Binary search used for lookups (O(log n))
- Fast insertion at the cost of occasional sorts

Design Notes: A vector was chosen over a map/tree structure to avoid compiler issues with default constructors and to improve memory locality. When the vector grows large enough, a sort is performed to maintain key order.

3.2.2 SSTable (Sorted String Table)

```
struct SSTable {
   string file_path;            // Path to the file
   size_t level;                // Level in the LSM tree
   map<string, size_t> index;    // Index mapping keys to file offsets
   string min_key;              // Minimum key in this SSTable
   string max_key;              // Maximum key in this SSTable
}
```

- Immutable on-disk structures
- Contain sorted key-value pairs
- Include sparse indexes for fast lookups
- Store metadata (min/max keys) for bloom filters and range queries

3.2.3 LRU Cache

```
struct CacheEntry {
   string key;          // Key of the cache entry
   string value;        // Value of the cache entry
   time_t timestamp;    // Timestamp of the cache entry
}

list<CacheEntry> cache_list;                // List to maintain LRU order
unordered_map<string, list<CacheEntry>::iterator> cache_map;  // Map for quick lookup
```

- Implements the Least Recently Used caching policy
- Combines a linked list and hash map for O(1) access and updates
- Configurable size limit (CACHE_SIZE constant)

4. System Architecture

4.1 Component Hierarchy

1. **Client Interface**: SET, GET, DEL, SYNC, DEBUG commands
2. **Storage Engine**: Central coordinator for operations
3. **MemTable Layer**: In-memory write buffer and recent reads
4. **SSTable Layer**: Multi-level persistent storage
5. **Compaction System**: Background maintenance

4.2 Data Flow

4.2.1 Write Path

1. Key-value pair received via set() method
2. Entry added to LRU cache for fast future access
3. Entry written to active MemTable
4. If MemTable size exceeds threshold:
    - Active MemTable becomes immutable
    - New active MemTable created
    - Background flush of immutable MemTable to Level 0 SSTable
    - Compaction triggered if Level 0 has too many SSTables

4.2.2 Read Path

1. Key lookup received via get() method
2. Check LRU cache (fastest)
3. If not found, check active MemTable (fast)
4. If not found, check immutable MemTable if present (fast)
5. If not found, check SSTables in level order (slower)
    - Level 0: Check all tables (may have overlapping ranges)
    - Level 1+: Binary search to find the right table, then lookup
6. Return most recent version of the key based on timestamps

4.2.3 Delete Path

1. Key deletion received via del() method
2. Remove from LRU cache
3. Write a tombstone record (special marker with deletion flag) to active MemTable
4. During compaction, tombstones eventually remove the actual data

5. Performance Optimizations

5.1 Read Optimizations

1. **LRU Cache**: Maintains frequently accessed items in memory
2. **Key Range Filtering**: Min/max key metadata allows skipping SSTables
3. **Binary Search**: Efficient lookup in sorted structures
4. **Sorted Vectors**: Improved cache locality compared to trees

5.2 Write Optimizations

1. **MemTable Buffering**: In-memory buffer for high write throughput

2. **Sequential I/O**: Batch writes to disk in sequential patterns
3. **Background Flushing**: Non-blocking writes by using separate threads
4. **Write Batching**: Group multiple operations for better efficiency

5.3 Space Optimizations

1. **Tombstone Compaction**: Old deletion markers are removed during compaction
2. **Size-Tiered Compaction**: Each level is larger than the previous, reducing space overhead
3. **Vector-Based MemTable**: Lower memory overhead than tree structures

5.4 Concurrency Optimizations

1. **Fine-Grained Locking**: Separate mutexes for different components:
    - memtable_mutex: Protects MemTable operations
    - compaction_mutex: Protects level structure during compaction
    - cache_mutex: Protects LRU cache operations
2. **Background Compaction**: Compaction runs in a separate thread
3. **Lock-Free Reads**: Common read path avoids locks when possible

6. Implementation Details

6.1 Compaction Strategy

The BLINK DB implementation uses a level-based compaction strategy:

1. Level 0: Accepts SSTables directly from MemTable flushes
2. Level 1+: SSTables must have non-overlapping key ranges
3. Level size ratio: Each level N+1 is configured to be 10× larger than level N
4. Compaction trigger: Level L triggers compaction when it has more than the allowed number of SSTables
5. Compaction process:
    - Select files to compact from level L
    - Find overlapping files in level L+1
    - Merge all selected files, removing duplicates and applying tombstones
    - Write new files to level L+1
    - Delete original files

6.2 Persistence and Recovery

1. **File Structure**:
    - Each SSTable is stored as a data file with a corresponding index file
    - Data file contains serialized key-value pairs
    - Index file contains key to offset mappings
2. **Recovery Process**:
    - On startup, the system scans the database directory
    - Loads all existing SSTables
    - Reconstructs the level structure based on file paths

6.3 Memory Management

1. **MemTable Size Limit**: Configurable maximum size (4MB by default)
2. **Cache Size Limit**: Fixed maximum number of entries (1024 by default)
3. **Eviction Policies**:
    - MemTable: Flushed to disk when size limit is reached
    - Cache: LRU eviction when size limit is reached

7. Trade-offs and Limitations

7.1 LSM Tree Limitations

1. **Read Amplification**: May need to check multiple files for a single read
2. **Space Amplification**: Same key may exist in multiple levels temporarily
3. **Background Compaction Overhead**: CPU and I/O resource consumption

7.2 Design Trade-offs

1. **Vector vs. Tree for MemTable**:
    - **Trade-off**: Simpler implementation and better cache locality vs. potentially slower insertions
    - **Mitigation**: Sort after batch insertions to amortize the cost
2. **Level-Based vs. Size-Tiered Compaction**:
    - **Trade-off**: Better read performance vs. higher write amplification
    - **Decision**: Level-based chosen for better read performance
3. **Synchronous vs. Asynchronous Flushing**:
    - **Trade-off**: Consistency vs. performance
    - **Decision**: Asynchronous flushing with SYNC command for manual control

8. Future Enhancements

1. **Bloom Filters**: Add bloom filters to SSTables to quickly determine if a key might exist
2. **Compression**: Implement data compression for SSTables
3. **Adaptive Compaction**: Dynamically adjust compaction strategy based on workload
4. **Multi-Threaded Compaction**: Parallelize compaction for better performance
5. **WAL (Write-Ahead Log)**: Add durability for unflushed MemTable data

9. Conclusion

The BLINK DB storage engine uses an LSM Tree architecture to provide excellent write performance while maintaining acceptable read performance. The design is particularly well-suited for write-heavy workloads such as logging, time-series data, and high-ingestion systems.

Key engineering decisions like the choice of data structures, the compaction strategy, and the multi-level organization work together to provide a performant and space-efficient key-value store. The implementation balances complexity with functionality, providing a robust foundation that can be extended with additional optimizations in the future.