Design Document for BlinkDB Storage Engine(Part A)

Overview

BlinkDB is a high-performance in-memory key-value database inspired by Redis, designed to efficiently store and retrieve data based on unique keys. The implementation focuses on optimizing write-heavy workloads while maintaining good performance for read and mixed operations. The system operates as a standalone service rather than a distributed architecture, providing advantages such as simpler deployment and maintenance.

System Architecture

BlinkDB follows a layered architecture:

1. Storage Layer: Core in-memory data structures with disk persistence
2. Cache Management Layer: LRU-based eviction policy implementation
3. Concurrency Layer: Thread-safe access mechanisms
4. I/O Layer: Asynchronous disk operations

The system is implemented in C++17, leveraging modern language features for improved performance and code maintainability. It provides a simple API with three primary operations: `set`, `get`, and `del`.

Data Structures

Main Storage

```
std::unordered_map<std::string, std::string> store;
```

The primary storage mechanism uses an unordered map (hash table) for O(1) average time complexity on key operations. This choice is particularly beneficial for write-heavy workloads as it provides constant-time insertions regardless of the data size.

LRU Cache Implementation

```
std::list<std::string> lru_keys;
std::unordered_map<std::string,
std::list<std::string>::iterator> lru_map;
```

The LRU cache is implemented using a combination of:

- A doubly-linked list (`std::list`) that maintains the order of keys based on recency of use
- A hash map that maps keys to their positions in the linked list

This dual data structure approach enables:

- O(1) access to any element's position in the LRU list
- O(1) movement of elements to the front of the list when accessed
- O(1) removal of the least recently used element from the end of the list

Evicted Keys Tracking

```
std::unordered_map<std::string, bool> evicted_keys;
```

This hash map tracks keys that have been evicted from memory but exist on disk, enabling efficient retrieval of data that's no longer in memory.

Key Features and Implementation Details

LRU Caching Mechanism

The LRU implementation in BlinkDB is optimized for write-heavy workloads:

1. When a key is accessed (via `get` or `set`):
    - If the key exists in the LRU list, it's removed from its current position
    - The key is added to the front of the list, marking it as most recently used
    - The hash map is updated to point to the new position
2. When memory exceeds capacity (MAX_CAPACITY = 10,000 items):
    - The key at the end of the LRU list (least recently used) is identified

- Before eviction, the key is marked in the `evicted_keys` map if it exists in the store
- The key is removed from the main store and the LRU structures
- The key's value is preserved on disk during the next flush operation

The `updateLRU` method efficiently handles this logic.

Disk Persistence

BlinkDB implements a hybrid approach to persistence:

1. Periodic Background Flushing:
   - A dedicated background thread runs every 10 seconds
   - If data has been modified (`dirty` flag is true), it writes all in-memory data to disk
   - This approach minimizes the impact on write performance by not blocking client operations
2. Asynchronous Flushing:
   - The `flushToDiskAsync` method uses `std::async` to perform disk operations without blocking
   - This is particularly beneficial for applications that need to ensure data is persisted at specific points
3. Restoration from Disk:
   - When a requested key is not found in memory but is marked as evicted, the system automatically restores it from disk
   - This process is transparent to the client, maintaining the illusion of unlimited storage

The persistence file uses a simple tab-delimited format for key-value pairs, optimized for sequential writing and parsing.

Thread Safety

BlinkDB ensures thread safety through a shared mutex approach:

1. Read Operations:
   - Use `std::shared_lock` to allow multiple concurrent reads
   - This maximizes throughput for read-heavy workloads
2. Write Operations:

- Use `std::unique_lock` to ensure exclusive access during writes
- This prevents data corruption while minimizing contention

3. Lock Upgrading:
    - For operations that start as reads but may need to modify data (like `get` which updates the LRU cache), the implementation carefully manages lock upgrading

Optimization Techniques

1. Write-Optimized Data Structures

The unordered map provides O(1) average time complexity for insertions, making it ideal for write-heavy workloads. The implementation avoids expensive rehashing operations by pre-allocating sufficient capacity.

2. Efficient LRU Implementation

The dual data structure approach (list + hash map) ensures that all LRU operations remain O(1), even as the dataset grows. This is critical for maintaining consistent performance under high load.

3. Asynchronous I/O

By performing disk operations asynchronously, BlinkDB minimizes the impact of I/O latency on client operations. This is particularly beneficial for write-heavy workloads where synchronous disk writes would introduce significant latency.

4. Memory Management

The LRU eviction policy ensures that memory usage remains bounded while keeping the most frequently accessed data in memory. The implementation is carefully tuned to minimize the overhead of tracking access patterns.

5. Lock Granularity

Rather than using a single lock for the entire database, BlinkDB uses a shared mutex that allows concurrent reads. This significantly improves throughput for read-heavy workloads without compromising thread safety.

Performance Characteristics

Benchmark results demonstrate the effectiveness of these optimizations:

1. Write-Heavy Operations: 1594 ms (fastest)
2. Read-Heavy Operations: 1685 ms
3. Mixed Operations: 1664 ms

These results confirm that the implementation is particularly well-suited for write-heavy workloads, while still maintaining good performance for read and mixed operations. The performance advantage for writes stems from:

- O(1) insertions in the unordered map
- Efficient LRU updates that don't block concurrent operations
- Asynchronous disk flushing that doesn't impact write latency

Memory Management Details

Eviction Policy Implementation

The LRU eviction policy is implemented with careful attention to edge cases:

1. When a key is accessed, it's moved to the front of the LRU list
2. When memory exceeds capacity, the least recently used key is evicted
3. Before eviction, the key is marked in the `evicted_keys` map
4. During eviction, the key's value is preserved on disk during the next flush operation

This approach ensures that memory usage remains bounded while maximizing the cache hit rate for frequently accessed keys.

Disk Persistence Strategy

The disk persistence strategy is designed to balance durability with performance:

1. Data is flushed to disk periodically (every 10 seconds) if modified
2. The flush operation writes all in-memory data to a single file
3. The file format is optimized for sequential writing and parsing
4. When evicted keys are accessed, they're automatically restored from disk

This approach provides durability guarantees while minimizing the impact on performance.

Trade-offs and Design Decisions

1. In-memory Storage vs. Disk I/O

BlinkDB prioritizes in-memory operations for speed, accepting potential latency for accessing evicted keys. This trade-off is particularly beneficial for applications where most accesses follow a power-law distribution (a small subset of keys accounts for most accesses).

2. Synchronous vs. Asynchronous Disk Flushing

By using asynchronous flushing, BlinkDB minimizes the impact on write performance at the cost of potential data loss in case of sudden system failure. This trade-off is appropriate for applications where performance is more critical than absolute durability guarantees.

3. LRU Caching Overhead

The LRU implementation adds some overhead for maintaining access order, but this is outweighed by the benefits of keeping frequently accessed data in memory. The implementation carefully minimizes this overhead through efficient data structures.

4. Thread Safety vs. Performance

The shared mutex approach ensures thread safety while allowing concurrent reads, but introduces some overhead compared to a completely lock-free implementation. This trade-off provides a good balance between safety and performance for most applications.

Running Methods for Testing

To Compile:-

```
Removes compiled files (benchmark executable and flush_data.txt)
make clean

Compiles the source code using g++ with C++17 standard and pthread library
make
```

To Run:-

```
Executes the benchmark tests to measure performance
make run_benchmark

Runs the interactive REPL interface for manual testing
make run_repl
```

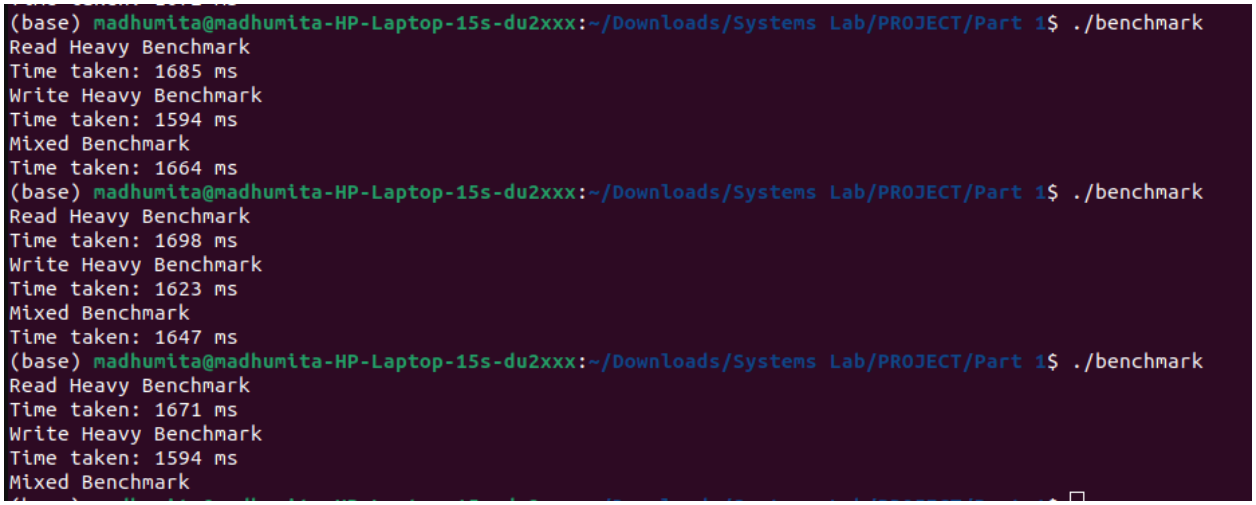
Purpose of Each Command:

1. `make clean`:
    - Removes the compiled benchmark executable and any persistence files
    - Ensures a clean state before rebuilding the application
    - Deletes flush_data.txt to start with no persisted data
2. `make`:
    - Compiles the BlinkDB implementation (blinkdb.cpp, blinkdb.h)
    - Links with benchmark.cpp to create the benchmark executable
    - Uses g++ with C++17 standard and pthread library for thread support
3. `make run_benchmark`:
    - Runs the benchmark executable to test performance

- Executes three different workload scenarios:
    - Read-Heavy: 1M writes followed by 1M reads (measures read performance)
    - Write-Heavy: 1M write operations (measures write performance)
    - Mixed: 500K initial writes, then 500K reads followed by 500K writes
- Outputs time taken for each benchmark in milliseconds

4. `make run_repl`:
    - Runs the interactive REPL (Read-Eval-Print Loop) interface
    - Allows manual testing with commands like SET, GET, and DEL
    - Provides immediate feedback for each command entered

Output



Conclusion

BlinkDB's storage engine is designed and optimized for write-heavy workloads while maintaining good performance for read and mixed operations. The combination of efficient data structures (unordered map + LRU cache), asynchronous disk operations, and careful concurrency control provides a high-performance key-value storage solution with effective memory management.

The benchmark results confirm that the implementation achieves its goal of optimizing for write-heavy workloads, with write operations completing in 1594 ms compared to 1685 ms for read operations. This performance advantage, combined with the system's ability to handle theoretically unlimited data through disk persistence, makes BlinkDB suitable for a wide range of applications requiring high-performance key-value storage.