BlinkDB Storage Engine Design Document

1. Introduction

BlinkDB is a key-value, high-performance, in-memory database inspired by Redis. It is designed to efficiently store and retrieve data using unique keys while maintaining an optimal balance between memory usage and persistence. This document outlines the design decisions, optimizations, and data structures used in the implementation of the BlinkDB storage engine.

2. Workload Optimization

For this implementation, BlinkDB is optimized for a **read-intensive workload**, where reads (GET) significantly outnumber writes (SET). This ensures that data retrieval operations are as fast as possible, making it ideal for applications requiring high read performance, such as content management systems, caching layers, and analytics platforms.

3. Data Structures

3.1 Key-Value Store

- **Unordered Map (unordered_map)**: Used as the primary data structure for storing key-value pairs in memory. It provides **O(1) average time complexity** for insert, lookup, and delete operations.
- **Least Recently Used (LRU) Cache**: Implemented using a **doubly linked list (list)** and a **hash map (unordered_map)** to maintain a fixed-capacity cache. This helps in efficiently managing memory and ensuring that frequently accessed keys remain in memory, reducing the need for disk lookups.

3.2 Persistent Storage

- **Append-Only File (AOF) Logging**: Every write (SET/DEL) operation is appended to a file (blinkdb.aof). This enables data recovery in case of a system crash.
- **Preloading Hot Data**: Frequently accessed data is loaded into memory on startup to minimize disk accesses during runtime.
- **Lazy Loading from Disk**: If a key is not found in memory, the database searches the AOF file for the last stored value, ensuring a fallback mechanism for cold data.
- **Periodic Flush to Disk**: The least recently used keys are flushed to disk when the memory limit is reached, preventing memory overflow.

4. Query Processing

4.1 SET Command

1. Check if the key already exists.
    - If it does, update the value and move the key to the front of the LRU list.
    - If it doesn't and the cache is full, evict the least recently used (LRU) key.
2. Add the new key-value pair to the in-memory store.
3. Append the operation to the AOF file.

4.2 GET Command (Optimized for Read-Intensive Workloads)

1. **Check Memory First**: If the key exists in memory, return it immediately (O(1) lookup time).
2. **If Not in Memory, Load from Disk**:
    - Scan the AOF file to retrieve the latest value for the key.
    - Load it into memory for faster subsequent access.
3. **Avoid Frequent Disk Access**:
    - Frequently accessed keys remain in memory due to the LRU caching mechanism.
    - Optimized for scenarios where the same keys are repeatedly accessed.

Example of Query Execution:

1. User runs SET user Krishna
2. The key user is stored in memory and appended to the AOF file.
3. Later, a GET use request is executed:
    - Since user is in memory, the value Krishna is returned instantly.
    - The key is moved to the front of the LRU cache to indicate recent usage.
4. If user were not in memory, the AOF file would be searched, and the key would be loaded back into memory.

4.3 DEL Command

1. If the key exists in memory, remove it from the hash map and LRU list.
2. Append the deletion operation to the AOF file.
3. If the key is not found, return an appropriate message.

5. Storage Mechanism & Eviction Policy

- **Capacity Management**: BlinkDB has a fixed-capacity LRU cache (default: 100 keys).
- **Eviction Policy**: Uses the **Least Recently Used (LRU) algorithm** to remove stale keys and free memory.
- **Persistence Strategy**:
    - Append-only logging for durability.
    - Background process for log compaction to reduce AOF file size.
    - Frequently accessed keys are reloaded into memory upon retrieval.

6. Performance Considerations

6.1 Read Performance (Highly Optimized)

- **Memory-First Approach**: Since most reads are served from memory (O(1) time complexity), retrieval is nearly instantaneous.

- **Fallback to Disk**: If a key is not in memory, a linear scan of the AOF file is performed ($O(n)$ worst case).
- **Re-inserting Hot Keys**: Once a key is read from disk, it is restored to memory for faster future access.
- **Batch Preloading for Hot Data**: Frequently used keys are preloaded into memory at startup to enhance performance.

6.2 Write Performance

- **Optimized for Fast Reads at the Cost of Writes**: Writing to an unordered map is $O(1)$, and appending to a file is $O(1)$, but frequent writes are deprioritized in favor of rapid retrieval.
- **Log Compaction Strategy**: Periodic cleanup of the AOF file to optimize performance and prevent excessive growth.

6.3 Delete Performance

- **Fast Deletes**: Deletion from the unordered map is $O(1)$, and deletion is recorded in the AOF file.
- **Delayed Disk Update**: Instead of removing keys from disk immediately, deletion records are added to AOF for eventual cleanup.

7. Future Enhancements

1. **AOF Log Compaction**: Implement periodic log trimming to remove obsolete entries and optimize file size.
2. **Asynchronous Disk Storage**: Implement background writing to disk for improved performance.
3. **Prefetching Mechanism**: Use machine learning-based predictions to preload frequently requested data into memory.
4. **Network Layer**: In Part 2, BlinkDB will support network-based communication using the Redis protocol.

8. Conclusion

BlinkDB provides a simple yet efficient in-memory key-value store with persistence. By optimizing for a **read-heavy workload**, it ensures that data retrieval is as fast as possible while leveraging disk storage as a fallback. The LRU caching mechanism ensures that frequently accessed data remains in memory, reducing reliance on slower disk reads and enhancing overall performance.