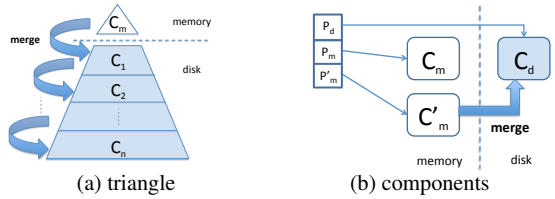

(a) triangle (b) components

Figure 1: **LSM-DS architecture: current (mutable) memory component C_m, previous (immutable) memory component C_m', and a disk component C_d. The merge incorporates the content of C_m' into C_d, while new items are added to C_m. All three components are accessible through global pointers P_m, P_m', P_d.**

of the components $C_1, \ldots, C_n$ reside on disk. For simplicity, in the context of this work, they are perceived as a single component, C_d. The architecture of LSM-DS is depicted in Figure 1 (ignore C_m' for now). An additional important building block is the *merge* procedure, which incorporates the contents of the memory component into the disk. It is executed in the background as an automatic maintenance service.

In order to differentiate the interface of the internal map data structure from that of the entire LSM-DS, we refer to the corresponding functions of the in-memory data structure as `insert` and `find`.

`insert(k,v)` – inserts the key-value pair (k, v) into the map. If k exists, the value associated with it is overwritten. A key is removed by inserting a $\perp$ value.

`find(k)` – returns a value v such that the map contains an item (k, v), or $\perp$ if no such value exists.

Initially, a put operation inserts an item into the main memory component C_m, and logs it in a sequential file for recovery purposes. When C_m reaches its size limit, which can be hard or soft, it is merged with component C_d, in a way reminiscent of merge sort. When considering multiple disk components C_m is merged with component C_1. Similarly, if a disk component C_i becomes full its data is migrated to the next component C_{i+1}. Component merges are often referred to as *compaction*, and are performed as background processes. The items of both C_m and C_d, are scanned and merged. The new merged component is then migrated to disk in bulk fashion, replacing the old component.

The get operation may require going through multiple components until the key is found. But when get operations are applied mostly to recently inserted keys, the search is completed in C_m. Moreover, the disk component utilizes a large RAM cache. Thus, in workloads that exhibit locality, most requests that do access C_d are actually satisfied from RAM as well.

To allow put operations to be executed while rolling the merge, the memory component becomes immutable, at which point it is denoted as C_m'. A new memory component C_m then becomes available for updates (see Figure 1). The put and get operations access the components through three global pointers: pointers P_m and P_m' to the *current* (mutable) and *previous* (immutable) memory components, and pointer P_d to the disk component. When the merge is complete the previous memory component is discarded. Allowing multiple puts and gets to be executed in parallel is discussed in the sequel.

3. cLSM Algorithm

We now present cLSM, our algorithm for concurrency support in an LSM-DS. Section 3.1 presents our basic approach for providing scalable concurrent get and put operations; this solution is generic, and can be integrated with many LSM-DS implementations. In Section 3.2, we extend the LSM-DS's functionality with snapshot scans, which are implemented by state-of-the-art NoSQL key-value stores (e.g., [2, 20]). This extension assumes that the in-memory data structure supports ordered iterated access with weak consistency (explained below), as various in-memory data structures do (e.g., [4, 5, 14]). Finally, in Section 3.3, we provide general-purpose non-blocking atomic read-modify-write operations. These are supported in the context of a specific implementation of the in-memory store as a skip list data structure (or any collection of sorted linked lists).

cLSM optimizes in-memory access in the LSM-DS, while ensuring correctness of the entire data store. Specifically, if the in-memory component's operations ensure serializability, then the same is guaranteed by the resulting LSM-DS.

cLSM is almost entirely non-blocking (lock-free) in that it does not block threads during normal in-memory operation: If the underlying implementation of the in-memory component is non-blocking, then, other than inherent blocking of the LSM-DS (when get searches the disk component or put blocks due to hard size limits of the in-memory components), cLSM *never* blocks get operations and only blocks put operations during short intervals before and after a merge occurs, whilst the global pointers are being updated.

3.1 Put and Get Operations

In this section we present our support for `put` and `get` operations. We assume an existing thread-safe map data structure for the in-memory component, i.e., its operations can be executed by multiple threads concurrently. Numerous data structure implementations, (e.g., see [5, 19, 22]), provide this functionality in a non-blocking and atomic manner. The disk component and merge function are implemented in an arbitrary way.

We implement our concurrency support in two hooks, `beforeMerge` and `afterMerge`, which are executed im-