CS205 Assignment 3 Documentation

Architecture

Modularizing a hierarchical storage structure

To represent the hierarchical structure of our storage – memory cache, disk storage, and cloud storage – we created the `ChainableStore` class. Each `ChainableStore` is backed by a `Repo`.

Multiple instances of `ChainableStore` can be chained together to provide the following behaviour:

1. If a resource is requested at the top-level `ChainableStore`, the accompanying `Repo` will first be checked for the resource. If no resource exists, then the chain will be traversed down the hierarchy until the resource is found or the end of the chain is reached.
2. If a resource is found in a lower-level store, then all stores above will be updated to save the resource for caching.
3. If a resource is deleted from a store, then all cached instances of the resource will be deleted from stores above.

Highly concurrent data-fetching with minimal thread waiting

The call to `Repo` is done in a background thread, which is freed as soon as a result is returned. This allows background threads to be reused for other purposes before entering the lower store with a potentially long blocking time.

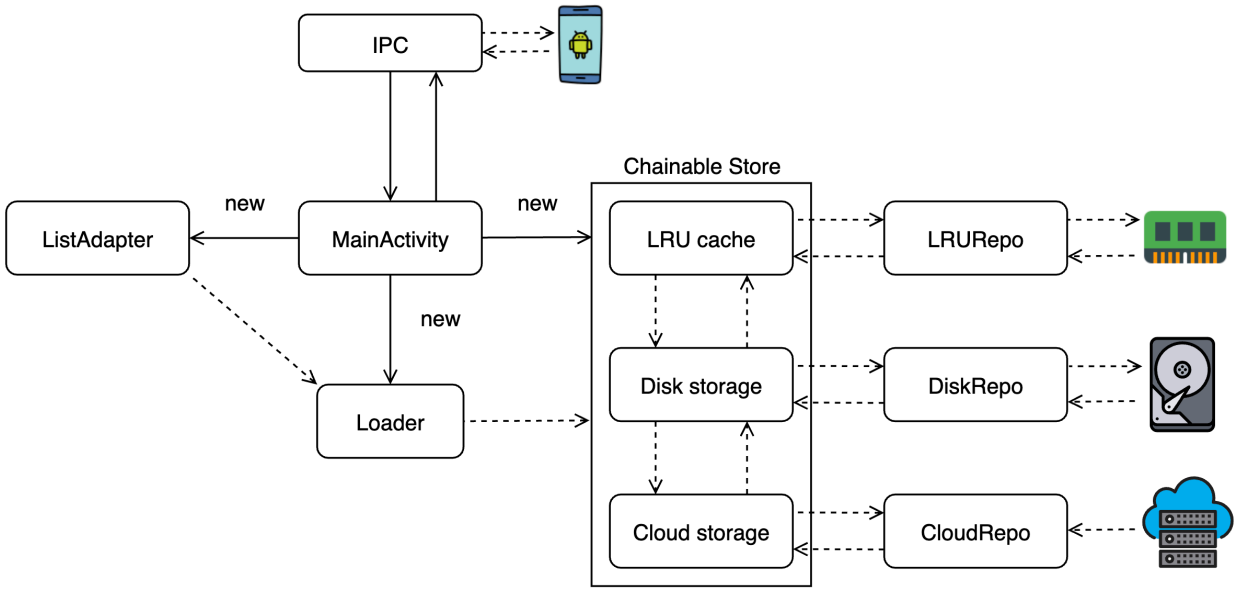
For example, if a thread is searching for key "A" in a main memory cache but cannot find an entry, it will prepare to search secondary storage. However, secondary storage has a long blocking time. Instead of continuing to explore secondary storage with the same thread, we allow other requests to be serviced first. A request for key "B" in main memory could be serviced quickly before the thread is reassigned to the current task of checking secondary storage, thereby reducing turnaround time.

Repo

Implementations of `Repo` represent different strategies to retrieve a resource. The repo should perform its data fetching synchronously and minimize spawning new threads, as `ChainableStore` will handle background threading.

One-way data flow

Data stores and loaders are defined in the main activity, and injected into lower-level dependencies for inversion of control. As such, all data constructs are defined and instantiated in `MainActivity` before being passed down into adapters and views.



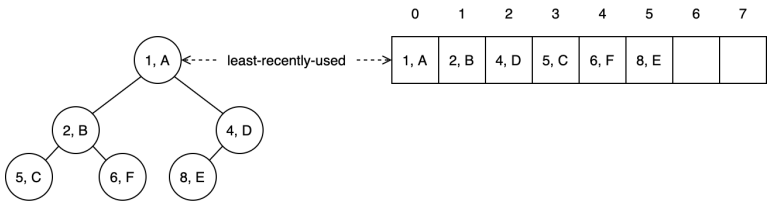
LRU Array Implementation

The standard library does not contain a convenient implementation of an LRU cache with a limit condition being dependent on total memory size. The closest data structure, `LinkedHashMap`, only allows for the removal of **one** eldest entry per insertion. Therefore, we decided to implement our own LRU cache using an array-backed priority queue and an array-backed hashmap.

Hashmap

key	pqOffset	value
A	0	first
B	1	second
C	3	third
D	2	fourth
E	5	fifth
F	4	sixth

Priority Queue



Each entry in the priority queue contains a counter `label`, and the map key as its `value`. Whenever an item with key `value` is used, its `label` is updated to the maximum counter. This ensures that the item with minimum counter `label` represents the least-recently used item. If the item is down-or-upheaped, `pqOffset` of the corresponding map entry is updated.

Each entry of the hashmap contains a tuple of `pqOffset` and the stored `value`. Whenever an entry is "used", `pqOffset` allows us to find the priority queue item in constant time, and update `label` before the heapifying.

This allows our LRU cache to achieve $O(1)$ insertion, $O(logn)$ updates, and $O(logn)$ deletions.