Generic Local Cache

Task:

You're tasked with writing a spec for a generic local cache with the following property: If the cache is asked for a key that it doesn't contain, it should fetch the data using an externally provided function that reads the data from another source (database or similar). What features do you think such a cache should offer? How, in general lines, would you implement it?

Features:

1. Obviously the first feature we need is configuring what is the memory boundaries available for the cache layer. We can try to do it with actual memory consumption which might be hard to do or by approximation by limiting the amount of data items we cache. Both approaches will need to have an eviction mechanism in place.
2. What should be the eviction policy if we need to free memory? We might add configuration for switching between eviction policies.
   - LRU (Least Recently Used) eviction policy nicely fits most of the use cases for caching. The idea is to evict entries that are not used as often based on the least recently used order.
   - LFU (Least Frequently Used) eviction policy usually involve the system keeping track of the number of times an item is referenced. When the cache is full and requires more room the system will purge the items with the lowest reference frequency.
   - FIFO eviction policy is based on First-In-First-Out (FIFO) algorithm which ensures that entry that has been in cache the longest will be evicted first. It is different from LruEvictionPolicy because it ignores the access order of entries.
   - Random eviction policy which randomly chooses entries to evict. This eviction policy could be used for debugging and benchmarking purposes.
   - We could add an expiration time for each item to remove items even if the memory limit was not achieved. That way we could free up items that had not been touched for a long time without waiting for memory to spike up.
3. When does eviction happen? We could have each non cached read do the clean up task or we could have a background thread do the job at discrete intervals. If we want to add capabilities like expiration time to free memory we would probably need to have a background thread in place. Also it might be more performant to have a background thread do the cleanup instead of having the job done by client processes.
4. The task here describes a cache where actual access to the data store is done externally by the cache clients through the use of an external function. There could two options here which are to allow the external function to be provided on a key based level through the get method or to use the same function for all the keys those providing the external function on the creation of the cache.
5. The cache described in the task sounds like a read cache which reads from a data store that is not managed by the cache. This implies that we could end up with stale data on the cache layer. This means that we will need some kind of pub/sub mechanism to notify the cache on data store data changes or to allow for an update period using expiration time for a simpler but less consistent system. The later might also hinder the cache performance as we will be evicting members that might not have changed.

6. In terms of QPS (queries per seconds)/latency we might see the most bang for the bucks if we ensure that we are caching frequently used data. Because we are running locally we have minimum latency in terms of returning cached data but the limited amount of local memory might cause more frequent calls to uncached data which will in term produce a higher latency.
7. Mutual exclusive lock should be acquired on the internal map updating to synchronize concurrent cache usage. When evicting members or when filling in missing cache items we will need to lock the internal map.
8. The data types that we can cache depends on the cache process model. If we are looking to provide an in process caching mechanism then all types could be cached. If we are looking to provide a local cache that can be accessed from different processes then we will have to only accept serializable data types.

Implementation Overview (in-proc):

1. CacheStoreProvider - an interface or an abstract class that will be required to implement and provide and instance which will get and set values to the backend store.
    - readFromStore(key) - A single method that returns the value from the data store.
2. CacheConfig - a class with definitions for cache behavior such as the eviction policy.
    - Eviction policy configuration.
    - Default TTL configuration.
3. Cache - a class that will be the entry point for caching interaction.
    - Cache class API:
        - ctor(cacheStoreProvider, cacheConfig)
            - cacheStoreProvider - an instance of a class implementing or extending CacheStoreProvider.
            - cacheConfig - an instance of class CacheConfig defining the behavior of the cache.
        - get(key) - a method that given a key returns the value based on cached value or if it is not available using the CacheStoreProvider. If value is not cached we should lock on the internal map in order to make sure that we only fetch the value from the store once.
        - get(key, cacheStoreProvider) - We might want to define different cache store providers to different keys. So having an overload or this signature as the only option could be a useful approach.
    - Cache class implementation:
        - The cache class should hold a single member such as the LRUMap which will be a map that tracks it's entries by order, frequency, expiration and provided a call to a cleanup method will evict the relevant members. The member type will be based on the configuration provided in the construction of the Cache class.
        - On creation of the cache class a low priority thread will be invoked that will periodically call the cleanup method of the Map implementing member. This will evict members based on the configuration provided when the Cache class was created.
        - The class should have a close method that terminates the background thread.

Implementation References (java):

1. LRUMap - A Map implementation with a fixed maximum size which removes the least recently used entry if an entry is added when full.