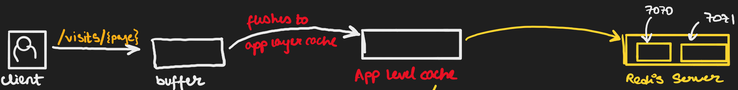
Assignment on Caching & Sharding

I. High Level Architecture



↳ stores key: atomic value pair for 5 seconds before flushing to Application level cache (caffeine cache)

↳ stores key: atomic value for 30 secs before flushing to global sharded cache.

II. Choice of Data Structures

→ buffer - HashMap<String, AtomicInteger>

→ localCache - caffeine cache<String, AtomicInteger>

→ redisShards - Map<String, RedisTemplate<String, Integer>>

↳ choice of atomicinteger buffer & local cache is done so that there is no loss of data during concurrent requests hitting the buffer.

→ using caffeine cache is preferred as that is the most used application layer cache in spring Boot.

III. Class structure, Design & overview.

Note: consistent hashing hasn't been perfectly implemented. The use case of addition of a server isn't properly done. so inorder we add a new virtual node, the migration of data to that shard isn't happening in this project.

→ ShardingAlgorithm

↳ Responsible for consistent hashing algorithm to distribute data.

→ addServer(String server): This method adds a server to the consistent hash ring by creating 2 replicas for each server and hashing them using MD5. It computes the hash of server name concatenated with replica index which ensures each replica has a unique identifier in the hash ring.

→ getServer(String key): Given a key, this method calculates its hash and finds the nearest server in the consistent hash ring to handle the request. This ensures that the data is consistently routed to the right server, even if one server is removed or added.

→ generateHash(String key): method generates MD5 hash of a given key & returns 32-bit integer representation of hash.

→ RedisShardManager

↳ this manages redis shards & manages connection b/w hash algorithm & allows us to get appropriate redis shard for a given key.

↳ 2 ports — 7070, 7071 → both added to the treeset of consistent hashing algorithm.

→ getShard(String key) — This method is the key connection point b/w the ShardingAlgorithm & Redis. It queries the ShardingAlgorithm to get the correct Redis shard based on the page no. hash and returns the corresponding RedisTemplate.

↳ RedisTemplate is used to interact with Redis for the specific shard.

→ getShardId(String key) - returns the shard id for a given key. It's useful for knowing which shard the key belongs to.

→ addShard(String shardId, RedisTemplate<String, Integer>) - Allows adding shards dynamically. This method basically adds the new shard to the shards map and to the consistent hash ring dynamically

↳ websiteVisitService class

→ This class handles the core logic for counting website visits, interacting with both caffeine cache and redis instances.

→ incrementVisit(pageNumber) - atomically increases the counter for a given page no. in the buffer, which holds all the updates in the form of batches for 5 seconds, before flushing it out to the local cache (app level cache)

↳ uses atomic integer to ensure thread safety during concurrent access.

→ getVisitResult(pageNumber) - this method retrieves the total number of visits for a given page:

↳ ① first checks the buffer
② then checks the local cache for any stored counts.
③ if there are any counts in the memory, it combines them with the buffer count & localcount & returns the result as in-memory.
④ If the total in memory count is zero, it fetches the count from the appropriate redis shard using RedisShardManager. This involves fetching the value from redis based on page no.

workflow

① Page visit increment

↳ when a page is visited → websiteVisitService.incrementVisit is called → increments counter in buffer.

② website count retrieval

↳ we first look into the buffer & local cache, if the value exists, we return it from memory. If it doesn't, we go to the specific shard to fetch the value.

→ This assignment implementation is more focused on write optimizations.