Solution

Functional Requirements

Before we consider Amazon and its scale, let's view this problem on a single node with a single user and the entire solution being done without a persistent database. There are two primary activities

1. Recording the fact that user viewed the item : write operation
2. Retrieve the history i.e. all viewed items in previous X days based on current timestamp with a max limit on the number of items shown (read operation). For instance,
    a. All items viewed in last 6 months with timestamp
    b. The max number of items allowed in display history is 100
    c. All items in the view must be unique : for instance, if the same item was viewed multiple times in the last 6 months, the count will only include the last time the item was viewed
    d. The history displayed is sorted in time-order : most recent going backwards to 6 months previous.

Single Node solution

Let's say the information about the viewing is stored in records of the following form.

```
typedef struct record {
        int timestamp_;  // at which the record was viewed
        int item_id_; // name of the item

        record(int item_id, int timestamp) :
           item_id_(item_id),  timestamp_(timestamp) {}

        bool operator<(const struct record &x) const {
            return (timestamp_ < x.timestamp_);
        }
```

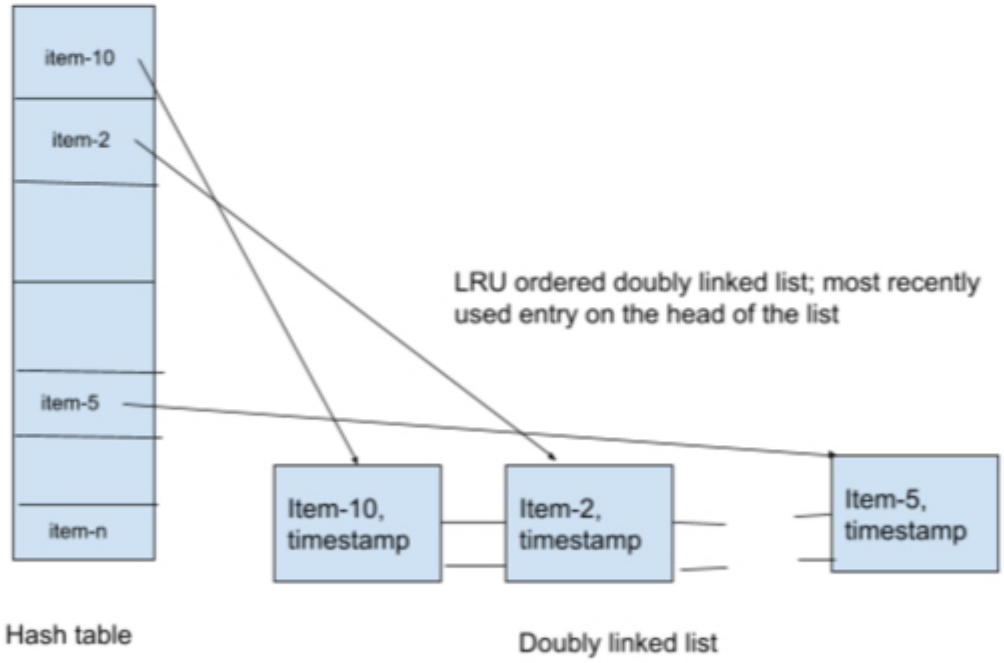
```
    } record_t;
```

The requirements map to a ring buffer per user of the items viewed by the user. However, the items in the ring buffer need to be unique. A simple ring-buffer does not provide the ability to maintain uniqueness across all the items in the buffer. The requirements for this question can thus be mapped to data structures used in the LRU cache leetcode question (Leetcode 146). However, the cache is user-specific.

For this case,

- We store the record for each viewed item in a doubly linked list ordered by timestamp with the most recently viewed item at the head of the list and the Least recently viewed item at the end
- The max size of the lru list is the max limit on the number of items to be shown. And this lru list contains only unique entries.
- The uniqueness of items on the list is maintained by using a hash table. The hash table contains the items to their position on the lru list.

The recording of the viewing activity thus translates to simply adding the record to the LRU list.




```
LRUCache lru_cache(MAX_CACHE_SIZE);

void viewed_item(int item_id, int timestamp) {
```

```
       lru_cache.put(item_id, timestamp)
  }

vector<record_t>  get_most_recently_viewed_items() {
     int least_timestamp =  std::chrono::system_clock::now() -  6-month; //
logical timestamp
     return lru_cache.get_all_keys_in range(least_timestamp);
}

  class LRUCache {
public:

    int size;
    int max_capacity;

    list<pair<int, int>> lru;  // (item_id, timestamp)
    map<int, pair<int, list<int>::iterator>> cache;  // item_id →
(timestamp, position in list)

    LRUCache(int capacity) {
        size = 0;
        max_capacity = capacity;
    }

    void move_to_head(int key, /* item_id */
                      int value,  /* timestamp */
                      list<int>::iterator& it) {
        lru.erase(it);
        lru.push_front(make_pair(key, value));
        cache[key] = make_pair(value, lru.begin());
    }

    void remove_from_tail() {
        auto lit = lru.back();
        int lrukey = lit.first;
        lru.pop_back();

        auto mit = cache.find(lrukey);
        assert(mit != cache.end());
        cache.erase(mit);
        size--;
    }
```

```
    vector<record_t> get_all_keys_in range(int least_timestamp) {
        vector<record_t> ret;

        for (auto& it : lru) {
              if (it.second.first < least_timestamp)
                 break;

              record_t r(it.first, it.second);
              ret.push_back(r);
        }

        return (ret);
    }

    void put(int key, int value) {
        auto mit = cache.find(key);
        if (mit == cache.end()) {

            if (size == max_capacity) {
                remove_from_tail();
            }

            lru.push_front(make_pair(key, value));
            cache[key] = make_pair(value, lru.begin());

            size++;
            return;
        }

        move_to_head(key, value, mit->second.second);
    }
};
```

Scale of the API

With millions of users the scale of the solution changes. Amazon has at least a few 50 million active users a day . For the API we will consider that

- the response latency should be under 200 millisecond
- Its ok for the API to be eventually consistent

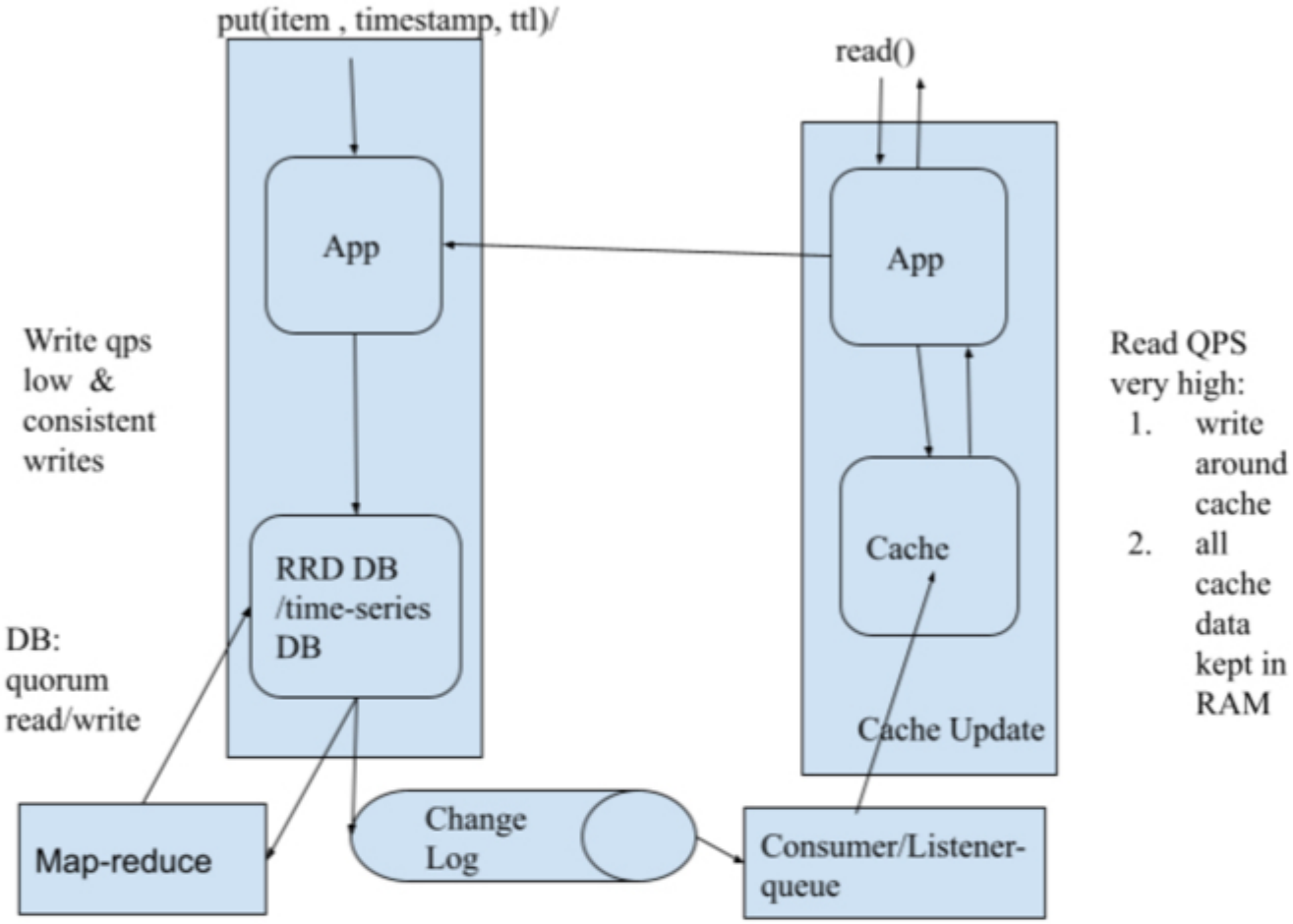
Basic components of the system :

1. Write microservice :
    a. Product/Item services produces an ItemViewed event on a queue
    b. A listener for the event queue will
        i. Store the ItemViewed event info in persistent DB :
            1. The DB could be a time-series DB or an RRD database
            2. The DB is used only for recovery purposes
            3. Or doing offline analysis for generating recommendations for users
        ii. Store the ItemViewed event info in a in-memory ring-buffer
            1. Sorted-sets for the event
2. The Reader part of the API will use the cache for reading the last viewed items
    a. This could be a sorted-set in memory

The ItemViewed event could contain the following information

```
{
      Timestamp,
      Item_id,
      User_id

}
```



Read (item, timestamp) pairs -> evaluate TTL->
issue delete in DB for expired keys

On the large-scale system,

1. Either we have a Map-Reduce job associated with deleting the items that have been viewed for the past 6 months.
2. Or, we have a separate time-partitioned table for the row-oriented time-series DB so that we do not need a Map-reduce job; a simple cron job can read the table and issue deletes for the expired viewed items .