Functional Requirements

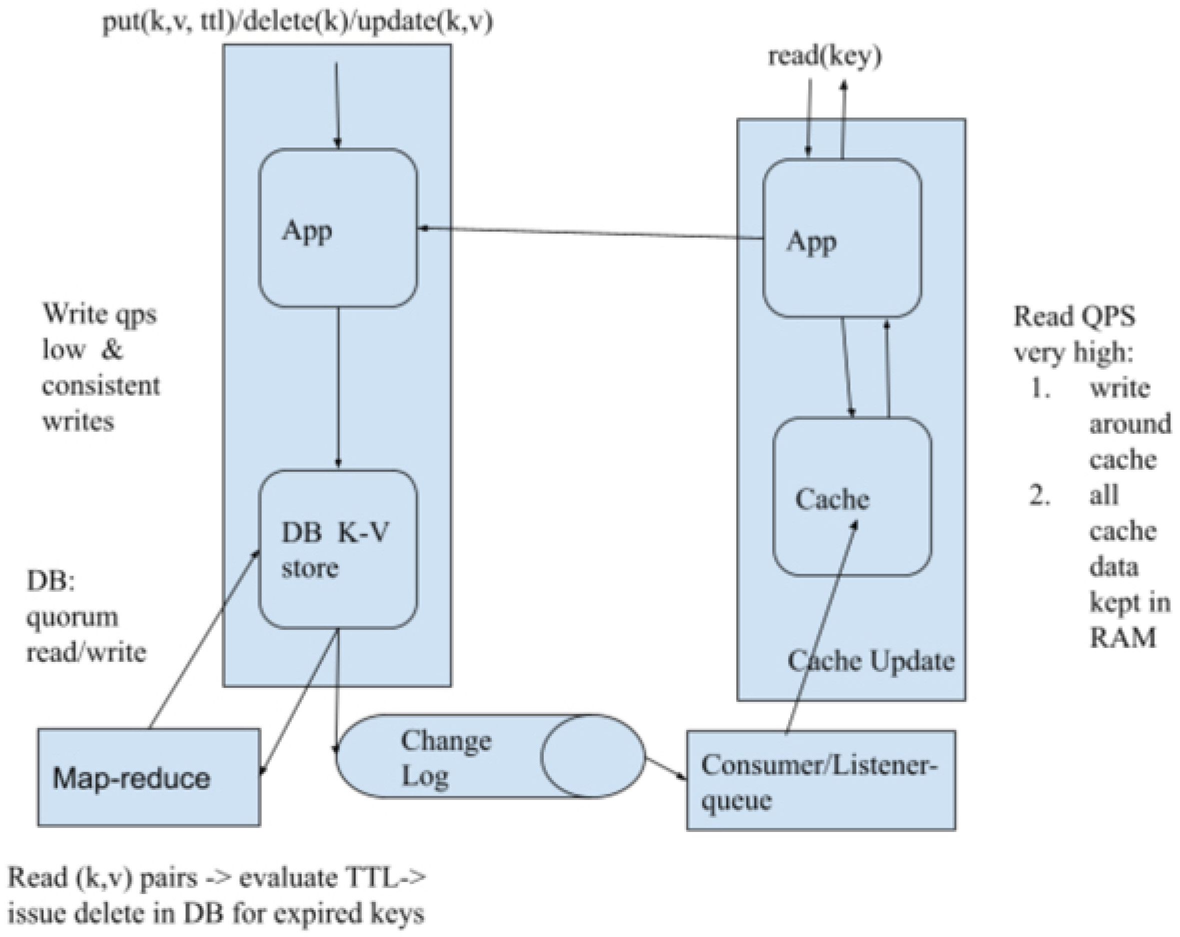
1. Store key/value (K,V) pairs
2. Unique keys
3. Keys :
    a. Primitive types: int, double
    b. OR, string , 1KB text
4. Values: complex objects
5. CRUD operations :
    a. GET (key)
    b. PUT(key, value) : ensures key does not already exist
    c. UPDATE(key, value) : ensures key exist
    d. Delete(key)
6. Optional Time-to-live(TTL) on each (K,V) pairs : TTL is independent of the usage policy

Capacity Requirements

1. 10 pow 9 (K,V) pairs
2. 2K writes/sec (QPS)
    a. Creates
    b. Updates
    c. Deletes
3. 100K reads/ sec (QPS)
4. Value size : <= 10KB

[Note: good to keep flexibility in requirements (not too rigid design) “add search based on value”]

Concurrent access to shared state in cache entries via a single reader-writer lock is non-performant on hot keys. Lock striping to reduce contention also does not provide much benefit because hot-keys dominate the lock contention by far. An alternate strategy could be to use deletion during updates thus avoiding locking. Hence the DB change log shard-ed by the consistently hashed key-space and replayed asynchronously into the cache by scheduling a replay activity is chosen in the design.


put(k,v, ttl)/delete(k)/update(k,v)
read(key)
App
App
Write qps low & consistent writes
Read QPS very high:
1. write around cache
2. all cache data kept in RAM
DB K-V store
Cache
DB: quorum read/write
Cache Update
Change Log
Map-reduce
Consumer/Listener-queue
Read (k,v) pairs -> evaluate TTL-> issue delete in DB for expired keys

The SLA of expired keys on TTL is 24 hours.

If it is a write-intensive system, it may make sense to have an in-memory updateable cache on both microservices.

Data Model

Primary Key	Value	TTL	Creation time	Update timestamp

Type 1 byte)	1 KB	10 KB	4 B	8B	8 B

1. No alternate indexes
2. 12 KB per entry

High-end server

1. 4 GHz

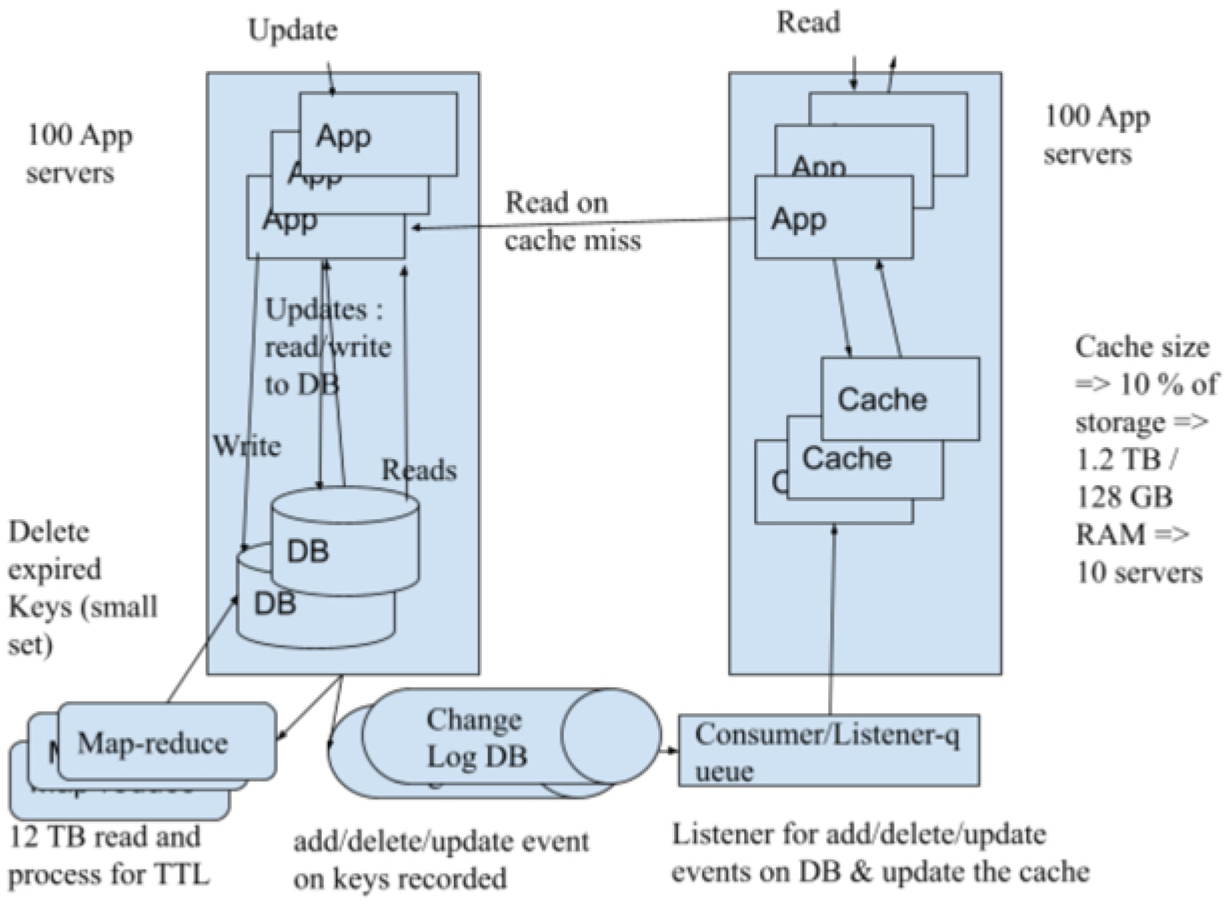
2. 8 core
3. 128 GB RAM
4. 2 TB of locally mounted HDD
5. 1K reads/sec
6. 200 writes/sec

Storage servers :

1. Total : 1 Billion keys X 12 KB => 12 TB data
2. From above, each high-end server has 2TB of locally mounted HDD
3. Sharding of keys => 12 TB/2TB => 6 shards , each shard with 1-2 TB of keys

App servers:

1. Put Microservice
    a. Needed 2K writes/sec
    b. From above, each high end server can do 200 writes/sec
    c. 2K/200 writes/sec => 100 app servers
2. Get Microservice
    a. Needed 100K reads/sec
    b. From above, each high end server supports 1K reads/sec
    c. 100K/1K => 100 app servers


Update
Read
100 App servers
App
App
App
Read on cache miss
App
App
100 App servers
Updates : read/write to DB
Write
Reads
DB
DB
Cache
Cache
Cache size => 10 % of storage => 1.2 TB / 128 GB RAM => 10 servers
Delete expired Keys (small set)
Map-reduce
Change Log DB
Consumer/Listener-queue
12 TB read and process for TTL
add/delete/update event on keys recorded
Listener for add/delete/update events on DB & update the cache

If a lot of keys have the same TTL, then a large number of delete keys may be issued by the MAP-Reduce jobs causing Hotspot. To avoid hotspots, the delete requests from Map-reduce jobs are rate-limited.

To avoid reading the entire DB to clear up keys whose TTL has expired, one strategy that can be applied is to introduce a separate table to hold the TTL timestamp information for keys. Using the table, instead of the Map-reduce job, a simple cron job can read only those keys whose TTL has expired from the table and issues deletes.

Update reads the key from the DB and updates the DB. Not all DB writes need to be cached. Consumer/listener-queue can have rules to decide which keys need to be updated in the cache.

Horizontal scaling of the key-space is done using consistent hashing.

Cache Eviction policy

LRU Cache (Leetcode 146)

```
class LRUCache {
public:

    int size;
    int max_capacity;

    list<int> lru;
    map<int, pair<int, list<int>::iterator>> cache;

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

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

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

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

    int get(int key) {
        auto mit = cache.find(key);
        if (mit == cache.end()) {
            return (-1);
        }

        int value = mit->second.first;
        move_to_head(key, value, mit->second.second);

        return (value);
    }

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

            size++;
            return;
        }

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

LFU Cache (LeetCode 460)

```
class LFUCache {
public:
    int cap;
    set<int> freq;
    unordered_map<int, list<int>> freq_key_map;  // freq -> key
    unordered_map<int, list<int>::iterator> key_location_map;  // key
-> freq_key_map iterator position
    unordered_map<int, pair<int, int>> key_value_freq_map;  // key ->
(value, freq)

    LFUCache(int capacity) {
        cap = capacity;
    }

    void incrementFrequency(int key) {
        unordered_map<int, pair<int, int>>::iterator it =
key_value_freq_map.find(key);
        if (key_value_freq_map.end() != it)
        {
            int prevfreq = it->second.second;
            int newfreq = ++it->second.second;
            list<int>::iterator loc = key_location_map[key];
            freq_key_map[newfreq].splice(freq_key_map[newfreq].begin(),
freq_key_map[prevfreq], loc);
            freq.insert(newfreq);
            if (freq_key_map[prevfreq].size()  == 0) {
                freq.erase(prevfreq);
                freq_key_map.erase(prevfreq);
            }
        }
    }

    void deleteLFU() {
        int least_used_freq = *freq.begin();
        freq.erase(least_used_freq);

        int key_to_be_removed = freq_key_map[least_used_freq].back();
        key_location_map.erase(key_to_be_removed);

        freq_key_map[least_used_freq].pop_back();
        if(freq_key_map[least_used_freq].size()  == 0) {
            freq_key_map.erase(least_used_freq);
        }
        key_value_freq_map.erase(key_to_be_removed);
    }

    void addNewKey(int key, int value) {
        key_value_freq_map[key] = pair(value, 1);
        freq.insert(1);
        freq_key_map[1].push_front(key);
        key_location_map[key] = freq_key_map[1].begin();
    }

    int get(int key) {
        int res = -1;
        unordered_map<int, pair<int, int>>::iterator it =
key_value_freq_map.find(key);
        if(key_value_freq_map.end() != it)
        {
            res = it->second.first;
            incrementFrequency(key);
        }
        return res;
    }

    void put(int key, int value) {
        if(cap == 0)
            return;
        unordered_map<int, pair<int, int>>::iterator it =
key_value_freq_map.find(key);
        if(key_value_freq_map.end() != it)
        {
            it->second.first = value;
            incrementFrequency(key);
        }
        else
        {
            if(key_value_freq_map.size() >= cap)
                deleteLFU();
            addNewKey(key, value);
        }
    }
};
```

Count Min Sketch

With billions of keys, both LRU and LFU cache eviction uses a lot of space (number of billions * size of keys * internal data structure in implementation as described above).

Count-min-sketch is a space-efficient data structure to decide a score for each key which can be used for the cache-eviction policy.

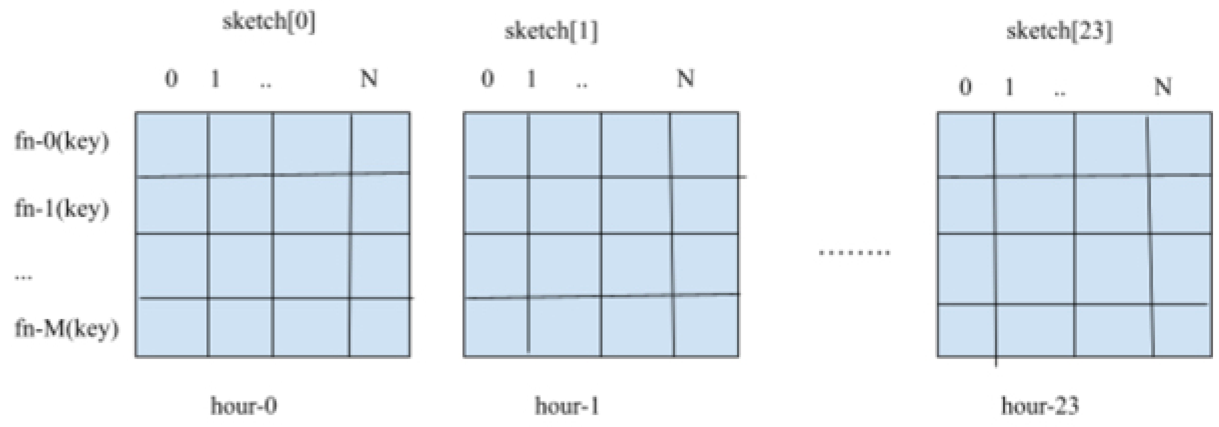
Let's say we keep keys in the cache for 24 hours. We have a table representing each hour in the past 24 hours named sketch[0] , sketch[1]...sketch[23]

Each sketch[N] table dimension

1. M rows : corresponds to M hash functions
2. N cols : a fixed size number of columns

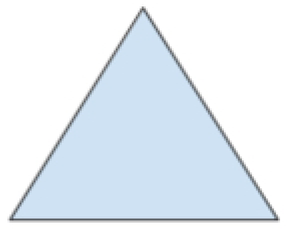
These dimensions decide the correctness and accuracy of determining the hot keys . For instance, a larger number of hash functions (M) means lesser chance of collision and thus better accuracy. Of course, this means increasing space usage a bit.

Space usage is 24 * M * N bytes .



ComputeTrendScore(key)

1. Step 1.
    a. fn-0(key) => hash-result-0
    b. fn-1(key) => hash-result-1
    c. fn-M(key) => hash-result-M
2. Step 2 : calculate score
    a. Minimum(sketch[0][hash-result-0], sketch[1][hash-result-1], ….sketch[n][hash-result-n])



Min-heap of cache-size of computed scores to decide which key to evict

Building a cache/key-value store from scratch