Single Node limited In-memory Solution:

Definition of the problem

A single node problem definition could be like the following:
https://leetcode.com/problems/design-hit-counter/

Design a hit counter which counts the number of hits received in the past 5 minutes (i.e., the past 300 seconds).

Your system should accept a `timestamp` parameter (in seconds granularity), and you may assume that calls are being made to the system in chronological order (i.e., `timestamp` is monotonically increasing). Several hits may arrive roughly at the same time.

Implement the `HitCounter` class:

- `HitCounter()` Initializes the object of the hit counter system.
- `void hit(int timestamp)` Records a hit that happened at `timestamp` (in seconds). Several hits may happen at the same `timestamp`.
- `int getHits(int timestamp)` Returns the number of hits in the past 5 minutes from `timestamp` (i.e., the past 300 seconds).

Ring Buffer Data Structure and Algorithm

To solve the above , the data structure to use would be a ring buffer with 60X N buckets (60 X N seconds in N minutes) , each bucket corresponding to a single second in the last N minutes.

```
class HitCounter {
public:

    vector<pair<int,int>> timestamp_to_hits;
    int num_of_buckets = 300; // 300 seconds in 5 mins

    /** Initialize your data structure here. */
    HitCounter() {
        // ring buffer; stores {timestamp, hits}
        timestamp_to_hits.resize(num_of_buckets, make_pair(INT_MAX, 0));
    }

    /** Record a hit.
        @param timestamp - The current timestamp (in seconds granularity).
*/
```

```
    void hit(int timestamp) {
        // the timetstamps come in monotonically increasing order
        int bucket = timestamp % num_of_buckets;

        if (timestamp_to_hits[bucket].first != timestamp) {
            timestamp_to_hits[bucket] = make_pair(timestamp, 1);
        } else {
            timestamp_to_hits[bucket].second++;
        }
    }

    /** Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity).
*/
    int getHits(int timestamp) {
        int count = 0;

        for (int i=0; i < 300; ++i) {
            if ((timestamp - timestamp_to_hits[i].first) < 300) {
                count += timestamp_to_hits[i].second;
            }
        }

        return count;
    }
};
```

High-end server and Counters

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

A single atomic uint64_t has a data hotspot problem.

Scale for

1. Storage: No
2. Throughput: No
3. Hotspot: Yes, say tens of thousands of searches per/sec, likes per sec and so on

4. API parallel: No
5. Availability: Yes
6. Geo-Distribution: Yes

When the QPS on the counter far exceeds the above, the counting would need to be distributed across multiple machines and to read the exact value of the counter the data would need to be accumulated. The delta refers to the increments in the counter value per time-stamped bucket.

```
INSERT INTO counter_table (counter_key, counter_value) VALUES (:key,
:delta)
ON DUPLICATE KEY UPDATE counter_value = counter_value + :delta;
```

This is quite synchronous since previous updates need to finish before the next update can go in. Batching requests to update the counter (thus an inherent delay) can solve the issue at the expense of being able to tolerate some temporary inaccuracies in the aggregated value and thus avoiding database overload. Real-time accurate counting is thus compromised.

Optimistic Locking

Assuming the DB layer does not have inherent support for distributed counters, one approach to maintaining/updating such counters is the standard optimistic locking. The counter is always associated with a version, all stale updates will be rejected by the DB.

```
CREATE TABLE counter (counter_key varchar PARTITION KEY, value
integer, version integer)

UPDATE counter SET value = :new_value, version = :expected_version +
1 WHERE counter_key = :counter_key ONLY IF version =
:expected_version

while true:
 counter = get_counter(key)
 if update_counter(key, counter.value + delta, counter.version):
   Break;
```

The problem with this approach is that if we use it without batching, we will issue all N concurrent requests while only one request will succeed leaving N-1 requests to be reissued.

Thus lots of retries and not a feasible solution at scale. And with network partition problems in distributed systems, this retry approach with versions (may get out of sync due to network partition) makes the solution infeasible.

Sharded Counters

With the Distributed systems one alternate approach is the sharded counter: every shard may maintain its own ring-buffer for the counter and replicate other shard's counters, and the counter requested from the client is simply the sum of all of those sharded counters.

```
CREATE TABLE virtual_counter (counter_key varchar PARTITION KEY,
node_id varchar CLUSTERING KEY, value integer)

counter_key    | shard_id      | value
---------------|---------------|-------
ad:1:views     | shard_1       | 1
ad:1:views     | shard_2       | 2
```

Every shard has its own space of the counter and even in the case of a network partition, it is safe to update the part of the counter. In case of any conflict on the shard, the Last Writer Wins works for that sharded region of the counter.

To know the exact counter value either the counter table is replicated across all the nodes or a designated aggregator queries all the shards and accumulates the value and returns.
Replication

Let's say there are a few data centers across the world for an organization. Say 7 data centers in the USA, two in Asia, one in Europe. One possible scheme could be the counter being partitioned for each data center.

Key	Value per data center
Counter	DC_USA_0 | DC_USA_1 |.. |DC_USA_7| DC_ASIA_0| DC_ASIA_1 | DC_EUROPE_0

Sharding the counter further within a data center can reduce the hotspot. For instance, 5K counter updates /sec shared across 10 data centers means 500 counter updates/sec on each

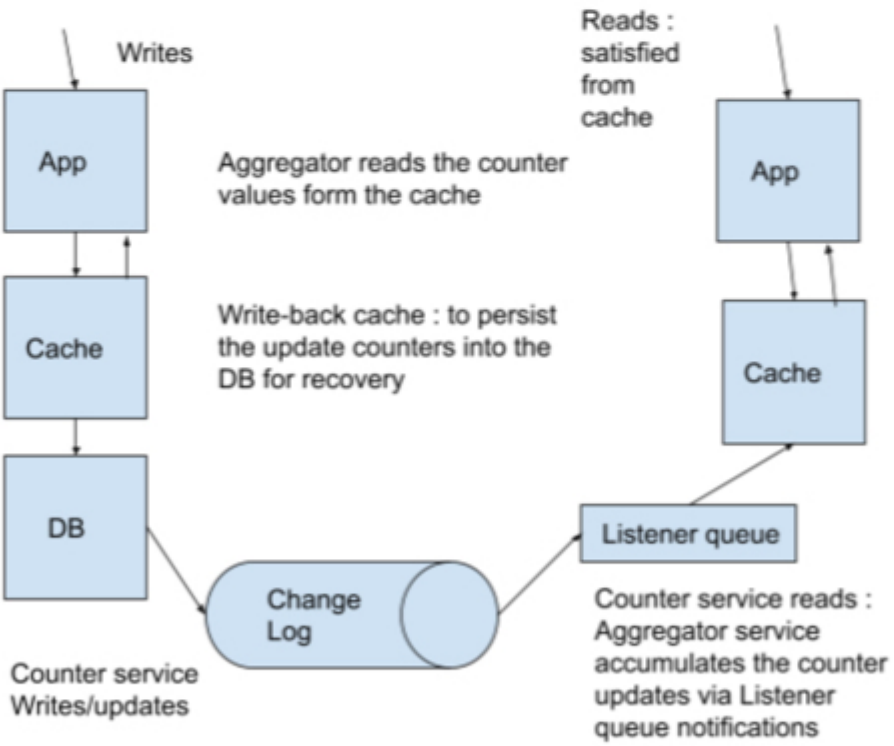
data center. Sharding further within the data center reduces this further so that each high-end server can handle the updates (max 200 Qps/sec) after the sharding.

Key	Value per data center shard-ed within the data center
Counter	DC_USA_0_VAL_0 | DC_USA_0_VAL_1| DC_USA_0_VAL_2| DC_USA_1_VAL_0|.. DC_USA_7_VAL_0| DC_ASIA_0_VAL_0| DC_ASIA_1_VAL_1| DC_EUROPE_0_VAL_0

1. The sharded counter approach is a columnar approach for the counter.
2. After updating the value in the cache, the {DC_COUNTRY_{#}_VAL_{#} } is replicated within a Data Center. Updates are faster within a data center.

There are two levels of aggregator:

1. The data center aggregator aggregates within a data center.
2. The Global aggregator aggregates across the data centers.



In the multi-datacenter setup, the accumulation of the counter value needs reads across all shards across all the data centers because the data is not replicated across all the data centers. Even if the strategy was to be replicated there is a replication delay involved.

CRDT: Conflict-free replicated data types

A different approach to distributed counters, in this case, is CRDT: conflict-free replicated data types.

CRDTs are objects which can be updated in distributed systems without expensive synchronization and they are guaranteed to converge eventually if all concurrent updates are commutative and if all updates are executed by each replica eventually.

There are two approaches to replication :

1. State-based replication: when a replica receives an update from a client, it updates its local state via a commutative merge function and then sends its full replica to another state. So each replica at some point is sending its full state to another replica.
2. Operation-based replication: since the full replica state may be huge, a different approach each replica can take is to apply the commutative merge function on receiving an update from a client and then sending only the update delta to the other replicas instead of the full state.

For instance, if the hit counter is depicted as an integer vector say [3, 5] on a replica and then it sends the state to another replica [4,2] the latter replica wil merge the values into [4, 5]. If the final state is always defined as it should be monotonically increasing i.e. apply the maximum value at any index i for the integer vector, the state will eventually converge at all replicas.

CRDT's need a global broadcast and are eventually consistent. Using a CRDT DB (Redis, Riak, CosmosDB) might be an option to store the hit counter but the high QPS requirements may not be satisfied by the global broadcast requirements of the CRDTs and staleness.

Distributed Time-sensitive Rolling Counter

To count the number, each shard works independently to count its own users from the past minute in the case of a sharded counter. When we request the global number, we just need to add all sharded counters together in the aggregator.