Overview

Social media platforms like reddit shows the number of users currently viewing a post. This is done in Realtime using distributed counter. It's value changes depending on when the user enters or exist the website.

Clarification to be done with the interviewer

- What is the purpose of distributed counter.
  E.g in this case to show number of active users using a webpage.
- Is the webpage public facing
  Yes.
- What is maximum number of users that uses the platform.
  1bliilon.
- How many avg concurrent users viewing the webpage
  100million
- What are max concurrent users
  300milliom
- What is the read to write ratio
  10:1 r:w
- Is the distributed counter strongly consistent
  No
- Is the distributed counter accurate
  Yes

Functional Requirements

- Platform should show the currently active users in real time
- In case the users exit, the counter should be decremented

Non-Functional Requirements

- Eventual Consistent
- Highly available
- Low latency
- Scalable
- Fault tolerant
- Accurate

API

- REST
- Communication Protocol: WebSocket / Server Sent Events

a. GET
/{webpage}/counter

The client executes the GET request to get the value of distributed counter.

Response: 200 Ok

```
{
"counter": 200,
"updated_at": "2030-10-10T12:11:42Z"
}
```

b. PUT
/{webpage}/counter

```
{
"action": “increment”
}
```

The client executes a PUT request to update a counter

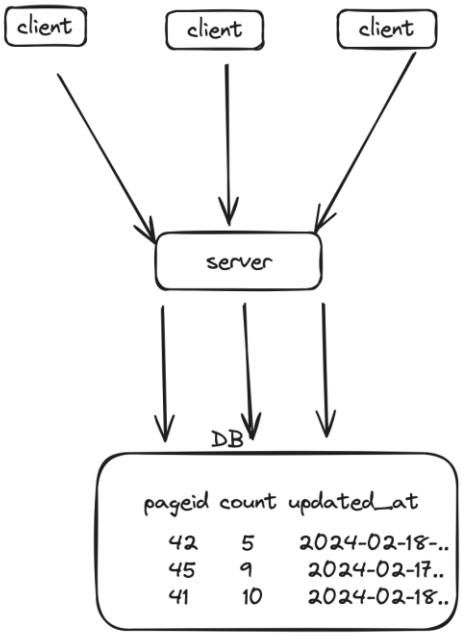
Distributed Counter High Level Design

Relational Database

Schema

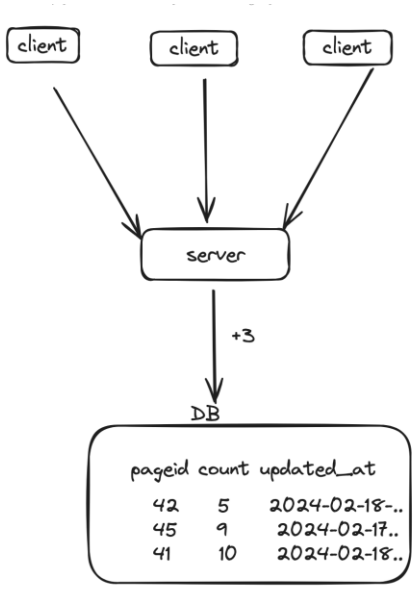
Table is normalized to reduce the data redundancy and save the storage cost.

Trivial Implementation



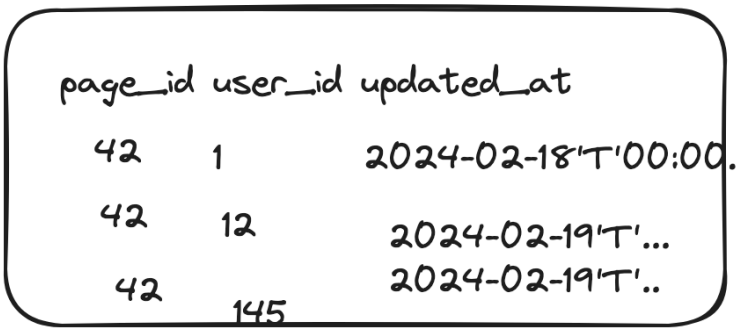
- Low traffic website can use a single instance database.
- Implementation should aquire lock to increment a counter corresponding to the particular webpage.
- Read write lock can be used to increase the performance of read.
- High concurrent writes will not be scaled with above approach.

Optimization to trivial implementations



- Server can aggregate the count in his local memory and sent the periodic updates to database.
- Concurrency still needs to be taken care of.
- Will reduce the write load on the database.
- Limitation being approach is not durable in case the server crashes as well as counter will be inaccurate because of delay in updating of the database.
- Still not scalable for high concurrent writes.

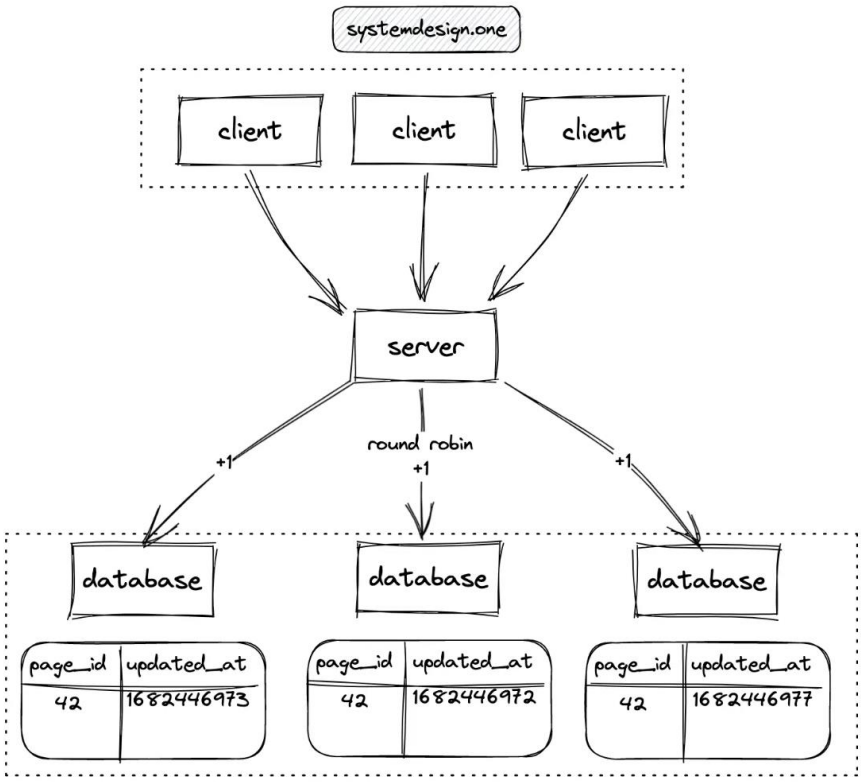
How to scale high concurrent writes



- Instead of updating the count table for a particular webpage, store them as separate record in the table.
- To get the number of active users currently active on the webpage, do
  select distinct(user_id) from table where page_id=42 and updated_at > now-1.
- This approach supports high concurrent writes because there is no need of lock.
- Limitation being reads will be very slow because of full table scan.
- Storage cost will be very high.
- One optimization can be to create an secondary index on updated_at at the expense of slower writes.

How to take care of storage

- Sharding.
- Each shard will now store its fair share of the data.
- Each shard will have the follower's node to evenly distribute the read load as well as increasing the high availability.
- Partition key:
    - pageid -> Can create the hotspot in case a particular webpage become too popular.
    - User_id : Updates in the counter for a particular webpage will be applied to lot of shards.
    - The value of the count will be fetched by parallel querying all the shards and aggregating the result.
    - Read will be slow because of querying all the shards.



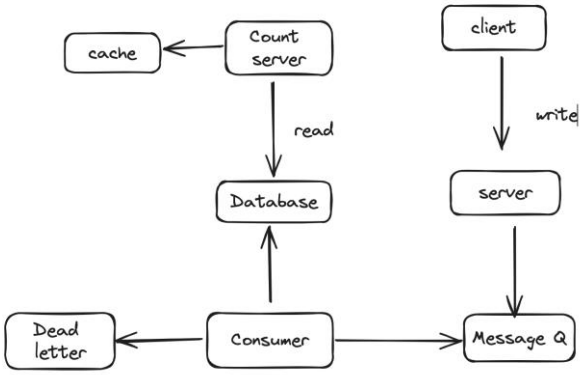
Limitation of using the relational database

- Distributed counter design should support high read and write load along with high concurrency.
- High write load can cause write congestion and poor latency (because of single leader).
- Transaction support is not needed in this case.
- Don't use relational database to build highly scalable and concurrent distributed counter.

NoSQL Database

- LSM based NoSQL are optimized for high write throughput which is requirement for distributed counter although at the cost of slower reads.
- Frequent disk access can also be bottleneck for high read loads.
- To improve reads, we might need to add additional cache layer.
- Downside of the approach will be counter consistency will not be guaranteed.
- Operational complexity will also be high with this approach.

Message Queue

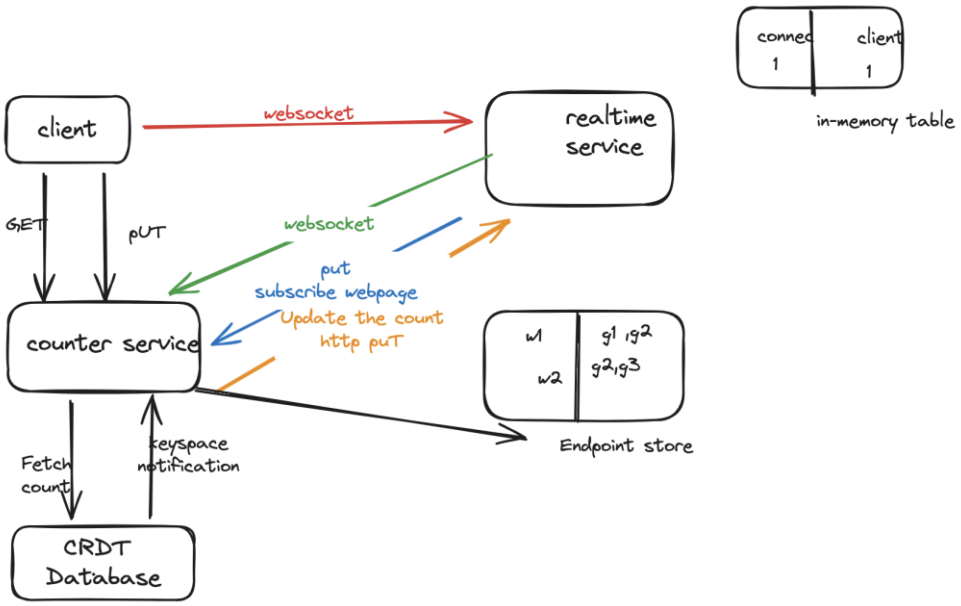


- Extension of above design. Decouple the write to further scale the write throughput.
- Counter service will serve the read request.
- Read request will have the same limitation as above design.
- Advantages being highly scalable for writes.
- Disadvantages being -> operationally complex, increased storage (message Q). Along with guaranteeing the ordering and exactly once delivery on the message Q to make the sure the counter is accurate.

Redis

- Redis offers extremely high support and throughout with its in-memory store.
- Redis can be configured in HA mode using Redis cluster.
- Also support CRDT (Conflict replicated data types -> uses mathematical rules to resolve the conflict) data structures.
- Persistence can be configured for durability.
- Which Redis data structure to user?
    - Simple key which will be incremented when a webpage is visited by a user and vice versa.
      Pro's -> Memory efficient and insertion and deletion is O (1)
      Con's -> Because INR and DECR are not idempotent which can lead to incorrect counter. E.g. in case of network issue, we will not know whether the INR ore DECR has been successfully executed on Redis or not. Retrying will cause the issue.

    - Hyper loglog -> uses probabilistic algorithm to count the cardinality (distinct element) of the set.
      Pro's -> Memory efficient.
      Con's -> Inaccurate. Deletion not supported.
    - Set -> Store and delete the user'd id. Cardinality of set will be the answer.
      Pro's -> Accurate. Idempotent. O (1) size. O(logn) insetion.
      Con's -> Memory
    - Verdict -> Go with set.
- We can use CRDT datatype (SET). This is offered by Redis enterprise.



- Client
    - Initiate a GET request to counter service to fetch the current count corresponding to webpage
    - PUT request to update the counter by 1 for this webpage
    - Establish a WebSocket connection to Realtime service for continuous update of realtime counter.
- Realtime service
    - Maintains an in-memory subscription table having connection id to client information along with which client is interested in what webpage.
    - In case of any update, will fanout the events to all the clients for that webpage.
    - Also establish a WebSocket connection for sending the status of client connection information to counter service. This is needed so that counter service can update the crdt database in case client disconnect a particular webpage.
    - Also sent a subscribe HTTP request to counter service to update the endpoint store that I'm interested in this webpage updated.

- Counter service
    - Receive the key spaces notification that this particular key has been updated along with the count.
    - Queries the persistent endpoint store to get the list of gateway store to fanout this information to.
    - Sent the information over HTTP put request.
    - Update the crdt database in case the Realtime service notifies the client has been disconnected.

Across different Datacentre

- The above design can be extended across different datacentre with CRDT database needs to be replicated.

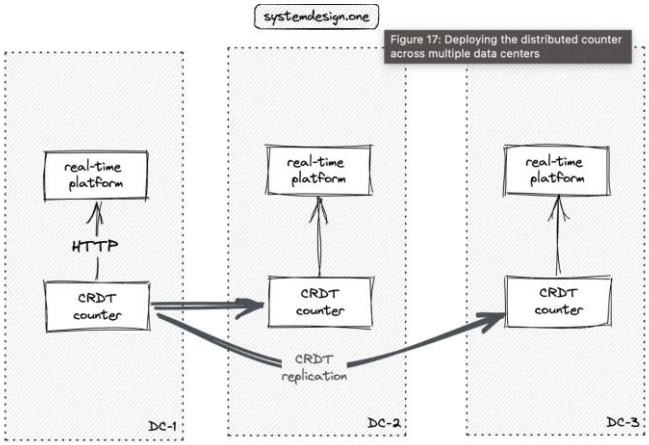


Figure 17: Deploying the distributed counter across multiple data centers

This design is inspired from LINK