How would users be able to saerch for words by keyword?

The naive solution to this problem is to keep all of the posts in a relational database and use a query like SELECT \* FROM Posts LIKE '%$keyword%';. This would technically return the correct results for a given query!

The above is a very bad solutino because it’s slow ehre

Using inverted index in elastic search optoin

1. An inverted index is a data structure used to store mapping from content, such as words or numbers, to its locations in a database, or in this case, documents.

Suppose we want to find all the words that contain the word “great”, how do we do that?

A screenshot of a computer screen

Description automatically generated

The above is the solution you are looking for here

1.

Case study here:

How would Users be able to get search results sorted by recency or like count?

Using multiple indexes here

A different approach would be to have two separate indexes: one sorted by the creation time and one sorted by the like count (I'll refer to these as the creation index and likes index going forward).

One for the creation index and like index here, these 2 are very important as said.

Using our Redis-based approach from earlier, we can have separate keys depending on whether we're sorting by Likes or Creation date.

For the creation index keys, we can use a standard Redis list. We're always going to appending to this list and our queries will only be taking from the last elements.

For the likes index, each key can use a [**Redis sorted set**](https://www.hellointerview.com/learn/system-design/deep-dives/redis#redis-for-leaderboards). The sorted set allows us to keep a list of items ordered by a score in the same way that a priority queue or sorted list might work, with the same time complexity of insertions and queries.

When a new post is created, we'll add it to both indexes for every keyword it contains. When a like event happens, we'll update the score in our sorted set for the likes index.

1) How can we handle the large volume of requests from users?

1. Using redis

2. Using CDN cache here

Our in-memory reverse-index based system is quite fast, but we're going to be handling a lot of traffic. We had some convenient requirements earlier that might make our job even easier. Two requirements in particular:

1. We do not have personalization, so if you and I are searching for the same thing with the same parameters, we should get the same results!
2. We have up to 1 minute before a post needs to appear in the search results.

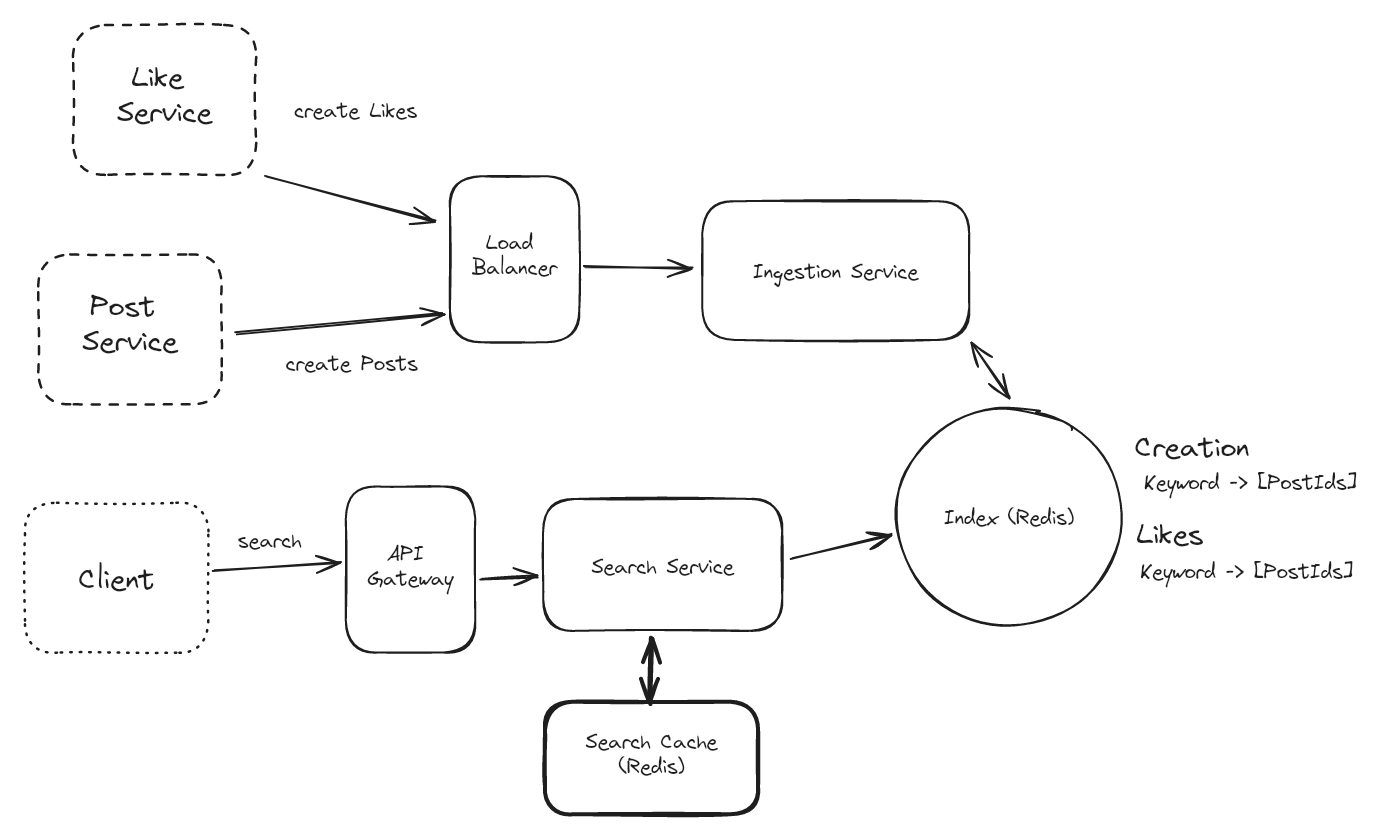
Caching sticks out here as the obvious tool for us! Any time we can tolerate stale data and we have duplicate requests coming through, we should consider whether caching is appropriate.

**Good Solution: Use a distributed cache alongside our search service**

Approach

One option for us is to add a distributed cache alongside our search service. This cache would be responsible for storing the most recent results for a given search query. When a search is performed, the service would first check the cache to see if the results are available. If they are, the service would return the results from the cache. If they are not, the service would perform a full search and store the results in the cache for future requests.

We'll want to put an eviction policy on our cache to ensure stale results don't stick around. Since we have up to 1 minute SLA on new posts, we can institute a TTL of < 1 minute on our cache. This will guarantee we're never serving results that might not contain newly created posts.



Search Cache

**Great Solution: Use a CDN to cache at the edge**

**Approach**

In addition to the Redis search cache, we can also utilize edge caching via a Content Delivery Network (CDN) like Cloudflare or AWS Cloudfront. Most CDNs operate like a big set of geographically-spread HTTP caches. You configure an "origin" or target for the cache and configure DNS to route through the CDN. If the cache has the item the user is looking for, it can return it faster than almost any alternative option: most CDNs have locations very close to most users. If it doesn't hit the cache, the CDN will proxy the request back to your servers to handle.

Using the CDN here is simple: on the response to our /search endpoint, we can add cache-control headers which tell our CDN when and for how long to cache a result. When a user tries to hit our search service, they'll first go through a geographically local CDN host. In the case of a cache hit, these users will get results in 10s of milliseconds vs the 100s which might have been required if they had to go through our API gateway, to the search service, via a call to the search cache, and back through. If it's not in the cache, we get a request as usual.

How do we handle the large volume of writes for post?

1. Using kafka is step 1, fan out the writes to muliiptle instances here

What about for the like event?

1. One approach we can take is to batch the writes for likes. Instead of writing every like update to our indexes, we can batch likes for a given postId over a period (like 30 seconds). Then, instead of needing to write 500 times for a particularly viral post, we can make 1 update with an increment of 500.

To do this we’ll need a secondary "batcher" service which accepts like events and aggregates them over a fixed window (maybe 30 seconds) before writing them back to Kafka to be consumed by the ingestion service.

How to optimize the storage with our current system?

Like how do we deal with too many indexes?

Use batch job to remove them when they are no longer needed here.

we can run a batch job to move rarely used keywords to a less frequently accessed but ultimately cheaper storage. One way to do this is to move these keyword indexes to cold, blob storage like S3 or R2.

On a regular basis we'll determine which keywords were rarely (or not at all) accessed in the past month. We'll move these indexes from our in-memory Redis instance to a blob in our blob storage. When the index needs to be queried, we'll first try to query Redis. If we don't get our keyword there, we can query the index from our blob storage with a small latency penalty.