Ask me anything about this topic!

Common Problems

Design Bit.ly

Scaling Reads

By Evan King · Published Jul 24, 2024 · easy · Asked at: +6

Watch Video Walkthrough

Watch the author walk through the problem step-by-step

Watch Now

Understanding the Problem

🔗**What is Bit.ly?**
Bit.ly is a URL shortening service that converts long URLs into shorter, manageable links. It also provides analytics for the shortened URLs.

Designing a URL shortener is a very common beginner system design interview question. Whereas in many of the other breakdowns on Hello Interview we focus on depth, for this one, I'm going to target a more junior audience. If you're new to system design, this is a great question to start with! I'll try my best to slow down and teach concepts that are otherwise taken for granted in other breakdowns.

Functional Requirements

The first thing you'll want to do when starting a system design interview is to get a clear understanding of the requirements of the system. Functional requirements are the features that the system must have to satisfy the needs of the user.

In some interviews, the interviewer will provide you with the core functional requirements upfront. In other cases, you'll need to determine these requirements yourself. If you're familiar with the product, this task should be relatively straightforward. However, if you're not, it's advisable to ask your interviewer some clarifying questions to gain a better understanding of the system.

The most important thing is that you zero in on the top 3-4 features of the system and don't get distracted by the bells and whistles.

We'll concentrate on the following set of functional requirements:

Core Requirements

1. Users should be able to submit a long URL and receive a shortened version.
   - Optionally, users should be able to specify a custom alias for their shortened URL (ie. "**www.short.ly/my-custom-alias**")
   - Optionally, users should be able to specify an expiration date for their shortened URL.
2. Users should be able to access the original URL by using the shortened URL.

Below the line (out of scope):

- User authentication and account management.
- Analytics on link clicks (e.g., click counts, geographic data).

These features are considered "below the line" because they add complexity to the system without being core to the basic functionality of a URL shortener. In a real interview, you might discuss these with your interviewer to determine if they should be included in your design.

Non-Functional Requirements

Next up, you'll want to outline the core non-functional requirements of the system. Non-functional requirements refer to specifications about how a system operates, rather than what tasks it performs. These requirements are critical as they define system attributes like scalability, latency, security, and availability, and are often framed as specific benchmarks—such as a system's ability to handle 100 million daily active users or respond to queries within 200 milliseconds.

Core Requirements

1. The system should ensure uniqueness for the short codes (no two long URLs can map to the same short URL)

2. The redirection should occur with minimal delay (< 100ms)
3. The system should be reliable and available 99.99% of the time (availability > consistency)
4. The system should scale to support 1B shortened URLs and 100M DAU

Below the line (out of scope):

- Data consistency in real-time analytics.
- Advanced security features like spam detection and malicious URL filtering.

> An important consideration in this system is the significant imbalance between read and write operations. The read-to-write ratio is heavily skewed towards reads, as users frequently access shortened URLs, while the creation of new short URLs is comparatively rare. For instance, we might see 1000 clicks (reads) for every 1 new short URL created (write). This asymmetry will significantly impact our system design, particularly in areas such as caching strategies, database choice, and overall architecture.

Here is what you might write on the whiteboard:

Bit.ly Non-Functional Requirements

The Set Up

Defining the Core Entities

We recommend that you start with a broad overview of the primary entities. At this stage, it is not necessary to know every specific column or detail. We will focus on the intricacies, such as columns and fields, later when we have a clearer grasp. Initially, establishing these key entities will guide our thought process and lay a solid foundation as we progress towards defining the API.

> Just make sure that you let your interviewer know your plan so you're on the same page. I'll often explain that I'm going to start with just a simple list, but as we get to the high-level design, I'll document the data model more thoroughly.

In a URL shortener, the core entities are very straightforward:

1. **Original URL**: The original long URL that the user wants to shorten.
2. **Short URL**: The shortened URL that the user receives and can share.
3. **User**: Represents the user who created the shortened URL.

In the actual interview, this can be as simple as a short list like this. Just make sure you talk through the entities with your interviewer to ensure you are on the same page.

Bit.ly Entities

The API

The next step in the delivery framework is to define the APIs of the system. This sets up a contract between the client and the server, and it's the first point of reference for the high-level design.

Your goal is to simply go one-by-one through the core requirements and define the APIs that are necessary to satisfy them. Usually, these map 1:1 to the functional requirements, but there are times when multiple endpoints are needed to satisfy an individual functional requirement.

9/10 you'll use a REST API and focus on choosing the right HTTP method or verb to use.

- **POST**: Create a new resource
- **GET**: Read an existing resource
- **PUT**: Update an existing resource

- **DELETE**: Delete an existing resource

To shorten a URL, we'll need a POST endpoint that takes in the long URL and optionally a custom alias and expiration date, and returns the shortened URL. We use post here because we are creating a new entry in our database mapping the long url to the newly created short url.

```
// Shorten a URL
POST /urls
{
  "long_url": "https://www.example.com/some/very/long/url",
  "custom_alias": "optional_custom_alias",
  "expiration_date": "optional_expiration_date"
}
->
{
  "short_url": "http://short.ly/abc123"
}
```

For redirection, we'll need a GET endpoint that takes in the short code and redirects the user to the original long URL. GET is the right verb here because we are reading the existing long url from our database based on the short code.

```
// Redirect to Original URL
GET /{short_code}
-> HTTP 302 Redirect to the original long URL
```

We'll talk more about which HTTP status codes to use during the high-level design.

High-Level Design

We'll start our design by going one-by-one through our functional requirements and designing a single system to satisfy them. Once we have this in place, we'll layer on depth via our deep dives.

1) Users should be able to submit a long URL and receive a shortened version

The first thing we need to consider when designing this system is how we're going to generate a short url. Users are going to come to us with long urls and expect us to shrink them down to a manageable size.

We'll outline the core components necessary to make this happen at a high-level.

Create a short url

1. **Client**: Users interact with the system through a web or mobile application.
2. **Primary Server**: The primary server receives requests from the client and handles all business logic like short url creation and validation.
3. **Database**: Stores the mapping of short codes to long urls, as well as user-generated aliases and expiration dates.

When a user submits a long url, the client sends a POST request to `/urls` with the long url, custom alias, and expiration date. Then:

1. The Primary Server receives the request and validates the long URL. This is to ensure that the URL is valid (there's no point in shortening an invalid URL) and that it doesn't already exist in our system (we don't want collisions).
    - To validate that the URL is valid, we can use popular open-source libraries like is-url or write our own simple validation.
    - To check if the URL already exists in our system, we can query our database to see if the long URL is already present.
2. If the URL is valid and doesn't already exist, we can proceed to generate a short URL unless
    - For now, we'll abstract this away as some magic function that takes in the long URL and returns a short URL. We'll dive deep into how to generate short URLs in the next section.
    - If the user has specified a custom alias, we can use that as the short code (after validating that it doesn't already exist).

3. Once we have the short URL, we can proceed to insert it into our database, storing the short code (or custom alias), long URL, and expiration date.
4. Finally, we can return the short URL to the client.

2) Users should be able to access the original URL by using the shortened URL

Now our short URL is live and users can access the original URL by using the shortened URL. Importantly, this shortened URL exists at a domain that we own! For example, if our site is located at `short.ly` , then our short urls look like `short.ly/abc123` and all requests to that short url go to our Primary Server.

Redirect to original url

When a user accesses a shortened URL, the following process occurs:

1. The user's browser sends a GET request to our server with the short code (e.g., `GET /abc123`).
2. Our Primary Server receives this request and looks up the short code (`abc123`) in the database.
3. If the short code is found and hasn't expired (by comparing the current date to the expiration date in the database), the server retrieves the corresponding long URL.
4. The server then sends an HTTP redirect response to the user's browser, instructing it to navigate to the original long URL.

There are two main types of HTTP redirects that we could use for this purpose:

1. **301 (Permanent Redirect)**: This indicates that the resource has been permanently moved to the target URL. Browsers typically cache this response, meaning subsequent requests for the same short URL might go directly to the long URL, bypassing our server.

The response back to the client looks like this:

```
HTTP/1.1 301 Moved Permanently
Location: https://www.original-long-url.com
```

1. **302 (Temporary Redirect)**: This suggests that the resource is temporarily located at a different URL. Browsers do not cache this response, ensuring that future requests for the short URL will always go through our server first.

The response back to the client looks like this:

```
HTTP/1.1 302 Found
Location: https://www.original-long-url.com
```

In either case, the user's browser (the client) will automatically follow the redirect to the original long URL and users will never even know that a redirect happened.

For a URL shortener, a 302 redirect is often preferred because:

- It gives us more control over the redirection process, allowing us to update or expire links as needed.
- It prevents browsers from caching the redirect, which could cause issues if we need to change or delete the short URL in the future.
- It allows us to track click statistics for each short URL (even though this is out of scope for this design).

Potential Deep Dives

At this point, we have a basic, functioning system that satisfies the functional requirements. However, there are a number of areas we could dive deeper into to reduce the likelihood of collision, support scalability, and improve performance. We can now look back at our non-functional requirements and see which ones still need to be satisfied or improved upon.

1) How can we ensure short urls are unique?

In our high-level design, we abstracted away the details of how we generate a short url but now it's time to get into the nitty-gritty! There are a handful of constraints we need to keep in mind as we design:

1. We need to ensure that the short codes are unique.
2. We want the short codes to be as short as possible (it is a url shortener afterall).
3. We want to ensure codes are efficiently generated.

Let's weigh a few options and consider their pros and cons.

Bad Solution: Long Url Prefix

Great Solution: Hash Function

Great Solution: Unique Counter with Base62 Encoding

2) How can we ensure that redirects are fast?

When dealing with a large database of shortened URLs, finding the right match quickly becomes crucial for a smooth user experience. Without any optimization, our system would need to check every single pair of short and original URLs in the database to find the one we're looking for. This process, known as a "full table scan," can be incredibly slow, especially as the number of URLs grows into the millions or billions.

Pattern: Scaling Reads

URL shorteners showcase the extreme read-to-write ratio that makes **scaling reads** critical. With potentially millions of clicks per shortened URL, aggressive caching strategies become essential.

Learn This Pattern

Good Solution: Add an Index

Great Solution: Implementing an In-Memory Cache (e.g., Redis)

Great Solution: Leveraging Content Delivery Networks (CDNs) and Edge Computing

3) How can we scale to support 1B shortened urls and 100M DAU?

We've done much of the hard work to scale already! We introduced a caching layer which will help with read scalability, now lets talk a bit about scaling writes.

We'll start by looking at the size of our database.

Each row in our database consists of a short code (~8 bytes), long URL (~100 bytes), creationTime (~8 bytes), optional custom alias (~100 bytes), and expiration date (~8 bytes). This totals to ~200 bytes per row. We can round up to 500 bytes to account for any additional metadata like the creator id, analytics id, etc.

If we store 1B mappings, we're looking at `500 bytes * 1B rows = 500GB` of data. The reality is, this is well within the capabilities of modern SSDs. Given the number of urls on the internet is our maximum bound, we can expect it to grow but only modestly. If we were to hit a hardware limit, we could always shard our data across multiple servers but a single Postgres instance, for example, should do for now.

So what database technology should we use?

The truth is: most will work here. We offloaded the heavy read throughput to a cache and write throughput is pretty low. We could estimate that maybe 100k new urls are created per day. 100k new rows per day is ~1 row per second. So any reasonable database technology should do (ie. Postgres, MySQL, DynamoDB, etc). In your interview, you can just pick whichever you have the most experience with! If you don't have any hands on experience, go with Postgres.

But what if the DB goes down?

It's a valid question, and one always worth considering in your interview. We could use a few different strategies to ensure high availability.

1. **Database Replication**: By using a database like Postgres that supports replication, we can create multiple identical copies of our database on different servers. If one

server goes down, we can redirect to another. This adds complexity to our system design as we now need to ensure that our Primary Server can interact with any replica without any issues. This can be tricky to get right and adds operational overhead.

2. **Database Backup**: We could also implement a backup system that periodically takes a snapshot of our database and stores it in a separate location. This adds complexity to our system design as we now need to ensure that our Primary Server can interact with the backup without any issues. This can be tricky to get right and adds operational overhead.

Now, let's point our attention to the Primary Server.

Coming back to our initial observation that reads are much more frequent than writes, we can scale our Primary Server by separating the read and write operations. This introduces a microservice architecture where the Read Service handles redirects while the Write service handles the creation of new short urls. This separation allows us to scale each service independently based on their specific demands.

Now, we can horizontally scale both the Read Service and the Write Service to handle increased load. Horizontal scaling is the process of adding more instances of a service to distribute the load across multiple servers. This can help us handle a large number of requests per second without increasing the load on a single server. When a new request comes in, it is randomly routed to one of the instances of the service.

But what about our counter?

Horizontally scaling our write service introduces a significant issue! For our short code generation to remain globally unique, we need a single source of truth for the counter. This counter needs to be accessible to all instances of the Write Service so that they can all agree on the next value.

We could solve this by using a centralized Redis instance to store the counter. This Redis instance can be used to store the counter and any other metadata that needs to be shared across all instances of the Write Service. Redis is single-threaded and is very fast for this use case. It also supports atomic increment operations which allows us to increment the counter without any issues. Now, when a user requests to shorten a url, the Write Service will get the without any issues. Now, when a user requests to shorten a url, the Write Service will get the next counter value from the Redis instance, compute the short code, and store the mapping in the database.

Final Design

But should we be concerned about the overhead of an additional network request for each new write request?

The reality is, this is probably not a big deal. Network requests are fast! In practice, the overhead of an additional network request is negligible compared to the time it takes to perform other operations in the system. That said, we could always use a technique called "counter batching" to reduce the number of network requests. Here's how it works:

1. Each Write Service instance requests a batch of counter values from the Redis instance (e.g., 1000 values at a time).
2. The Redis instance atomically increments the counter by 1000 and returns the start of the batch.
3. The Write Service instance can then use these 1000 values locally without needing to contact Redis for each new URL.
4. When the batch is exhausted, the Write Service requests a new batch.

This approach reduces the load on Redis while still maintaining uniqueness across all instances. It also improves performance by reducing network calls for counter values.

To ensure high availability of our counter service, we can use Redis's built-in replication features. Redis Enterprise, for example, provides automatic failover and cross-region replication. For additional durability, we can periodically persist the counter value to a more durable storage system.

Test Your Knowledge

Take a quick 15 question quiz to test what you've learned.

Start Quiz

Login to track your progress

How would you rate the quality of this article?

1 2 3 4 5 6 7 8 9 10

Poor Great

Login To Join The Discussion

Your account is free and you can post anonymously if you choose.

Search comments

Popular

A **AbundantCopperSwift328**

Top 5% • 1 year ago • edited 16 days ago

In API definition section, GET endpoint return 301 redirect response.

I think, it needs to be updated to 302 as you have reasoning in high-level design

24

Evan King

Admin • 1 year ago • edited 16 days ago

Will update!

10

R

Renjie Zhang

Top 5% • 1 year ago • edited 2 days ago

I have a question about using Redis as the global counter. what if the Redis is down, and we lost the records of the counter.

21

Evan King

Admin • 1 year ago • edited 3 days ago

Enable high availability with build in replication for recovery

26

C

CapableScarletFly802

Premium • 10 months ago • edited 3 days ago

Even with availability (even using Redis WAIT) you will steal deal with a possibility of a Redis replica containing stale data, if the master went down before it could replicate. How would the system handle this?

10

A

Alok Nayal

Premium • 1 month ago

Not an expert, but shouldn't the replication be happening through a stream, which would be created almost as soon as write happens on the master?

0

d

FullSilverToucan503

• 1 year ago

This replication would be within say an AWS region with multiple AZs? If it were just a single redis cluster, a power event/coordinated failure would be a risk?

2

A

aniu.reg

Top 5% • 1 year ago • edited 23 days ago

Redis can also be backed by a DB. Just like the write service gets next batch of IDs from Redis say (1000 to 2000), Redis can retrieve its "available IDs" from a DB say (1,000,000 to 1,100,000). Once the the id allocation hits 200,000 Redis goes to the RDB and increment the DB record by 1 (means 100 000 new ids)

6

P

pure.disc7055

Premium • 7 months ago • edited 1 month ago

Yeah to me, I'm not convinced that using Redis has any advantages over using a SEQUENCE in Postgres, or just a record in DynamoDB, if you're already using a replicated Postgres or DynamoDB for the URL storage. Especially if you add the batching functionality, the potential performance advantage of Redis is minimal.

6

G

GlamorousHarlequinGecko508

Premium • 4 months ago • edited 1 month ago

Actually yes , I think using postgresql sequence would be better option than global redis,

4

I

its_aa

Premium • 4 months ago • edited 3 days ago

If you have more than one write DB instance, how would you guarantee a globally unique sequence across them without adding coordination overhead — isn’t that exactly what Redis solves with atomic counters?

9

Raju Muke

Premium • 2 months ago • edited 1 day ago

Postgres SEQUENCE is not horizontally scalable and it will add latency like 1 -2 ms . As redis is horizontally scalable and latency is sub milliseconds only.

4

d

FullSilverToucan503

• 1 year ago

Yes, that should hopefully address the coordinated failure risk. DB would need to have cross -region write consistency to guarantee correctness. Likely doable as write qps for every 100000 write is quite low. Not sure what technologies are out there which could be used for this?

0

Vijay Kumar

• 2 months ago • edited 9 days ago

Why we can't use twitter snowflake unique id generator for generating unique id and then encode them in Base62 and then use that as short url

3

Evan King

Admin • 2 months ago • edited 9 days ago

You can. that is the same as the "Great Solution: Hash function"

5

kk khatri

• 1 month ago

u can ..yeah.. and leave the interview in 5 mins with an answer of 'thank you for your time'

0

Aamreen

• 24 days ago

why? isn't that a good approach?

0

A AssistantVioletCardinal950

Top 5% • 1 year ago • edited 14 days ago

1. The POST logic involves checking whether the long URL already exists. However, the design does not mention having an index on the long URL. Should we consider adding an index to the long URL for faster lookups?
2. There is a chance that multiple short URLs could be generated for the same long URL. For instance, if two POST requests are received simultaneously and both check the database while the long URL is not yet present, each request might create a different short URL for the same long URL. [I think this can be addressed by having a uniq constraint on long URL, but the write may be slower]
3. In a multi-data-center setup with multiple writers, how can we ensure that short URL duplicates are avoided and that there are no multiple short URLs for the same long URL?

12

Anton

Premium • 21 days ago • edited 13 days ago

I don't think it's important and maybe not even desirable to generate the same short URL for the same long URL.

Each short URL can have its own properties (e.g. different expiration date).

2

R **Rajeev Ranjan**

Top 1% • 1 year ago • edited 16 days ago

Can a global counter with a batch mechanism lead to easily guessable URLs?

8

Evan King

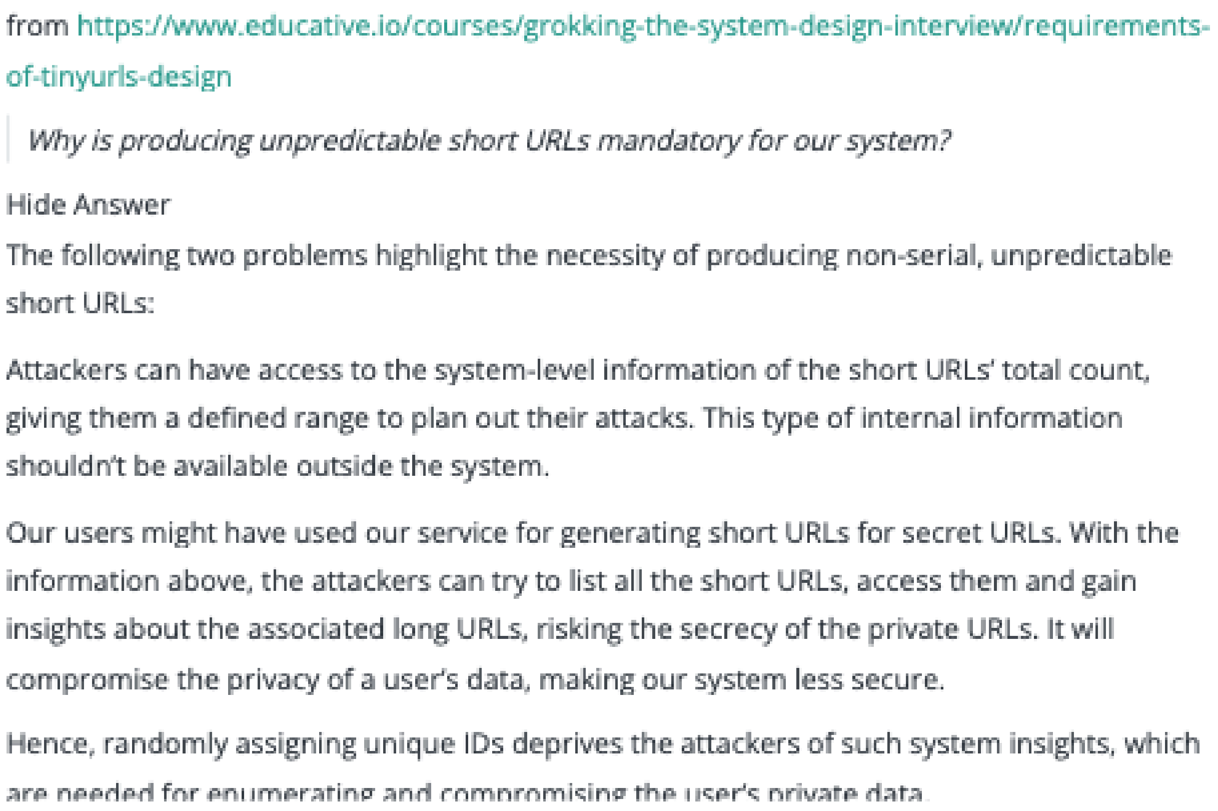
Admin • 1 year ago • edited 9 days ago

I don't see a reason to be worried about guessing short urls :)

11

P **panaali2**

Top 5% • 1 year ago • edited 13 days ago



from https://www.educative.io/courses/grokking-the-system-design-interview/requirements-of-tinyurls-design

> *Why is producing unpredictable short URLs mandatory for our system?*

Hide Answer

The following two problems highlight the necessity of producing non-serial, unpredictable short URLs:

Attackers can have access to the system-level information of the short URLs' total count, giving them a defined range to plan out their attacks. This type of internal information shouldn't be available outside the system.

Our users might have used our service for generating short URLs for secret URLs. With the information above, the attackers can try to list all the short URLs, access them and gain insights about the associated long URLs, risking the secrecy of the private URLs. It will compromise the privacy of a user's data, making our system less secure.

Hence, randomly assigning unique IDs deprives the attackers of such system insights, which are needed for enumerating and compromising the user's private data.

Show More

8

Evan King

Admin • 1 year ago • edited 1 month ago

Valid. But, personally I'd say you should not be shortening private urls. From a product perspective, I'd even make that clear with some copy under the "shorten url" button.

If it became a requirement that we needed to support private urls, we'd:

- Add a layer of indirection (e.g., hash the sequential ID)
- Use a larger ID space to make enumeration impractical
- Implement rate limiting and monitoring for suspicious access patterns (we'd do this anyway).

I'd be less worried about private urls than I'd be about exposing how many urls we've shortened to our competitors. In any case, this is a great discussion in your interview if you have the time, but otherwise, it's a distraction unless clearly agreed upon in the set of requirements.

8

P

panaali2

Top 5% • 1 year ago • edited 6 hours ago

Agreed. easy/clean solution! we can also use a crypto hash with a salt instead of using a larger ID space.

2

Stefan Mai

Admin • 1 year ago

Or a fixed-width block cipher so you don't have to worry about the extraneous collisions :)

3

M

Mohit rao

Premium • 10 months ago • edited 1 month ago

Hey Evan, what do we mean larger ID space here? TIA.

1

I

its_aa

Premium • 4 months ago • edited 1 month ago

number of characters in short code.

1

H **Harsh wardhan Kumar**

• 10 months ago

It creates security, privacy, and abuse risks for the system.
Let's say I am generating short URLs for my unlisted videos in youtube or some documents in my GDrive. The receiver can see all other videos and files by guessing - privacy risks.

Brute force attack - Bots can try all possible numbers starting from 1 and create a replica of our database, they can find valuable contents like files, private meet links, unlisted youtube videos etc.

But Base62 is already harder to guess than other converters like hexadecimal

2

R **RetiredHarlequinHarrier460**

Premium • 9 months ago

Are there any other concerns to using global counters from a security perspective? Competitors can determine the number of url's you have.

0

L **LeftAmethystMite819**

• 1 year ago

Hi Evan,
Our DB writes are not very high write throughput, even if we assume some peak it won't cross 1K QPS. Why not use simple auto increment id and take base62 encoding on it? One potential problem with this approach is our counter metrics are exposed outside but it's not a big deal though. You will have less moving parts in the system. What am I missing?

6

A **aniu.reg**

Top 5% • 1 year ago

I think the problem here is that when the RDB is sharded, it is impossible to keep a global counter - each RDB shard has its own auto increment, if one is down, the new instance wont be able to know which new id to start from.

4

M **MarxistHarlequinMarsupial399**

Top 10% • 1 year ago

Yes, this setup seems quite fine to me too.

At which point should we pass from auto-incremented ID to a Redis counter - this would be nice to have this idea discussed above as it's the simplest of all.

A single Postgres DB with the right hardware, and read replicas, could handle an heavy load quite well.

2

R

Rishabh Sharma

Premium • 1 month ago

I don't think DB sharding should be an issue given we have assumed we'll need 500GB of data at max, let's assume it goes to 1TB, then also sharding is not needed. If you're thinking from read perspective, I don't believe sharding is done primarily for reading first of all, but if we do, we also have to notice that the DB is not what we'll want to rely most of the time upon to return the results. It's the cache we'll want to have the data most of the time, sharding DB during reads to populate cache later on seems like an overkill to me.

0

Show All Comments

Questions

Meta SWE Interview Questions

Amazon SWE Interview Questions

Google SWE Interview Questions

OpenAI SWE Interview Questions

Engineering Manager (EM) Interview Questions

Learn

Learn System Design

Learn DSA

Learn Behavioral

Learn ML System Design

Learn Low Level Design

Guided Practice

Links

FAQ

Pricing

Gift Mock Interviews

Gift Premium

Become a Coach

Our Coaches

Hello Interview Premium

Legal

Terms and Conditions

Privacy Policy

Contact

About Us

Product Support

7511 Greenwood Ave North
Unit #4238 Seattle
WA 98103