**System Design – Interview Questions**

Contents

[ **Concepts** 2](#_Toc172452647)

[ **System Design Interview questions** 11](#_Toc172452648)

[ **Design a URL-shortening service, TinyURL** 11](#_Toc172452649)

[ **Design a social media app?** 22](#_Toc172452650)

[ **Design a web cache (distributed cache), LRU cache** 31](#_Toc172452651)

[ **Design Autocomplete for a Search Engine, Type-ahead system** 42](#_Toc172452652)

[ **Object-oriented design interview questions** 46](#_Toc172452653)

[ **References** 46](#_Toc172452654)

* **Concepts**
* What skills do you need to design Distributed Systems?
* **Technical skills**:

Basic programming: proficiency in at least one programming language: Java, C++, Python.

Software development methodologies: Agile, Scrum, Waterfall for executing projects, managing time.

Fundamental design principles: modularity, abstraction, encapsulation.

Basic UX/UI design knowledge

Deployment and scaling: Cloud infrastructure, containerization, load balancing.

Modeling tools: UML, SysML, for creating visual representations of systems components, relationships and behaviors.

Version control systems: Git, Apache Subversion (SVN), code management, change tracking.

API integration

* **Soft skills:**

Problem-solving

Communication

Teamwork: multidisciplinary teams: software engineers, hardware engineers, UX/UI designers, project managers.

Adaptability

Time management

* Define Load Balancing. Why is it important in System Design?

Load balancing is a technique used in computing and networking for distributing workload evenly across multiple resources (servers, processors, network links), preventing any single resource from becoming a bottleneck or point of failure.

Optimizes how resources are utilized to enhance system performance, and ensures both **high availability** and **reliability.**

* Differentiate between horizontal and vertical scaling.

Are methods of increasing the system’s capacity to handle increased workloads. However, they differ in **how resources are added or adjusted to increase this capacity.**

* **Horizontal scaling (scaling out):** adds more **server** nodes or **instances** to a system to distribute the workload evenly. Capacity is increased by connecting multiple servers, usually with a **load balancer**.

Advantages: Easier to scale, better load distribution, and improved fault tolerance because the failure of a single node won’t crash the entire system.

Drawbacks: More nodes make systems more complex, have higher costs due to additional hardware, and can make applications sensitive to latency because of the physical distance between server nodes.

* **Vertical scaling (scaling up):** involves upgrading or increasing the resources (CPU, RAM, storage) of an existing node or server to handle larger workloads.

Advantages: Less complex, fewer nodes simplify system management, and pretty cost-effective for smaller-scale applications or systems.

Drawback: Single point of failure due to reliance on a single node or server, upgrades require downtime, and there’s an upper limit to how much you can augment a single node or server.

* What are **Sharding** and **Partitioning**?

Are performance optimization techniques used in **database management** to distribute and organize data across multiple **nodes** or **servers**.

* **Sharding**: is a method used to **horizontally** partition a database into smaller, more manageable pieces called shards. Each shard contains a subset of the data, and the shards are distributed across multiple nodes or database servers. Sharding allows the database workload to be spread across several machines, improving performance and enabling the system to handle larger datasets and more user requests.
* **Partitioning**: divides a database into smaller segments based on specific criteria, such as a range of values or a certain attribute. There are two main types of partitioning:

**Horizontal** partitioning: divides a database into multiple tables, each containing a subset of the **rows** from the original table.

**Vertical** partitioning: divides a table into smaller tables or partitions by **columns**, so each partition contains a subset of columns from the original table.

**Horizontal** partitioning offers faster queries unlike **vertical** partitioning, but it causes issues like **data skew**. **Vertical** partitioning is useful because it reduces redundancy and increases data security but **complicates** **query optimization**.

* What are your best practices for **testing** and **debugging**?

A good testing and debugging approach involve constant **refinement** and **iteration** over time. What helps in this process is defining test cases for system evaluation and utilization of third-party tools to assist automation, monitoring, and other debugging practices.

* What is the **cap theorem**?

It describes the trade-off between 3 properties. It says: a distributed system can only guarantee two of these three properties simultaneously:

**Consistency**: every read operation receives the most recent write or an error.

**Availability**: every request (both read and write) receives a non-error response, but does not guarantee that it contains the most recent data.

**Partition Tolerance**: the system continues to function even when communications between nodes are partially or completely disrupted due to network failures.

Different databases can offer different advantages:

An RDBMS (like MySQL, Postgres, etc.) can give Consistency and Availability at the same time, but not Partition Tolerance.

Redis, HBase databases and MongoDB can guarantee Consistency and Partition Tolerance, but not Availability.

Cassandra and CouchDB allow Availability and Partition Tolerance simultaneously.

* What is the difference between an **open** and **closed system**?

An **open system** allows communication with **external** **systems** and has a more complex structure as it incorporates communication with entities.

On the other hand, **closed systems** do **not** transmit information to the external world.

* What is the link between **scalability** and **performance**?

**Scalability**: refers to a system’s ability to handle an increasing workload by efficiently utilizing additional resources (processing power, memory, network bandwidth).

**Performance**: refers to the **responsiveness**, throughput, and overall efficiency of the system when executing tasks or responding to user requests.

A system can have degrees of scalability and performance, such as:

* High performance and high scalability: capable of processing requests or tasks efficiently, even as user traffic or workloads increase. As resources are added to this system, it maintains or even improves its performance.
* High performance and low scalability: capable of processing requests or tasks efficiently for a limited number of users. As traffic or workloads increase, the system struggles to maintain its performance, resulting in slower response times and reduced throughput.
* Low performance and low scalability: struggles to process tasks and requests, even with a limited number of users or workload. As the workload increases, this system’s performance worsens.
* Low performance and high scalability: this system can handle increasing workloads efficiently by utilizing additional resources, but its baseline performance is suboptimal, and it has inherently slow response times or low throughput.
* How do you measure **system performance**?

System performance is measured by the following factors:

Average response time: time (ms) it takes for a system to process a request and return a response.

Latency: speed at which data is exchanged between a client and server.

Throughput: amount of data packets produced or processed in a given period. e.g.: bits per second or HTTP operations per day.

Request rate: number of requests received by a system in a given period, often measured in requests per second (RPS) or transactions per second (TPS).

Error rate: percentage of requests that result in errors or failures.

Availability: percentage of time a system is operational and accessible to users.

Scalability: ability of a system to handle increased workloads by adding resources.

Reliability: ability of a system to perform consistently and accurately over time, without errors or failures.

Apdex (Application Performance Index): standardized method for measuring user satisfaction.

Garbage collection by the system: affects the memory usage and responsiveness of a system. A system with poor garbage collection can lead to excessive memory usage, slower response times, and crashes.

* What are the different types of documentation created during system design?

Software documentation includes:

* **Product** documentation:

1. System documentation
2. Requirements document
3. Software architecture documentation
4. Source code documentation
5. Quality assurance documentation
6. Maintenance and help guide
7. User documentation
8. End-user documentation
9. System administrators documentation

* **Process** documentation:

1. Plans, estimates, and schedules
2. Reports and metrics
3. Working papers
4. Standards

* What are some of the factors to consider when designing a system’s data architecture?
* Types of data the system will be computing.
* The databases that the data will be interacting with.
* Whether the system will require offline functionality and/or real-time streaming abilities.
* Alignment with the organization’s goals.
* Data volume and growth.
* Privacy of data.
* How the data will be accessed.
* What is a **controller** in System Design?

A controller is an essential part of the system as it receives input from a user or external system and manages the flow of data or operations within a system.

The controller acts as an intermediary between the user/external system and the underlying system components, such as a database.

Controller directs other components and makes decisions for the entire system.

* What are the different types of consistency patterns in System Design?
* **Weak consistency**: After a write, reads may or may not see it.

The read request does not get the newly written data.

e.g.: Memcached, real-time uses (VoIP, video chat, online multiplayer games), if you are on a phone call and lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss, because losing information from some seconds ago does not matter in these situations.

* **Eventual consistency**: After a write, reads will eventually see it (within ms).

For highly available systems (DNS, email systems), where data is replicated asynchronously so the written data is read within milliseconds.

e.g.: mail, DNS, SMTP, Amazon S3, SimpleDB

* **Strong consistency**: After a write, reads will see it.

When the data is written in this type of consistency, subsequent reads see the latest data instantly in a synchronous manner. RDBMS and file systems follow this for data transaction.

e.g.: datastore, File systems, RDBMSes, Azure tables

* What is a Content Delivery Network (**CDN**)?

Is a **proxy server network** distributed globally to deliver content to users more efficiently and quickly.

End-users in different locations receive HTML, CSS, JS files, images, and videos from the VDN closest to them. This reduces the user’s waiting time and prevents any single server from becoming overloaded.

* What is a **Web Crawler**?

Are automated programs used by **search engines** like Google to **search documents** on the **web** and retrieve **information** from websites, with the purpose to **index** the contents of websites and **gather** **data** that can be used for various purposes (**engine indexing**, **content analysis**, **data mining**).

* Which structure tools are used in System Design?
* Data Flow Diagrams (DFD): show the flow of data from external entities to its logical storage.
* Decision trees: It is a tree-like model of decisions and their consequences.
* **Pseudocode**: a program outline written in a formal natural language instead of a programming language.
* Data dictionary: lists the information (default label, description, units, etc.) of all the data items that are in the DFD.
* Structured English: Structured English uses straightforward English to break down the program code into an easily understandable logical sequence of steps.
* What is your strategy for designing a recommendation system?

When designing a recommendation system for maximum user satisfaction, it is important to include the following:

* Features should be based on **user requirements** from the system (movies, music, apparel shopping recommendations, etc).
* Mechanism to recommend relevant content in **real-time**.
* Collaborative filtering approach.
* Evaluation component to understanding the system’s working.
* What is **Consistent Hashing** and How does it work?

Technique used in Distributed System (data does not fit on single server) to distributed cluster of nodes efficiently.

Maps data to physical nodes and ensures that only a small set of keys move when servers are added or removed.

Uses:

for Data partitioning: Amazon web services, DynamoDB, Apache, Cassandra

for Distribute web contents evenly: Content Delivery Networks (Akamai)

for Distribute persistent connections evenly: Load balancers

How it works:

Hash Space: circular hash space each node and data are positioned based on hash function.

Node Assignment: nodes (servers) placed on this circle based on their hash values.

Data Assignment

Rebalancing

Every record and server is mapped on unit circle.

Each record is assigned to the first server that appears on the circle in a clockwise direction, it brings even distribution of records.

A diagram of a server

Description automatically generated

**Add server to the unit circle**: records next to the **server** need to be **updated** whereas all the other **records** **maintain** the previous assignments.

**Delete server from the unit circle**: only records associated to this server need to be updated.

* + **Simple Hashing vs Consistency Hashing**

**Simple hashing**:

e.g.: when a set of records needs to be assigned to **n number of servers**. For that, evenly distribute records in each server, we can use MOD operation: id server = record MOD n

problem: when a new server is added or deleted, almost all the keys (record) need to be remapped (**record MOD n**), cause overhead.

**Consistent hashing**:

adding or deleting a server only requires redistribution of a fraction of the keys (**record / n**).

benefits: scalability, Load Distribution, quick Replication and Partitioning of data, faster retrieval of keys as each server holds a limited number of keys.

<https://www.educative.io/answers/what-is-consistent-hashing>

Consistent Hashing | Algorithms You Should Know #1

<https://www.youtube.com/watch?v=UF9Iqmg94tk>

* What is an API Gateway?

Is a server between clients and backend services. Accepts all API calls and return result.

Functionalities:

Routing: directs client request to the appropriate backend services.

Composition: combines responses from services into a single response, reducing client requests.

Security: authentication, authorization, rate limiting.

Monitoring and Logging

Load Balancing: distributes requests from instances of services, ensure availability and reliability.

* What is the difference between SQL and NoSQL?
* **SQL** (Structured Query Language):

Schema: SQL databases use **fixed schema**, **structure** of data (tables, rows, columns) must be defined before data can be inserted. Data integrity and complex queries.

Transactions: **ACID** properties (Atomicity, Consistency, Isolation, Durability) for **financial** systems.

e.g.: MySQL, PostgreSQL, Oracle.

- **NoSQL** (Not only SQL):

Schema: NoSQL databases use **flexible schema**, **unstructured** storage (document, key-value, wide-column, graph stores).

Scalability: **horizontal scaling**, **high traffic** loads, **big data** apps and **real-time** web apps.

Transactions: availability and partition tolerance over consistency (**CAP theorem**).

e.g.: MongoDB (document store), Redis (key-value store), Cassandra (wide-column store), Neo4j (graph database).

* What is Caching and what are different strategies around caching?

Caching is temporal storing of data in a cache, improve data retrieval, reduce the load.

Strategies:

Cache-aside (Lazy Loading)

Write-through

Write-back (Write-behind)

Time-to-Live (TTL): cached data is set to expire, ensure balancing performance and data accuracy.

Eviction Policies: which data to remove when cache is full. e.g.: Least Recently Used (LRU), Most Recently Used (MRU), Least Frequently Used (LFU).

* What is the difference between a row and columnar database?

Row database:

Storage: row-by-row, for transactional operations where row is accessed or modified frequently.

Access Pattern: for apps with high volume of write operations and complex transactional queries.

e.g.: MySQL, PostgreSQL, Oracle.

Columnar database:

Storage: column-by-column, for read operations, analytical queries with aggregate functions.

Access Pattern: for read-heavy workloads and OLAP (Online Analytical Processing).

e.g.: Apache Cassandra, HBase, Google BigQuery.

* What is a CDN and how is it helpful?

CDN (Content Delivery Network) is a geographically distributed network of servers to deliver content to users more quickly and efficiently based on their location.

Benefits:

Reduced Latency: caching content closer to end-users, faster load times.

Improved Load Times

Reliability and Redundancy: distributes content across servers, availability if some server fail.

Scalability: handle large traffic spikes by distributing the load across many servers.

* What’s the difference between Polling, Long Polling, WebSockets, and Server-Sent Events?

Polling:

Client periodically sends requests to server to check for new data. Simple but inefficient because unnecessary requests when there is no new data.

Long Polling:

Client sends request to server, server holds the request open until new data is available. Once ne data is sent, the client re-request. Reduces latency, more efficient than Polling.

WebSockets:

Full-duplex communication channels over a single, long-lived connection. For real-time, bidirectional communication client-server. Uses: live chat, gaming.

Server-Sent Events (SSE):

Server sends automatic updates to client over a single HTTP connection. Simpler to implement than WebSockets for one-way data streams: live news feed, real-time notifications.

* What is Data Partitioning?

Dividing a large database into smaller pieces to be distributed across storage nodes or servers. Improves scalability, performance, manageability of large databases and traffic.

Types of Partitioning:

Horizontal Partitioning (Sharding): divides data by rows.

Vertical Partitioning: divides data by columns.

Range Partitioning: distributes data based on a range of values (dates, numerical) for query processing for range queries.

Hash Partitioning: distributes data based on a hash function for columns, even distribution and minimizing the risk of hotspots.

* What is a Database Index and How does it help?

Is a data structure to improves speed of data retrieval operations on database table, quick access to rows, used in databases with complex queries.

Benefits:

Fast Query Performance: indexes reduce amount of data, quick search and retrieval operations.

Types of Indexes:

B-tree Indexes: more used, quick search, insert, delete, sequential access.

Hash Indexes: lookup constant-time, no suitable for range queries.

Full-text Indexes: for searching large text fields.

Trade-offs: indexes have read performance, but slow down write operations (overhead of maintaining the index).

* What are Load Balancers?

Are devices or software that distributes incoming network traffic across servers, improve availability and reliability.

* **System Design Interview questions**
* **Design a URL-shortening service, TinyURL**

TinyURL is a URL shortening web service that creates **shorter** aliases for **long URLs**. Users select these shortened URLs and are redirected to the original URL.

Uses:

Short links save space, abbreviate URL.

* + **Ask clarifying questions**
* Allow **custom** **URL**? Will users be able to customize the URL? Can the user create their own custom URL (short.com/custom) or it be automatically generated by the server (short.com/a7g3k2)?

User create their own URL, up to 16 chars.

* Short URL **expire**? How long do the URLs last before they expire?

Assume it should live for 100 years, forever.

* We don’t reuse the expired short URLs. Why do we need to delete them from our system?

May be default expiration time 5 years. If records forever, our datastore’s search index will grow without bound, querying time can make latency.

* What are the availability and latency requirements for this system?
* How many **shortens/month**?

100 million/month

* **Key features:**

Dynamic short URL range

Improved readability

* **Functional requirements**:

**Generate** Short URL, from long URL

**Redirection**, click short URL **redirects** to long URL

Create **custom short URL**, **16 chars** max

**Deletion**

**Update**: user update long URL

Short URL as small as possible, **7 chars**

**Expiry time**: user set expiration time, by default short URL stay in system **100 years**

System can handle multiple requests

* **Non-Functional requirements**:

**Availability**: running all the time, no downtime

**Scalability**: horizontal scale

**Readability**: short URL easy to read

**Latency**: low, smooth experience. URL redirect is fast, even if there’s a lot of traffic

**Unpredictability**: security to ensure next-in-line short URL is not serially produced (not easy to guess).

Service should expose REST API’s for devs.

* + **Design high-level**

Calculations: estimate **traffic** and **storage** need per **month**, **bandwidth** and memory requirements.

Define the APIs (SOAP, REST) and general database schema (URL mapping and user data).

* **Facts and Figures**:

**200:1 (reads:writes) ratio**

1 short URL created – 200 short URL clicked

**100 million short URLs created/mo**

40 short URLs created/sec - **8000 short URLs clicked (reads)/sec**

Short URL stored 100 years

100 million short URLs created/mo

12 mo = 1 year

100 yrs = 1200 mo => 1200\*100 million = 120 billion objects short URLs created

{ longURL, shsortURL, createdAt, userID }

1 object = **500 bytes**

120 billion objects \* 500 bytes = **60 TB storage**

8000 reads/sec\*60\*60\*24 = **700 million reads/day**

Assume 80% of the requests are for 20% of the data

20% short URLs: more popular, created by influencers with huge followings

Cache 20% of requests

20% \* 700 million \* 500 bytes = **70 GB cache memory**

**Summary**:

40 URLs generated/sec

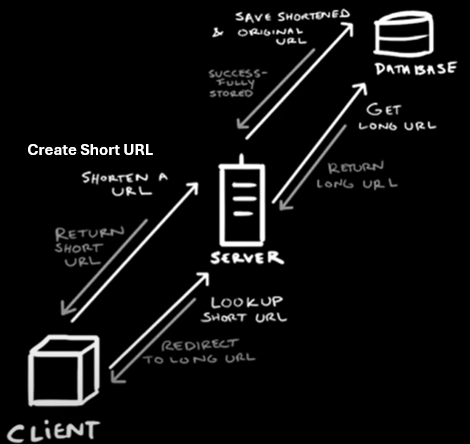
120 billion short URLs generated/100 years

8000 URLs reads/sec

60 TB storage

70 GB cache storage

* **Approach 1:**



Have one **server** is a **single point of failure**, if this server fails then the whole service fails. Might happen **DDoS (Distributed Denial-of-Service) attack** (cyber-attack): a malicious actor floods the system service with more request than service is capable of handling, causing the service to fall over, outage, downtime. It’s handled by adding **Server Redundancy** using **Load Balancer**.

* **Approach 2:**

**A diagram of a server

Description automatically generated**

**Algorithm**:

Store a lot of short URLs (120 billion)

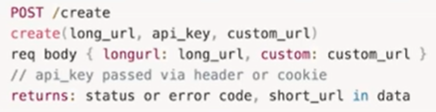
Short URL should be as short as possible (7 chars = minimum necessary to given all the possible combinations of all alpha numerical characters to sufficiently accommodate our expected traffic)

App should be resilient to load spikes (for URL generation (creation) and redirection (read))

Following short URL is fast

**System APIs: REST** endpoints

Create Short URL:



Lookup Short URL: when is clicked



**Server**:

Creation of short URL is always unique and 2 short URLs can not point the same long URL:

Option 1 (used): **hashing** strategy, using **base 62 encode**

A blackboard with white text and blue letters

Description automatically generated

Option 2: using **key generation service**

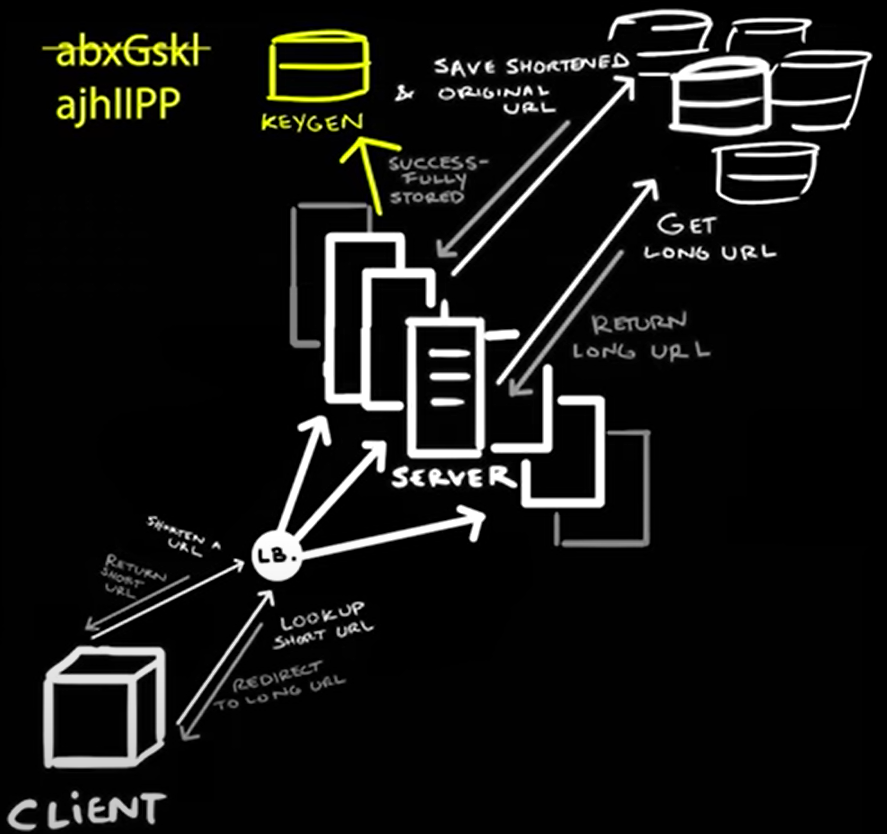
A blackboard with white text

Description automatically generated

**MD5** has collision attacks

Built-in URL encoding functions provide by programming language/framework

We would want to have a dedicated database (KEYGEN) to store already generated random 7-character strings, and whenever we want to use one of these seven character strings for a new URL we would go into that database and Mark that entry as used. Is more complicated (no single point of failure, concurrency issues).



**Database**:

**Database** schema:

User:

ID

name

email

creation date

URL/Link:

short URL (7 chars)

original log URL

user ID

creation date

**Relational database** (**SQL**): efficient to check if a URL exists in the database and handle **concurrent**, more **difficult** to **scale** in comparison to a NoSQL database. **ACID** compliance means a database **update** will get fully **committed** or fully **rolled back**.

**NoSQL**: **easy** to **scale**, consistent (if you perform to the database that change isn’t guaranteed to immediately propagate to all other parts of the system). Read immediately following a write could return **stale** information.

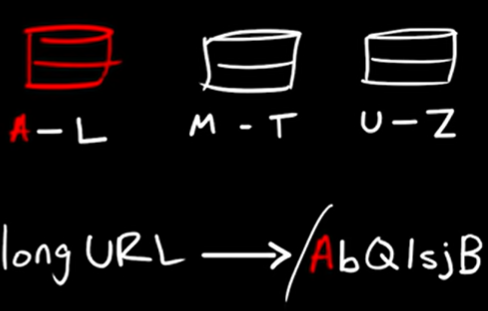
I choose **NoSQL**, because for this case, NoSQL database is a little better that SQL. Easier scaling for huge **storage** (60 TB) & **high reads** (8000/sec) / **writes** (40/sec).

We could scale with a **relational database** using **Customer Partitioning** and **Replication**, but is more difficult to develop and maintain in general, in contrast these features are available by default in **NoSQL** like **Mongo** and **Cassandra**, we want to **distribute** data across multiple machines using **Shards**.

**MongoDB** (used): replicas for **heavy reading**, ensures **atomicity** in concurrent write operations and **avoids collisions** by returning duplicate-key errors for record-duplication issues.

**Cassandra**, Riak, DynamoDB: need **read-repair**, **slower reads** to write **performance**, are leader-less NoSQL databases, **weaker** data **consistency**. **Cassandra** more **availability** than **MongoDB**.

When we’ve generated a **7-character short URL** and then use the **hash** as a Shard key, means is that we’re using the **hashed** version of the **URL** to determine which **bucket** it will live in. First, we hash the URL and that hash becomes the key and our database will automatically decide which bucket the URL will go into, then when we need to look up that information the database which bucket to look into based on the hash.



**Data partitioning and replication:**

Make scalable

**Range-based partitioning**: partitions based on first letter of its **hash key**. Cause **unbalanced** database servers, create unequal load.

**Hash-based partitioning**: take the hash of a stored object and calculate which partition to use. Cause overloaded partitions, to solve this use **Consistent hashing**.

**Cache**:

Cache server: MemCached, **Redis**, horizontal scale, read-intensive, contain long and short URL.

**Pareto principle**: 80/20, cache 20% of daily traffic, we can guess that 80% of the click to Links in our system will be clicks on about 20% of the links.

We need 70 – 100 GB of memory to Cache for 20% of traffic.

Modern-day Enterprise level server can have 64 -256 GB of memory, we need 1 – 2 servers. Good idea is to add **redundancy** (**replicas**, **backup**, **distribute** hottest URLs in multiples machines).

Is Cache is full?: apply policy **Least Recently Used** (**LRU**), use **Linked Hash Map** to store **URLs** by **hash** that track which ones have been most and least recently used. Linked Hash Map use Hash Map table (quick access bases on their keys, lookup O(1) constant time) and Linked List (maintain order elements)

**Cache miss happening**: when data requested is not in the cache, servers go look into database, we use the data retrieved from database to update the cache and cache replicas.

**Load Balancing**:

**Distribute requests among all available services** or given certain **priority** to **certain machines** in **certain scenarios**. Improve availability.

A diagram of a server

Description automatically generated

Use **Round Robin Strategy** to distribute requests evenly among all available servers. Best case: For 4 servers, each one handle 25% of traffic. Worst case: one server becomes overloaded or slow, we can implement **health check** which queries the servers and adjusts load distributions based on that.

* + **Drill down on your design**

Consider tradeoff: encoding actual URLs may turn out the same shortened URL for two different users who enter the same URL. System may not work for URLs with URL-encoding. **Concurrency** may cause problems, etc.

Where will you place load balancers?. How will you cache URLs?

Identify the common problems like handling user load and regulating database storage space.

Select the best type of database to store original URLs.

How do we ensure that two **concurrent** requests for a short URL do not **overwrite**?

**MongoDB** ensures **consistency** by **locking** and **concurrency** control protocols, preventing the users from modifying the **same data simultaneously**.

* + **Bring it all together**

Is the system you’ve designed highly available, so that URLs will not break if the servers go down?. Does it meet any potential business objectives laid out at the start of the interview?

**Edge case**:

When the customer wants to create their own custom short URL, system generates random URLs 7-characters. How do we make sure that we can give these customers the most freedom to choose whatever URL they want with the least likelihood that their custom URL will conflict with one of the random ones the system has already generated.

Solution: any custom URLs be at least 8 character. Additional, for paid service we could have a dedicated **database** instance for these custom URLs as well as dedicated **web servers**.

**Availability**: replication, backups the **storage** and cache daily with **Amazon S3 storage service**. **Load Balancers** to handle system’s **traffic**. Rate Limiters to limit each user’s resource allocation.

**Scalability**: distribute horizontal **sharded** databases, consistency **hashing** scheme to balance the load, MongoDB.

**Readability**: use base-62 encoder or base-58 encoder instead of base-64 encoder to generate short URLs. Remove non-alphanumeric characters.

**Latency**: MongoDB low latency.

**Unpredictability**

Companies: Google

refs:

<https://www.educative.io/blog/system-design-tinyurl-instagram>

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/system-design-tinyurl>

System Design: Design a URL Shortener like TinyURL

<https://www.youtube.com/watch?v=zgIyzEEXfiA>

* Design a **Ride-Sharing service**.

1. Define the **architecture** for the system (suggested architecture: monolithic or microservices).
2. Choose the **database** to use.
3. Design an optimized **dispatch** **system** to match users with drivers.
4. Design the **mechanism** for **maps** and **routing**.
5. Define an approach to storing **geographical** locations.

* Design a **Video Streaming Service** like Youtube.

A **video-sharing** platform would allow users to **upload**, **watch**, **share**, and **comment** on video-based content. It is a huge system that will be transmitting **petabytes** of data, it will have to be scalable so that a large number of users can view and share the video content simultaneously. You can use the following approach:

* Define the components for this system, which would be:

1. Client: Devices to use the service like mobile phones, smart TV, etc.
2. CDN (Content Delivery Network): all videos are stored in the CDN which allows viewers to stream videos.
3. API (Application Programming Interface) servers: performs functionalities like playlist, creation feed recommendation, user signup, etc.

* Design the approach to record video stats.
* Decide how users would be able to add comments in real-time.
* Design the Uber backend.

<https://www.educative.io/blog/uber-backend-system-design?eid=5082902844932096>

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/system-design-twitter?eid=5082902844932096>

* Design Quora
* Design Google Maps
* Design a Proximity Service/Yelp
* Design WhatsApp
* Design Typehead Suggestion
* Design a Collaborative Document Editing Service / Google Docs
* Facebook System Design

<https://www.educative.io/blog/facebook-system-design-interview>

Design Facebook Newsfeed

Design Messenger

Design Instagram

* Others:

Design Ticketmaster

Design an API rate limiter

* Design a Web Crawler

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/system-design-web-crawler?eid=5082902844932096>

* Design YouTube

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/system-design-youtube?eid=5082902844932096>

* **Design TikTok**
  + **High-Level Architecture Overview**

1. **Client Application: Mobile and Web apps.**
2. **Backend Services:**

API Gateway: centralized entry point for all client requests.

User Management Service: authentication (login/signup), authorization (role-based access control), profile management.

Content Management Service: manages video uploads, processing (transcode), storage (cloud).

Feed Generation Service: personalizes the content feed for users based on preferences and interactions.

Search and Discovery Service: Facilitates content search and discovery through hashtags, trends, and recommendations.

Social Interaction Service: manages likes, comments, follows, shares.

Notification Service: handles **push notifications** and in-app notifications.

Analytics Service: collects and processes usage data for insights and recommendations.

Advertising Service: manages ad placements, targeting, tracking.

1. **Databases and Storage:**

User Database: stores user profiles, authentication tokens.

Content Database: stores metadata about videos, hashtags.

Media Storage: cloud-based storage for video files, thumbnails.

Cache Layer: caching for frequently accessed data to improve performance (Redis, Memcached).

1. **Video Processing Pipeline:**

Upload Handler: handles video uploads from users.

Transcoder: converts videos to various formats.

Thumbnail Generator: creates video thumbnails.

CDN Integration: Distributes video content globally via a Content Delivery Network for fast and reliable access.

1. **Recommendation Engine:**

Data Collection: collect data on user interactions, preferences, behaviors.

Machine Learning Models: uses algorithms to personalize content feeds and recommend videos.

Real-Time Processing: adjusts recommendations in real-time based on user activity.

1. **Infrastructure:**

Cloud Services: uses cloud infrastructure (AWS, GCP, Azure) for scalability and reliability.

Microservices Architecture

Load Balancers: distributes incoming traffic across multiples servers.

Scalability Mechanisms: auto-scaling and load balancing to handle varying traffic loads.

refs:

Google system design interview: Design TikTok (with ex-Google EM)

<https://www.youtube.com/watch?v=NHqdG-aZxOk>

1. Clarifications questions

* Narrow down the question scope.
* Don’t be afraid to make suggestions about which part of the system to focus on.
* Make sure the interviewer is on board your approach.

Is a video sharing application, social media platform.

Give me some uses cases to explain

Focus on a back-end distributed system that supports uploads and downloads, rather than the TikTok app.

In terms of sign in and sign up and things like that can we kind of skip that, because is pretty standard I think.

So you want to focus on sort of the video into the system and then streaming the videos or consuming the videos I guess?

Tell me about, How many users?, How many videos uploaded?, What’s the scale of this?

1B users, 150 countries, 1B video views a day, 10B videos uploaded in on the year.

Start in **Google drawing**.

Do you have any success metrics?

Success metrics: time in app // How long are people spending in the app?

A close up of a paper

Description automatically generated

1. Non-functional requirements

* Make some calculations to determine the scale of the system.

Don’t do the maths for the user data storage req uirements, instead do to make the traffic calculations and queries per second calculations

* Practice multiplying and dividing big numbers.
* Prioritize those calculations which will most impact your design – in this case, storage.

Detailed assumptions: I assumed that is a vertical form factor video sharing

A close up of a sign

Description automatically generated

Is possibly some replication to have redundancy, also possibly we have to have different formats for different types of devices. Is a lot storage.

Videos are blobs or objects, a storage solution like a blob storage solution.

Does that seem reasonable so far in terms of calculations?

A close up of a sign

Description automatically generated

To do calculations of video metadata, like ID for a video when was created, how long is it, maybe a number of likes ultimately because we want to be able to track how many people like this video.

These videos don’t need to relate to each other, so I probably would use some sort of non-relational (No SQL) database solution. Would be like key-value pair storage.



Lookup network traffic:

Uploading 300 videos per second

100000 sec per day aprox.

A close up of a screen

Description automatically generated

A close up of numbers

Description automatically generated

1. High level design (components)

* Map out the main components of your high-level design.

TikTok app service: run in the cloud, microservice.

TikTok upload service.

1. High level design (upload flow with databases)

* Map out the upload flow.

Let’s talk about the video storage database: use hosted solutions

Video blob storage, raw videos, tiered Amazon S3 (because if a video has a year old and nobody’s really looked at it, it could go into really slow offline storage, which means it doesn’t cost as much, is more cost effective).

Amazon web services: you can replicate the data for better availability, and you can tier the storage data.

Video metadata: DynamoDB

How you think the different regions and countries could be reflected in your design?

29:14

* **Design a social media app?**

Twitter (X.com), Instagram, Facebook

* + **Ask clarifying questions**
* Core features or a high-level overview of the whole service.
* Constraints of the system
* Assumptions (traffic distribution, number of active users and tweets, read vs write-heavy).
* Key features:

1. **Tweeting**: send a tweet
2. **Timeline**: where you see other people’s tweets (that you are following and you aren’t following) and your own tweets.

**User timeline**: your own timeline on your profile, you see your **own tweets** and **retweets**.

Take your own action and list them up.

**Home timeline**: you see all the **tweets** from **people** you **follow** on the **home page**, and we merge them in chronological order.

1. **Following**: you click following someone else’s profile and you want to **see** that **someone else’s tweets** in **your home timeline**.

- Functional requirements:

- Non-Functional requirements:

* + **Design high-level**

Back-of-the-envelope calculations: average KBs per tweet, size of new tweet content per month, read requests and tweets per second.

High-level components: write, read, search APIs, types of databases, SQL vs NoSQL.

* Facts and Figures:

Summary:

**- Approach 1: Naive solution for Home timeline**

Synchronous DB queries

Using **relational databases** (MySQL) where you have a **user** and **tweet** **table**.

The user tweets a message and that’s just one record in the table.

A diagram of a user

Description automatically generated

User accesses is home timeline:

Who this user is following? and do a big select statement over Tweets table, means tweets from all the people I follow and give me them in chronological order. Is big amount of data, uses indexes.

**- Approach 2: Better solution for Home timeline**

Characteristics of the system. What to optimize for?

People **read more tweets than write**. Need is to have a **super-fast read function** where you can access your whole timeline in less than a second.

**Consistency perspective**: Is less important. if you tweet (or post in Facebook somebody else has to see that post as fast as possible). Is **eventual consistency**.

**Availability perspective**: more important. Is really bad if one of those two people can access your network.

Give high level overview:

A screenshot of a computer screen

Description automatically generated

<https://docs.google.com/drawings/d/1vwEyQUxpFNTyaWLtDZ7hUzba6yhoJk5O-EN8Ljmxxgo/edit>

**in-memory database = RAM**

**fan-out**: they take your tweet and then put in an **in-memory database**, means **pre-computing** (means **calculate** and **store in memory**) all home timeline of everybody you follow.

* Mixed approach: In-memory + synchronous calls

What could be a performance problem with this architecture?. Weaknesses, think about edge cases.

Took very long to update those user Redis lists.

**Edge case**: people would react on a tweet and you wouldn’t see the original tweet people reacted.

A celebrity (Justin Bieber) sends a tweet, his/her millions of followers see and react that tweet, while other people didn’t see the original tweet they only saw the reaction of their friends, because the friend who you’re following has only a hundred followers, this replication is a lot of faster than the celebrity

**- Approach: Solution for Following**

When tweet something, go to the **Followers table**, then this table will give us all the Redis list IDs, we have to pre-compute. (see imagen of approach 2)

**Trade-offs of architecture**: Time vs. Space

Need to have fast reads

Space:

Twitter has a limitation of 140 characters per tweet.

**Replicate** user Redis list in **home timeline** consumes a lot space.

* **Approach: Solution for User timeline**

What happens when Bob accesses this **timeline**

We do computations when someone is tweeting, and when the user go to Twitter browser get a fast response.

A diagram of a process flow

Description automatically generated

<https://docs.google.com/drawings/d/1aZpJOMrCwwemGkPKNOhR_g6vFi7hZjgcp3ed4LQQgWg/edit>

* + **Drill down on your design**

Potential bottlenecks: load balancer with multiple servers, scalability issues, **fanout service** slowing down tweets and @replies.

Components: how a user views the home timeline or posts a tweet, the intricacies of the database design.

* + **Bring it all together**

Consider:

Does the final design address the bottlenecks you’ve identified?

Does it meet the goals you discussed at the beginning of the interview?

Do you have any questions for the interviewer?

* In-depth topics:

Search

Monetization

Push notifications:

SMS notifications

Advertisement: social networks know about their customers, preferences, products they’re interesting. Talk about how ingest all those analytics data and place effective advertisement tweets in a timeline.

Companies: Google

refs:

Design the Twitter timeline and search

<https://github.com/donnemartin/system-design-primer/blob/master/solutions/system_design/twitter/README.md>

System Design: How to design Twitter? Interview question at Facebook, Google, Microsoft

<https://www.youtube.com/watch?v=KmAyPUv9gOY>

* **Design X game**

Tic-tac-toe:

Noughts and crosses, Os and Xs, paper-and-pencil game.

Strategy, Observation.

A red tic tac toe game

Description automatically generated

Chess

Boggle:

Word game in which players try to find as many words as they can from a grid of lettered dice.

Paper and writing utensil, 2+ players

Language

A computer screen shot of a game

Description automatically generated

Minesweeper:

Logic puzzle video game

A screenshot of a game

Description automatically generated

Cards/poker

* + Ask clarifying questions

What are the rules of the game?

How many players are there? Are there spectators?

Do we need a timer? Are any other special functions required?

* + Design high-level

Possible classes for the game\_ board, piece, spot, etc.

Methods that will be required for things like moving pieces.

* + Drill down on your design

Identify important attributes for each class: grid coordinates, color of each spot.

Define how the game will prevent illegal moves and recognize a victory.

* + Bring it all together

Check your design. Confirm whether it has met all of the requirements.

* + **A Two-Player Online Chess Game**

<https://medium.com/double-pointer/system-design-interview-a-two-player-online-chess-game-4a150ad8ea78>

* Problem Statement:

Online chess game, chessboard and chess pieces, played by 2 logged in to the game from different devices.

* Chessboard:

A chess board with chess pieces

Description automatically generated

* Game Rules:

8x8 checkerboard

6 types of chess pieces: king, queen, knight, bishop (alfil), rook (torre), pawn

Can be 2 outcomes of the game: checkmate (one side wins) or stalemate (draw=empate).

* Design:

Object oriented design problem.

Classes: Board, Spots (cells on the chessboard) of pieces (each player) and no pieces (NULL), Piece (color and status), Player (pieces and color), Move, Game (board and 2 players)

A graph with numbers and a grid

Description automatically generated with medium confidence

A line drawing of different angles

Description automatically generated with medium confidence

Class Diagram:

A diagram of a computer

Description automatically generated

* **Design a Parking Lot**
  + Ask clarifying questions

Is this a multiple floor parking garage or a single level parking lot?

How many entry and exit points will be needed, and for what types of vehicles?

Are there monetary goals for this parking lot?

* + Design high-level

Possible use cases: customers parking and paying for their spot, admin managing the system, parking attendants maintaining the lot and helping customers, etc.

Possible classes of the system: ParkingLot, ParkingFloor, Account, ParkingTicket, Vehicle, etc.

* + Drill down on your design

How will you diagram specific activities? (e.g.: customers paying for parking tickets, display panels showing available spots, etc).

What are the required enums, data types, constants of the eventual code for the parking lot system?

* + Bring it all together

Will this system meet the requirements you’ve laid out with the interviewer in the beginning of the session?

* + **Parking Lot System Design**

Show your Object-Oriented Design Skills, Design Patterns, Tackling Concurrency.

Selecting a parking spot near some entrance.

Finding the nearest parking spot.

* Requirements:

A white board with black lines

Description automatically generated

1. Big Parking Lots, 10K - 30K
2. 4 entrances, 4 exits
3. Customers collect tickets at the entrances and the parking spot is assigned on the ticket.
4. Parking spot assigned to a vehicle should be near to the entrance.
5. The system should not allow more vehicles than the capacity of the parking lot.
6. Types of parking spot: for handicapped, compact cars, large cars, motorcycle.
7. Hourly rate: the system should support hourly parking rate and the final parking fee should be based on the total time that a vehicle is spent in the parking lot.
8. The customers can pay parking fee with cash and credit.
9. Monitoring System: how many cars are entering and exiting.

* Design Patterns:

Creational: deals with how the objects are instantiated.

Structural: how different objects and classes are composed in order to form larger structure.

Behavioral: deals with possibilities of the objects and how they interact with each other.

* Design approaches:

Top-Down Design: first design the high-level object, then identify smaller sub-components.

**Bottom-Up Design**: first design the smallest component, then design bigger components. Aligned to **Object Oriented Design**.

* Identify different objects in our system:

1. Parking Lot System
2. Entry and Exit terminals: have printers, payment processors.
3. Parking Spot

Other use Vehicle class, but it don’t involve in our parking lot, so don’t use it.

1. Ticket
2. Database
3. Monitoring System

* Interfaces, Classes, Components using Bottom-Up approach:

Parking Spot: abstract class (we cannot instantiate an object of Parking Spot).

Sub classes: Handicap, Compact, Large, Motorcycle Parking Spot.

A whiteboard with black text

Description automatically generated

Parking Ticket: id, parkingSpotId (buying slot number), parkingSpotType, time (time when the vehicle enters the parking lot).

A white board with black text

Description automatically generated

* Types of Terminals: Entry and Exit terminal.

A diagram of a ticket

Description automatically generated with medium confidence

16:45

Strategy Design Pattern

* **Design a web cache (distributed cache), LRU cache**

**LRU** is a **cache eviction policy**, means the cache evicts the least recently used items first. Evicts data from cache when cache is full.

eviction = replacement

Uses:

RAM of multiple machines can be accessed in a single in-memory store quickly and reliably.

Distributed Message Queue, Notification Service, Rate Limiter

* + **Ask clarifying questions**

**- Functional requirements:**

put (key, value): store object in cache

get (key): retrieve objects from cache

assume key, value are strings

**- Non-Functional requirements:**

for Distributed System:

Scalability:

scales out increasing requests and data.

create more **shards** and have more data stored in memory.

Availability:

survives hardware/network failures

**shards** to more data in-memory. Is not there at all, if some shard dies or becomes unavailable due to a network partition, all cache data for that shard is lost and all requests to that shard will result in a cache miss until keys are re-hashed. **hot shard problem**.

Performance:

fast cache

fast puts and gets

LRU cache uses constant time operations.

**Cache** client picks **Cache Server** in log(n) time very fast.

Connection between Cache Client and Cache Server is done over TCP or UDP, fast

Durability: for data persistence.

**CAP theorem**: Availability replaced Consistency

* + **Design high-level**

Possible data structures for storing data: hash table, queues, doubly linked list.

Options to distribute the cache: **dedicated cache clusters vs co-located caches**.

* **Approach 1:**

Client -> Web Service -> Datastore

Data store is database or web service.

Disadvantages:

Calls to the data store take long time.

Use lot system resources

Improvements to do:

Storing data in memory (**distributed cache**) improves latency, only if data is unavailable or stale we call to the datastore.

A diagram of a web service

Description automatically generated

* **Approach 2: Local cache**

Client -> Local Cache (Service) -> Datastore

Store data in-memory local, **limited capacity**.

Data structure: Hash table (key-value), retrieve constant time O(1).

Problem: when reach maximum Hash Table size and cannot add any more elements, Hash table **do not track** which entry has been used recently, must evict or replacement policy old data.

Solution:

We use 2 Data structures: a **HashMap** (Hash table, retrieve O(1)) and **Doubly Linked List** (track the most **recently** accessed elements, constant time O(1) for add, update, delete).

**LRU cache algorithm:**

A diagram of a diagram

Description automatically generated with medium confidence

**head** of list is **most recently** used, **tail** of list is **least recently** used. When cache is full, to free space for new item, we remove item at **tail** of list.

Simulation:



**LRU cache algorithm implementation:**



**Distributed System:**

A comparison of a service host

Description automatically generated

Dedicated cache cluster:

Cache and the service do not share memory and CPU anymore

Co-located cache:

We do not need a separate cluster.

Cache and the service scale out at the same time.

Clients call the cache process using TCP or UDP connection.

**Choosing a cache host:** How do cache clients decide which **cache shard** to call?

**MOD hashing:**

CacheHostNumber = HASH\_FUNCTION(key) **MOD** NumberOfCacheHosts

CacheHostNumber: remainder as an index in the array of cache hosts.

e.g.: we have 3 cache hosts

hash = 8 (HASH\_FUNCTION(key)),

8 MOD 3 = 2: so the cache host with index 2 will be selected by the service to store this item in the cache and while retrieving the item from the cache.

A diagram of a service

Description automatically generated

What happens when we add a new cache host or some host dies due to hardware failures?: the MOD function will start to produce completely different results. 8 MOD 4 = 0. Service hosts will start choosing completely different cache hosts than they did previously resulting in a high percentage of cache misses. **Uses**: no production, for testing purposes. **MOD hashing is a poor choice**.

A diagram of a service

Description automatically generated

**Consistent hashing (better):**

Map each object on circle. Pick arbitrary point assign 0. Move clockwise along the circle and assign values (232). Take a list of cache hosts and calculate each **hash** based on host identifier (IP address, name), this **hash value** tell us where on the consistent hashing circle that host lives, to help us to assign a list of hash ranges each cache host owns.

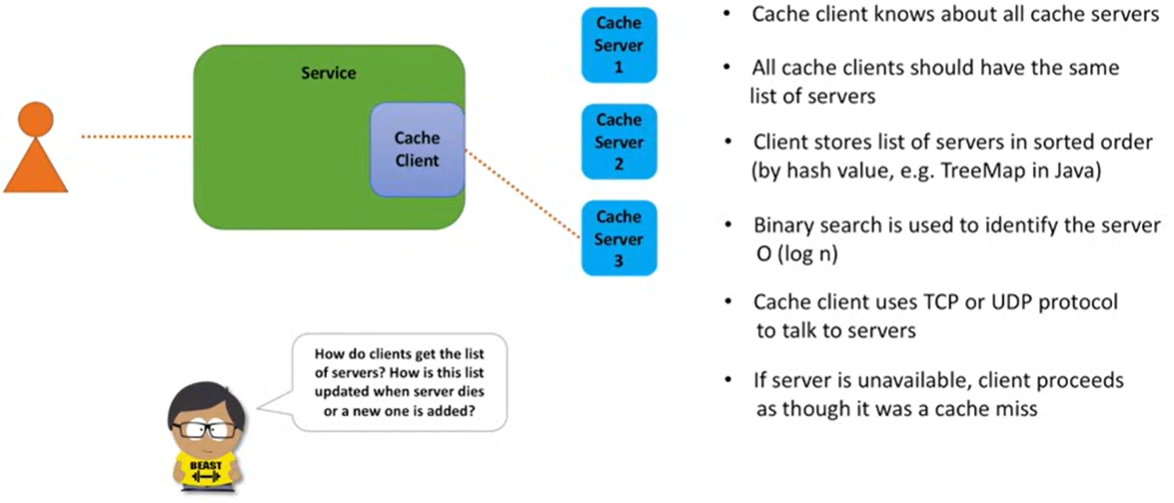
Each host will own all the cache items that live between this hosts and the nearest clockwise neighbor.

A diagram of a network

Description automatically generated

Who is responsible for running all these hash calculations and routing requests to the selected cache host?

**Cache Client**: library integrated with the service code, responsible for the **cache host selection**.



Maintaining a list of cache servers: How list is created, maintained and shared among all clients.

**Option 1**: store list of cache hosts in a file and deploy it to service hosts using **Continuous Deployment Pipeline**. Simple, no flexible. Every time list changes, code change, **redeploy**.



**Option 2**: simplify deployment process. Put **file** to **shared storage**, make service hosts poll for the file periodically from common location (S3 storage service). **Daemon** process runs on each **service** host and polls data from storage once a minute.

Disadvantages: need maintain the file manually, make changes and deploy it to the shared storage every time cache host dies or new host is added.

A diagram of a server

Description automatically generated

**Option 3 (better)**: We could monitor cache **server health**. If bad happens then all service hosts are notified and stop sending any requests to the unavailable cache server. If a new cache server is added, all service hosts are also notified and start sending request to it. For that, we need **Configuration Service** to discover **cache hosts** and **monitor** their **health**.

Each cache server registers itself with the configuration service and sends heartbeats to the configuration service periodically. As long as heartbeats come, server is keep registered in the system. If heartbeats stop coming, the configuration service unregisters a cache server that is no longer alive or inaccessible.

Every cache client grabs the list of registered cache servers from the configuration service.

Is the hardest of implementing, operational cost is higher, but it helps to fully **automate the list maintenance**.

A diagram of a service

Description automatically generated

**Summary**:

To **store more data in memory** we partition data into **shards**, put each shard on its own server.

Every **cache client** knows about all **cache shards**, use **consistency hashing algorithm** to pick a shard for storing and retrieving a particular cache key.

* + **Drill down on your design**

Tradeoffs: maximum hash table size will prevent from adding more elements, shards may become “hot” (process more requests than others).

Data replication could help with “hot” shard bottleneck.

* + **Bring it all together**

Is fast, highly scalable, available distributed cache.

**Availability**:

**Solve hot shard problem for high availability**: use Data Replication. 2 categories of data replication protocols:

1. Includes set of probabilistic protocols like Gossip, Epidemic broadcast, Trees, Bimodal multicast: **favor eventual consistency**.
2. Includes consensus protocols like 2 or 3 phase commit, Paxos, Raft, Chain Replication: **favor strong consistency**.

We use **Leader-Follower (aka master-slave) replication**: Leader = Master, Replica = Follower

Every time the connection between master and replica breaks, replica attempts to automatically reconnect to the master.

Calls to a cache shard are now spread across several nodes, is much to deal with hot shards.

Leader are elected whether call a separate component (**Configuration service**) or implement Leader election in cache cluster.

**Configuration service**: is distributed service for monitoring leaders, followers, failover. If some leader is not working as expected, Configuration service can promote follower to leader. Is used by Cache clients to discover all cache servers. e.g.: **Zookeeper** (used), Redis Sentinel.

A diagram of a service

Description automatically generated

**Points of failure**: Data replication asynchronously for better performance, we don’t want to wait until Leader server replicates data to all Followers. If Leader server got some data and failed before this data was replicated by any of the Followers, data is lost (cache miss).

**First priority of cache is to be fast.**

**Consistency:**

We built favors performance and availability over consistency.

Things lead to inconsistency:

1. We replicate data asynchronously to have a better performance. A get call processed by the master node, may return a different result than a get call for the same key but processed by a read replica.
2. Clients have a different list of cache servers. Cache servers may go down and go up again, and it is possible that a client write values that no other clients can read. It is **solved** by using **Synchronous Replication** to make sure a single view of the cache servers list, but this will increase latency and overall complexity of the system.

**Data expiration:**

LRU evicts data from cache when cache is full, but if cache is not full some items may sit for a long time, such items may **become stale**. To **solve** this issue we can introduce some **metadata** for a cache entry and include time-to-live attribute. Approaches for expired items are cleanup from cache:

1. We can passively expire an item, when some client tries to access it, and the item is found to be expired.
2. We can actively expire, when create a maintenance thread that runs at regular intervals and removes expired items. Usually, some probabilistic algorithms are used, when several random items are tested with every run.

**Local and remote (distributed) cache:**

If data is not found in the local cache, then call to the distributed cache. Better is used Local cache inside a Cache client.

Third party implementations: Guava cache.

**Security:**

Not expose cache servers directly to the internet, use **firewall** to restrict access to cache server ports and ensure only approved clients can access the cache.

Clients may also encrypt data before storing it in cache and decrypt it on the way out.

**Monitoring and Logging:**

Metrics: number of faults while calling the cache, latency, numbers of hits and misses, CPU and memory utilization on cache hosts, network I/O. Information who and when accessed the cache, key and status code.

**Cache Client:**

Maintain a **list of cache servers**, pick a shard to route a request to, handle a remote call and any potential failures, emit metrics.

Use **proxy** between **Cache Clients** and **Cache Serves** and will be responsible for picking a Cache shard. e.g.: **twemproxy project by Twitter**.

Another idea is to make Cache Servers responsible for picking a shard. Client sends request to a random Cache Server who applying **Consistent Hashing** (or other **Partitioning algorithm**) and redirects request to the shard that stores the data. e.g.: Redis cluster.

**Consistent hashing:**

Flaws:

1. Domino effect: appear when Cache Server dies and all of its load is transferred to the next server, this transfer might overload the next server, and then that server would fail, causing a chain reaction of failures.
2. The fact that Cache Servers do not split the circle evenly, some servers may reside close to each other and some may be far apart, causing uneven distribution of keys among the Cache Servers. Solutions for this problem are modifications of the consistent hashing algorithm:
   1. Simple idea is to add each server on the circle multiple times. Read Jump Hash algorithm (paper published by Google in 2014) or Proportional hashing (algorithm used by Yahoo! Vide platform).

**Summary:**

**Local Cache** has limited capacity and does not scale, we decided to run our LRU Cache as a standalone process.

We use a **Consistent Hashing** ring, a logical structure that helps to assign owners for ranges of cache keys.

We use **Cache Client**, that is responsible of routing requests for each key to a specific shard that stores data for this key.

**Memcached** is open-source high-performance distributed cache is built on top of these principles.

For better **Availability** we use master-slave replication for monitoring leader and read replicas and provide failover support we use a Configuration Service that is also used by Cache Clients for discovering Cache Servers.

A diagram of a computer component

Description automatically generated

Companies: Google

ref:

LRUCache.java

System Design Interview - Distributed Cache

<https://www.youtube.com/watch?v=iuqZvajTOyA>

* + implement LRU cache

Problem Statement:

Least Recently Used (LRU) is a common caching strategy. It defines the policy to evict elements from the cache to make room for new elements when the cache is full, meaning it discards the least recently used items first.

Let’s take an example of a cache that has a capacity of 4 elements. We cache elements 1, 2, 3 and 4.

A green square with black numbers

Description automatically generated

The diagram above represents the cache state after first access of all four elements. We now need to cache another element “5”.

A green square with black numbers

Description automatically generated

In LRU cache, we evict the least recently used element (in this case “1”) in case a new element needs to be cached. Now “2” is next in line to be evicted if a new element needs to be cached. Let’s see what happens when “2” is accessed again.

A green square with black numbers

Description automatically generated

Now “3” becomes the next in line to be evicted from the cache.

Hint:

Doubly linked list

Hashing

Think about evictions

<https://www.educative.io/implement-least-recently-used-cache>

* **Design Heavy Hitters**

Aka Top k most frequent items, find the most frequent search queries on Google, most viewed videos on Youtube, most played songs on Spotify, most shared or liked posts on Twitter, Facebook or Instagram.

System Design Interview - Distributed Cache

<https://www.youtube.com/watch?v=iuqZvajTOyA>

* **Design Autocomplete for a Search Engine, Type-ahead system**

Uses:

Google Search Autocompletion

* + **Ask clarifying questions**
* Key features:

Response Time: <100ms

Relevant/Context: the searching must be relevant to the prefix

Numbers/Sorted results: suggestions based on the popularity or ranking

* Functional requirements:

3 suggestions for each prefix to search

Sort suggestion based in the ranking

Ranking only used in valid words

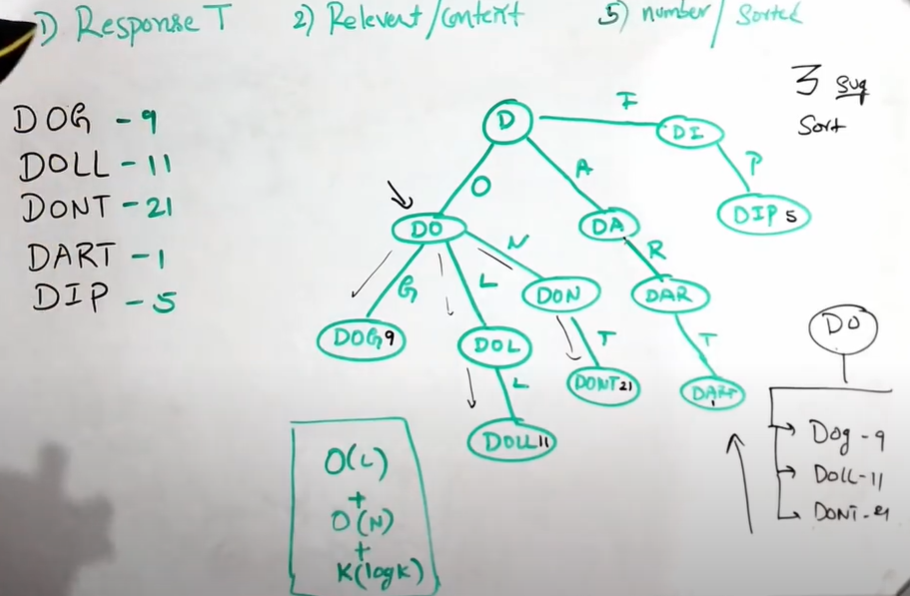
* Non-Functional requirements:
  + **Design high-level**
* Facts and Figures:

Summary:

Data structure to store the valid suggestions: **Trie**, O(L), L = length prefix, less space, use nodes (hash), every node has a prefix

**How to find prefix suggestions?**

Use ranking, is a value based on how many times user has searched that prefix, used only for valid suggestion.



valid suggestions ranking

DOG 9

DOLL 11

DONT 21

DART 1

DIP 5

DOLLAR 51

DOGE 15

Time Complexity:

O(L), L = length prefix to search

O(N), N = number of prefix nodes under a prefix we are trying to search

O(K log K), sort, K = number of words to sort

**Improvement**: precompute the top 3 suggestions by rankings (sorted) for each prefix (node) and save in a List

For D prefix: [DOLLAR, DONT, DOGE]

For DO prefix: [DOLLAR, DONT, DOGE]

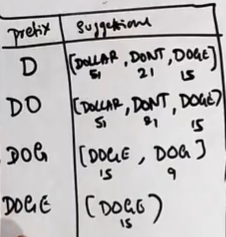
For DOG prefix: [DOGE, DOG]

For DOGE prefix: [DOGE]

**Persistent storage**:

Cache or database

In this case, use **Prefix Hash Table** that stores:

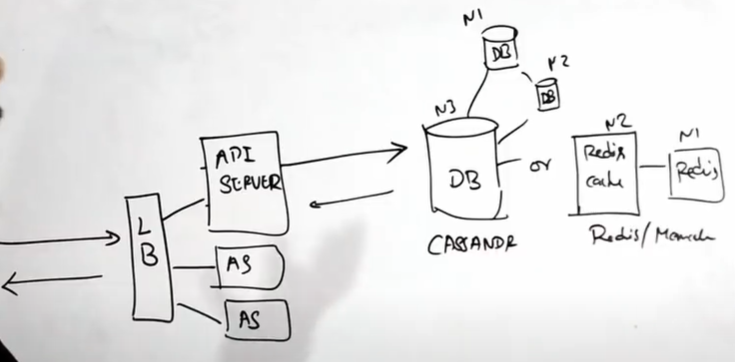


* **Architecture**:

API Server: to make a AJAX call

Database (Cassandra) or Redis Cache (faster, Redis or Memcache)

Load Balancer



For an efficient TYPE-Ahead System like Google, How to include **Contextual results** like Trending solution results:

1. User Base Search Suggestion: maintain a **table of history** of search keywords for a user
2. Trending Search Keys: compute the top trending search keywords, table has the top trending search keywords.

Build contextual and trending tables using prefix hash table.

Example:

User gives a prefix and follow these 3 queries:

1. All results from **Prefix Hash Table**
2. User System Results (**Base Search Suggestion**)
3. Trending Suggestions Results (**Trending Search Keys**)
   * Drill down on you design

How would you modify the frequency of the system without compromising availability or increasing latency.

Consider the fault tolerance of the system. How would you store the already built trie data structure so that in case of failure the system can be restored?

* + Bring it all together

Company: Google

refs:

Amazon interview question: System design / Architecture for auto suggestions | type ahead

<https://www.youtube.com/watch?v=xrYTjaK5QVM>

* **Design an API**

For Twitter API

* + Requirements

Reading and Posting tweets

Timeline and followers

* + Design

API Server

Rest API, SOAP, GraphQL (fetching and reading)

Authentication and Authorization

Pagination

Rate Limiting

ref:

System Design (TPM) Mock Interview: Twitter API

<https://www.youtube.com/watch?v=k-E4YdEs8qM>

* **Desing a messaging app**

Telegram

* + Requirements

Send and receive messages

refs:

System design mock interview: "Design WhatsApp or Telegram" (with ex-Google EM)

<https://www.youtube.com/watch?v=M6UZ7pVD-rQ>

* Design a File-Sharing System

Design Google Drive, DropBox, iCloud

ref:

Design file-sharing system like Google Drive / Dropbox (System design interview with EM)

<https://www.youtube.com/watch?v=4_qu1F9BXow>

* Design e-commerce

Design Amazon, eBay, FlipKart.

* Design a Ride-hailing or Delivery app

Design Uber. DoorDash

* **Object-oriented design interview questions**
* **References**
* System Design Interview questions

<https://www.codinginterview.com/system-design>

Go to System Design Question

The Top 10 Object-Oriented Design Interview Questions Developers Should Know

<https://hackernoon.com/the-top-10-object-oriented-design-interview-questions-developers-should-know-c7fc2e13ce39>

Top 10 Facebook system design interview questions

<https://www.educative.io/blog/facebook-system-design-interview>

Top 14 System Design interview questions for software engineers

<https://www.educative.io/blog/top-10-system-design-interview-questions>

The complete guide to the System Design Interview in 2023

<https://www.educative.io/blog/complete-guide-system-design-interview>

Basic System Design for Uber or Lyft | System Design Interview Prep

<https://www.youtube.com/watch?v=R_agd5qZ26Y>

<https://igotanoffer.com/blogs/tech/system-design-interviews>

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/design-and-deployment-of-tinyurl>

<https://www.educative.io/courses/grokking-modern-system-design-interview-for-engineers-managers/encoder-for-tinyurl>

Crack the system design interview

<https://tianpan.co/notes/2016-02-13-crack-the-system-design-interview>

* Mock Interviews:

<https://interviewing.io/mocks/google-system-design-distributed-databases>

* **System Design Interview Tips**

1. Communicate efficiently

Total time of interview last 45 min.

1. Scope the problem

Start clarifying the requirements with your interviewer and making clear assumptions if details are vague.

Focus on a crucial part of the system (Spotify – music recommendation engine).

1. Start drawing

Star in 15 min

1. Start with a simple design

Get to a working solution first, then iterate.

e.g.: We’ll need to use caching here but I’ll come back to that later.

5. Properly understand the problem

6. Practice, practice, practice!

Practice out loud with friends, experts, record yourself.

7. Explain your thinking

8. Get comfortable with the math

Get used to calculating queries per second and the storage capacity needed.

9. Use the drawing tool efficiently

10. Utilize a range of prep resources

Written and video content.

<https://igotanoffer.com/blogs/tech/system-design-interviews>

System Design Interviews: 10 Key Principles (with ex-Google EM)

<https://www.youtube.com/watch?v=8dG0qzNAVXI>

* How to prepare for system design interviews

1. Learn the concepts
2. Work through system design interview questions

Ask clarifying questions

Design high-level

Drill down on your design

Bring it all together

1. Practice answering example questions
2. Practice with someone else

Practice with ex-interviewers

Coach: <https://igotanoffer.com/en/interview-coaching/type/system-design-interview#buy-hours-anchor>

<https://igotanoffer.com/blogs/tech/system-design-interviews>

<https://igotanoffer.com/blogs/tech/system-design-interview-prep>

* Network protocols and proxies

<https://igotanoffer.com/blogs/tech/network-protocols-proxies-system-design-interview>

* TCP vs UDP connection

A screenshot of a computer

Description automatically generated

* Back-of-the-envelope calculations

<https://highscalability.com/google-pro-tip-use-back-of-the-envelope-calculations-to-choo/>

* + Powers of two table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name (symbol)** | **value 10x (decimal)** | **value 10x** | **value 2x**  **(binary)** | **value 2x** |
| kilobyte (kB) | 103 | 1 thousand | 210 | 1024 |
| megabyte (MB) | 106 | 1 million | 220 | 1,048,576 |
| gigabyte (GB) | 109 | 1 billion | 230 | 1,073,741,824 |
| terabyte (TB) | 1012 | 1 trillion | 240 | 1,099,511,627,776 |
| petabyte (PB) | 1015 |  |  |  |
| exabyte (EB) | 1018 |  |  |  |
| zettabyte (ZB) | 1021 |  |  |  |
| yottabyte (YB) | 1024 |  |  |  |

1 Kilobyte = 1000 bytes

1 Megabyte = 1000 Kilobytes

1 Gigabyte = 1000 Megabytes

1 Terabyte = 1000 Gigabytes

1 Petabyte = 1000 Terabytes

* + Latency numbers

A screenshot of a computer

Description automatically generated

Handy metrics based on numbers above:

Read sequentially from HDD: 30 MB/s

Read sequentially from 1 Gbps Ethernet: 100 MB/s

Read sequentially from SSD: 1 GB/s

Read sequentially from main memory: 4 GB/s

6-7 world-wide round trips per second

2,000 round trips per second within a data center

<https://github.com/donnemartin/system-design-primer?tab=readme-ov-file#appendix>