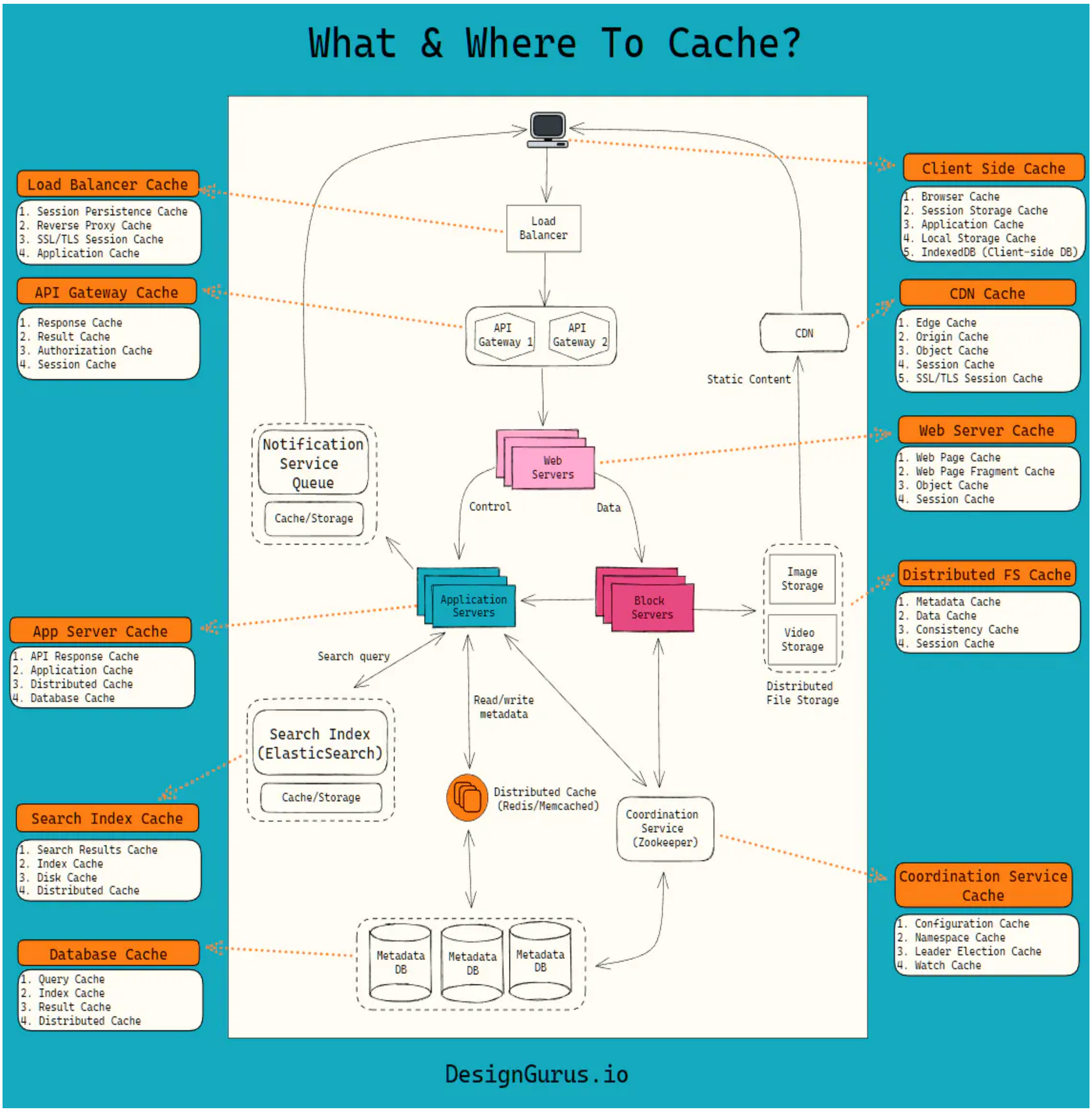
Caching

14 August 2023 11:23 PM

Caching is a crucial concept in system design that involves storing frequently accessed or computed data in a faster, closer-to-the-user storage location to improve performance and reduce the need to access the original source of the data. Caching is used to minimize latency and resource usage by serving data that has been previously retrieved, computed, or generated.



In the context of system design, caching can be implemented at various levels:

1. Client-Side Caching: Clients (user devices or browsers) can cache resources locally to reduce the need to request them from the server every time. This includes caching web pages, images, stylesheets, and JavaScript files.
2. Server-Side Caching: Servers can cache frequently requested data or the results of expensive computations. This can be done in-memory using technologies like Memcached or Redis, or on disk using file-based caching.
3. Database Caching: Databases can use caching to store frequently queried data, reducing the load on the database server and improving query response times. This can be done using database-specific caching mechanisms or through external caching layers.
4. Content Delivery Networks (CDNs): CDNs are a form of caching that involves distributing cached content to multiple geographically distributed servers. CDNs store copies of static assets like images, videos, and files, reducing the distance data needs to travel to reach users.
5. Application-Level Caching: Applications can cache data that is expensive to compute or fetch, such as API responses, search results, or processed data.

Benefits of Caching in System Design:

1. Improved Performance: Caching reduces the time it takes to retrieve and serve data, resulting in faster response times and better user experiences.
2. Reduced Load on Resources: Caching helps to offload backend servers, databases, and other resources by serving cached content instead of generating it from scratch.
3. Lower Latency: Cached data can be served quickly without the delay associated with fetching or computing the data in real-time.
4. Cost Savings: Caching can reduce the need for expensive computational or database resources, potentially saving operational costs.
5. Scalability: By reducing the load on backend systems, caching can contribute to improved system scalability and better handling of high traffic loads.

However, caching also presents challenges such as cache invalidation (ensuring that cached data remains up to date), cache coherence (ensuring consistency across caches), and managing cache size and eviction policies (determining which data to remove from the cache when it's full).

Effective caching strategies require careful consideration of the data access patterns, data volatility, and business requirements to strike the right balance between performance improvement and data consistency.

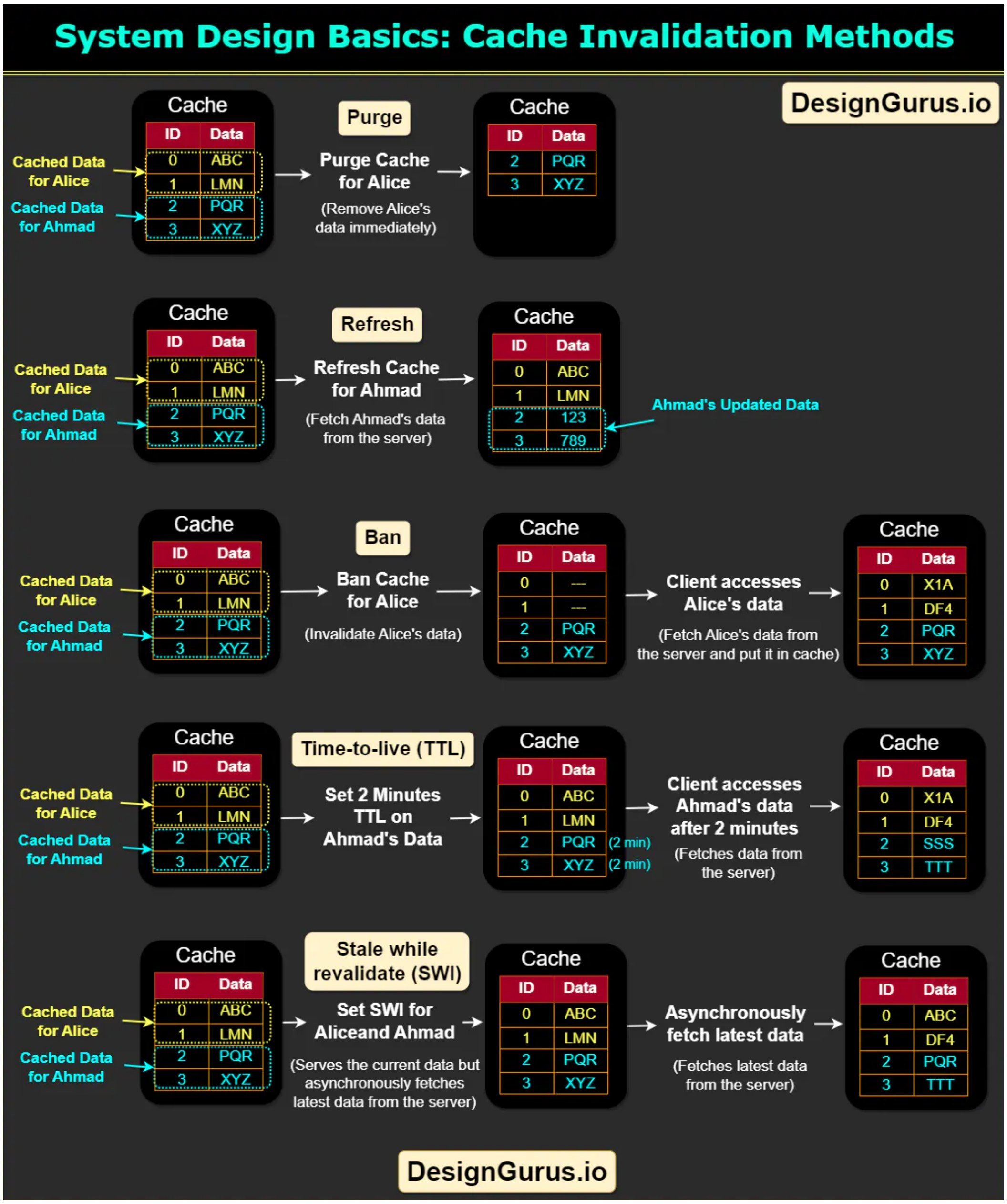
Cache invalidation :

It is the process of removing or updating cached data when the original data source is modified. It's a critical aspect of caching strategies to ensure that cached data remains accurate and up to date. In a distributed system, managing cache invalidation can be complex due to the need to coordinate cache updates across multiple nodes.

Here are some common strategies for cache invalidation:

1. Time-Based Expiration: Set an expiration time or time-to-live (TTL) for cached items. When the TTL expires, the cached data is considered invalid and is automatically removed from the cache. This approach is simple but may lead to serving stale data if the expiration time is too long.
2. Cache-Control Headers: When caching web resources like HTTP responses, servers can include Cache-Control headers to indicate how long a resource should be cached by the client. These headers provide instructions for caching behaviour.
3. Event-Based Invalidation: When the original data source changes, an event is triggered to notify the cache to invalidate or update the corresponding cached data. This approach requires a mechanism to detect and propagate data changes to the cache.
4. Write-Through Caching: In this strategy, updates to the data source trigger immediate updates to the cache. This ensures that the cache always holds the latest data, but it can add latency to write operations.
5. Write-Behind Caching: Updates are first made to the cache and then asynchronously propagated to the data source. This improves write performance but introduces the risk of temporarily inconsistent data between the cache and the source.
6. Versioning: Include version information with the cached data. When the data source changes, the version is updated, and caches can compare versions to determine if cached data is still valid.
7. Distributed Invalidation: In distributed systems, invalidation messages are propagated across nodes to ensure consistency. This can involve using a centralized invalidation service or a distributed pub/sub mechanism.
8. Manual Invalidation: Allow administrators or system operators to manually invalidate cached data. This can be useful for scenarios where data changes are infrequent or when changes are manually coordinated.

It's important to choose a cache invalidation strategy based on the specific requirements of your application. Different strategies have trade-offs in terms of complexity, performance impact, and consistency guarantees. Cache invalidation mechanisms need to be carefully designed to avoid issues like race conditions, data inconsistencies, and excessive cache invalidation traffic.



System Design Basics: Cache Invalidation Methods

DesignGurus.io

Purge

Cached Data for Alice
Cached Data for Ahmad

Cache

ID	Data
0	ABC
1	LMN
2	PQR
3	XYZ

Purge Cache for Alice
(Remove Alice's data immediately)

Cache

ID	Data
2	PQR
3	XYZ

Refresh

Cached Data for Alice
Cached Data for Ahmad

Cache

ID	Data
0	ABC
1	LMN
2	PQR
3	XYZ

Refresh Cache for Ahmad
(Fetch Ahmad's data from the server)

Cache

ID	Data
0	ABC
1	LMN
2	123
3	789

Ahmad's Updated Data

Ban

Cached Data for Alice
Cached Data for Ahmad

Cache

ID	Data
0	ABC
1	LMN
2	PQR
3	XYZ

Ban Cache for Alice
(Invalidate Alice's data)

Cache

ID	Data
0	---
1	---
2	PQR
3	XYZ

Client accesses Alice's data
(Fetch Alice's data from the server and put it in cache)

Cache

ID	Data
0	X1A
1	DF4
2	PQR
3	XYZ

Time-to-live (TTL)

Cached Data for Alice
Cached Data for Ahmad

Cache

ID	Data
0	ABC
1	LMN
2	PQR
3	XYZ

Set 2 Minutes TTL on Ahmad's Data

Cache

ID	Data	
0	ABC	
1	LMN	
2	PQR	(2 min)
3	XYZ	(2 min)

Client accesses Ahmad's data after 2 minutes
(Fetches data from the server)

Cache

ID	Data
0	X1A
1	DF4
2	SSS
3	TTT

Stale while revalidate (SWI)

Cached Data for Alice
Cached Data for Ahmad

Cache

ID	Data
0	ABC
1	LMN
2	PQR
3	XYZ

Set SWI for Aliceand Ahmad
(Serves the current data but asynchronously fetches latest data from the server)

Cache

ID	Data
0	ABC
1	LMN
2	PQR
3	XYZ

Asynchronously fetch latest data
(Fetches latest data from the server)

Cache

ID	Data
0	ABC
1	DF4
2	PQR
3	TTT

DesignGurus.io

Cache Patterns :

1. Cache-Aside (Lazy Loading) :
   - Applications explicitly request data from the cache.
   - If not in the cache, the app retrieves data from the source and then stores it in the cache for future use.
   - Simple to implement but can lead to cache misses if data isn't already cached.
2. Read-Through Cache:
   - When a cache miss occurs, the cache fetches the data from the underlying data source (e.g., database) and stores it in the cache before returning it to the client.
   - Reduces the load on the data source by storing frequently accessed data in the cache.
   - Can be combined with write-through caching to keep the cache and data source consistent.
3. Write-Through Cache :
   - When new data is written, the cache immediately writes it to both the cache and the source (database, for example).
   - Ensures data consistency between cache and source.
   - Can introduce write latency due to dual writes.
4. Write-Behind Cache :
   - Data is written to the cache first and then asynchronously written to the source.
   - Improves write performance by avoiding immediate writes to the source.
   - Potential for temporary inconsistency between cache and source.
5. Cache Invalidation :
   - Removing or updating cached data when it becomes outdated or changes in the source.
   - Ensures cache reflects the most accurate and up-to-date data.
   - Can be challenging to implement efficiently, as cache invalidation needs to be coordinated across multiple nodes.
6. Cache Preloading :
   - Populating the cache with data before it's requested by the application.

- Reduces cache misses and improves performance during peak times.
- Requires careful timing and management to ensure the cache remains relevant.

7. Cache Eviction Policies :
   - Strategies for choosing which data to remove from the cache when space is limited.
   - Common policies include removing the least recently used (LRU) data, the least frequently used (LFU) data, etc.
8. Cache-Through (Eager Loading) :
   - Applications request data from the cache, which fetches it from the source if not already cached.
   - Helps reduce cache misses compared to cache-aside.
   - Can add some complexity due to cache interaction.
9. Time-Based Expiration :
   - Setting a specific time after which cached data is considered invalid and needs to be refreshed from the source.
   - Simple to implement but might lead to serving outdated data if the expiration time is too long.
10. Cache Segmentation :
    - Dividing the cache into sections for different types of data or data with different access patterns.
    - Helps optimize cache space allocation and management.
11. Cache Federations :
    - Distributing cached data across multiple cache instances or nodes.
    - Useful for handling large datasets and high traffic by spreading the load.

Remember that choosing the right cache pattern depends on your application's specific requirements, usage patterns, and the level of data consistency needed between the cache and the source.

Cache eviction policies :

Following are some of the most common cache eviction policies:

1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
3. Least Recently Used (LRU): Discards the least recently used items first.
4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first. Assumes that items accessed more frequently are more likely to be accessed again. Less commonly used than LRU due to its counterintuitive nature.
5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.
6. Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

Cache Performance Metrics :

When implementing caching, it’s important to measure the performance of the cache to ensure that it is effective in reducing latency and improving system performance. Here are some of the most common cache performance metrics:

- Hit rate: The hit rate is the percentage of requests that are served by the cache without accessing the original source. A high hit rate indicates that the cache is effective in reducing the number of requests to the original source, while a low hit rate indicates that the cache may not be providing significant performance benefits.

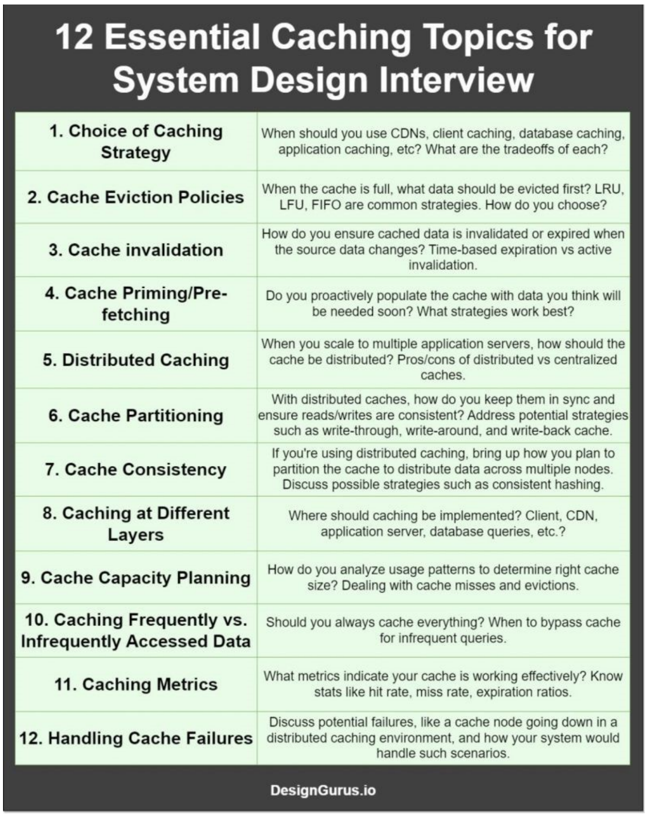
- Miss rate: The miss rate is the percentage of requests that are not served by the cache and need to be fetched from the original source. A high miss rate indicates that the cache may not be caching the right data or that the cache size may not be large enough to store all frequently accessed data.

- Cache size: The cache size is the amount of memory or storage allocated for the cache. The cache size can impact the hit rate and miss rate of the cache. A larger cache size can result in a higher hit rate, but it may also increase the cost and complexity of the caching solution.

- Cache latency: The cache latency is the time it takes to access data from the cache. A lower cache latency indicates that the cache is faster and more effective in reducing latency and improving system performance. The cache latency can be impacted by the caching technology used, the cache size, and the cache replacement and invalidation policies.

MISC :

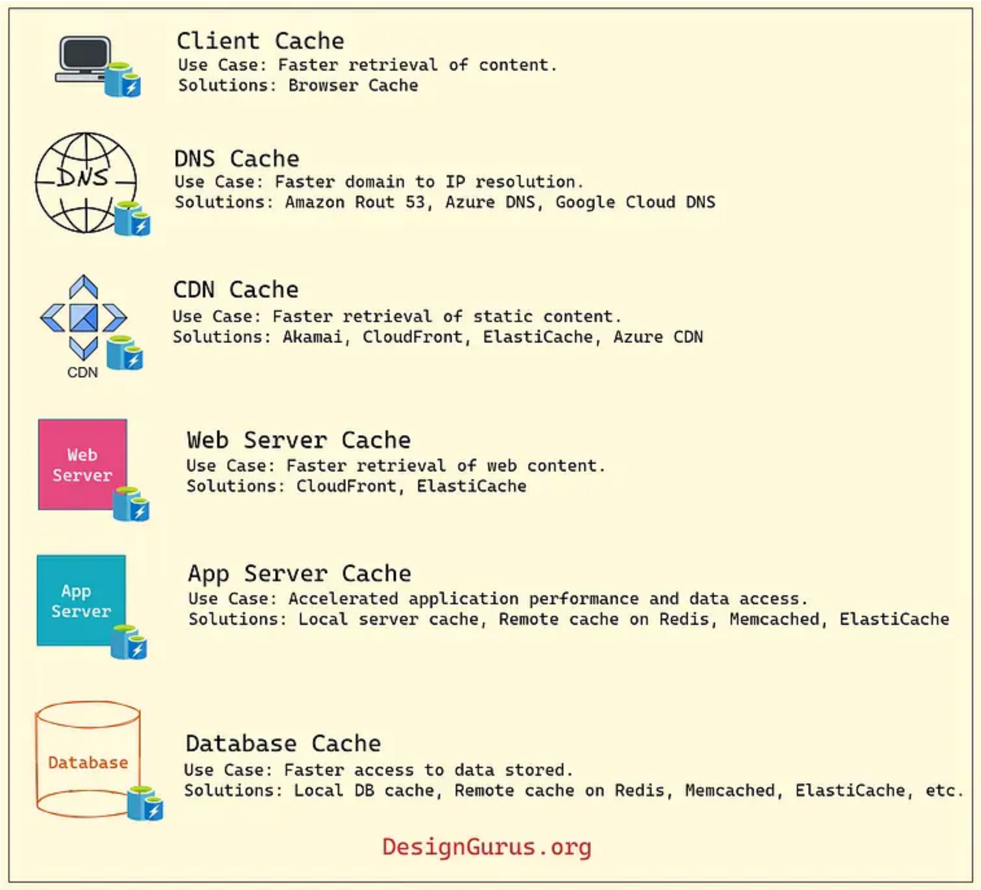
Refer : https://www.designgurus.io/blog/caching-system-design-interview



12 Essential Caching Topics for System Design Interview

1. Choice of Caching Strategy	When should you use CDNs, client caching, database caching, application caching, etc? What are the tradeoffs of each?
2. Cache Eviction Policies	When the cache is full, what data should be evicted first? LRU, LFU, FIFO are common strategies. How do you choose?
3. Cache invalidation	How do you ensure cached data is invalidated or expired when the source data changes? Time-based expiration vs active invalidation.
4. Cache Priming/Pre-fetching	Do you proactively populate the cache with data you think will be needed soon? What strategies work best?
5. Distributed Caching	When you scale to multiple application servers, how should the cache be distributed? Pros/cons of distributed vs centralized caches.
6. Cache Partitioning	With distributed caches, how do you keep them in sync and ensure reads/writes are consistent? Address potential strategies such as write-through, write-around, and write-back cache.
7. Cache Consistency	If you're using distributed caching, bring up how you plan to partition the cache to distribute data across multiple nodes. Discuss possible strategies such as consistent hashing.
8. Caching at Different Layers	Where should caching be implemented? Client, CDN, application server, database queries, etc.?
9. Cache Capacity Planning	How do you analyze usage patterns to determine right cache size? Dealing with cache misses and evictions.
10. Caching Frequently vs. Infrequently Accessed Data	Should you always cache everything? When to bypass cache for infrequent queries.
11. Caching Metrics	What metrics indicate your cache is working effectively? Know stats like hit rate, miss rate, expiration ratios.
12. Handling Cache Failures	Discuss potential failures, like a cache node going down in a distributed caching environment, and how your system would handle such scenarios.

DesignGurus.io



Client Cache
Use Case: Faster retrieval of content.
Solutions: Browser Cache

DNS Cache
Use Case: Faster domain to IP resolution.
Solutions: Amazon Rout 53, Azure DNS, Google Cloud DNS

CDN Cache
Use Case: Faster retrieval of static content.
Solutions: Akamai, CloudFront, ElastiCache, Azure CDN

Web Server Cache
Use Case: Faster retrieval of web content.
Solutions: CloudFront, ElastiCache

App Server Cache
Use Case: Accelerated application performance and data access.
Solutions: Local server cache, Remote cache on Redis, Memcached, ElastiCache

Database Cache
Use Case: Faster access to data stored.
Solutions: Local DB cache, Remote cache on Redis, Memcached, ElastiCache, etc.

DesignGurus.org