FullStack.Cafe - Kill Your Tech Interview

Q1: What is Caching? ☆

Topics: Caching

Answer:

In computing, a **cache** is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data’s primary storage location. *Caching allows you to efficiently reuse previously retrieved or computed data*.

Q2: Is Redis just a cache? ☆☆

Topics: Redis Caching

Answer:

Like a cache Redis offers:

- in memory key-value storage

But unlike a cash Redis:

- Supports multiple datatypes (strings, hashes, lists, sets, sorted sets, bitmaps, and hyperloglogs)
- It provides an ability to store cache data into physical storage (if needed).
- Supports pub-sub model
- Redis cache provides replication for high availability (master/slave)
- Supports ultra-fast lua-scripts. Its execution time equals to C commands execution.
- Can be shared across multiple instances of the application (instead of in-memory cache for each app instance)

Q3: What is Resultset Caching? ☆☆

Topics: Caching

Answer:

Resultset caching is storing the results of a database query along with the query in the application. Every time a web page generates a query, the applications checks whether the results are already cached, and if they are, pulls them from an in-memory data set instead. The application still has to render the page.

Q4: What is Cache Invalidation? ☆☆

Topics: Caching

Answer:

> There are only two hard things in Computer Science: cache invalidation and naming things.

> – Phil Karlton

HTTP caching is a solution for improving the performance of your web application. For lower load on the application and fastest response time, you want to cache content for a long period (TTL). But at the same time, you want your clients to see fresh (*validate the freshness*) content as soon as there is an update.

Cache invalidation gives you the best of both worlds: you can have very long TTLs, so when content changes little, it can be served from the cache because no requests to your application are required. At the same time, when data does change, that change is reflected without delay in the web representations.

Q5: What usually should be cached? ☆☆

Topics: Caching

Answer:

The results for the following processes are good candidates for caching:

- Long-running queries on databases,
- high-latency network requests (for external APIs),
- computation-intensive processing

Q6: Name some Cache Writing Strategies ☆☆

Topics: Caching

Answer:

There are two common strategies to write data in a cache:

1. **Pre-caching data**, for small pieces of data, usually during the application initialization, before any request.
2. **On-demand**, checking first if the requested data is in the cache (if the data is found, it is called a *cache hit*), using it, improving the performance of the application. Whenever the requested data has not been written to the cache (*cache miss*), the application will need to retrieve it from the slower source, then writing the results in the cache, thus saving time on subsequent requests for the same data.

Q7: What are some alternatives to Cache Invalidation? ☆☆☆

Topics: Caching

Answer:

There are three alternatives to cache invalidation.

- The first is to **expire** your cached content quickly by reducing its time to live (TTL). However, short TTLs cause a higher load on the application because content must be fetched from it more often. Moreover, reduced TTL does not guarantee that clients will have fresh content, especially if the content changes very rapidly as a result of client interactions with the application.
- The second alternative is to** validate the freshness** of cached content at every request. Again, this means more load on your application, even if you return early (for instance by using HEAD requests).
- The last resort is **to not cache volatile content** at all. While this guarantees the user always sees changes without delay, it obviously increases your application load even more.

Q8: Name some Cache Invalidation methods ☆☆☆

Topics: Caching

Answer:

- **Purge** - Removes content from *cache* immediately. When the client requests the data again, it is fetched from the application and stored in the cache. This method removes all variants of the cached content.

- **Refresh** - Fetches requested content from the application, even if cached content is available. The content previously stored in the cache is replaced with a new version from the application. This method affects only one variant of the cached content.

- **Ban** - A reference to the cached content is added to a blacklist (or ban list). Client requests are then checked against this blacklist, and if a request matches, new content is fetched from the application, returned to the client, and added to the cache. This method, *unlike purge*, does not immediately remove cached content from the cache. Instead, the cached content is updated after a client requests that specific information.

Q9: Compare caching at Business Layer vs Caching at Data Layer ☆☆☆☆

Topics: Caching

Answer:

- **Caching on the DAL is straightforward and simple**. Data access and persistence/storage layers are irresistibly natural places for caching. They're doing the I/Os, making them handy, easy place to insert caching. I daresay that almost every DAL or persistence layer will, as it matures, be given a caching function-if it isn't designed that way from the very start. Caching at the DAL/persistence layer risk having the "cold" reference data sitting there, pointlessly occupying X mb of cache and displacing some information that will, in fact, be intensively used in just a minute. Even the best cache managers are dealing with scant knowledge of the higher level data structures and connections, and little insight as to what operations are coming soon, so they fall back to guesstimation algorithms.
- **Caching in the business is flexible and potentially more efficient**. Application or business-layer caching requires inserting *cache management operations* or hints in the middle of other business logic, which makes the business code more complex. But the tradeoff is: Having more knowledge of how macro-level data is structured and what operations are coming up, it has a much better opportunity to approximate optimal ("clairvoyant" or "Bélády Min") *caching efficiency*.

Whether inserting cache management responsibility into business/application code makes sense is a judgment call, and will vary by applications. Lower complexity encourages higher correctness and reliability, and faster time-to-market. That is often considered a great tradeoff - less perfect caching, but better-quality, more timely business code.

Q10: What are Cache Replacement (or Eviction Policy) algorithms? ☆☆☆☆

Topics: Caching

Answer:

In computing, **cache algorithms **(also frequently called cache replacement algorithms or cache replacement policies or cache eviction policies) are optimizing instructions, or algorithms, that a computer program or a hardware-maintained structure can utilize in order to manage a cache of information stored on the computer. Caching improves performance by keeping recent or often-used data items in memory locations that are faster or computationally cheaper to access than normal memory stores. **When the cache is full, the algorithm must choose which items to discard to make room for the new ones.**

Q11: What are some disadvantages of Cache Invalidation? ☆☆☆☆

Topics: Caching

Answer:

Invalidating cached web representations when their underlying data changes can be very simple. For instance, invalidate `/articles/123` when article 123 is updated. However, data usually is represented not in one but in multiple representations. Article 123 could also be represented on the articles index (`/articles`), the list of articles in the current year (`/articles/current`) and in search results (`/search?name=123`). In this case, when article 123 is changed, a lot more is involved in invalidating all of its representations. In other words, **invalidation adds a layer of complexity to your application**.

Summary:

- Using invalidation to transfer new content can be difficult when invalidating multiple objects.
- Invalidating multiple representations adds a level of complexity to the application.
- Cache invalidation must be carried out through a caching proxy; these requests can impact performance of the caching proxy, causing information to be transferred at a slower rate to clients.

Q12: Why is Cache Invalidation considered difficult? ☆☆☆☆

Topics: Caching

Answer:

The **non-determinism** is why cache invalidation are unique and intractably hard problem in computer science.

- Computers can perfectly solve **deterministic** problems. But they can't predict when to *invalidate* a cache because, ultimately, we, the humans who design and build computational processes, shall agree on when a cache needs to be invalidated. The hard and unsolvable problem becomes: *how up-to-date do you really need data to be and when to change (or remove) it*?

- Another problem a cache is (often by nature) much smaller compared to the overall amount of data that needs to be stored and if you just keep adding and adding elements to your cache, it becomes a full copy of your data. Respectively, you *run out of memory* quickly.

Basically it's difficult to achieve a desirable balance between stale objects stinking up your cache, and frequent unnecessary refreshes of unchanged objects considering limitation of the cache size.

Q13: What is the difference between Cache replacement vs Cache invalidation? ☆☆☆☆

Topics: Caching

Answer:

- Frequently, the cache has a fixed limited size. So, whenever you need to write in the cache (commonly, after a_ cache miss_), you will need to determine if the data you retrieved from the slower source should or should not be written in the cache and, if the size limit was reached, what data would need to be removed from it. That process is called **Cache replacement** strategy (or policy). Some examples of Cache replacement strategies are:

  - **Least recently used (LRU)** - Discards the least recently used items first.
  - **Least-frequently used (LFU)** - Counts how often an item is needed. Those that are used least often are discarded first.

- **Cache invalidation** is the process of determining if a piece of data in the cache should or should not be used to service subsequent requests. The most common strategies for cache invalidation are:

  - **Expiration time**, where the application knows how long the data will be valid. After this time, the data should be removed from the cache causing a “cache miss” in a subsequent request;
  - **Freshness caching verification**, where the application executes a lightweight procedure to determine if the data is still valid every time the data is retrieved. The downside of this alternative is that it produces some execution overhead;
  - **Active application invalidation**, where the application actively invalidates the data in the cache, normally when some state change is identified.

Q14: Explain what is Cache Stampede ☆☆☆☆

Topics: Caching Software Architecture

Answer:

A **cache stampede (or cache miss storm)** is a type of cascading failure that can occur when massively parallel computingsystems with caching mechanisms come under very high load. This behaviour is sometimes also called **dog-piling**.

Under very heavy load, when the cached version of the page (or resource) expires, there may be sufficient concurrency in the server farm that multiple threads of execution will all attempt to render the content (or get a resource) of that page simultaneously.

To give a concrete example, assume the page in consideration takes 3 seconds to render and we have a traffic of 10 requests per second. Then, when the cached page expires, we have 30 processes simultaneously recomputing the rendering of the page and updating the cache with the rendered page.

Consider:

```
function fetch(key, ttl) {
  value ← cache_read(key)
  if (!value) {
    value ← recompute_value()
    cache_write(key, value, ttl)
  }
  return value
}
```

If the function `recompute_value()` takes a long time and the key is accessed frequently, many processes will simultaneously call `recompute\_value()` upon expiration of the cache value.

In typical web applications, the function `recompute_value()` may query a database, access other services, or perform some complicated operation (which is why this particular computation is being cached in the first place). When the request rate is high, the database (or any other shared resource) will suffer from an overload of requests/queries, which may in turn cause a system collapse.

Q15: When to use LRU vs LFU Cache Replacement algorithms? ☆☆☆☆☆

Topics: Caching

Answer:

- **LRU (Least Recently Used)** is good when you are pretty sure that the user will more often access the most recent items, and never or rarely return to the old ones. An example: a general usage of an e-mail client. In most cases, the users are constantly accessing the most recent mails. They read them, postpone them, return back in a few minutes, hours or days, etc. They can find themselves searching for a mail they received two years ago, but it happens less frequently than accessing mails they received the last two hours.

- On the other hand, LRU makes no sense in the context where the user will access some items much more frequently than others. An example: I frequently listen to the music I like, and it can happen that on 400 songs, I would listen the same five at least once per week, while I will listen at most once per year 100 songs I don't like too much. In this case, **LFU (Least Frequently Used)** is much more appropriate.

Q16: What are best practices for caching paginated results whose ordering/properties can change? ☆☆☆☆☆

Topics: Caching Software Architecture

Answer:

It seems what you need is a wrapper for all the parameters that define a page (say, `pageNumber`, `pageSize`, `sortType`, `totalCount`, etc.) and use this `DataRequest` object as the key for your caching mechanism. From this point you have a number of options to handle the *cache invalidation*:

- Implement some sort of timeout mechanism (TTL) to refresh the cache (based on how often the data changes).
- Have a listener that checks database changes and updates the cache based the above parameters (data refresh by server intent).
- If the changes are done by the same process, you can always mark the cache as outdated with every change and check this flag when a page is requested (data refresh by client intent).

The first two might involve a scheduler mechanism to trigger on some interval or based on an event. The last one might be the simpler if you have a single data access point. Lastly, it can quickly become an overly complicated algorithm that outweighs the benefits, so be sure the gain in performance justify the complexity of the algorithm.

Q17: Cache miss-storm: Dealing with concurrency when caching invalidates for high-traffic sites ☆☆☆☆☆

Topics: Caching Software Architecture

Problem:

For a high traffic website, there is a method (say `getItems()`) that gets called frequently. To prevent going to the DB each time, the result is cached. However, thousands of users may be trying to access the cache at the same time, and so locking the resource would not be a good idea, because if the cache has expired, the call is made to the DB, and all the users would have to wait for the DB to respond. What would be a good strategy to deal with this situation so that users don't have to wait?

Solution:

The problem is the so-called `Cache miss-storm (Cache Stampede or Dogpile)` - a scenario in which a lot of users trigger regeneration of the cache, hitting in this way the DB.

To prevent this, first you have to set *soft* and *hard* expiration date. Lets say the hard expiration date is 1 day, and the soft 1 hour. The hard is one actually set in the cache server, the soft is in the cache value itself (or in another key in the cache server). The application reads from cache, sees that the soft time has expired, set the soft time 1 hour ahead and hits the database. In this way the next request will see the already updated time and won't trigger the cache update - it will possibly read stale data, but the data itself will be in the process of regeneration.

Next point is: you should have procedure for *cache warm-up*, e.g. instead of user triggering cache update, a process in your application to pre-populate the new data.

The worst case scenario is e.g. restarting the cache server, when you don't have any data. In this case you should fill cache as fast as possible and there's where a warm-up procedure may play vital role. Even if you don't have a value in the cache, it would be a good strategy to "lock" the cache (mark it as being updated), allow only one query to the database, and handle in the application by requesting the resource again after a given timeout.

Q18: Name some Cache Stampede mitigation techniques ☆☆☆☆☆

Topics: Caching

Answer:

- **Locking** - a process will attempt to acquire the lock for the cache key and recompute it only if it acquires it. If implemented properly, locking can prevent stampedes altogether, but requires an extra write for the locking mechanism. Apart from doubling the number of writes, the main drawback is a correct implementation of the locking mechanism which also takes care of edge cases including failure of the process acquiring the lock, tuning of a time-to-live for the lock, race-conditions, and so on.

- **External recomputation** - This solution moves the recomputation of the cache value from the processes needing it to an external process. This approach requires one more moving part - the external process - that needs to be maintained and monitored. The recomputation of the external process can be triggered in different ways:

    - When the cache value approaches its expiration
    - Periodically
    - When a process needing the value encounters a cache miss

- **Probabilistic early expiration** - With this approach, each process may recompute the cache value before its expiration by making an independent probabilistic decision, where the probability of performing the early recomputation increases as we get closer to the expiration of the value. Since the probabilistic decision is made independently by each process, the effect of the stampede is mitigated as fewer processes will expire at the same time.