Caching

Background

In all technology (not just distributed systems), caching is a way of providing faster reads to the end client. It tends to do so by using some faster form of data storage (in distributed systems this generally means storing data in RAM, not disk).

Caching accomplishes three main objectives:

- Storing precomputed values (either previous calls to a database or aggregations)
- Fewer network calls to the database (which is probably located physically further away)
- Fewer load on the database, as it probably already is under plenty

Tradeoffs of Caching

Pros:

- Faster reads
  - However if we attempt to find our data in the cache and it is rarely there, cache slows things down
    - This can happen because of thrashing (value that we want in cache keeps getting replaced right before we want to read it since cache size is too small)
    - Can also happen because of poor cache eviction policies
- Potentially faster writes

Cons:

- Increased complexity on each write

Cache Eviction Policies

- First in first out
- Last in first out
- Least recently used
  - Out of these this is the most practical one for an interview answer, discard the piece of data that was accessed longest ago
  - If it comes up, this is implemented with a doubly linked list and a hashmap
- Least frequently used
- Random replacement
- Sliding window
  - Can work very well but still being researched, probably not worth mentioning in this video

Application Server and Global Cache

Two possible options for implementing cache are either to use memory on our application servers as cache or instead to use standalone nodes as dedicated cache servers.

Application Server Cache:

- No extra network calls required from server to hit cache

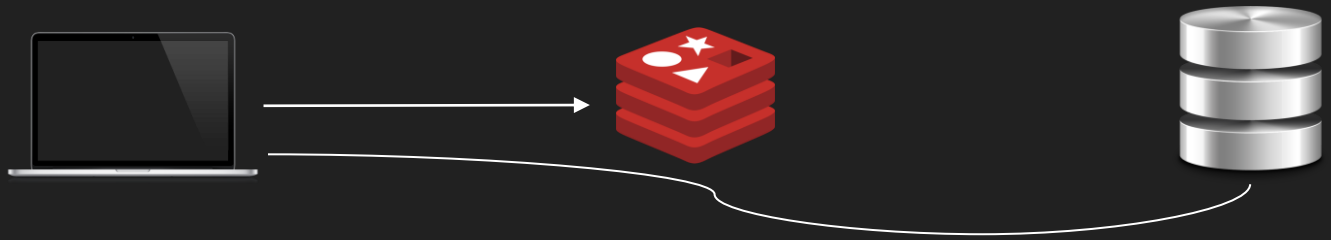
Global Cache (Generally better):

- Can scale independently of number of application servers and does not crash with application server
- Can be accessed by any application server
- Can be replicated and partitioned, we will see these in play with Redis and Memcached

Keeping Cache Updated On Writes

- Write through
- Write around
- Write back

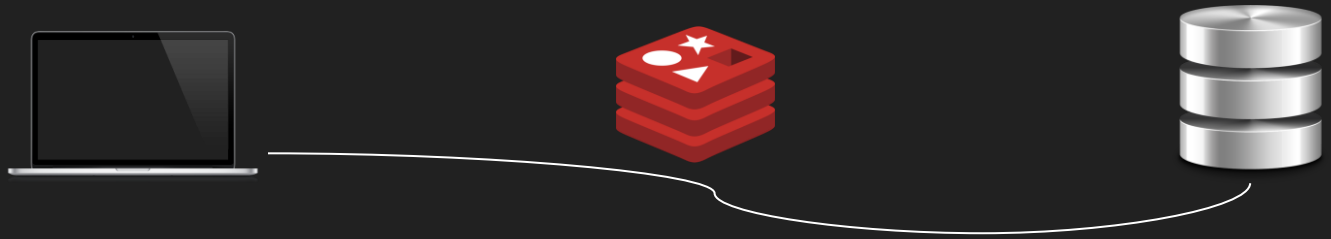
Write Through Cache



Write both the cache and database at the same time, in parallel!

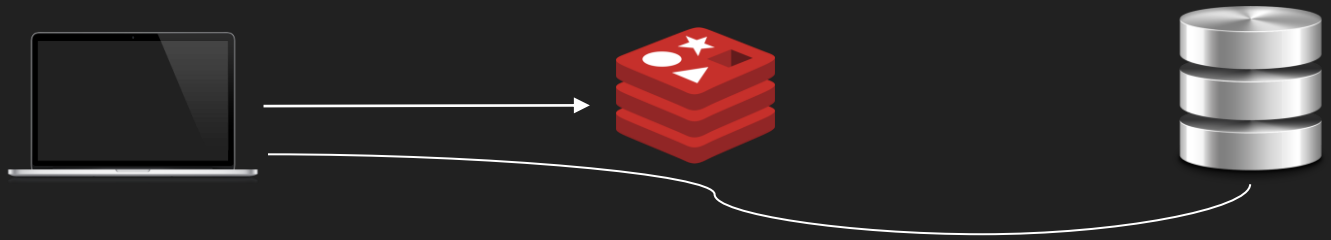
But what if only one update works? Depending on how much it matters, we may need a 2 phase commit protocol, which really slows things down.

Write Around Cache



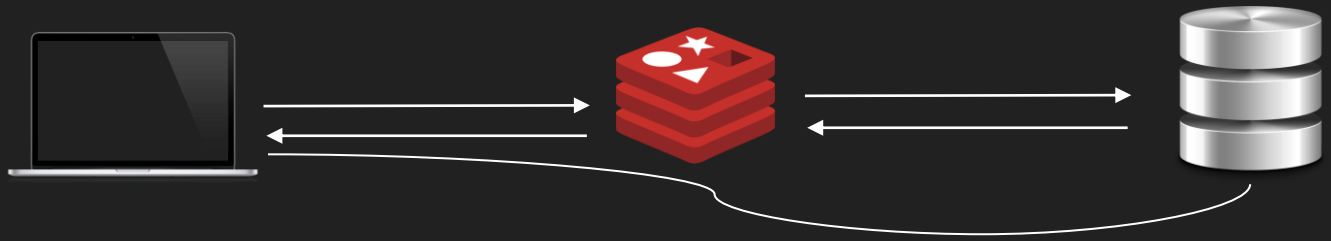
1) Write the database, and also invalidate the current cache data for that key

Write Around Cache



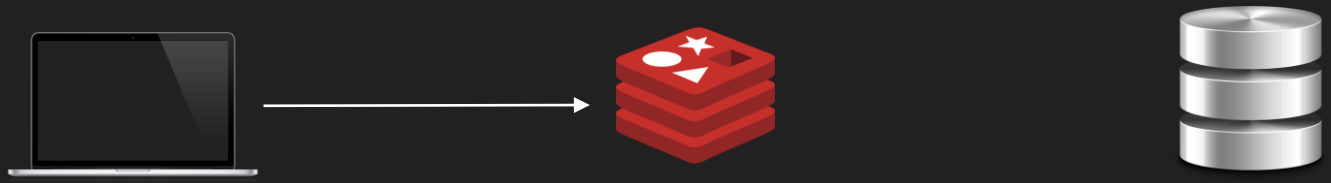
1) Write the database, and also invalidate the current cache data for that key
2) Client requests data for the given key

Write Around Cache



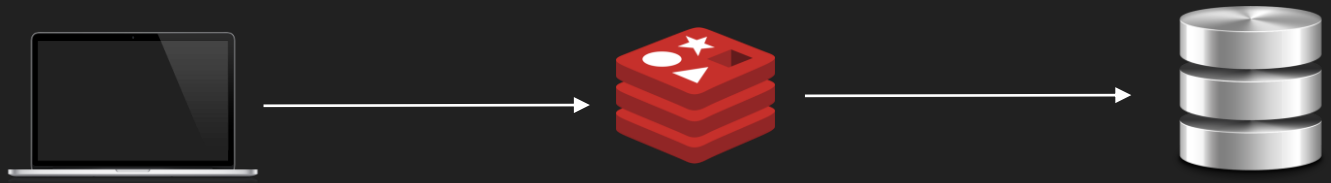
1) Write the database, and also invalidate the current cache data for that key
2) Client requests data for the given key
3) Cache requests data from the database for said key, and returns it to the client

Write Back Cache



1) Write data only to the cache

Write Back Cache



1) Write data only to the cache
2) Eventually, write data from cache to the database, possibly many writes at once so as to reduce network calls

While this approach has the fastest write speed for the client, it obviously does not work for data where we can not deal with eventual consistency such as password changes. In this scenario, such a write policy is unacceptable.

Write Back Cache with Cache Consistency

What if we did want to have fast writes using a write back cache, but also want other users to be able to see the changes held on the cache when they want to read the modified object?

We could use a distributed lock service (such as ZooKeeper), where a cache server needs to hold a lock in order to modify that object. When another server wants to read the object, the lock server informs the cache server that it must release its lock on the object, and write back the modifications to the database before doing so.

This obviously has the tradeoff that only one server can hold the lock at a time and therefore ruins the ability to have high write throughput from many servers.

Caching in Practice

In actual systems, cache servers treated very similarly to databases!

- Replication for fault tolerance
- Partitioning for large datasets
- Coordination services for cluster management
- Secondary indexes (not just a hash table?)

In the following videos, we will see actual existing technologies that have built out distributed caching solutions using all of these principles!

Caching Conclusion

Caching - Improves read latency, and depending on design, possibly write latency

Generally better to decouple cache from application servers for ability to independently scale and communicate with all servers (at the cost of more network requests).

As for writes:

- Write back for minimal write latency, but at the cost of eventual consistency
- Write through for maximum consistency, but may have to use distributed transactions
- Write around for simple implementation, but at the cost of an initial cache miss