CLOUD COMPUTING APPLICATIONS

Caching Technical Concepts

Prof. Reza Farivar

Topics

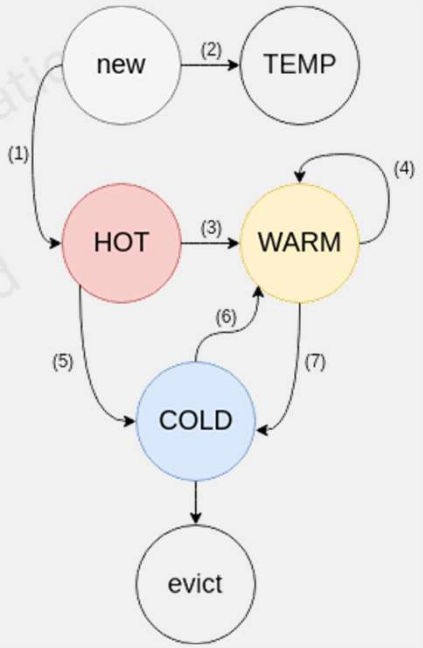
- Cache Replacement Policy
  - LRU, FIFO, etc.
- Cache Writing / Updating Policy
  - Cache Aside, Write-through, Write-back
- Cache Coherence

Reading from Cache

- A client first checks to see if a data piece is available in a cache
- Cache hit: the desired data is in the cache
- Cache miss: the desired data is not in the cache
  - In this case, the client needs to go to the "slow/big" source of data
  - Once data is retrieved, it is also copied in the cache, ready for next accesses (principle of locality)
    - But **where** in the cache should we put this new data?

Cache Replacement Policy

- Also known as:
  - Cache Eviction Policy
  - Cache Invalidation Algorithm
- Least Recently Used (LRU)
  - Replaces the oldest entry in the cache
  - Memcached uses segmented LRU
- Time-aware Least Recently Used (TLRU)
  - TTU: Time To Use
- Least Frequently Used (LFU)
- First in First Out (FIFO)
- Many others
  - LIFO, FILO, MRU, PLRU,



Cache Replacement

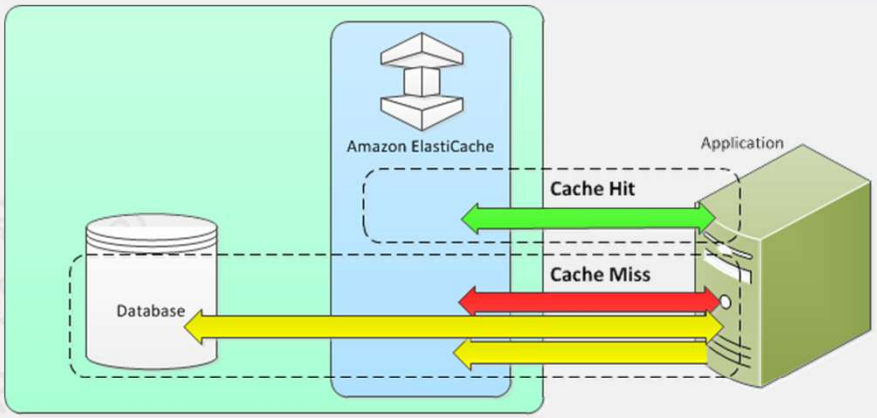
- maintain consistency between caches and the source of truth such as the database through cache invalidation
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Time to Live (TTL): Expiration time for records in the cache
  - Eventual consistency for coherence problem

Cache Writing/Updating Policies

- Whenever a new piece of data is created and needs to be stored, we also have a question regarding updating the caches
  - Cache Aside
    - Lazy Loading
  - Write-Through
    - Write is done synchronously both to the cache and to the backing store.
  - Write – Back
    - Write-Behind
    - Lazy Writing

Cache Aside (aka lazy loading)

- The ***application*** is responsible for reading and writing from storage
  - The cache does not interact with storage directly
- The ***application*** does the following:
  - Look for entry in cache, resulting in a cache miss
  - Load entry from the database
  - Add entry to cache
  - Return entry
- Memcached is usually used in this manner



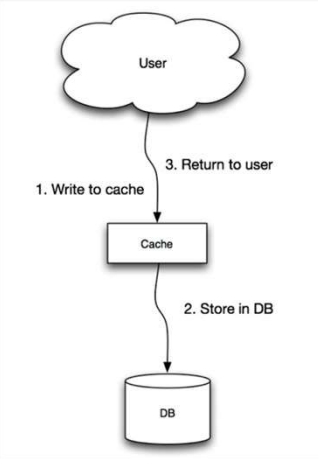

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return user
```

Disadvantages of Cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

Write-through writing/updating policy

- The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:
  - Application adds/updates entry in cache
  - Cache synchronously writes entry to data store
  - Return



Application code:

```
set_user(12345, {"foo":"bar"})
```

Cache code:

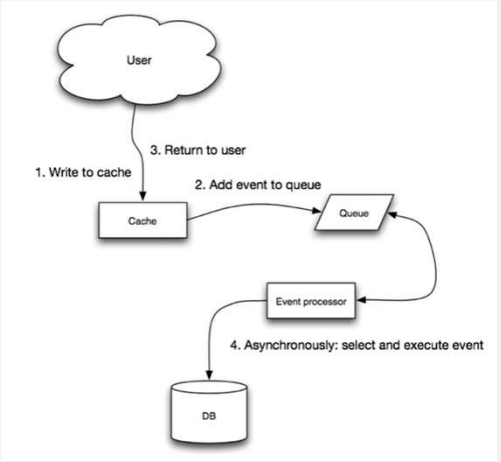
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

Disadvantages of Write-Through

- Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast
  - Users are generally more tolerant of latency when updating data than reading data.
- Data in the cache is never stale ☺
- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database
  - Cache-aside in conjunction with write through can mitigate this issue
- ***Cache Churn:*** Most data written might never be read → waste of resource
  - Can be minimized with a TTL

Write-back (Write-Behind) writing/updating policy

- Initially, writing is done only to the cache.
- The write to the backing store is postponed until the modified content is about to be replaced by another cache block.
  - Asynchronously write entry to the data store, improving write performance
- Write-back cache is more complex to implement, since it needs to track which of its locations have been written over, and mark them as *dirty* for later writing to the backing store
  - ***Lazy Write***: Data in these dirty locations are written back to the backing store only when they are evicted from the cache
- a read miss in a write-back cache (which requires a block to be replaced by another) will often require two memory accesses to service: one to write the replaced data from the cache back to the store, and then one to retrieve the needed data.



Img source: Scalability, Availability & Stability Patterns
Jonas Bonér

Write-around

- Writing is only done to the underlying data source
- **Advantage:** Good for not flooding the cache with data that may not subsequently be re-read
- **Disadvsntage:** Reading recently written data will result in a cache miss (and so a higher latency) because the data can only be read from the slower backing store
- The write-around policy is good for applications that don't frequently re-read recently written data
  - This will result in lower write latency but higher read latency which is an acceptable trade-off for these scenarios

Write Allocation Policies

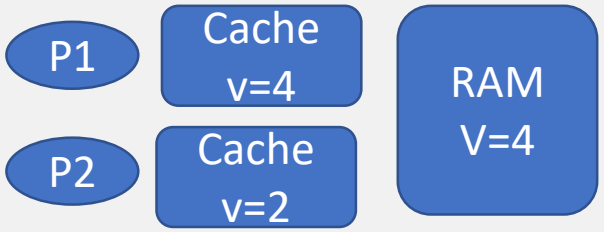
- Since no data is returned to the requester on write operations, a decision needs to be made on write misses, whether or not data would be loaded into the cache. This is defined by these two approaches:
  - *Write allocate* (also called *fetch on write*): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
  - *No-write allocate* (also called *write-no-allocate* or *write around*): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.

Write Allocation Policies

- A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.
- A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, since they still need to be written directly to the backing store.

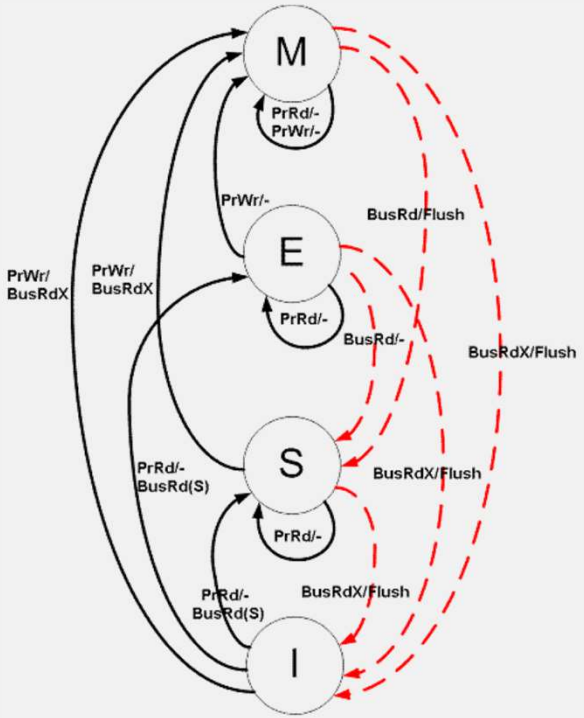
Cache Coherency

- *Note: do not confuse with Consistency!*
- Whenever cache is distributed, ***with the same data in more than one place***, we have the coherency problem
  - Write Propagation: Changes to the data in any cache must be propagated to other copies (of that value) in the peer caches
  - Transaction Serialization: Reads/Writes to a single memory location must be seen by all processors in the same order
- Coherence Protocols
  - **Write-invalidate**: When a write operation is observed to a location that a cache has a copy of, the cache controller ***invalidates*** its own copy
    - **MESI** protocol
  - **Write-Update**: When a write operation is observed to a location that a cache has a copy of, the cache controller ***updates*** its own copy



MESI protocol

- Aka the Illinois Protocol
- Supports “write-back-caches”
- Uses Cache-to-cache transfer
- States:
  - Modified
  - Exclusive
  - Shared
  - Invalid



Coherency Mechanisms

- Snooping
  - Send all requests for data to all processors
  - Processors “snoop” to see if they have a local copy
  - Requires broadcast
  - Works well with a “bus” → CPUs
  - Usually implemented with state machines
- Directory-based
  - Keep track of what is being share in a centralized location
    - In reality, every block in every cache
  - Send point-to-point requests
  - Scales better than snooping
  - Lcache, WP-Lcache
- TTL: Eventually consistent caches
  - Main mechanism for DNS caching records