Optimization in High-Performance Computing

Algorithms and Data Structures

Khanh Nguyen
University of the Cumberlands
knguyen19181@ucumberlands.edu

1 Overview

Caching[1] is an optimization technique that we can use in our applications to keep recent or often-used data in memory locations that are faster or computationally cheaper to access than their source.

Caching has a proven track record of helping companies of all sizes improve application performance. In fact, advanced caching services provide much more than just a quick key-value cache. But in order to understand what caching is all about and where it is headed, let's first review the basics.

The three primary types of caching are:

- **Key-value caching:** The key is a unique identifier for the data, and the value is the data itself.
- **Object caching:** stores entire objects instead of just key-value pairs, and is usually employed for storing complex data structures
- **Distributed caching:** stores data across multiple servers, offering enhanced scalability and redundancy

2 Cache

2.1 Cache Benefits

Effective caching aids both content consumers and content providers. Some of the benefits that caching brings to content delivery are:

- **Decreased network costs:** Content can be cached at various points in the network path between the content consumer and content origin. When the content is cached closer to the consumer, requests will not cause much additional network activity beyond the cache.
- **Improved responsiveness:** Caching enables content to be retrieved faster because an entire network round trip is unnecessary. Caches maintained close to the user, like the browser cache, can make this retrieval nearly instantaneous.
- **Increased performance on the same hardware:** For the server where the content originated, more performance can be squeezed from the same hardware by allowing aggressive caching. The content owner can leverage the powerful servers along the delivery path to take the brunt of certain content loads.
- **Availability of content during network interruptions:** With certain policies, caching can be used to serve content to end users even when it may be unavailable for short periods of time from the origin servers.

Preprint. Under review.

2.2 Cache Strategies

Different caching solutions [2] support different implementations, each with its pros and cons. Data architects and software engineers must, therefore, consider which solution supports these out of the box and the design that best suits the team's needs prior to implementation.

- **Read-through:** When looking for a specific ID, the application will hit the cache first; if the data is missing, then the cache will be responsible for pulling the data from the main database and updating it.

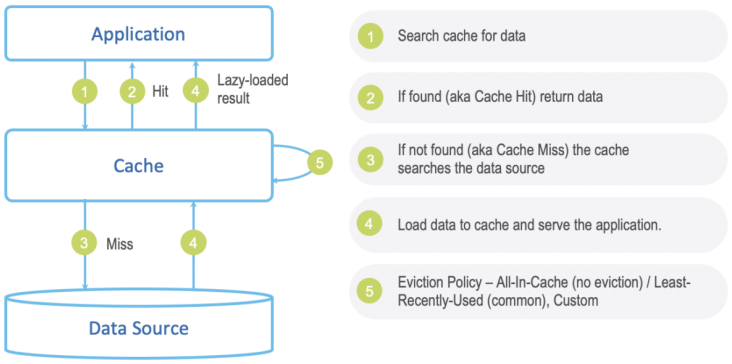

Figure 1: Read Through Cache

- **Write-Through:** The write-through caching pattern is similar to the read-through in that instead of writing directly to the main database, the application writes to the cache as if it were the main database. The cache then writes to the main database. This write to the cache is done synchronously and, in turn, impacts the write latency

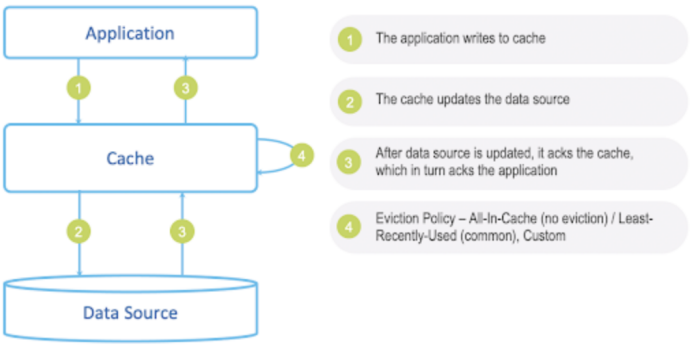

Figure 2: Write Through Cache

- **Write-Behind:** Write-behind caching takes a similar approach to write-through, except that writing to the main database is done asynchronously. This means write latency is only limited to the cache latency. Because it is volatile, however, there is a risk of data loss if the cache is down.

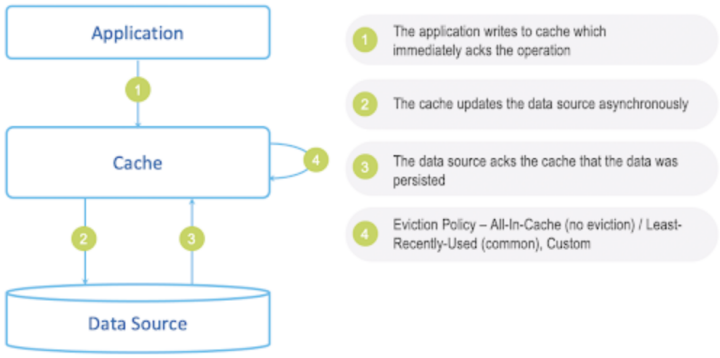

Figure 3: Write Behind Cache

2.3 Cache Eviction Policy

Whenever there is a key that does not exist in the cache, the service will add the key with the corresponding value to the cache. The cache will grow indefinitely, and it can lead to out-of-memory (OOM). For Cache applications such as Redis, it will store the "least-frequent used" in the disk when memory runs out (page swapping) [3]. Therefore, a cache eviction policy is needed to manage the cached information and choose which items to discard to make room for new ones. There are several different strategies to keep the cache from growing past its maximum size, and each has its use case:

Strategy	Eviction Policy	Use case
First In, First Out (FIFO)	Evicts the oldest of the entries	Newer entries are most likely to be reused
Last In, First Out (LIFO)	Evict the latest of the entries	Older entries are most likely to be reused
Least Recently Used (LRU)	Evict the least recently used entry	Recently used entries are most likely to be reused.
Most Recently Used (MRU)	Evict the most recently used entry	The least used entries are most likely to be reused.
Least Frequently Used (LFU)	Evict the least often accessed entry	Entries with a lot of hits are more likely to be reused.

Figure 4: Cache Eviction Policy

In the following code implementation, we will use the least recently used (LRU) [4] to demonstrate cache capabilities. Before that, we would need to understand how it works. A cache implemented using the LRU strategy organizes its items in order of use. Every time we access an entry, the LRU algorithm will move it to the top of the cache. This way, the algorithm can quickly identify the entry that's gone unused the longest by looking at the bottom of the list.

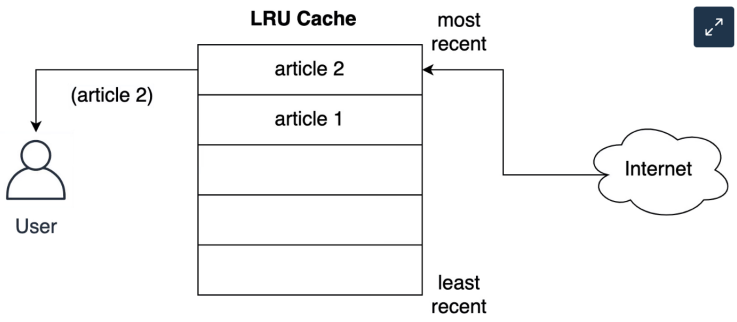

Figure 5: Least recently used strategy

The LRU strategy assumes that the more recently an object has been used, the more likely it will be needed in the future, so it tries to keep that object in the cache for the longest time.

One way to implement an LRU cache in Python is to use a combination of a doubly linked list and a hash map. The head element of the doubly linked list would point to the most recently used entry, and the tail would point to the least recently used entry.

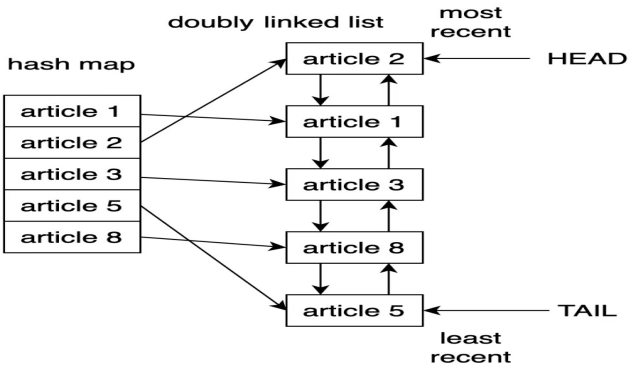

Figure 6: Cache with Linked List and Hash Map

3 Cache Implementation

3.1 Practical Problem

To demonstrate the cache's capability, we will use the following problem to demonstrate it: the staircase problem. A person is standing at the bottom of a staircase with n steps. The person can climb either 1 step, 2 steps, or in some variations, even 3 steps at a time. The problem is to count the number of distinct ways to reach the top of the staircase.

For example, the number of combinations for reaching the fourth stair will equal the total number of different ways you can reach the third, second, and first stair:

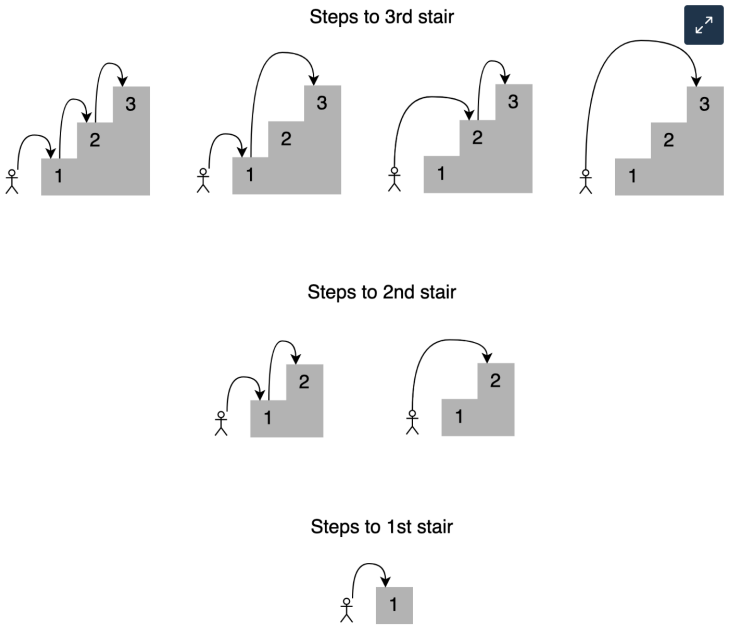

Figure 7: Stair case problem

Based on the picture, there are four different ways to reach the third stair and seven different ways to reach the fourth stair. Since each of the steps has three different ways to reach the next step, each step will have a variation of three, which cost us three recursive calls.

```python
# stepsTo will determine the possible combination of all possible ways from the ground
# to a given stair by combining all possible variations of the current step with 1 or 2
# or 3 steps
# Time complexity: O(3^n) (without cache since for each of the step, it will have 3 different ways)
# Space complexity: O(n)
def stepsTo(stair: int) -> int:
    if stair == 0:
        return 1
    elif stair < 0:
        return 0
    else:
        return stepsTo(stair-1) + stepsTo(stair-2) + stepsTo(stair-3)
```

Figure 8: Stair case's solution in Python with recursive

With the staircase of 7, we will have 44 different ways to reach, and the result will be the same as discussed in previous cases.

3.2 Practical Implementation

For the staircase problem, we will have the following solution in Python

```
MSCS532_HPC on  main [!?] via  v3.12.5 on ☁  kel.nguyen@vietna
mtechsociety.org(us-east1)
❯ python3 main.py
Number of combination that can reach to step 7 is  44
```

Figure 9: Stair case's solution output in Python with recursive

However, there are overlapping sub-problems when calculating the combination. For example, when computing stepsTo(stair), it computes stepsTo(stair-1) and stepsTo(stair-2) separately, but both may require calculating stepsTo(stair-3) again and again. As the stair grows, the number of repeated calculations increases exponentially, making the function inefficient. As a result, the recursive algorithm leads to $O(3^n)$ in time complexity

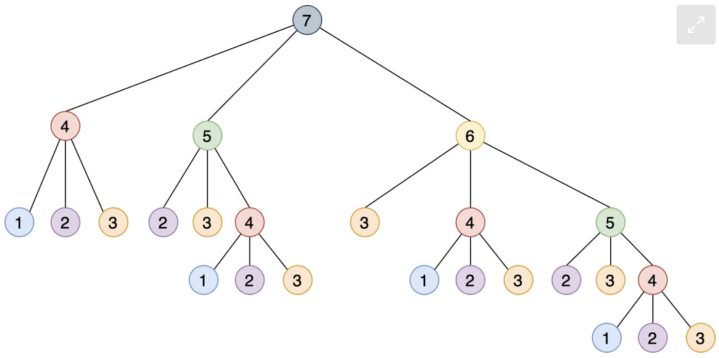

Figure 10: Stair case recursive calculation

From the aforementioned graph, stepsTo(4) has been used many times when derived from stepsTo(7), stepsTo(5) or stepsTo(6). The same can be said for other steps, such as stepsTo(2) or stepsTo(3). Therefore, to avoid the computation again and again, we can use memorization[5]. This approach ensures that a function does not run for the same inputs more than once by storing its result in memory and then referencing it later when necessary.

As an optimization for the staircase's recursive calculation, we will use the LRU cache as a technique to memorize the previously computed value and store it in memory with a size of 16 entries.

```python
@lru_cache(maxsize=16)
# stepsTo will determine the possible combination of all possible ways from the ground
# to a given stair by combining all possible variations of the current step with 1 or 2
# or 3 steps
# Time complexity: O(3^n) (without cache since for each of the step, it will have 3 different ways)
# Space complexity: O(n)
def stepsTo(stair: int) -> int:
    if stair == 0:
        return 1
    elif stair < 0:
        return 0
    else:
        return stepsTo(stair-1) + stepsTo(stair-2) + stepsTo(stair-3)
```

Figure 11: Stair case memorization with LRU Cache in Python

```
MSCS532_HPC on  main [!?] via  v3.12.5 on ☁  kel.nguyen@vietna
mtechsociety.org(us-east1)
❯ python3 main.py
Number of combination that can reach to step 7 is  44
CacheInfo(hits=12, misses=10, maxsize=16, currsize=10)
```

Figure 12: Stair case memorization's output with LRU Cache in Python

3.3 Comparison

When experimenting with the Recursive algorithm and Recursive with the Memorization algorithm, the difference between them can be clearly found:

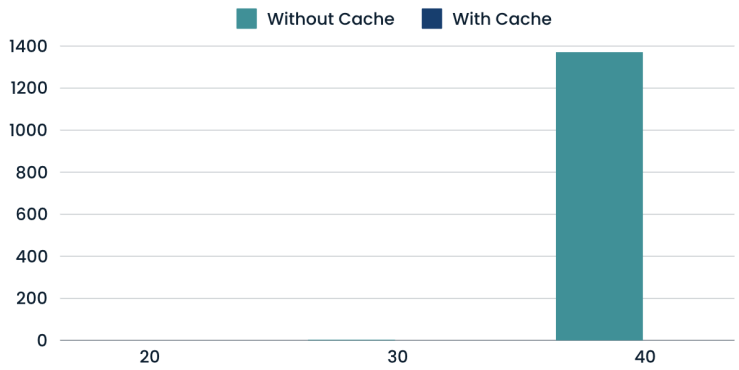

Figure 13: Staircase's Profiler (CPU(s)))

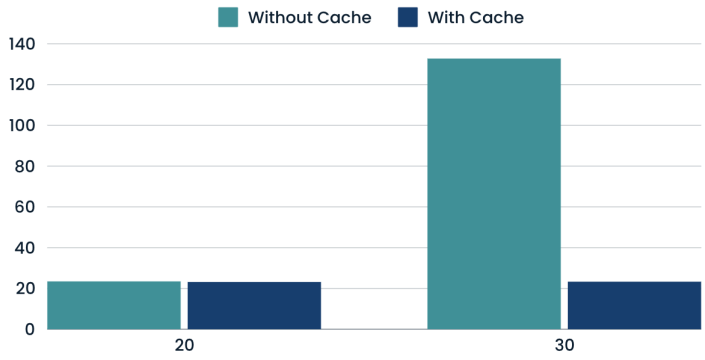

Figure 14: Staircase's Profiler (Memory(MiB)))

As a result, there has been an improvement from 1371 seconds to 0 seconds with Cache for the staircase problem at 40. As for space complexity, there has been an improvement from 132.76 to 23.312 MiB

References

[1] Kousik Nath (2018). *All things caching- use cases, benefits, strategies, choosing a caching technology, exploring some popular products*. Medium.

[2] Gigaspaces (2024). *In memory caches*.

[3] Blake Miner (2011). *What does Redis do when it runs out of memory?*. StackOverFlow

[4] Santiago Valdarrama (2021). *LRU Cache Python*. Real Python.

[5] Federico klez Culloca (2024). *Why Is It Called Memoization?*. StackOverFlow

[6] Fu Bang (2023). *GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings*. ACL Anthology

[7] Martin Doyhenard (2024). *Gotta cache 'em all: bending the rules of web cache exploitation*. PortSwigger

[8] Badrish Chandramouli (2024). *Introducing Garnet – an open-source, next-generation, faster cache-store for accelerating applications and services*. Microsoft

[9] Ruby B. Lee Zhenghong Wang (2005). *New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks*.

[10] Nguyen The Duy Khanh *High Performance Computing Assignment's Github Repository*