7 Cache Eviction Strategies You Should Know

Cache memory is limited - you can’t store everything

ASHISH PRATAP SINGH

JAN 02, 2025

Caching is a technique to make applications lightning fast, reduce database load, and improve user experience.

But, cache memory is **limited** - you can’t store everything.

So, how do you decide which items to keep and which ones to evict when space runs out?

This is where **cache eviction strategies** come into play. They determine which items are removed to make room for new ones.

In this article, we’ll dive into **Top 7 Cache Eviction Strategies** explaining what they are, how they work, their pros and cons.

If you’re enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

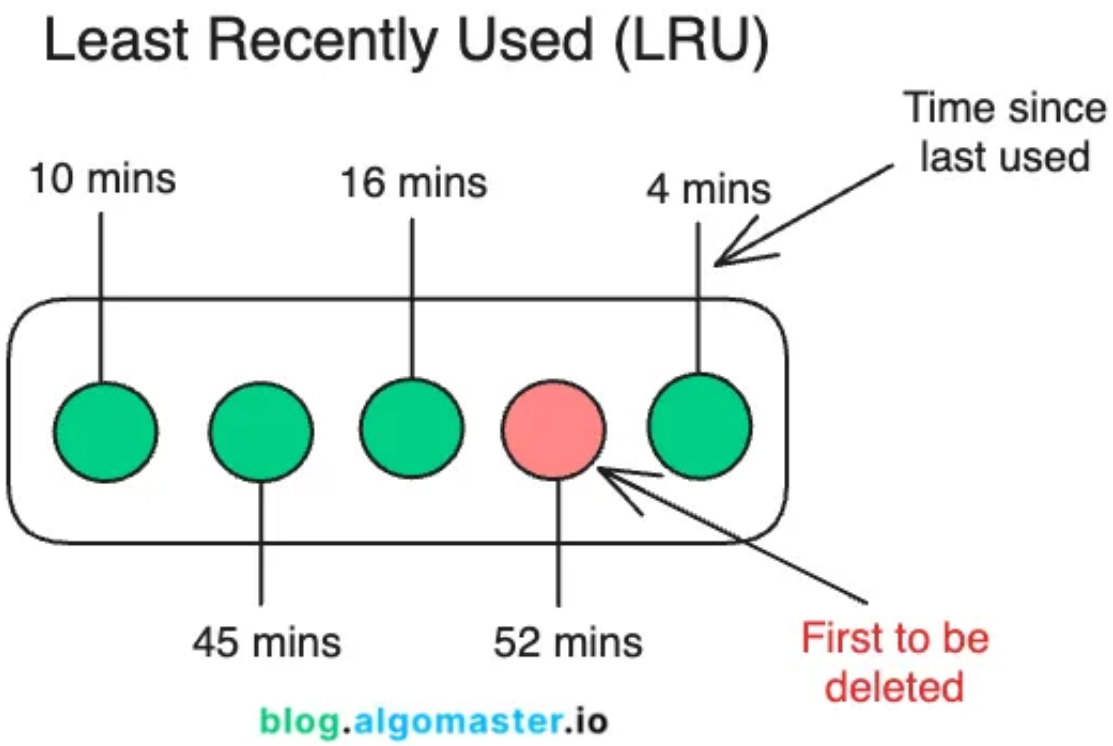
As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

---

1. Least Recently Used (LRU)

LRU evicts the item that hasn’t been used for the longest time.

The idea is simple: if you haven’t accessed an item in a while, it’s less likely to be accessed again soon.



Visualized using Multiplayer

How it Works

- **Access Tracking**: LRU keeps track of when each item in the cache was last accessed. This can be done using various data structures, such as a **doubly linked list** or a combination of a **hash map** and a **queue**.
- **Cache Hit** (**Item Found in Cache**): When an item is accessed, it is moved to the

most recently used position in the tracking data structure (e.g., moving it to the front of a list).

- **Cache Miss (Item Not Found in Cache):**
  - If the item isn't in the cache and the cache has free space, it is added directly.
  - If the cache is full, the **least recently used item** is evicted to make space for the new item.
- **Eviction:** The item that has been accessed least recently (tracked at the beginning of the list) is removed from the cache.

Consider a cache with a capacity of 3:

1. **Initial State**: Empty cache.
2. Add **A** → Cache: [A]
3. Add **B** → Cache: [A, B]
4. Add **C** → Cache: [A, B, C]
5. Access **A** → Cache: [B, C, A] (A becomes recently used)
6. Add **D** → Cache: [C, A, D] (B is evicted as it's the "least recently used")

Pros:

1. **Intuitive**: Easy to understand and widely adopted.
2. **Efficient**: Keeps frequently accessed items in the cache.
3. **Optimized for Real-World Usage**: Matches many access patterns, such as web browsing and API calls.

Cons:

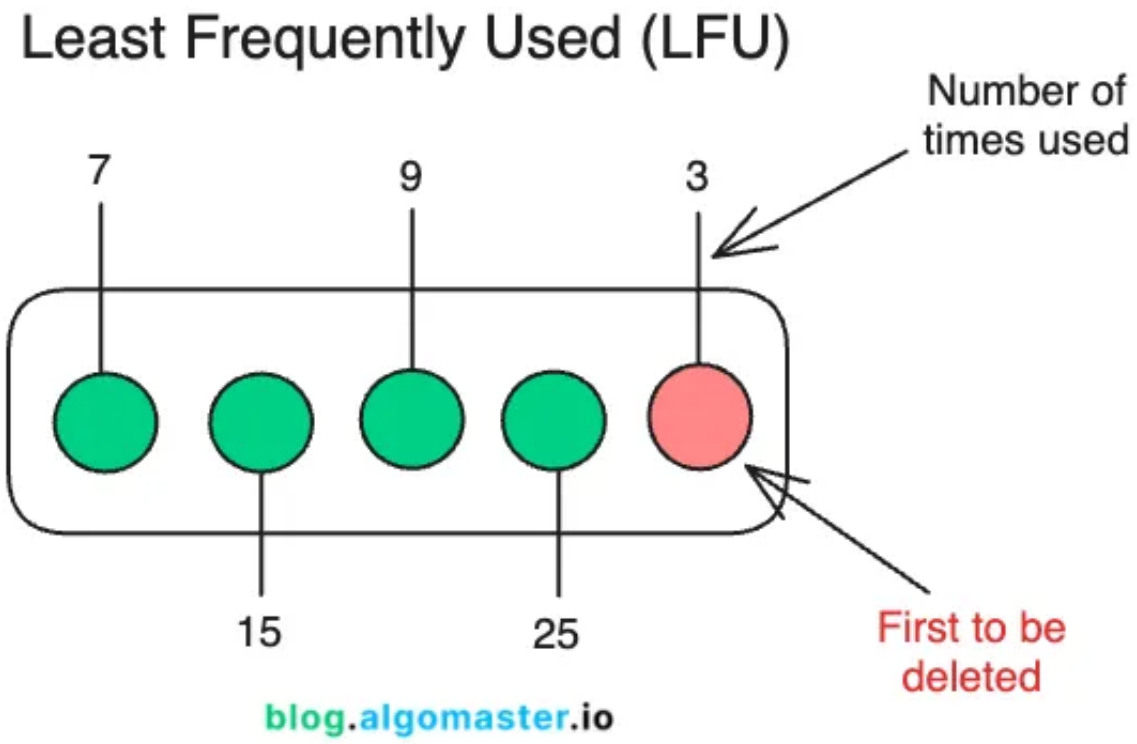
1. **Metadata Overhead**: Tracking usage order can consume additional memory.
2. **Performance Cost**: For large caches, maintaining the access order may introduce computational overhead.
3. **Not Adaptive**: Assumes past access patterns will predict future usage, which may not always hold true.

---

2. Least Frequently Used (LFU)

LFU evicts the item with the lowest access frequency. It assumes that items accessed less frequently in the past are less likely to be accessed in the future.

Unlike LRU, which focuses on **recency**, LFU emphasizes **frequency** of access.



Visualized using Multiplayer

How it Works

- **Track Access Frequency**: LFU maintains a frequency count for each item in the cache, incrementing the count each time the item is accessed.
- **Cache Hit (Item Found in Cache)**: When an item is accessed, its frequency count is increased.

- **Cache Miss (Item Not Found in Cache):**
  - If the cache has available space, the new item is added with an initial frequency count of 1.
  - If the cache is full, the **item with the lowest frequency** is evicted to make room for the new item. If multiple items share the same lowest frequency, a secondary strategy (like LRU or FIFO) resolves ties.
- **Eviction**: Remove the item with the smallest frequency count.

Consider a cache with a capacity of 3:

1. **Initial State**: Empty cache.
2. Add **A** → Cache: [A (freq=1)]
3. Add **B** → Cache: [A (freq=1), B (freq=1)]
4. Add **C** → Cache: [A (freq=1), B (freq=1), C (freq=1)]
5. Access **A** → Cache: [A (freq=2), B (freq=1), C (freq=1)]
6. Add **D** → Cache: [A (freq=2), C (freq=1), D (freq=1)] (B is evicted as it has the lowest frequency).
7. Access **C** → Cache: [A (freq=2), C (freq=2), D (freq=1)]

Pros:

1. **Efficient for Predictable Patterns**: Retains frequently accessed data, which is often more relevant.
2. **Highly Effective for Popular Data**: Works well in scenarios with clear "hot" items.

Cons:

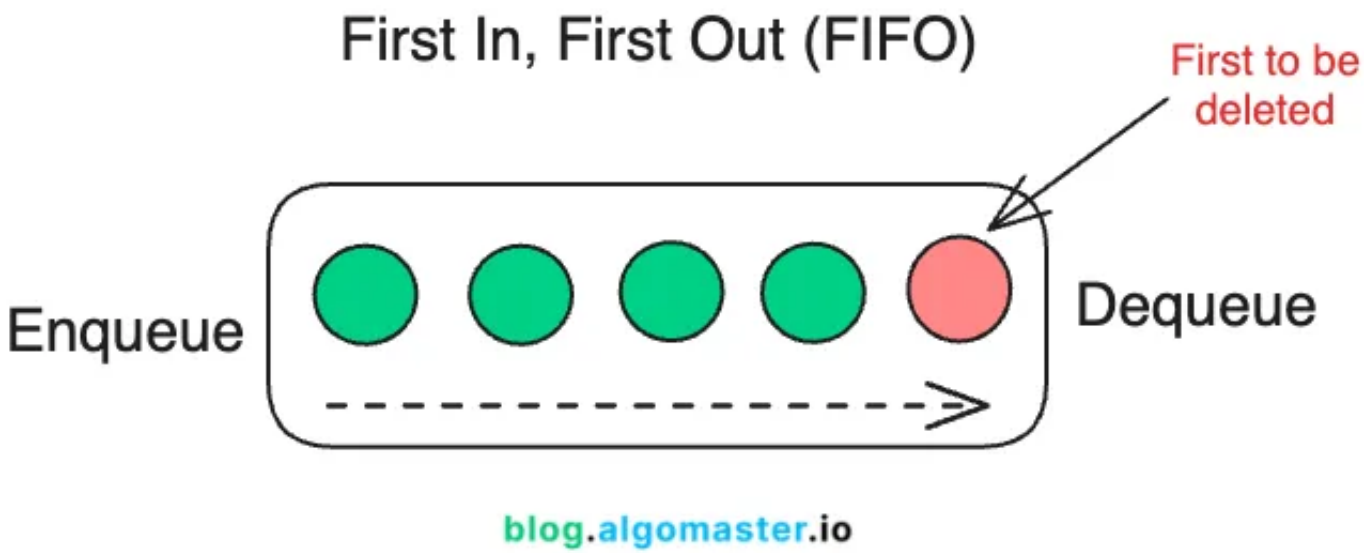
1. **High Overhead**: Requires additional memory to track frequency counts.
2. **Slower Updates**: Tracking and updating frequency can slow down operations.
3. **Not Adaptive**: May keep items that were frequently accessed in the past but are no longer relevant.

---

3. First In, First Out (FIFO)

FIFO evicts the item that was added first, regardless of how often it’s accessed.

FIFO operates under the assumption that items added earliest are least likely to be needed as the cache fills up.



Visualized using Multiplayer

How It Works

- **Item Insertion**: When an item is added to the cache, it is placed at the end of the queue.
- **Cache Hit (Item Found in Cache)**: No changes are made to the order of items. FIFO does not prioritize recently accessed items.
- **Cache Miss (Item Not Found in Cache)**:
    - If there is space in the cache, the new item is added to the end of the queue.

    - If the cache is full, the item at the front of the queue (the oldest item) is evicted to make space for the new item.
- **Eviction**: The oldest item, which has been in the cache the longest, is removed to make room for the new item.

Let's assume a cache with a capacity of 3:

1. Add **A** → Cache: [A]
2. Add **B** → Cache: [A, B]
3. Add **C** → Cache: [A, B, C]
4. Add **D** → Cache: [B, C, D] (A is evicted because it was added first).
5. Access **B** → Cache: [B, C, D] (Order remains unchanged).
6. Add **E** → Cache: [C, D, E] (B is evicted because it was the oldest remaining item).

Pros:

1. **Simple to Implement**: FIFO is straightforward and requires minimal logic.
2. **Low Overhead**: No need to track additional metadata like access frequency or recency.
3. **Deterministic Behavior**: Eviction follows a predictable order.

Cons:

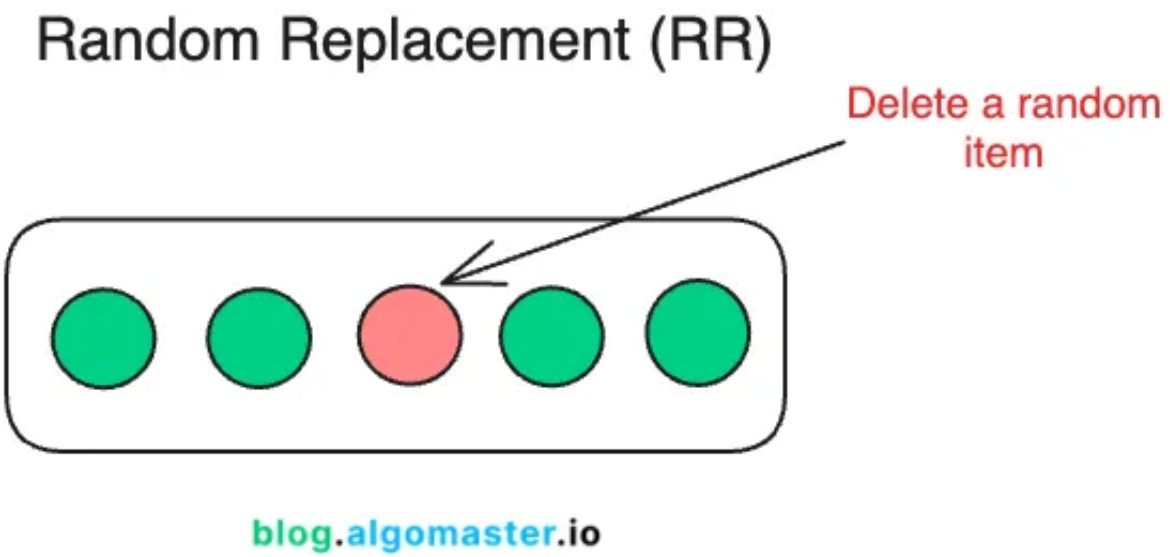
1. **Ignores Access Patterns**: Items still in frequent use can be evicted, reducing cache efficiency.
2. **Suboptimal for Many Use Cases**: FIFO is rarely ideal in modern systems where recency and frequency matter.
3. **May Waste Cache Space**: If old but frequently used items are evicted, the cache loses its utility.

---

4. Random Replacement (RR)

RR cache eviction strategy is the simplest of all: when the cache is full, it evicts a random item to make space for a new one.

It doesn't track recency, frequency, or insertion order, making it a lightweight approach with minimal computational overhead.



Visualized using Multiplayer

This simplicity can sometimes be surprisingly effective, especially in systems with unpredictable or highly dynamic access patterns.

How It Works

- **Item Insertion**: When an item is added to the cache and there is space, it is stored directly.
- **Cache Hit:** If the requested item exists in the cache, it is served, and no changes are made to the cache.

- **Cache Miss:** If the item is not in the cache and the cache is full, a random item is removed.
- **Eviction**: The randomly selected item is removed, and the new item is added to the cache.

Let's assume a cache with a capacity of 3:

1. Add **A** → Cache: [A]
2. Add **B** → Cache: [A, B]
3. Add **C** → Cache: [A, B, C]
4. Add **D** → Cache: [B, C, D] (A is randomly evicted).
5. Add **E** → Cache: [C, E, D] (B is randomly evicted).

Pros:

1. **Simple to Implement**: No need for metadata like access frequency or recency.
2. **Low Overhead**: Computational and memory requirements are minimal.
3. **Fair for Unpredictable Access Patterns**: Avoids bias toward recency or frequency, which can be useful in some scenarios.

Cons:

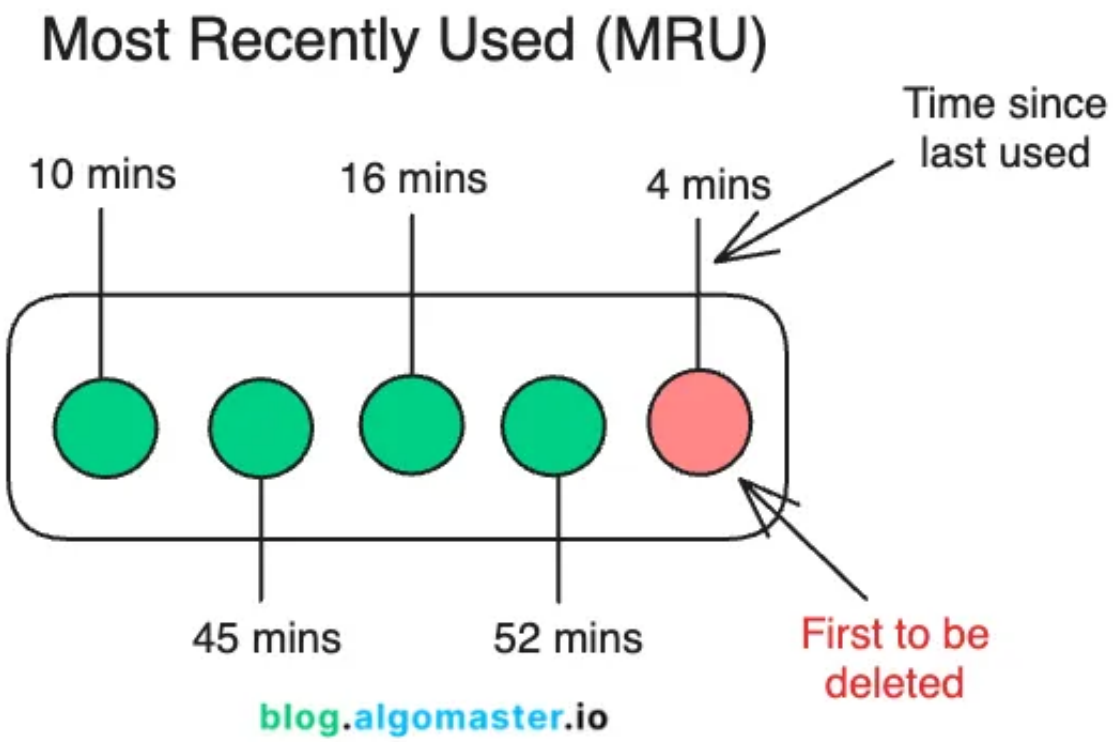
1. **Unpredictable Eviction**: A frequently used item might be evicted, reducing cache efficiency.
2. **Inefficient for Stable Access Patterns**: Doesn't adapt well when certain items are consistently accessed.
3. **High Risk of Poor Cache Hit Rates**: Random eviction often leads to suboptimal retention of important items.

---

5. Most Recently Used (MRU)

MRU is the opposite of **Least Recently Used** (**LRU**). In MRU, the item that was accessed most recently is the first to be evicted when the cache is full.

The idea behind MRU is that the most recently accessed item is likely to be a temporary need and won't be accessed again soon, so evicting it frees up space for potentially more valuable data.



Visualized using Multiplayer

How It Works

- **Item Insertion**: When a new item is added to the cache, it is marked as the most recently used.
- **Cache Hit** (**Item Found in Cache**): When an item is accessed, it is marked as the most recently used.

- **Cache Miss (Item Not Found in Cache):**
  - If the cache has available space, the new item is added directly.
  - If the cache is full, the most recently used item is evicted to make room for the new item.
- **Eviction**: The item that was accessed or added most recently is removed.

Let's assume a cache with a capacity of 3:

1. Add **A** → Cache: [A]
2. Add **B** → Cache: [A, B]
3. Add **C** → Cache: [A, B, C]
4. Access **C** → Cache: [A, B, C] (C is marked as the most recently used).
5. Add **D** → Cache: [A, B, D] (C is evicted as it was the most recently used).
6. Access **B** → Cache: [A, B, D] (B becomes the most recently used).
7. Add **E** → Cache: [A, D, E] (B is evicted as it was the most recently used).

Pros:

1. **Effective in Specific Scenarios**: Retains older data, which might be more valuable in certain workloads.

2. **Simple Implementation**: Requires minimal metadata.

Cons:

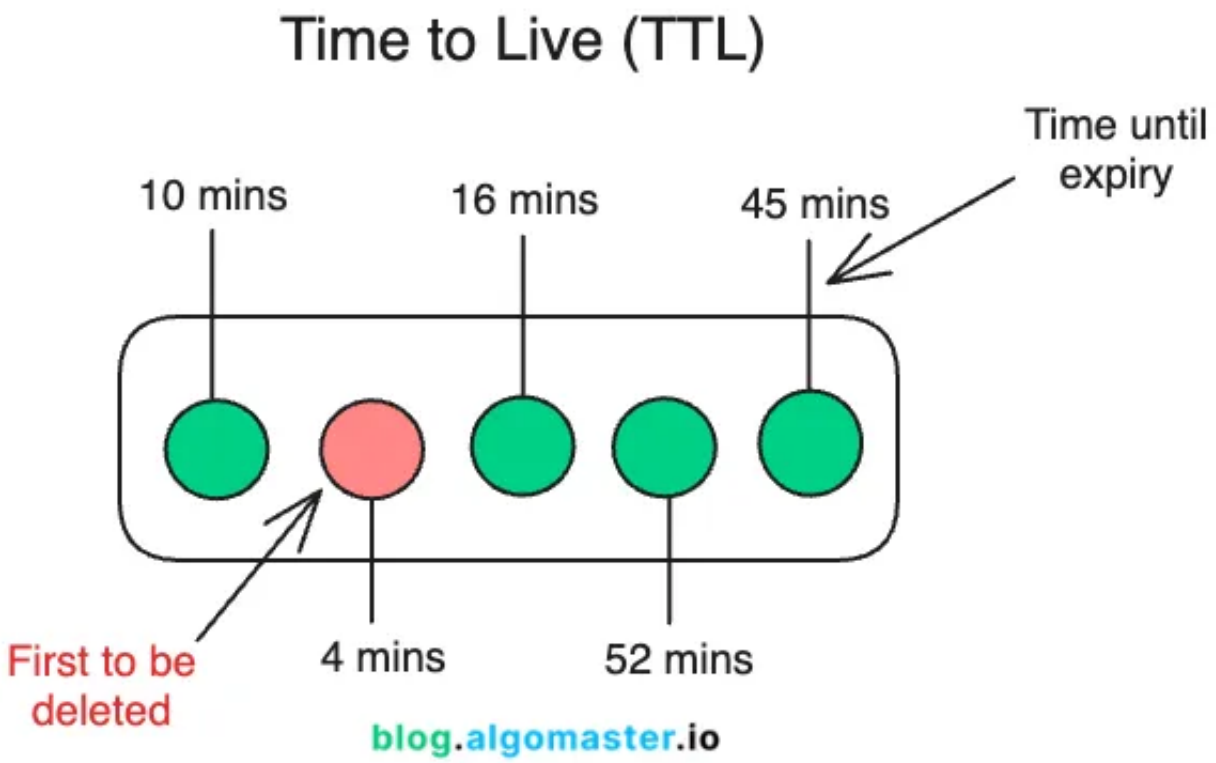
1. **Suboptimal for Most Use Cases**: MRU assumes recent data is less valuable, which is often untrue for many applications.
2. **Poor Hit Rate in Predictable Patterns**: Fails in scenarios where recently accessed data is more likely to be reused.
3. **Rarely Used in Practice**: Limited applicability compared to other strategies like LRU or LFU.

---

6. Time to Live (TTL)

TTL is a cache eviction strategy where each cached item is assigned a fixed lifespan. Once an item's lifespan expires, it is automatically removed from the cache, regardless of access patterns or frequency.

This ensures that cached data remains fresh and prevents stale data from lingering in the cache indefinitely.



Visualized using Multiplayer

How It Works

- **Item Insertion**: When an item is added to the cache, a TTL value (e.g., 10 seconds) is assigned to it. The expiration time is usually calculated as `current time + TTL`.
- **Cache Access (Hit or Miss)**: When an item is accessed, the cache checks its

expiration time:

    - If the item is expired, it is removed from the cache, and a cache miss is recorded.
    - If the item is valid, it is served as a cache hit.
- **Eviction**: Expired items are automatically removed either during periodic cleanup or on access.

Let's assume a cache with a TTL of 5 seconds:

1. Add **A** with TTL = 5s → Cache: [A (expires in 5s)]
2. Add **B** with TTL = 10s → Cache: [A (5s), B (10s)]
3. After 6 seconds → Cache: [B (expires in 4s)] (A is evicted because its TTL expired).
4. Add **C** with TTL = 5s → Cache: [B (4s), C (5s)]

If an item is accessed after its TTL expires, it results in a cache miss.

TTL is often implemented in caching systems like **Redis** or **Memcached**, where you can specify expiration times for each key.

Pros:

1. **Ensures Freshness**: Automatically removes stale data, ensuring only fresh items remain in the cache.
2. **Simple to Configure**: TTL values are easy to assign during cache insertion.
3. **Low Overhead**: No need to track usage patterns or access frequency.
4. **Prevents Memory Leaks**: Stale data is cleared out systematically, avoiding cache bloat.

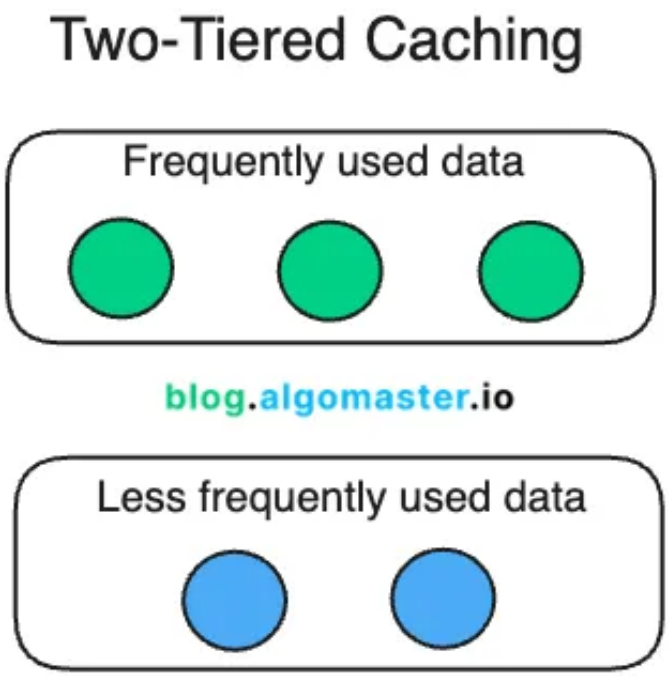
Cons:

1. **Fixed Lifespan**: Items may be evicted prematurely even if they are frequently accessed.
2. **Wasteful Eviction**: Items that haven't expired but are still irrelevant occupy cache space.
3. **Limited Flexibility**: TTL doesn't adapt to dynamic workloads or usage patterns.

---

7. Two-Tiered Caching

Two-Tiered Caching combines two layers of cache—usually a **local cache** (in-memory) and a **remote cache** (distributed or shared).



Visualized using Multiplayer

The local cache serves as the first layer (hot cache), providing ultra-fast access to frequently used data, while the remote cache acts as the second layer (cold cache) for items not found in the local cache but still needed relatively quickly.

How It Works

1. **Local Cache (First Tier)**:
    - Resides on the same server as the application, often in memory (e.g., `HashMap`,

`LRUCache` in the application)..

- Provides ultra-fast access to frequently accessed data, reducing latency and server load.
- Examples: In-memory data structures like `HashMap` or frameworks like **Guava Cache**.

2. **Remote Cache** (**Second Tier**):
   - Shared across multiple servers in the system. Slightly slower due to network overhead but offers larger storage and shared consistency.
   - Used to store data that is not in the local cache but is still frequently needed.
   - Examples: Distributed cache systems like **Redis** or **Memcached**.

Workflow:

- A client request checks the **local cache** first.
- If the data is not found (cache miss), it queries the **remote cache**.
- If the data is still not found (another cache miss), it retrieves the data from the primary data source (e.g., a database), stores it in both the local and remote caches, and returns it to the client.

Pros:

1. **Ultra-Fast Access**: Local cache provides near-instantaneous response times for frequent requests.
2. **Scalable Storage**: Remote cache adds scalability and allows data sharing across multiple servers.
3. **Reduces Database Load**: Two-tiered caching significantly minimizes calls to the backend database.
4. **Fault Tolerance**: If the local cache fails, the remote cache acts as a fallback.

Cons:

1. **Complexity**: Managing two caches introduces more overhead, including synchronization and consistency issues.
2. **Stale Data**: Inconsistent updates between tiers may lead to serving stale data.
3. **Increased Latency for Remote Cache Hits**: Accessing the second-tier remote cache is slower than the local cache.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a **paid subscriber**.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all **premium courses** on **algomaster.io**.

There are group discounts, gift options, and referral bonuses available.

Checkout my **Youtube channel** for more in-depth content.

Follow me on **LinkedIn**, **X** and **Medium** to stay updated.

Checkout my **GitHub repositories** for free interview preparation resources.

I hope you have a lovely day!

See you soon,
Ashish

110 Likes · 7 Restacks

← Previous

Next →

Discussion about this post

Comments | Restacks

Write a comment...

Nitish Nandwana 10 Jan

Liked by Ashish Pratap Singh

👏👏👏

LIKE (1) REPLY SHARE

Rohan 2 Jan *Edited*

Hi Ashish

What is the best caching mechanism?

It would be more helpful if you provided use cases for which mechanism needs to be used in which scenario

Thanks, in advance!!

LIKE (1) REPLY SHARE

1 reply

3 more comments...