A Crash Course in Caching - Final Part

ALEX XU

MAR 29, 2023 · PAID

This is part 3 of the Crash Course in Caching series.

Managing Operational Challenges in Caching

When implementing caching systems, it is crucial to not only choose the best cache strategies but also address the operational challenges that may arise. The following section dives into common caching problems by category and provides insight into how to tackle these challenges effectively.

Reliability Challenges

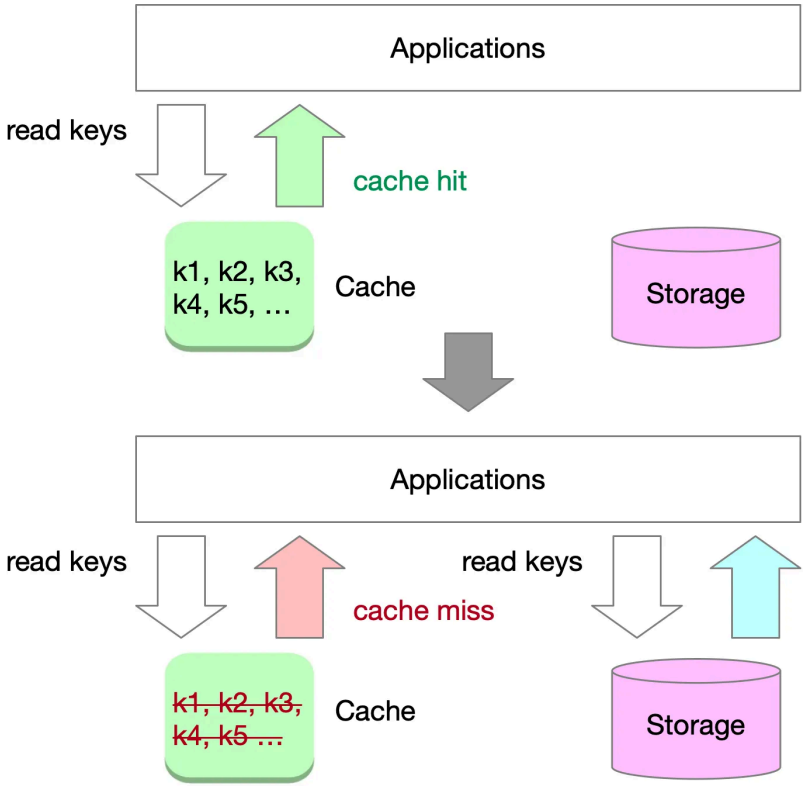
Cache reliability is important for maintaining stable system performance. Common reliability issues include cache avalanche, cache stampede, and cache penetration.

Mitigating cache avalanche and cache stampede

Caching systems play a crucial role in protecting databases by maintaining high cache hit ratios and reducing traffic to storage systems. Under normal conditions, most read requests result in

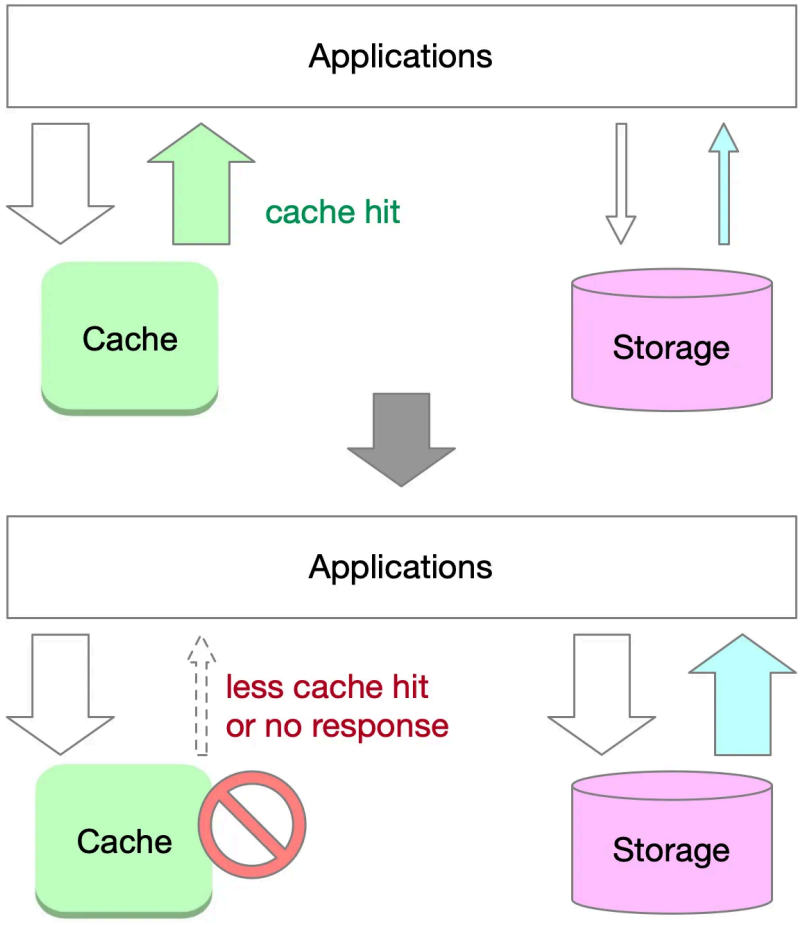
cache hits, with only a small portion leading to cache misses. By design, the cache handles the majority of the traffic, and the storage system experiences significantly less load.

Cache avalanche and cache stampede are related but distinct phenomena that can occur in large scale systems, causing significant performance degradation or even system-wide outages. Cache avalanche happens when multiple or all cache entries expire simultaneously or within a short time window, leading to a sudden surge in requests to the underlying data store.



Cache avalanche

Cache stampede, also known as thundering herd, occurs in large scale systems when a sudden influx of requests overwhelms the system, causing performance degradation or even system-wide outages. This can happen due to various reasons, such as cache misses on popular items, a sudden spike in user traffic, or service restarts after maintenance.



Thundering herd

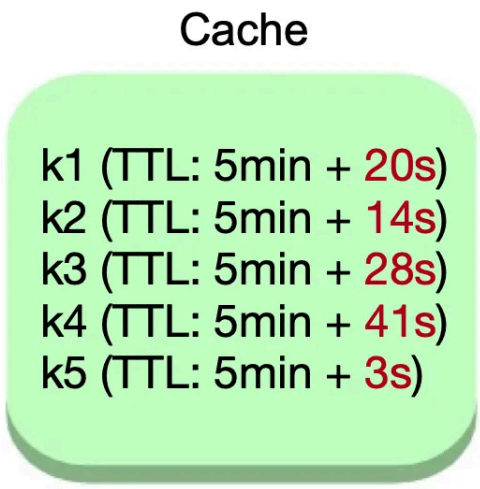
When a cache or part of it fails, a large number of read requests result in cache misses, causing a massive influx of traffic to the storage system. This sudden increase in traffic can significantly reduce storage performance or even lead to system crashes. In this scenario, the failure of the cache serves as a trigger that combines elements of both cache avalanche and cache stampede, resulting in a high-stress situation for the storage system.

Traffic peaks, such as during high-traffic events like Black Friday sales on e-commerce sites in the United States, can also trigger these phenomena. A single cache node might crash due to the increased demand for popular keys, causing these keys to be rehashed to another node, which then crashes as well, eventually leading to the collapse of the entire cache system.

To maintain the stability and efficiency of large scale systems, it is essential to implement strategies that mitigate the risks of both cache avalanche and cache stampede. There are several techniques to help prevent and alleviate the impact of these events.

Staggered expiration

For cache avalanche prevention, use staggered expiration by combining a base time-to-live (TTL) value with a random delta. This approach spreads out the expiration times of cached entries, reducing the likelihood of many items expiring at once.



Staggered expiration

Consistent hashing

Consistent hashing can be used to distribute cache entries across multiple cache servers evenly. This technique reduces the impact of cache avalanche and cache stampede by sharing the load among the remaining servers and preventing a single point of failure.

Circuit breakers

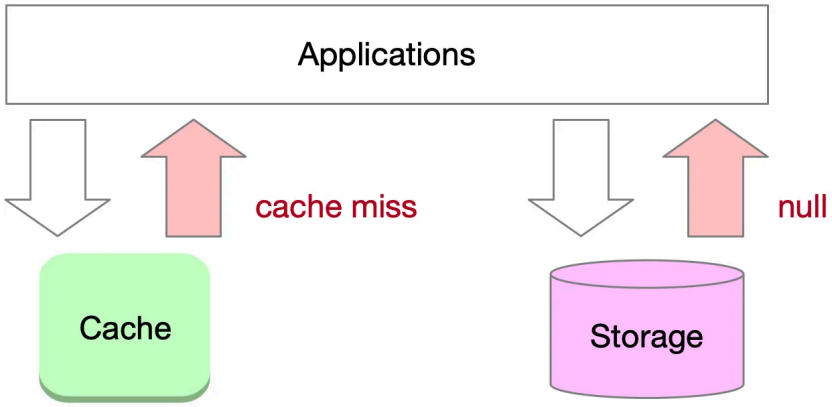
Implementing circuit breakers in the system can help prevent cache avalanche and cache stampede from escalating into more severe issues. Circuit breakers monitor the system's health and prevent overloading by stopping excessive requests to the cache and data store if the system is under high stress.

Request rate limiting and load shedding

Employing request rate limiting and load shedding can specifically address cache stampede by controlling the rate at which requests are processed and preventing the system from being overwhelmed. These techniques can be applied on a per-user, per-client, or system-wide basis to maintain stability during high-load situations.

Addressing Cache Penetration Issues

Cache penetration occurs when an application attempts to access non-existent data, bypassing the cache and leading to potential performance issues. When an application requests a non-existent key, it results in a cache miss, followed by a request to the storage system, which also returns an empty result. This bypassing of the cache to reach the backing storage is referred to as cache penetration. If a large number of non-existent key requests occur, it can impact the performance of the storage layer and destabilize the overall system.

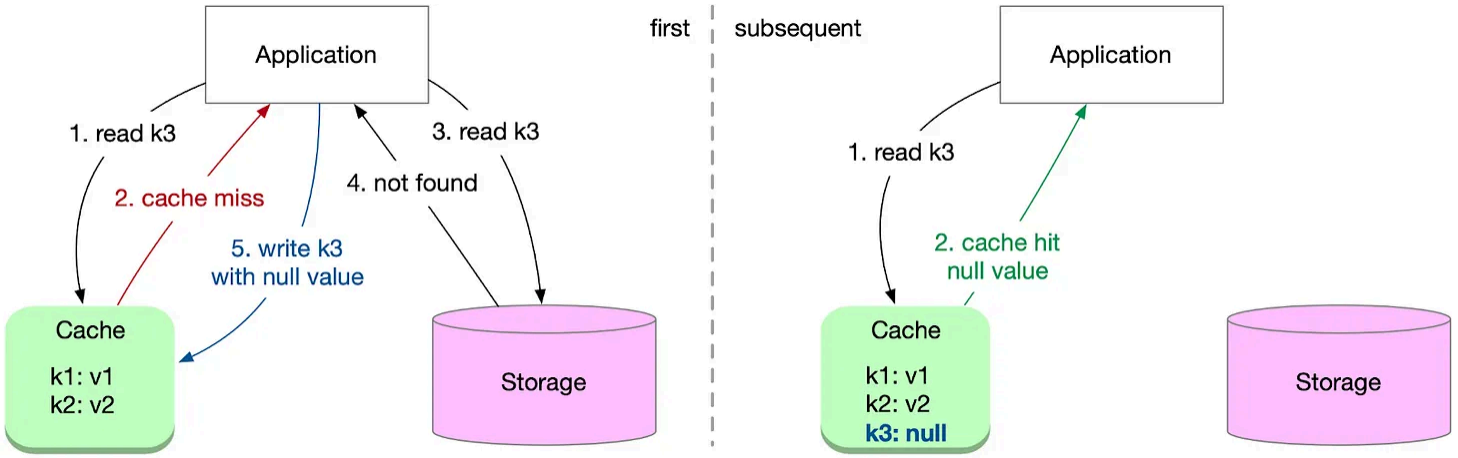


Cache penetration

A common example of cache penetration occurs when numerous users try to query non-existent orders on a website within a short period. Large websites typically implement protective

measures to mitigate this issue.

To mitigate this, store a placeholder value in the cache to represent non-existent data. Subsequent requests for the same non-existent data will hit the cache instead of the data store. Set an appropriate TTL for these placeholder entries to prevent them from occupying cache space indefinitely.

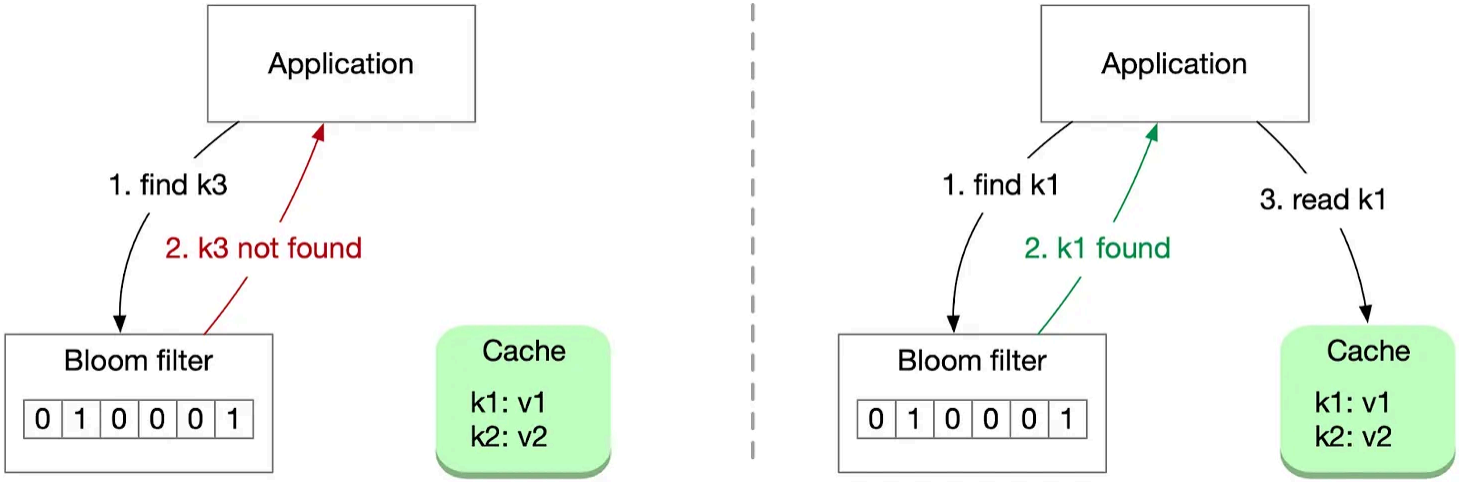


Cache placeholder value with a non-existent key

While this method is simple and effective, it can consume significant cache resources if the application frequently queries a large number of non-existent keys.

Another solution involves using a bloom filter, a space-efficient, probabilistic data structure that tests whether an element belongs to a set. By using a bloom filter, the system can quickly identify non-existent data, reducing cache penetration and unnecessary data store requests.

When a record is added to storage, its key is also recorded in a bloom filter. When fetching a record, the application first checks whether the key exists in the bloom filter. If the key is absent, it won't exist in the cache or storage either, and the application can return a null value directly. If the key is present in the bloom filter, the application proceeds to read the key from the cache and storage as usual. Since a bloom filter sometimes returns a false positive, a small number of the cache reads will result in a cache miss.



Bloom filter before the cache

The challenge of using a bloom filter lies in its capacity limitations. For instance, 1 billion keys with a 1% false positive rate would require approximately 1.2 GB of capacity. This solution is best suited for smaller data sets.

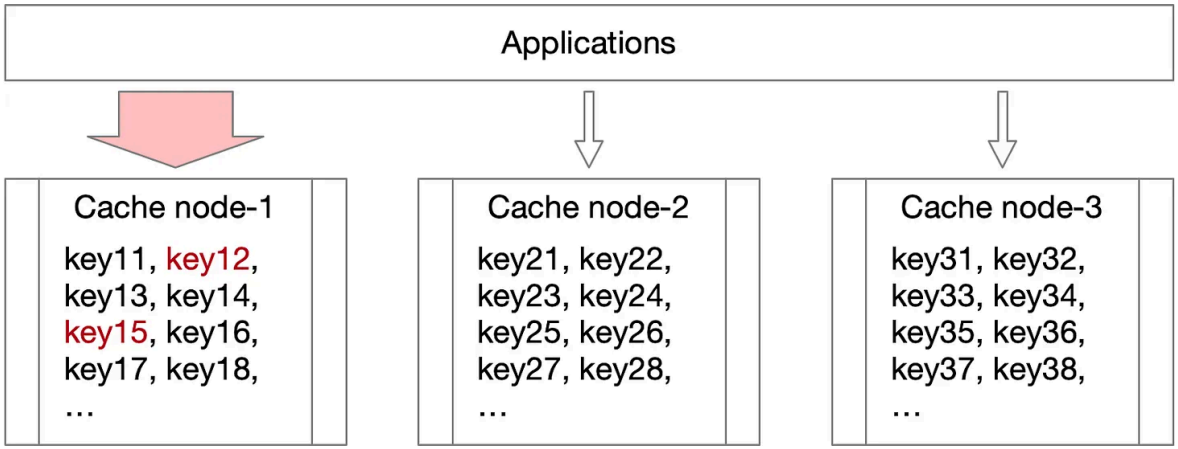
Traffic Pattern Challenges

Hot key problem

The hot key problem occurs when high traffic on a single key results in excessive traffic pressure on the cache node, potentially leading to performance issues.

In a distributed cache system, keys are shared across different nodes based on a sharding strategy. Traffic for each key can vary greatly, with some keys experiencing exceptionally high traffic. As illustrated in the diagram below, substantial traffic for keys like "key12" and "key15" could potentially exceed the resource capacity of node 1.

For instance, when tweets are distributed in the cache system by their IDs, and a few of them become popular quickly, the corresponding cache node experiences increased traffic, possibly surpassing its resource capabilities. Hot key issues can arise in various scenarios, such as operational emergencies, festivals, sports games, flash sales, etc.



Hot key in distributed cache

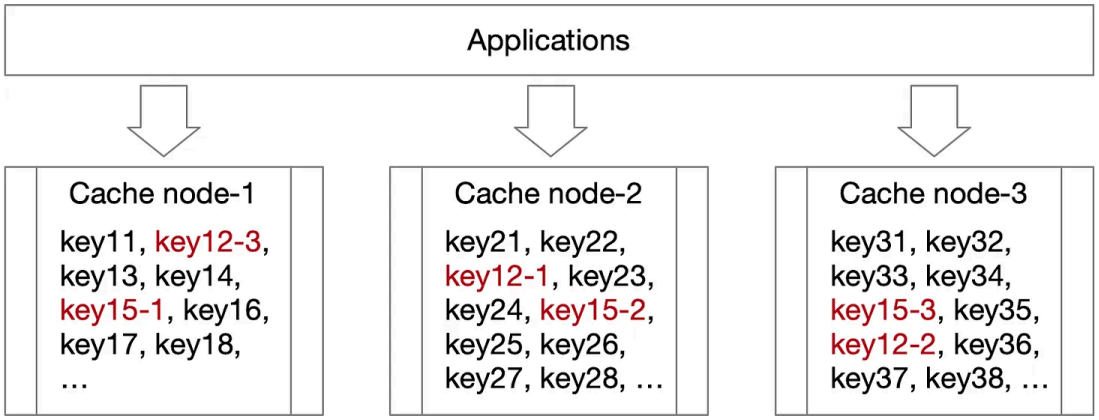
To address this problem, two steps are necessary:

1. Identify the hot keys.
2. Implement special measures for these keys to reduce traffic pressure.

In step 1, for predictable events like important holidays or online promotions, it is possible to evaluate potential hot keys beforehand. However, for emergencies, advance evaluation is not feasible. One solution is to conduct real-time traffic analysis to promptly detect emerging hot keys.

In step 2, there are several potential solutions:

- Distribute traffic pressure across the entire cache system. As illustrated in the diagram below, split "key12" into "key12-1", "key12-2", up to "key12-N", and distribute these N keys across multiple nodes. When an application requests a hot key, it randomly selects one of the suffixed keys, enabling traffic to be shared across many nodes and preventing a single node from becoming overloaded.



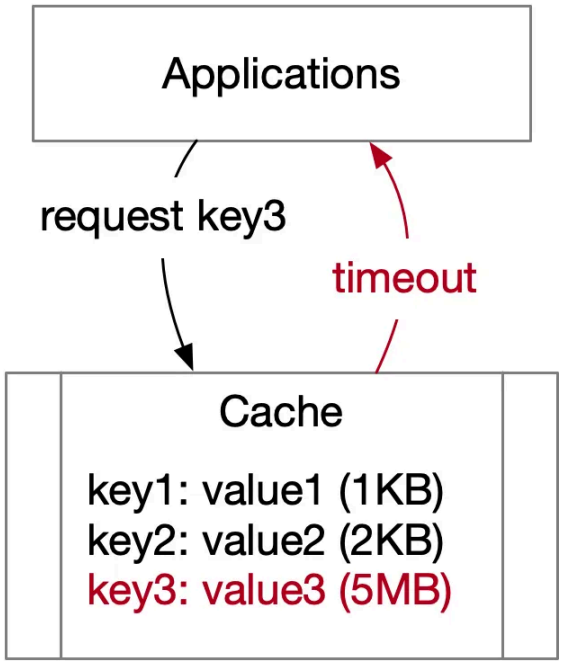
- If numerous hot keys exist, real-time monitoring can enable the cache system to expand quickly, reducing the traffic impact.
- As a last resort, applications can store hot keys in a local cache, decreasing traffic to the remote cache system. This approach alleviates the pressure on the cache nodes and helps maintain overall system performance.

Large key problem

When the value size of a key is significantly large, it can result in access timeouts, leading to the large key problem.

The impact of the large key problem is more severe than it may initially seem:

- Frequent access to large keys can consume significant network bandwidth, leading to slow lookup performance.
- Any partial update of a large key will result in the modification of the entire value, further contributing to performance issues.
- If a large key becomes invalid or is evicted from the cache, reloading it from storage can be time-consuming, leading to additional slow query performance.



Large keys are encountered in specific scenarios. For example, the content of some social media posts may be much larger than average, and accessing these larger posts will naturally take more time

Several methods can be employed to address the large key problem:

- Compress large keys in cache to reduce data size.
- Set a longer Time-To-Live (TTL) for large keys to minimize eviction or invalidation.
- Divide large keys into multiple smaller ones to minimize the presence of large keys in the system.

- Prevent the creation of large keys by modifying business requirements, such as limiting the length of tweets. This approach helps to avoid the large key problem from the outset.

Consistency Challenges

Cache consistency is a crucial aspect of any caching system, involving two primary concerns:

- Consistency within the cache system itself.
- Consistency between the cache and the underlying data storage.

Cache consistency is defined as follows: if a key exists in the cache, its value should eventually match the value in the underlying data store.

To optimize performance, caching systems make trade-offs between consistency and speed. The best achievable consistency in these systems is eventual consistency, where the cache and storage eventually have the same data.

There are several scenarios that could lead to cache inconsistency

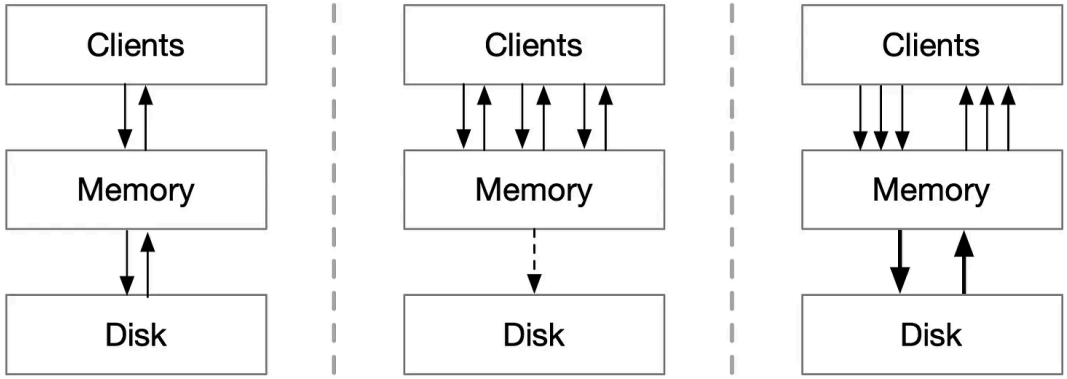
Cache system without replicas

In a cache system without replicas, each shard of cache data has just one copy stored in a single cache node, typically in memory for improved performance.

When the cache process or node restarts, all cached keys are cleared, causing requests for these keys to result in cache misses. Memcached is an example of such a system. To prevent data loss from memory, disk backup can in theory be employed. With disk backup, memory can be

recovered after the cache process restarts. Disk backup can be implemented using either a write-through or write-back strategy, each with its pros and cons.

- Write-through strategy: In this approach, any memory update also updates the disk synchronously. Data consistency between memory and disk is guaranteed, but at the cost of reduced write performance.
- Write-back strategy: This method updates the disk asynchronously in batches after the memory is updated. While it generally offers better performance since caching systems prioritize speed, the write-back cache acknowledges write requests before writing to the disk, which may cause data loss if a crash occurs during this process.
- Modified write-back strategy: An alternative write-back strategy acknowledges write requests only after writing to the disk. If upstream clients don't receive a timely response, they can retry, preventing data loss. However, this approach increases the response time for a single write across the entire batch of writes. The period of batch writing is a critical factor, and with an appropriate duration, this strategy can be an effective choice.



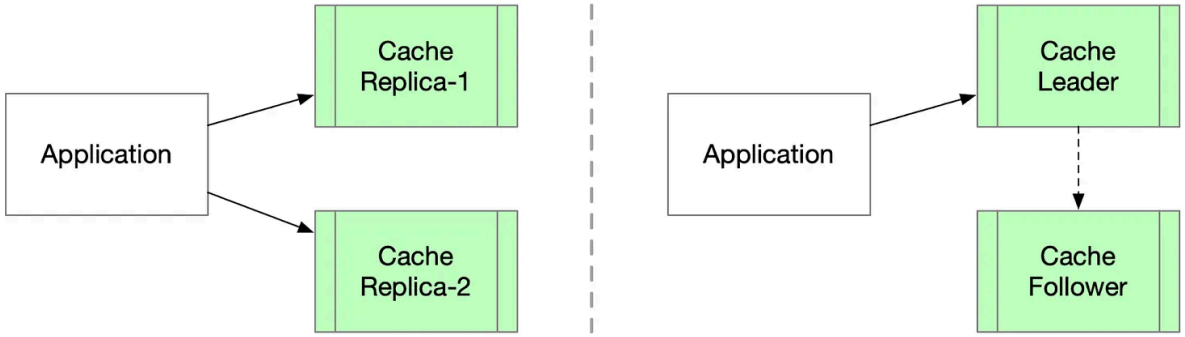
In practice, restoring a large cache with a vast amount of data from disk after a crash can take too long to be practical. This is why disk backup strategies are rarely used, even with products like Redis that offer persistence. In most situations, cache replicas are a more efficient and reliable approach for ensuring data durability and fault tolerance.

Multiple cache replicas

In a distributed cache system with more than one replica for cached data, the system can tolerate the crash of a single node. However, this introduces the complexity of ensuring consistency among different replicas for the same cached key. This issue is fundamentally about data consistency in a distributed system. As previously mentioned, caching systems typically prioritize speed, making strict consistency difficult to achieve.

When writing a key to a cache with multiple replicas, there are several approaches:

- Parallel writing to all replicas: The application writes to all replicas in parallel, waiting for responses from each. Response time is determined by the slowest replica, but if any single replica is unreachable, the write process is blocked.
- Writing to a quorum of replicas: This improved method writes to more than half of the replicas, but it introduces additional complexity and must be combined with a quorum-based read process. Generally, it is not worth implementing such a complex mechanism for cache replicas.
- Writing to the leader replica: In this approach, data is written only to the leader replica and is then asynchronously replicated to the followers. This popular choice can result in reading stale data from follower replicas if they are not in sync with the leader. However, this method works well in scenarios with infrequent or no data updates.



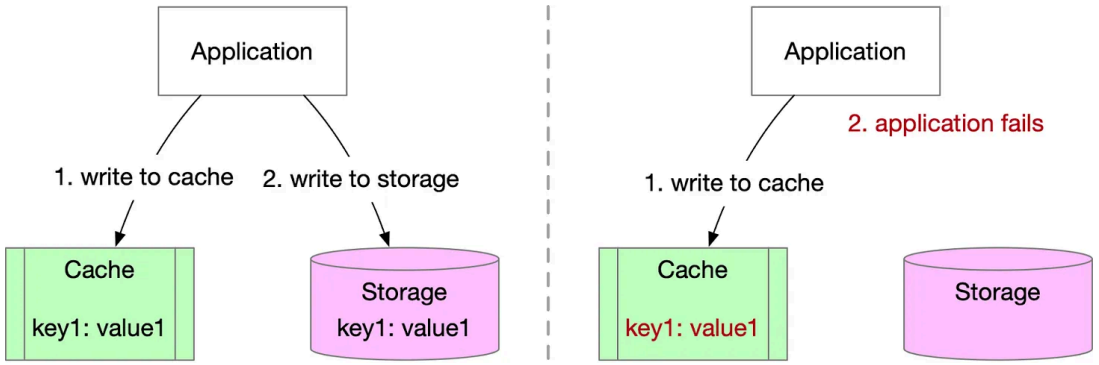
Write to multiple replicas

Writing order of cache and storage

The cache and storage are typically separate systems, and it is highly unlikely that both can be updated simultaneously. Distributed transactions may be an option, but they require both systems to support transactional operations and often result in poor performance, making them an impractical choice.

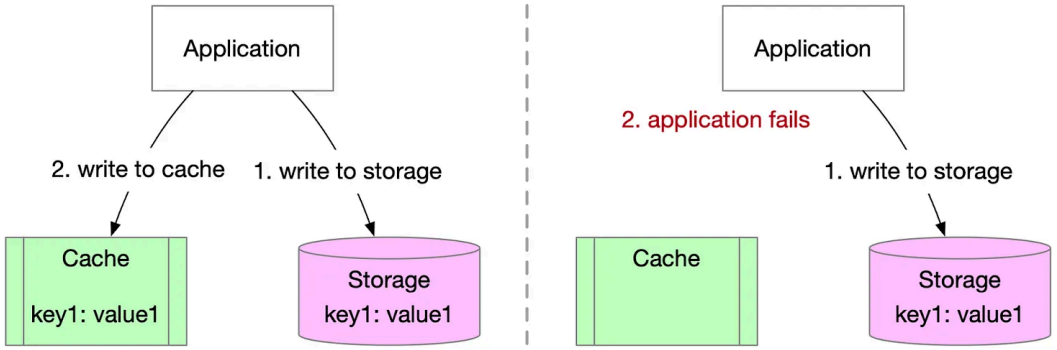
Consequently, the writing order to the cache and storage significantly impacts their data consistency. There are two main approaches:

- Writing to the cache first, then to storage: As illustrated below, data is written to the cache before storage. If the application fails before writing to storage, the cache will contain extra data not present in the durable storage. This situation can negatively affect system behavior, as the extra cached data should not be visible.



Cache first, storage next

- Writing to storage first, then to the cache: As shown below, data is written to storage before the cache. If the application fails before writing to the cache, no harm is done if the cache does not contain the key. However, if the key was already cached, the cached value becomes stale.



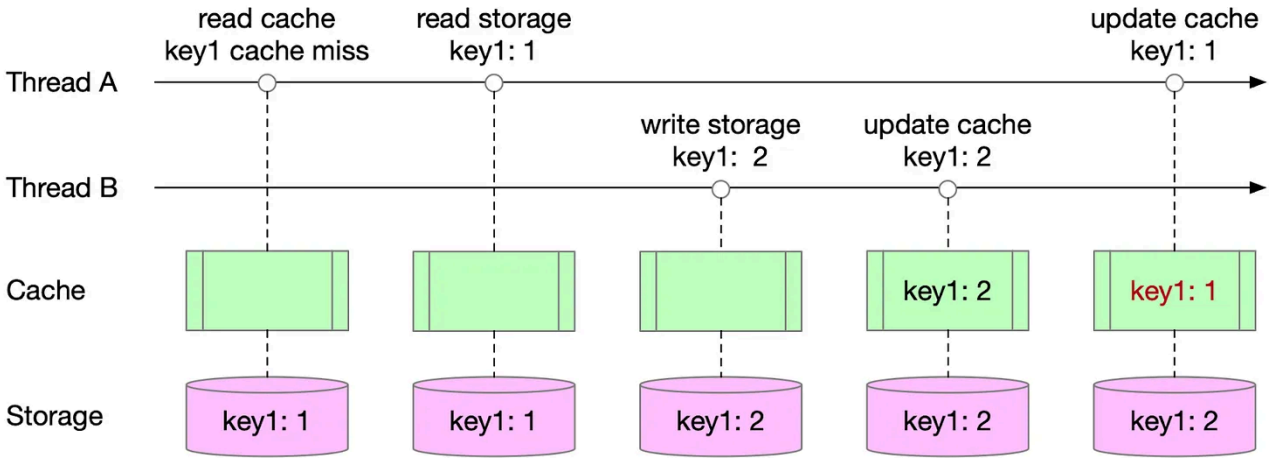
Storage first, cache next

Both writing orders can result in data inconsistency between cache and storage, with the latter generally being safer in most scenarios. For the latter, there is an option to invalidate the cache

entry first before writing to the storage. This would eliminate the data inconsistency, but it may not work well for hot keys.

Concurrent cache updates

Data inconsistency can occur when multiple writers update the cache concurrently, as illustrated in the following example:



Concurrent cache updates

1. Thread A reads a key from the cache, resulting in a cache miss.
2. Thread A falls back to read from storage, and gets the value 1, and prepares to write the value back to the cache.
3. Thread B writes a new value 2 for the key into storage.
4. Thread B updates the key with new value 2 in cache or invalidates the key (either option yields the same result).

5. Thread A writes the initially read value 1 back to the cache, creating an inconsistency between the cache and storage.

This classic problem arises from multiple writers updating the cache. Both the read and write paths can update the cache, which increases the complexity of the problem.

Using a distributed lock is an option, but it adds a heavy workload to the reading process and is not ideal. The CAS (compare and swap) mechanism is a better alternative. When reading from the cache, the application retrieves the value along with a version number that increments upon value changes. Writing back to the cache requires the previously read version number, and the value is successfully written only if the version number matches the current one in the cache.

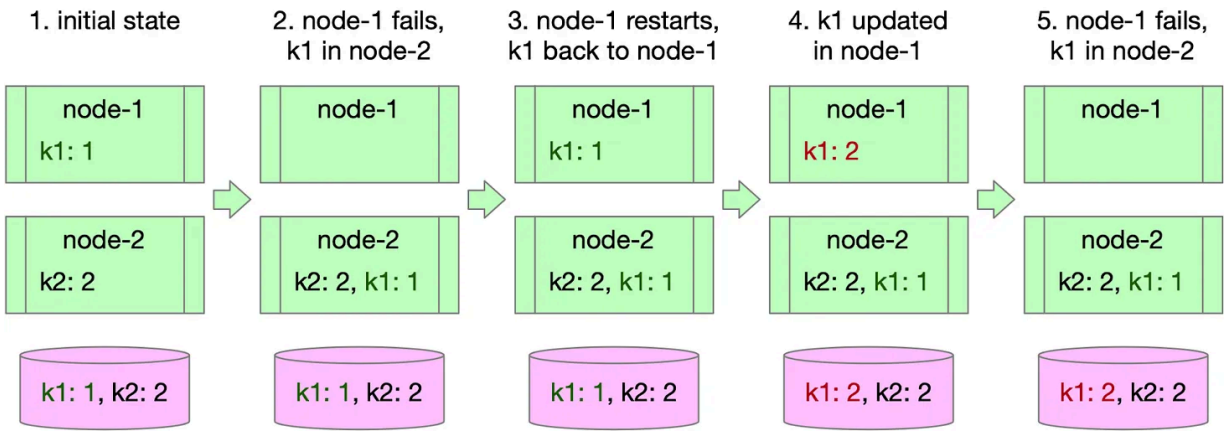
Reserving a dedicated version number for each key demands significant extra capacity, while using a global version number for all keys increases the probability of write failure due to frequent updates. A balanced approach involves maintaining a version number for each shard of keys, requiring minimal space while maintaining an acceptable write failure rate. This approach does, however, introduce complexity.

Handling Data Inconsistency in Distributed Cache Systems with Consistent Hashing

In distributed cache systems, consistent hashing is a widely used sharding strategy. When a cache node fails, the data partitions on it are automatically rehashed to other nodes. However, this flexible mechanism can also lead to data inconsistencies, as shown in the following example:

- Initially, cache node-1 stores k1, and node-2 stores k2.
- Node-1 fails, and k1 is rehashed to node-2, where it is loaded upon request.

- Node-1 restarts, and k1 is rehashed back to node-1, but k1's value in node-2 remains.
- k1's value is updated, affecting only node-1 and storage.
- Node-1 fails again, and k1 is rehashed back to node-2, where the previously loaded value is still valid, causing data inconsistency between cache node-2 and storage.



Data inconsistency caused by repeated node failures

This issue arises from a node's repeated failures and restarts and updates that affect only one node, leading to dirty data in other nodes. Several solutions can help address this problem or minimize its impact:

- Shorten the TTL (time-to-live) of cached data, allowing it to expire sooner and be reloaded from storage.
- Disable the automatic rehash function to prevent dirty data in the cache. In this case, standby nodes should automatically take over failed nodes to avoid cache avalanches.

- If the automatic rehash function remains enabled, find a way to eliminate or invalidate dirty data. However, this approach can be complex and may not be worth the effort.

Production and operational considerations

In this final section, we'll cover key production and operational considerations for caching, including monitoring and alerting, fault tolerance, and scalability. These elements are crucial for maintaining a reliable, high-performance caching system in a production environment.

Monitoring & alerting

Effective monitoring and alerting are essential for maintaining a healthy caching system. By continuously tracking key metrics and receiving timely alerts, we can quickly identify and address issues before they escalate. Consider the following best practices:

- Monitor cache hit ratio: Keep an eye on the cache hit ratio, which is the percentage of cache requests that return a cached value. A low cache hit ratio could indicate an inefficient caching strategy or the need for a larger cache size.
- Track cache evictions: Monitor the number of evictions occurring in the cache. Frequent evictions might suggest the cache is too small or the eviction policy is not optimal for your use case.
- Measure latency: Track the time it takes to read or write data to the cache and storage. High latency can impact the overall performance of your system.

- Alert on anomalies: Set up alerts for unusual spikes or dips in key metrics. This will help you detect potential problems early and take corrective action.

Fault tolerance

In distributed systems, it is common for individual nodes to experience failures. While data is often persisted in durable storage, ensuring no data loss even if all cache nodes are down, it is still crucial to design a cache system with high availability to maintain optimal performance.

To achieve fault tolerance in a distributed cache system, consider the following strategies:

Consistent hashing

Consistent hashing is a widely-used sharding strategy in distributed cache systems. When a node fails, the associated data shards are redistributed and rehashed across the remaining nodes. This approach allows the system to tolerate node failures while minimizing the impact on overall performance.

Data replication

Creating multiple replicas for each data shard can also improve fault tolerance. If a node fails, the corresponding data can still be accessed from its replicas. However, maintaining data consistency between replicas can be challenging. One approach to address this challenge is to use a primary-replica mode, as implemented in Redis replication. In this configuration, the primary node synchronizes data updates with its replica nodes, ensuring data consistency while providing redundancy in case of node failures.

Scalability

In distributed systems, scalability is a key consideration. A distributed cache system should be designed to expand as needed to support increased data capacity and traffic.

Sharding strategies play a crucial role in achieving scalability, as they partition the entire dataset into smaller, more manageable shards. The number of shards effectively determines the upper limit of the number of nodes that can be used. Choosing the most appropriate sharding strategy for your specific use case is essential.

In particular, consistent hashing is a popular sharding strategy due to its ability to maintain cluster availability during node additions or removals. This means that the cache system can scale seamlessly without causing downtime or disrupting performance, making it an attractive choice for many distributed caching scenarios.

Summary

Caching is a powerful, yet complex technique for optimizing data access and reducing storage traffic in various scenarios. To effectively employ caching, it's crucial to understand the key concepts and considerations. Here are some key takeaways to help guide you when working with caches:

- Caches are transient, fast, and store a partial dataset, making them ideal for speeding up data access and reducing storage traffic.

- Caching is best suited for scenarios with a high read-to-write ratio and minimal data updates.
- When designing your caching solution, consider:
    - Whether to use local or remote cache.
    - Cache replacement and invalidation strategies.
    - Cache capacity and data partitioning.
    - Cache access strategies.
    - Identifying and resolving cache-related issues.
- Measuring cache efficiency is essential, with the cache hit ratio serving as a useful metric.
- High availability and scalability are vital for distributed cache systems, which can be achieved through sharding strategies like consistent hashing and replication techniques.

By understanding these key points, you'll be better equipped to design, implement, and optimize cache solutions for your specific use cases. We hope this crash course has provided valuable insights and guidance for your caching endeavors. Thank you for reading!

188 Likes · 4 Restacks

6 Comments

Write a comment...

Oliver Mar 31, 2023 Liked by Alex Xu

Here is a paragraph from the article : 'The challenge of using a bloom filter lies in its capacity limitations. For instance, 1 billion keys with a 1% false positive rate would require approximately 1.2 GB of capacity. This solution is best suited for smaller data sets."

Could you please explain how you calculated the required capacity of 1.2GB? Thanks.

LIKE (5) REPLY SHARE

1 reply by Alex Xu

Julia May 6, 2023

It comes to me that all the challenges mentioned on cache actually can be seen challenges to any distributed system.

This article really inspired me to think more about the potential challenges and solutions on designing a distributed system.

Could you please have an article on that next? I would be thrilled to read it.

Thanks!

LIKE (2) REPLY SHARE

4 more comments...