**Chapter 4. Caching Policies and Strategies**

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In computing, a cache is a component or mechanism used to store frequently accessed data or instructions closer to the processor, reducing the latency of retrieving the information from slower storage or external resources. Caches are typically implemented as high-speed, small-capacity memories located closer to the CPU than the main memory. The goal is to improve overall system performance by reducing the time required to access data or instructions.

The concept of cache revolves around the principle of locality, which suggests that programs tend to access a relatively small portion of their data or instructions repeatedly. By storing this frequently accessed information in a cache, subsequent access to the same data can be served faster, resulting in improved performance.

When data is accessed from a cache, there are two possible outcomes: cache hit and cache miss. A cache hit occurs when the requested data is found in the cache, allowing for fast retrieval without accessing the slower main memory or external resources. On the other hand, a cache miss happens when the requested data is not present in the cache, requiring the system to fetch the data from the main memory or external storage. Cache hit rates measure the effectiveness of the cache in serving requests without needing to access slower external storage, while cache miss rates indicate how often the cache fails to serve requested data.

This chapter will cover important information to help you understand how to use data caching effectively. We’lll talk about cache eviction policies, which are rules for deciding when to remove data from the cache to make retrieval of important data faster. We’ll also cover cache invalidation, which ensures the cached data is always correct and matches the real underlying data source. The chapter will also discuss caching strategies for both read and write intensive applications. We’ll also cover how to actually put caching into action, including where to put the caches to get the best results. You’ll also learn about different ways caching works and why Content Delivery Networks (CDNs) are important. And finally, you’ll learn about two popular open-source caching solutions. So, let’s get started with the benefits of caching.

**Caching Benefits**

Caches play a crucial role in improving system performance and reducing latency for several reasons:

*Faster Access*

Caches offer faster access times compared to main memory or external storage. By keeping frequently accessed data closer to the CPU, cache access times can be significantly lower, reducing the time required to fetch data.

*Reduced Latency*

Caches help reduce latency by reducing the need to access slower storage resources. By serving data from a cache hit, the system avoids the delay associated with fetching data from main memory or external sources, thereby reducing overall latency.

*Bandwidth Optimization*

Caches help optimize bandwidth usage by reducing the number of requests sent to slower storage. When data is frequently accessed from the cache, it reduces the demand on the memory bus or external interfaces, freeing up resources for other operations.

*Improved Throughput*

Caches improve overall system throughput by allowing the CPU to access frequently needed data quickly, without waiting for slower storage access. This enables the CPU to perform more computations in a given amount of time, increasing overall system throughput.

Amdahl’s Law and the Pareto distribution provide further insights into the benefits of caching:

*Amdahl’s Law*

Amdahl’s Law states that the overall speedup achieved by optimizing a particular component of a system is limited by the fraction of time that component is utilized. Caches, being a critical optimization component, can have a significant impact on overall system performance, especially when the fraction of cache hits is high. Amdahl’s Law emphasizes the importance of efficient caching to maximize the benefits of performance optimization.

*Pareto Distribution*

The Pareto distribution, also known as the 80/20 rule, states that a significant portion of the system’s workload is driven by a small fraction of the data. Caching aligns well with this distribution by allowing the frequently accessed data to reside in a fast cache, serving the most critical operations efficiently. By focusing caching efforts on the most accessed data, the Pareto distribution can be leveraged to optimize performance for the most important workloads.

In summary, caches provide faster access to frequently accessed data, reducing latency and improving overall system performance. They help optimize bandwidth, increase throughput, and align with principles such as Amdahl’s Law and the Pareto distribution to maximize performance benefits.

The next section will cover different policies to perform cache eviction, like techniques such as least recently used (LRU) and least frequently used (LFU), which can help you choose the best caching policy for different situations.

**Cache Eviction Policies**

Caching plays a crucial role in improving the performance and efficiency of data retrieval systems by storing frequently accessed data closer to the consumers. Caching policies determine how the cache handles data eviction and replacement, when its capacity is reached. Cache eviction policies try to maximize the cache hit ratio—the percentage of time the requested item was found in the cache and served. Higher cache hit ratio reduces the necessity to retrieve data from external storage, resulting in better system performance. In this section, we will explore various caching policies, including Belady’s algorithm, queue-based policies (FIFO, LIFO), recency-based policies (LRU, TLRU, MRU, SLRU), and frequency-based policies (LFU, LFRU).

**Belady’s Algorithm**

Belady’s algorithm is an optimal caching algorithm that evicts the data item that will be used furthest in the future. It requires knowledge of the future access pattern, which is usually impractical to obtain. Belady’s algorithm serves as a theoretical benchmark for evaluating the performance of other caching policies.

*Queue-Based Policies*

Queue-based cache eviction policies involve managing the cache by treating it like a queue. When the cache reaches its capacity, one of the queue-based policies is used to remove data to make space for new data.

*FIFO (First-In-First-Out)*

FIFO is a simple caching policy that evicts the oldest data item from the cache. It follows the principle that the first data item inserted into the cache is the first one to be evicted when the cache is full. FIFO is easy to implement but may suffer from the “aging” problem, where recently accessed items are evicted prematurely.

*LIFO (Last-In-First-Out)*

LIFO is the opposite of FIFO, where the most recently inserted data item is the first one to be evicted. LIFO does not consider the access pattern and can result in poor cache utilization and eviction decisions.

*Recency-Based Policies*

Recency-based cache eviction policies focus on the time aspect of data access. These policies prioritize keeping the most recently accessed items in the cache.

*LRU (Least Recently Used)*

LRU is a popular caching policy that evicts the least recently accessed data item from the cache. It assumes that recently accessed items are more likely to be accessed in the near future. LRU requires tracking access timestamps for each item, making it slightly more complex to implement.

*MRU (Most Recently Used)*

MRU evicts the most recently accessed data item from the cache. It assumes that the most recently accessed item is likely to be accessed again soon. MRU can be useful in scenarios where a small subset of items is accessed frequently.

*Frequency-Based Policies*

Frequency-based cache eviction policies prioritize retaining items in the cache based on how often they are accessed. The cache replaces items that have been accessed the least frequently, assuming that rarely accessed data may not be as critical for performance optimization.

*LFU (Least Frequently Used)*

LFU evicts the least frequently accessed data item from the cache. It assumes that items with lower access frequency are less likely to be accessed in the future. LFU requires maintaining access frequency counts for each item, which can be memory-intensive.

*LFRU (Least Frequently Recently Used)*

LFRU combines the concepts of LFU and LRU by considering both the frequency of access and recency of access. It evicts the item with the lowest frequency count among the least recently used items.

**Allowlist Policy**

An allowlist policy for cache replacement is a mechanism that defines a set of prioritized items eligible for retention in a cache when space is limited. Instead of using a traditional cache eviction policy that removes the least recently used or least frequently accessed items, an allowlist policy focuses on explicitly specifying which items should be preserved in the cache. This policy ensures that important or high-priority data remains available in the cache, even during periods of cache pressure. By allowing specific items to remain in the cache while evicting others, the allowlist policy optimizes cache utilization and improves performance for critical data access scenarios.

Caching policies serve different purposes and exhibit varying performance characteristics based on the access patterns and workload of the system. Choosing the right caching policy depends on the specific requirements and characteristics of the application.

By understanding and implementing these caching policies effectively, system designers and developers can optimize cache utilization, improve data retrieval performance, and enhance the overall user experience. Let’s discuss different cache invalidation strategies, which are applied post identifying which data to evict based on the above cache eviction policies.

**Cache Invalidation**

Cache invalidation is a crucial aspect of cache management that ensures the cached data remains consistent with the underlying data source. Effective cache invalidation strategies help maintain data integrity and prevent stale or outdated data from being served. Here are three common cache invalidation techniques: active invalidation, invalidating on modification, invalidating on read, and time-to-live (TTL).

*Active Invalidation*

Active invalidation involves explicitly removing or invalidating cached data when changes occur in the underlying data source. This approach requires the application or the system to actively notify or trigger cache invalidation operations. For example, when data is modified or deleted in the data source, the cache is immediately updated or cleared to ensure that subsequent requests fetch the latest data. Active invalidation provides precise control over cache consistency but requires additional overhead to manage the invalidation process effectively.

*Invalidating on Modification*

With invalidating on modification, the cache is invalidated when data in the underlying data source is modified. When a modification operation occurs, such as an update or deletion, the cache is notified or flagged to invalidate the corresponding cached data. The next access to the invalidated data triggers a cache miss, and the data is fetched from the data source, ensuring the cache contains the most up-to-date information. This approach minimizes the chances of serving stale data but introduces a slight delay for cache misses during the invalidation process.

*Invalidating on Read*

In invalidating on read, the cache is invalidated when the cached data is accessed or read. Upon receiving a read request, the cache checks if the data is still valid or has expired. If the data is expired or flagged as invalid, the cache fetches the latest data from the data source and updates the cache before serving the request. This approach guarantees that fresh data is always served, but it adds overhead to each read operation since the cache must validate the data’s freshness before responding.

*Time-to-Live (TTL)*

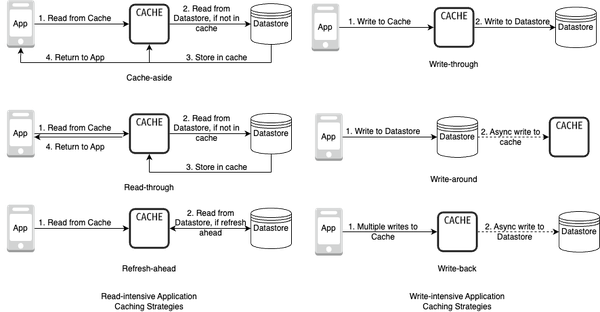
Time-to-Live is a cache invalidation technique that associates a time duration with each cached item. When an item is stored in the cache, it is marked with a TTL value indicating how long the item is considered valid. After the TTL period elapses, the cache treats the item as expired, and subsequent requests for the expired item trigger cache misses, prompting the cache to fetch the latest data from the data source. TTL-based cache invalidation provides a simple and automatic way to manage cache freshness, but it may result in serving slightly stale data until the TTL expires.

The choice of cache invalidation strategy depends on factors such as the nature of the data, the frequency of updates, the performance requirements, and the desired consistency guarantees. Active invalidation offers precise control but requires active management, invalidating on modification ensures immediate data freshness, invalidating on read guarantees fresh data on every read operation, and TTL-based invalidation provides a time-based expiration mechanism. Understanding the characteristics of the data and the system’s requirements helps in selecting the appropriate cache invalidation strategy to maintain data consistency and improve overall performance.

The next section covers different caching strategies to ensure that data is properly consistent between the cache and the underlying data source.

**Caching Strategies**

Caching strategies define how data is managed and synchronized between the cache and the underlying data source. In this section, we will explore several caching strategies as shown in [Figure 4-1](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch04.html#fig_1_caching_strategies), including cache-aside, read-through, refresh-ahead, write-through, write-around, and write-back.



**Figure 4-1. Caching Strategies**

The left-hand side of the diagram displays read-intensive caching strategies, focusing on optimizing the retrieval of data that is frequently read or accessed. The goal of a read-intensive caching strategy is to minimize the latency and improve the overall performance of read operations by serving the cached data directly from memory, which is much faster than fetching it from a slower and more distant data source. This strategy is particularly beneficial for applications where the majority of operations involve reading data rather than updating or writing data.

Let’s take a look at those in more detail:

*Cache-Aside*

Cache-aside caching strategy, also known as lazy loading, delegates the responsibility of managing the cache to the application code. When data is requested, the application first checks the cache. If the data is found, it is returned from the cache. If the data is not in the cache, the application retrieves it from the data source, stores it in the cache, and then returns it to the caller. Cache-aside caching offers flexibility as the application has full control over caching decisions but requires additional logic to manage the cache.

*Read-Through*

Read-through caching strategy retrieves data from the cache if available; otherwise, it fetches the data from the underlying data source. When a cache miss occurs for a read operation, the cache retrieves the data from the data source, stores it in the cache for future use, and returns the data to the caller. Subsequent read requests for the same data can be served directly from the cache, improving the overall read performance. This strategy offloads the responsibility of managing cache lookups from the application unlike Cache-aside strategy, providing a simplified data retrieval process.

*Refresh-Ahead*

Refresh-ahead caching strategy, also known as prefetching, proactively retrieves data from the data source into the cache before it is explicitly requested. The cache anticipates the future need for specific data items and fetches them in advance. By prefetching data, the cache reduces latency for subsequent read requests and improves the overall data retrieval performance.

The right-hand side of the diagram displays the write-intensive strategies, focussing around optimizing the storage and management of data that is frequently updated or written. Unlike read-intensive caching, where the focus is on optimizing data retrieval, a write-intensive caching strategy aims to enhance the efficiency of data updates and writes, while still maintaining acceptable performance levels. In a write-intensive caching strategy, the cache is designed to handle frequent write operations, ensuring that updated data is stored temporarily in the cache before being eventually synchronized with the underlying data source, such as a database or a remote server. This approach can help reduce the load on the primary data store and improve the application’s responsiveness by acknowledging write operations more quickly.

Let’s take a look at those in more detail:

*Write-Through*

Write-through caching strategy involves writing data to both the cache and the underlying data source simultaneously. When a write operation occurs, the data is first written to the cache and then immediately propagated to the persistent storage synchronously before the write operation is considered complete. This strategy ensures that the data remains consistent between the cache and the data source. However, it may introduce additional latency due to the synchronous write operations.

*Write-Around*

Write-around caching strategy involves bypassing the cache for write operations. When the application wants to update data, it writes directly to the underlying data source, bypassing the cache. As a result, the written data does not reside in the cache, reducing cache pollution with infrequently accessed data. However, subsequent read operations for the updated data might experience cache misses until the data is fetched again from the data source and cached.

*Write-Back*

Write-back caching strategy allows write operations to be performed directly on the cache, deferring the update to the underlying data source until a later time. When data is modified in the cache, the change is recorded in the cache itself, and the update is eventually propagated to the data source asynchronously on schedule or or when specific conditions are met (e.g., cache eviction, time intervals). Write-back caching provides faster write operations by reducing the number of immediate disk writes. However, it introduces a potential risk of data loss in the event of a system failure before the changes are flushed to the data source.

Each caching strategy has its own advantages and considerations, and the selection of an appropriate strategy depends on the specific requirements and characteristics of the system.

By understanding these caching strategies, system designers and developers can make informed decisions to optimize data access and improve the overall performance of their applications. Let’s cover different deployment options for a cache in the overall system and how it affects the performance and data sharing.

**Caching Deployment**

When deploying a cache, various deployment options are available depending on the specific requirements and architecture of the system. Here are three common cache deployment approaches: in-process caching, inter-process caching, and remote caching.

*In-Process Caching*

In in-process caching, the cache resides within the same process or application as the requesting component. The cache is typically implemented as an in-memory data store and is directly accessible by the application or service. In-process caching provides fast data access and low latency since the cache is located within the same process, enabling direct access to the cached data. This deployment approach is suitable for scenarios where data sharing and caching requirements are limited to a single application or process.

*Inter-Process Caching*

Inter-process caching involves deploying the cache as a separate process or service that runs alongside the applications or services. The cache acts as a dedicated caching layer that can be accessed by multiple applications or processes. Applications communicate with the cache using inter-process communication mechanisms such as shared memory, pipes, sockets, or remote procedure calls (RPC). Inter-process caching allows multiple applications to share and access the cached data, enabling better resource utilization and data consistency across different components. It is well-suited for scenarios where data needs to be shared and cached across multiple applications or processes within a single machine.

*Remote Caching*

Remote caching involves deploying the cache as a separate service or cluster that runs on a different machine or location than the requesting components. The cache service is accessed remotely over a network using protocols such as HTTP, TCP/IP, or custom communication protocols. Remote caching enables distributed caching and can be used to share and cache data across multiple machines or even geographically distributed locations. It provides scalability, fault-tolerance, and the ability to share cached data among different applications or services running on separate machines. Remote caching is suitable for scenarios that require caching data across a distributed system or when the cache needs to be accessed by components running on different machines.

The choice of cache deployment depends on factors such as the scale of the system, performance requirements, data sharing needs, and architectural considerations. In-process caching offers low latency and direct access to data within a single process, inter-process caching enables sharing and caching data across multiple applications or processes, and remote caching provides distributed caching capabilities across multiple machines or locations. Understanding the specific requirements and characteristics of the system helps in selecting the appropriate cache deployment strategy to optimize performance and resource utilization. Let’s cover different caching mechanisms to improve application performance in the next section.

**Caching Mechanisms**

In this section, we will explore different caching mechanisms, including client-side caching, CDN caching, web server caching, application caching, database caching, query-level caching, and object-level caching.

*Client-side Caching*

Client-side caching involves storing cached data on the client device, typically in the browser’s memory or local storage. This mechanism allows web applications to store and retrieve static resources, such as HTML, CSS, JavaScript, and images, directly from the client’s device. Client-side caching reduces the need to fetch resources from the server on subsequent requests, leading to faster page load times and improved user experience.

*CDN Caching*

Content Delivery Network (CDN) caching is a mechanism that involves caching static content on distributed servers strategically located across different geographic regions. CDNs serve cached content to users based on their proximity to the CDN server, reducing the latency and load on the origin server. CDN caching is commonly used to cache static files, media assets, and other frequently accessed content, improving the overall performance and scalability of web applications.

*Web Server Caching*

Web server caching refers to caching mechanisms implemented at the server-side to store frequently accessed content. When a request is made to the server, it first checks if the requested content is already cached. If found, the server serves the cached content directly, avoiding the need to regenerate the content. Web server caching is effective for static web pages, dynamic content with a long expiration time, and content that is expensive to generate.

*Application Caching*

Application caching involves caching data within the application’s memory or in-memory databases. It is typically used to store frequently accessed data or computation results that are costly to generate or retrieve from other data sources. Application caching improves response times by reducing the need for repeated data retrieval and computation, enhancing the overall performance of the application.

*Database Caching*

Database caching focuses on improving the performance of database operations by caching frequently accessed data or query results. This caching mechanism can be implemented at different levels: query-level caching and object-level caching.

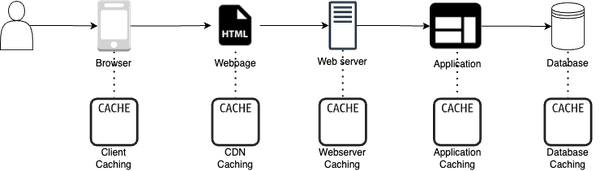
*Query-Level Caching*

Query-level caching involves storing the results of frequently executed queries in memory. When the same query is executed again, the cached result is served instead of querying the database again, reducing the database load and improving response times.

*Object-Level Caching*

Object-level caching caches individual data objects or records retrieved from the database. This mechanism is useful when accessing specific objects frequently or when the database is relatively static. Object-level caching reduces the need for frequent database queries, improving overall application performance.

By employing these caching mechanisms as shown in [Figure 4-3](https://learning.oreilly.com/library/view/system-design-on/9781098146887/ch04.html#fig_3_redis_deployment_setups), organizations and developers can optimize data retrieval, reduce latency, and improve the scalability and responsiveness of their systems. However, it is essential to carefully consider cache invalidation, cache coherence, and cache management strategies to ensure the consistency and integrity of the cached data.



**Figure 4-2. Caching mechanisms employed at different stages**

Out of the above mechanisms, Content Delivery Networks (CDNs) play a crucial role in improving the performance and availability of web content to end-users by reducing latency and enhancing scalability by caching at edge locations. Let’s cover CDNs in detail in the next section.

**Content Delivery Networks**

CDNs employ various strategies and architectural models to efficiently distribute and cache content across geographically distributed servers. This section explores different types of CDNs, including push and pull CDNs, optimization techniques for CDNs, and methods for ensuring content consistency within CDNs.

CDNs can be categorized into two main types: push and pull CDNs.

*Push CDN*

In a push CDN, content is pre-cached and distributed to edge servers in advance. The CDN provider proactively pushes content to edge locations based on predicted demand or predetermined rules. With push CDNs, content is only uploaded when it is new or changed, reducing traffic while maximizing storage efficiency. This approach ensures faster content delivery as the content is readily available at the edge servers when requested by end-users.

Push CDNs are suitable for websites with low traffic or content that doesn’t require frequent updates. Instead of regularly pulling content from the server, it is uploaded to the CDNs once and remains there until changes occur.

*Pull CDN*

In a pull CDN, content is cached on-demand. The CDN servers pull the content from the origin server when the first user requests it. The content is then cached at the edge servers for subsequent requests, optimizing delivery for future users. The duration for which content is cached is determined by a time-to-live (TTL) setting. Pull CDNs minimize storage space on the CDN, but there can be redundant traffic if files are pulled before their expiration, resulting in unnecessary data transfer.

Pull CDNs are well-suited for websites with high traffic since recently-requested content remains on the CDN, evenly distributing the load.

CDNs employ different optimization techniques to improve the performance of caching at the edge server. Let’s cover some of these optimization techniques in detail.

*Dynamic Content Caching Optimization*

CDNs face challenges when caching dynamic content that frequently changes based on user interactions or real-time data. To optimize dynamic content caching, CDNs employ various techniques such as:

*Content Fragmentation*

Breaking down dynamic content into smaller fragments to enable partial caching and efficient updates.

*Edge-Side Includes (ESI)*

Implementing ESI tags to separate dynamic and static content, allowing dynamic portions to be processed on-the-fly while caching the static fragments.

*Content Personalization*

Leveraging user profiling and segmentation techniques to cache personalized or user-specific content at the edge servers.

*Multi-tier CDN architecture*

Multi-tier CDN architecture involves the distribution of content across multiple layers or tiers of edge servers. This approach allows for better scalability, fault tolerance, and improved content delivery to geographically diverse regions. It enables efficient content replication and reduces latency by bringing content closer to end-users.

*DNS Redirection*

DNS redirection is a mechanism employed by CDNs to direct user requests to the nearest or most suitable edge server. By resolving DNS queries to the most appropriate server based on factors like geographic proximity, network conditions, and server availability, CDNs optimize content delivery and reduce latency.

*Client Multiplexing*

Client multiplexing refers to the technique of combining multiple HTTP requests and responses into a single connection between the client and the edge server. This reduces the overhead of establishing multiple connections and improves efficiency, especially for small object requests, resulting in faster content delivery. Content Consistency in CDNs

Ensuring content consistency across multiple edge servers within a CDN is crucial to deliver the most up-to-date and accurate content. CDNs employ various methods to maintain content consistency, including:

*Periodic Polling*

CDNs periodically poll the origin server to check for updates or changes in content. This ensures that cached content is refreshed to reflect the latest version.

*Time-to-Live (TTL)*

CDNs utilize Time-to-Live values, specified in HTTP headers or DNS records, to determine how long cached content remains valid. Once the TTL expires, the CDN fetches updated content from the origin server.

*Leases*

CDNs use lease-based mechanisms to control the duration of content caching at the edge servers. Leases define a specific time window during which the content remains valid before requiring renewal or revalidation.

**NOTE**

AWS offers Amazon Cloudfront, a pull CDN offering built for high performance, security, and developer convenience, which we will cover in more detail in Chapter 9 - AWS Network Services.

Before ending the section, let’s also understand that using a CDN can come with certain drawbacks also due to cost, stale content and frequent URL changes. CDNs may involve significant costs depending on the amount of traffic. However, it’s important to consider these costs in comparison to the expenses you would incur without utilizing a CDN. If updates are made before the TTL expires, there is a possibility of content being outdated until it is refreshed on the CDN. CDNs require modifying URLs for static content to point to the CDN, which can be an additional task to manage.

Overall, CDNs offer benefits in terms of performance and scalability but require careful consideration of these factors and the specific needs of your website. At the end of this chapter, let’s dive deeper into two popular open-source caching solutions to understand their architecture and how they implement the caching concepts discussed in the chapter.

**Open Source Caching Solutions**

Open source caching solutions, such as Redis and Memcached, have gained popularity due to their efficiency, scalability, and ease of use. Let’s take a closer look at Memcached and Redis, two widely adopted open-source caching solutions.

**Memcached**

Memcached is an open-source, high-performance caching solution widely used in web applications. It operates as a distributed memory object caching system, storing data in memory across multiple servers. Here are some key features and benefits of Memcached:

*Simple and Lightweight*

Memcached is designed to be simple, lightweight, and easy to deploy. It focuses solely on caching and provides a straightforward key-value interface for data storage and retrieval.

*Horizontal Scalability*

Memcached follows a distributed architecture, allowing it to scale horizontally by adding more servers to the cache cluster. This distributed approach ensures high availability, fault tolerance, and improved performance for growing workloads.

*Protocol Compatibility*

Memcached adheres to a simple protocol that is compatible with various programming languages. This compatibility makes it easy to integrate Memcached into applications developed in different languages.

*Transparent Caching Layer*

Memcached operates as a transparent caching layer, sitting between the application and the data source. It helps alleviate database or API load by caching frequently accessed data, reducing the need for repetitive queries.

Let’s take a look at Memcached’s architecture.

**Memcached Architecture**

Memcached’s architecture consists of a centralized server that coordinates the storage and retrieval of cached data. When a client sends a request to store or retrieve data, the server handles the request and interacts with the underlying memory allocation strategy.

Memcached follows a multi-threaded architecture that enables it to efficiently handle concurrent requests and scale across multiple CPU cores. In this architecture, Memcached utilizes a pool of worker threads that can simultaneously process client requests. Each worker thread is responsible for handling a subset of incoming requests, allowing for parallel execution and improved throughput. This multi-threaded approach ensures that Memcached can effectively handle high traffic loads and distribute the processing workload across available CPU resources. By leveraging multiple threads, Memcached can achieve better performance and responsiveness, making it suitable for demanding caching scenarios where high concurrency is a requirement.

In terms of memory allocation, Memcached employs a slab allocation strategy. It divides the allocated memory into fixed-size chunks called slabs. Each slab is further divided into smaller units known as pages. These pages are then allocated to store individual cache items. The slab allocation strategy allows Memcached to efficiently manage memory by grouping items of similar sizes together. It reduces memory fragmentation and improves memory utilization.

When a new item is added to the cache, Memcached determines the appropriate slab size for the item based on its size. If an existing slab with enough free space is available, the item is stored in that slab. Otherwise, Memcached allocates a new slab from the available memory pool and adds the item to that slab. The slab allocation strategy enables efficient memory utilization and allows Memcached to store a large number of items in memory while maintaining optimal performance.

Overall, Memcached’s architecture and memory allocation strategy work together to provide a lightweight and efficient caching solution that can handle high traffic loads and deliver fast data access times. By leveraging memory effectively and employing a scalable architecture, Memcached enables applications to significantly improve performance by caching frequently accessed data in memory.

**Redis**

Redis, short for **Re**mote **Di**ctionary **S**erver, is a server-based in-memory data structure store that can serve as a high-performance cache. Unlike traditional databases that rely on iterating, sorting, and ordering rows, Redis organizes data in customizable data structures from the ground up, supporting a wide range of data types, including strings, bitmaps, bitfields, lists, sets, hashes, geospatial, hyperlog and more, making it versatile for various caching use cases. Here are some key features and benefits of Redis:

*High Performance*

Redis is designed for speed, leveraging an in-memory storage model that allows for extremely fast data retrieval and updates. It can handle a massive number of operations per second, making it suitable for high-demand applications.

*Persistence Options*

Redis provides persistence options that allow data to be stored on disk, ensuring durability even in the event of system restarts. This feature makes Redis suitable for use cases where data needs to be retained beyond system restarts or cache invalidations.

*Advanced Caching Features*

Redis offers advanced caching features, such as expiration times, eviction policies, and automatic cache invalidation based on time-to-live (TTL) values. It also supports data partitioning and replication for scalability and fault tolerance.

*Pub/Sub and Messaging*

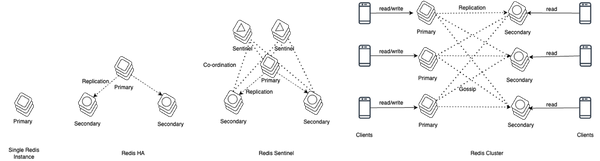
Redis includes publish/subscribe (pub/sub) messaging capabilities, enabling real-time messaging and event-driven architectures. This makes it suitable for scenarios involving real-time data updates and notifications.

Redis serves as an in-memory database primarily used as a cache in front of other databases like MySQL or PostgreSQL. By leveraging the speed of memory, Redis enhances application performance and reduces the load on the main database. It is particularly useful for storing data that changes infrequently but is frequently requested, as well as data that is less critical but undergoes frequent updates. Examples of such data include session or data caches, leaderboard information, and roll-up analytics for dashboards.

Redis architecture is designed for high performance, low latency, and simplicity. It provides a range of deployment options for ensuring high availability based on the requirements and cost constraints. Let’s go over the availability in Redis deployments in detail, followed by persistence models for redis durability and memory management in Redis.

**Availability in Redis Deployments**

Redis supports different deployment architectures as shown in Figure 4-3, including a single Redis instance, Redis HA (High Availability), Redis Sentinel, and Redis Cluster. Each architecture has its trade-offs and is suitable for different use cases and scalability needs.



**Figure 4-3. Redis Deployment Setups**

*Single Redis Instance*

In a single Redis instance setup, Redis is deployed as a standalone server. While it is straightforward and suitable for small instances, it lacks fault tolerance. If the instance fails or becomes unavailable, all client calls to Redis will fail, affecting overall system performance.

*Redis HA (High Availability)*

Redis HA involves deploying a main Redis instance with one or more secondary instances that synchronize with replication. The secondary instances can help scale read operations or provide failover in case the main instance is lost. Replication ID and offset play a crucial role in the synchronization process, allowing secondary instances to catch up with the main instance’s data.

*Redis Sentinel*

Redis Sentinel is a distributed system that ensures high availability for Redis. Sentinel processes coordinate the state and monitor the availability of main and secondary instances. They also serve as a point of discovery for clients, informing them of the current main instance. Sentinel processes can start a failover process if the primary instance becomes unavailable.

*Redis Cluster*

Redis Cluster enables horizontal scaling by distributing data across multiple machines or shards. Algorithmic sharding is used to determine which Redis instance (shard) holds a specific key. Redis Cluster employs a hashsloting mechanism to map data to shards and allows for seamless resharding when adding new instances to the cluster. Gossip Protocol is used in Redis Cluster to maintain cluster health. Nodes constantly communicate to determine the availability of shards and can promote secondary instances to primary if needed.

**Durability in Redis Deployment**

Redis provides two persistence models for data durability: RDB files (Redis Database Files) and AOF (Append-Only File). These persistence mechanisms ensure that data is not lost in case of system restarts or crashes. Let’s explore both models in more detail:

**RDB Files (Redis Database Files)**

RDB is the default persistence model in Redis. It periodically creates snapshots of the dataset and saves them as binary RDB files. These files capture the state of the Redis database at a specific point in time. Here are key features and considerations of RDB persistence:

*Snapshot-based Persistence*

RDB persistence works by periodically taking snapshots of the entire dataset and storing it in a file. The frequency of snapshots can be configured based on requirements.

*Efficiency and Speed*

RDB files are highly efficient in terms of disk space usage and data loading speed. They are compact and can be loaded back into Redis quickly, making it suitable for scenarios where fast recovery is essential.

*Full Data Recovery*

RDB files provide full data recovery as they contain the entire dataset. In case of system failures, Redis can restore the data by loading the most recent RDB file available.

However, it’s worth noting that RDB files have some limitations. Since they are snapshots, they do not provide real-time durability and may result in data loss if a crash occurs between two snapshot points. Additionally, restoring large RDB files can take time and impact the system’s performance during the recovery process.

**AOF (Append-Only File)**

AOF persistence is an alternative persistence model in Redis that logs every write operation to an append-only file. AOF captures a sequential log of write operations, enabling Redis to reconstruct the dataset by replaying the log. Here are key features and considerations of AOF persistence:

*Write-ahead Log*

AOF persists every write operation to the append-only file as a series of commands or raw data. This log can be used to rebuild the dataset from scratch.

*Durability and Flexibility*

AOF offers more durability than RDB files since it captures every write operation. It provides the ability to recover data up to the last executed command. Moreover, AOF offers different persistence options (such as every write, every second, or both) to balance durability and performance.

*Append-only Nature*

AOF appends new write operations to the end of the file, ensuring that the original dataset is never modified. This approach protects against data corruption caused by crashes or power failures.

However, AOF persistence comes with its own considerations. The append-only file can grow larger over time, potentially occupying significant disk space. Redis offers options for AOF file rewriting to compact the log and reduce its size. Additionally, AOF persistence typically has a slightly higher performance overhead compared to RDB files due to the need to write every command to disk.

In practice, Redis users often employ a combination of RDB and AOF persistence based on their specific requirements and trade-offs between performance, durability, and recovery time objectives.

It’s important to note that Redis also provides an option to use no persistence (volatile mode) if durability is not a primary concern or if data can be regenerated from an external source in the event of a restart or crash.

**Memory Management in Redis**

Redis leverages forking and copy-on-write (COW) techniques to facilitate data persistence efficiently within its single-threaded architecture. When Redis performs a snapshot (RDB) or background saving operation, it follows these steps:

1. 1. Forking: Redis uses the fork() system call to create a child process, which is an identical copy of the parent process. Forking is a lightweight operation as it creates a copy-on-write clone of the parent’s memory.
2. 2. Copy-on-Write (COW): Initially, the child process shares the same memory pages with the parent process. However, when either the parent or child process modifies a memory page, COW comes into play. Instead of immediately duplicating the modified page, the operating system creates a new copy only when necessary.

By employing COW, Redis achieves the following benefits:

*Memory Efficiency*

When the child process is initially created, it shares the same memory pages with the parent process. This shared memory approach consumes minimal additional memory. Only the modified pages are copied when necessary, saving memory resources.

*Performance*

Since only the modified pages are duplicated, Redis can take advantage of the COW mechanism to perform persistence operations without incurring a significant performance overhead. This is particularly beneficial for large datasets where copying the entire dataset for persistence would be time-consuming.

*Fork Safety*

Redis uses fork-based persistence to avoid blocking the main event loop during the snapshot process. By forking a child process, the parent process can continue serving client requests while the child process performs the persistence operation independently. This ensures high responsiveness and uninterrupted service.

It’s important to note that while forking and COW provide memory efficiency and performance benefits, they also have considerations. Forking can result in increased memory usage during the copy-on-write process if many modified pages need to be duplicated. Additionally, the fork operation may be slower on systems with large memory footprints.

Overall, Redis effectively utilizes forking and copy-on-write mechanisms within its single-threaded architecture to achieve efficient data persistence. By employing these techniques, Redis can perform snapshots and background saving operations without significantly impacting its performance or memory usage.

Overall, Redis offers developers a powerful and flexible data storage solution with various deployment options and capabilities.

Both Redis and Memcached are excellent open-source caching solutions with their unique strengths. The choice between them depends on specific requirements and use cases. Redis is suitable for scenarios requiring versatile data structures, persistence, pub/sub messaging, and advanced caching features. On the other hand, Memcached shines in simple, lightweight caching use cases that prioritize scalability and ease of integration.

**NOTE**

AWS offers Amazon Elasticache, compatible with both Redis and Memcached for real-time use cases like caching, session stores, gaming, geo-spatial services, real-time analytics, and queuing, which we will cover in more detail in Chapter 10 - AWS Storage Services.

**Conclusion**

In concluding this chapter on caching, we have journeyed through a comprehensive exploration of the fundamental concepts and strategies that empower efficient data caching. We’ve covered cache eviction policies, cache invalidation mechanisms, and a plethora of caching strategies, equipping you with the knowledge to optimize data access and storage. We’ve delved into caching deployment, understanding how strategic placement can maximize impact, and explored the diverse caching mechanisms available. Additionally, we’ve touched upon Content Delivery Networks (CDNs) and open-source caching solutions including Redis and Memcached, that offer robust options for enhancing performance. By incorporating Redis or Memcached into your architecture, you can significantly improve application performance, reduce response times, and enhance the overall user experience by leveraging the power of in-memory caching.

As we move forward in our exploration of enhancing system performance, the next chapter will embark on an exploration of scaling and load balancing strategies. Scaling is a pivotal aspect of modern computing, allowing systems to handle increased loads gracefully. We will also delve into strategies for load balancing in distributing incoming traffic efficiently. Together, these topics will empower you to design and maintain high-performing systems that can handle the demands of today’s dynamic digital landscape.