redislabs
home of redis

whitepaper

15 Reasons to use Redis as an Application Cache

Author: Itamar Haber

Table of Contents

Executive Summary

Caching is a technique used to accelerate application response times and help applications scale by placing frequently needed data very close to the application. Redis, an open source, in-memory, data structure server is frequently used as a distributed shared cache (in addition to being used as a message broker or database) because it enables true statelessness for an applications' processes, while reducing duplication of data or requests to external data sources. This paper outlines 15 reasons that make Redis an ideal caching solution for every application.

Introduction

Software processes are made of data; every process' instructions, inputs, runtime state and outputs are data stored somewhere. Based on the assumption that a datum's speed can't be greater than that of a photon's, the closer the data is to its processing unit, the more performant that unit's processing will be.

In modern computing environments, processes are run at different layers, from low-level processes that are embedded in the hardware to high-level abstractions, such as clusters. While this paper focuses on typical standalone stateless application server processes, most of the concepts herein are applicable for a wide range of use cases. The main benefit of using stateless standalone application processes is application scalability while distributing multiple processes.

Let's break down a typical application server process into three post-bootstrapped stages:

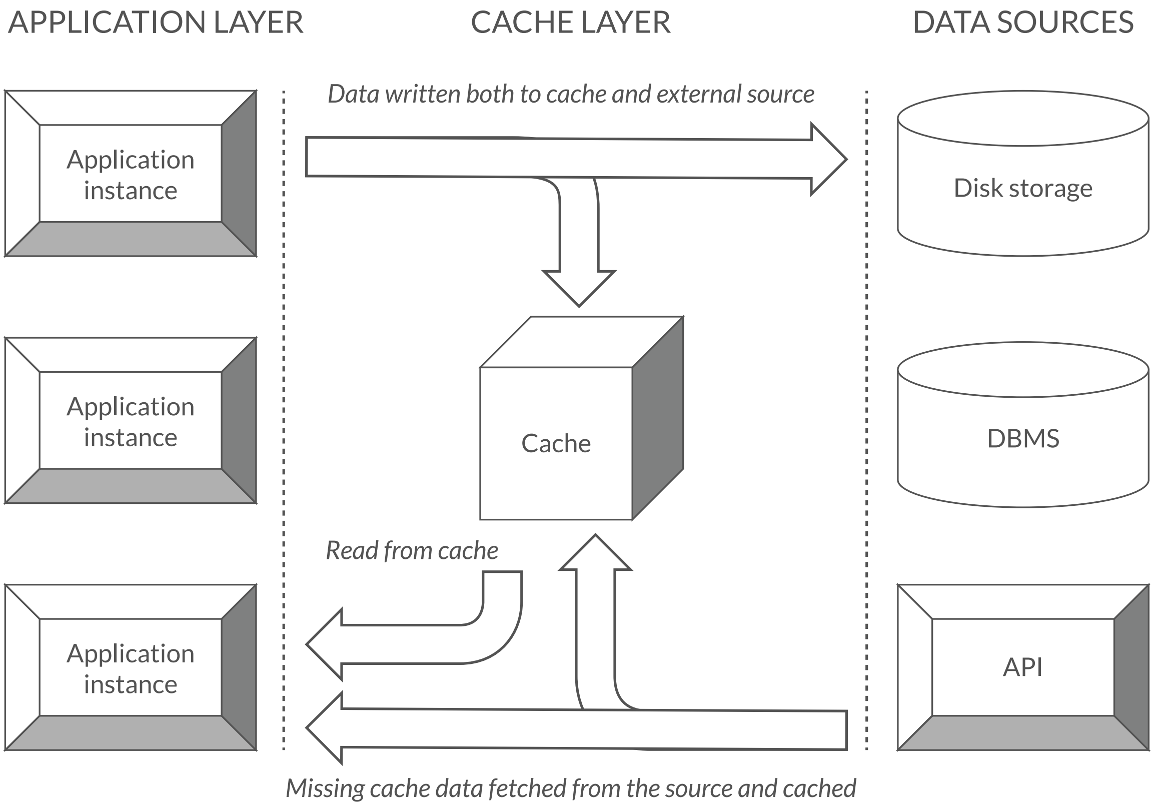
1. Receive client request
2. Process request to prepare the response
3. Return response

Any meaningful processing in stage 2 requires data other than that which is immediately available to the process (e.g. request inputs, initialization information). Furthermore, the process generates additional data, whether intermediate computational results or the final reply. The server's main memory, a.k.a. RAM, is used for storing the data but not all data is or can be stored in RAM. A server's RAM is finite, volatile[1], usually non-sharable, and in most cases comparatively expensive. Hence data is stored in a variety of process-external data sources (e.g. disk or a DBMS) for better scalability, durability, concurrency and cost reductions.

[1] *Excluding emerging durable memory technologies*

What is an application cache

Once an application process uses an external data source, its performance can be bottlenecked by the throughput and latency of said data source. When a slower external data source is used, frequently accessed data is often moved temporarily to a faster storage that is closer to the application to boost the application's performance. This faster intermediate data storage is called the application's cache, a term originating from the French verb "cacher" which means "to hide." The following diagram shows the high-level architecture of an application cache:



Why cache

The cache's main purpose is to reduce the time needed to access data stored outside of the application's main memory space. Additionally, caching is also an extremely powerful tool for scaling up external data sources and mitigating the effects that usage spikes have on them.

An application-side cache effectively reduces all resource demands needed to serve data from external sources, thus freeing these resources for other uses. Without the use of a cache, the application interacts with the data source for every request, whereas when a cache is employed only a single request to the external data source is needed, with subsequent access served from the cache.

Additionally, a cache also contributes to the application's availability. External data sources may experience failures that result in degraded or terminated service. During such outages the cache can still serve data to the application and thus retain its availability.

What is in the cache

An application's cache is intended for storing data that the application's process requires rapidly during execution. Though each application is unique and can potentially store any kind of data in its cache, an application would typically use a cache for:

- **Configuration settings:** Information that the application requires to make bootstrapping and runtime decisions is often stored in relatively slow storage (such as text files on disk or a shared configuration repository). By keeping cached copies of these settings, the application can access the data with minimal latency.
- **Localization and internationalization data:** User-facing applications often provide localized variants of their interface to accommodate international audiences. Internationalization data is usually stored external to the application so it can be managed separately. Because this data is needed to serve most requests, caching it yields improvements in application response times.
- **Templates and partially rendered responses:** Many applications compose their replies by adding data to templates and pre-prepared content (such as HTML fragments and JSON snippets). While the data itself is dynamic, these static properties are required for every request. Caching them mitigates the cost of their frequent use.
- **Reusable results of compute-intensive functions:** Sometimes an application's workflow requires the generation of resource-intensive results. Once these results are obtained, there are cases in which the results could be later reused, such as when performing partial aggregates. The cache acts as an ideal intermediate medium for retaining such results between requests.
- **Session data:** Caching user session data is an integral part of building responsive applications that can scale. Because every user interaction requires access to the session's data, keeping it in the cache ensures the fastest response times to the application user. Caching session data at the application level is superior to alternative methods. Keeping sessions sticky at the load-balancer level, for example, practically forces all requests in a session to be processed by a single app server, while caching allows the requests to be processed by any app server without losing users' states.
- **DBMS data:** Most traditional databases are designed for providing robust functionality rather than speed at scale. The application cache is often used for storing copies of lookup tables and the replies to expensive queries from the DBMS, both to improve the application's performance as well as to reduce the load of the data source.
- **API responses:** Modern applications are built using loosely coupled components that communicate via an API. An application component uses the API to make requests for service from other components, whether inside (e.g. in a microservices architecture) or outside (in a SaaS use case, for instance) the application itself. In cases where an API's reply can be stored in cache, even if only for a relatively short duration, the application's performance is improved by avoiding the inter-process communication.
- **Application objects:** Any object that the application generates and that could be reused at a later time can be stored in the cache. The method for caching objects varies by application, but most caches allow storing both serialized as well as raw object data. A typical example of an often-cached application object is a user profile that is made up of data from multiple sources.

What isn't in the cache

All data in the cache share a distinctive property: ephemerality. Since the cache's contents are either read from an external source or are the results of a computation that can be repeated, that data is considered transient. Data that's missing from the cache, either because it was never there to begin with or because it was removed, can always be fetched/computed and then cached when needed.

Application cache types

Private on-heap caches

A basic type of cache that an application can use is a private cache, i.e. a cache that can only be accessed by the process that owns it. Such caches are trivially set up in the application's memory space by using the programming language's constructs or the use of embedded libraries. While allowing the fastest access, the main disadvantage of on-heap caches is that they consume the same pool of RAM resources as the application process itself. This creates contention between the cache's and the application's memory requirements, one in which the application always wins because the cache's transient nature allows its contents to be removed when RAM is limited. Frequent cache evictions carry two major costs: de-allocating (or garbage collecting) the memory that was claimed by the evicted contents, and re-populating the cache with the data once it is needed again.

Private caches are also a suboptimal means for scaling. They require provisioning each instance of the application with ample resources for caching, thus resulting in more expensive application servers. Also, while buffering the application from the pressure of external data sources, each private cache takes its toll on their resources, which may present a substantial load. Lastly, because the caches are private to each instance, their data may be (and often is) duplicated across the various instances, which is wasteful.

Shared off-heap caches

An alternative approach to on-heap caches is one in which the cached data is stored outside the application process. At the expense of a decrease in speed to access the data, an off-heap cache offers looser coupling between the application and its cache's resource requirements. The increased latency of accessing data in an off-heap cache is practically negligible because the majority of time spent by the application is on processing the request rather than communicating with the cache.

An off-heap cache residing on the same server as the application can help by off-loading cache management from the application process. When multiple instances of the application are running on the server, the same off-heap cache can be shared by all instances, thus reducing the overall resources needed.

Distributed shared cache

The off-heap cache may reside on a server remotely or a cluster of servers that provide caching services to the application. The remote cache, also referred to as a distributed cache, offers several desirable advantages, including minimizing the duplication of cached data and requests to external data sources. The use of a distributed cache also enables true statelessness for the application's processes by using the cache to store state information between requests and even between a request's different processing stages. This allows independent application processes to participate in providing the service, greatly increasing the application's ability to scale.

Redis as a distributed shared cache

Redis is an open source, in-memory Data Structure Store, used as a database, a caching layer or a message broker. Sometimes referred to as the “Leatherman of Databases”, it’s simple yet flexible design philosophy makes it an effective choice for solving a multitude of demanding data processing tasks. Redis data structures resolve very complex programming problems with simple commands executed within the data store, reducing coding effort, increasing throughput, and reducing latency.

Caching is one of Redis’ most popular use cases as Redis perfectly meets the requirements of an application’s distributed caching layer.

1. **In-memory store** 内存
   All data in Redis is stored in RAM, delivering the fastest possible access times to the data for both read and write requests.
2. **Optimized for speed** 速度优化
   Redis is designed and implemented for performance. Being written in ANSI C, it compiles into extremely efficient machine code and requires little overhead. It uses a (mostly) single-threaded event loop model that optimally utilizes the CPU core that it is running on. The data structures used internally by Redis are implemented for maximum performance and the majority of data operations require constant time and space.
3. **Support for arbitrary data** 支持任意数据
   Data stored in Redis can be in any form and size. Redis is binary safe so it can store any data, from human readable text to encoded binaries. A single data element in Redis can range in size from 0 bytes to 0.5GB, allowing it to cache practically any datum.
4. **Key-based access** 基于key-value
   Redis is based on the key-value model in which data is stored and fetched from Redis by key. Key-based access allows for extremely efficient access times and this model maps naturally to caching, with Redis providing the customary GET and SET semantics for interacting with the data.
5. **Multi-key operations** 多key
   Several of Redis’ commands operate on multiple keys. Multi-key operations provide better overall performance compared to performing the operations one after the other, because they require substantially less communication and administration.
6. **Atomicity of operations and transactions** 原子操作
   Every operation in Redis is atomic. This ensures the integrity of the cached data and provides a consistent view of it to the processes sharing it. Enforcing a higher level of consistency in the data model is possible by grouping together several operations into transactions, wherein they are executed sequentially without any other interleaving operations.
7. **Data expiration** 数据到期-超时
   Keys in Redis can be set with a time to live (TTL), after which they are expired. Until they expire, such keys are called “volatile” keys. A volatile key’s TTL is unaffected by changes to the data that the key holds and it is possible to update the TTL independently. Once they’ve outlived their time, expired keys are removed automatically by Redis without any further action required of the application.

Expiration is a key concept in caching that serves two important purposes: first, it is a simple yet extremely effective cache invalidation approach. Secondly, expiration is an indispensable tool for keeping the cache's size under control, hence it is usually recommended to assign an expiration date to all cached data.

8. **Eviction policies** 驱逐策略

   When used as a cache, Redis supports a configurable eviction policy that is triggered when its memory is running low. Because cached data is ephemeral, it is possible to evict the existing data that is in the cache to make room for new data. Eviction in Redis can apply either to all of cache's keys or only to the volatile ones. Furthermore, Redis allows you to choose between two eviction approaches: random and usage-based. The random eviction policy is suitable for cases in which all data is accessed uniformly, whereas the usage-based approach evicts the least recently used (LRU) keys.

9. **Intelligent caching** 智能缓存

   Redis' true value is providing advanced data structures and operations to application developers. "Intelligent caching" is more than just leveraging these data structures with GET and SET operations; it's the act of exploiting their unique properties to efficiently and optimally manipulate data. While a comprehensive overview of Redis' extensive features is outside the scope of this paper, two prime examples of how it enables intelligent caching are its commands that modify the data in the server and its ability to execute embedded Lua scripts.

10. **Distributed network server** 分布式网络服务

    Redis is designed to be a remote, network-attached server. It uses a lightweight TCP protocol that has client implementations in every conceivable programming language.

11. **Request pipelining** 请求管道

    To reduce the penalties incurred by sending many small packets over the network, and in cases where the multiple requests need to be sent to the cache without waiting for responses, the Redis protocol supports a pipelining mode. When pipelining, the application can group together a batch of requests without waiting for the response of each one and send these together in bulk to the server for processing. By doing so, the overhead of network traffic is mitigated substantially and the average time per request in the pipeline is greatly reduced. Request pipelining also improves Redis server performance, as it allows the Redis process to execute multiple commands in a row, eliminating the overhead caused by context switches.

12. **Highly available** 高可用

    A cache is only effective if it is available to the application. The data in Redis and the service that Redis provides can be made highly available via the use of Redis' built-in replication mechanism. A Redis cache can be set up with a replica (i.e. slave), which is continuously updated with modification from the primary (i.e. master). To ensure the cache's availability, watchdog and management process are employed that monitor the cache's processes and manage the failover when needed.

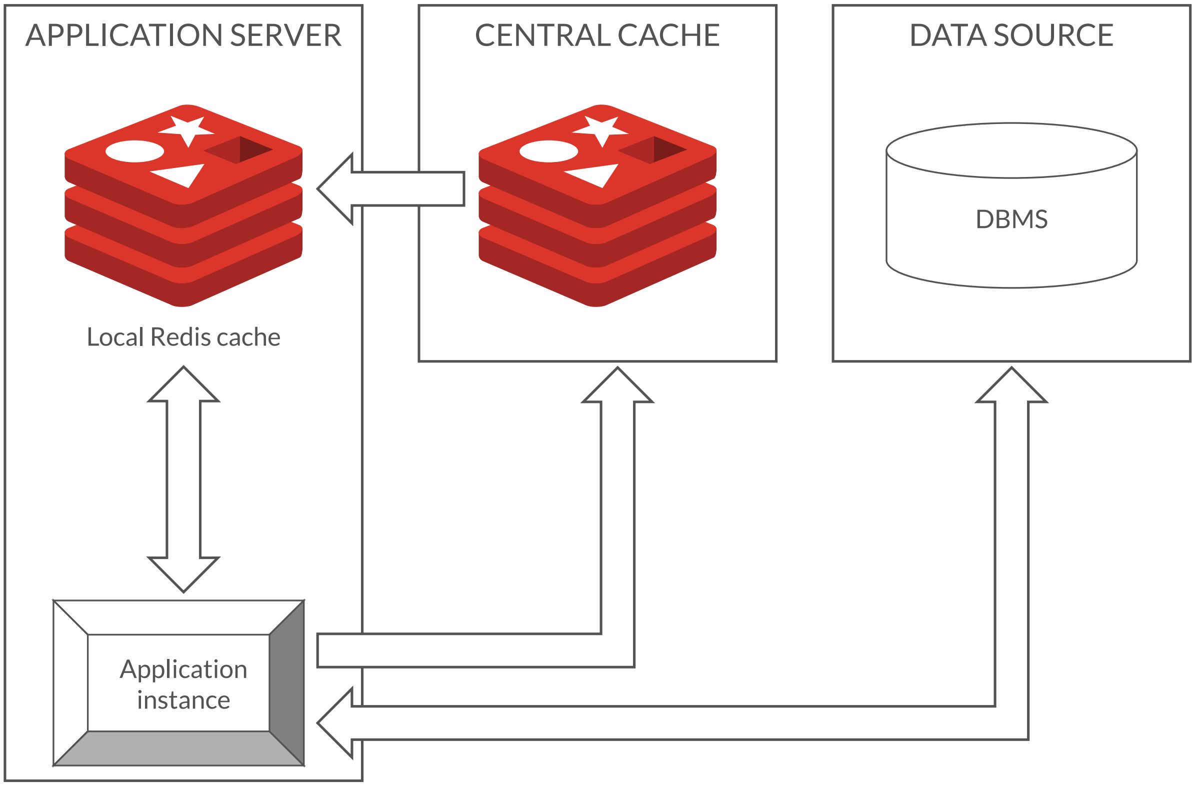
13. **Data persistence** 数据坚持

    Despite cached data's temporary existence, it is sometimes desirable for the cache to persist data to durable storage when cached data is relatively static or for faster recovery times from outages by allowing a swift recovery of the cache. Another benefit of recovering the cache's contents from persistent storage is the prevention of the often overwhelming load on

external data sources that would have otherwise been requested for the data (a phenomena referred to as “the thundering herd” or “a pile of dogs”).
Redis offers optional and tunable data persistence mechanisms that write the contents of the cache to durable disk storage. It is possible to create snapshots of the cache at any moment in time as well as to maintain a journal of operations that allows recovery up to the instant before the outage.

14. **Scalable shared-nothing clustering** 可扩展的共享无群集
Redis can be scaled horizontally to meet any increase in demand for RAM, computation or network resources. A Redis cluster is a set of processes, possibly on multiple nodes, that work together to provide the caching service. The cluster is made up of multiple Redis servers (i.e. shards), with each one of these being responsible for a subset of the cache’s keyspace. This allows scaling out the cluster simply by adding more shards to it and redistributing the data.

15. **Local cache replicas** 本地缓存副本
Redis can be deployed on the same server that runs the application’s processes in order to act as a local private cache. Once co-located with the application, Redis’ replication can be leveraged for maintaining local replicas of a central shared cache as well. This deployment mode improves the administrative aspects of cache management while providing extreme data locality for the applications’ processes as shown in the diagram below.



The Redis Labs Enterprise Cluster (RLEC) enables you to install an enterprise-grade cluster that acts as a container for managing and running multiple Redis databases in a highly available and scalable manner, with predictable and stable top performance. For zero touch as-a-service deployment, Redis Cloud provides similar functionality over the major public clouds (AWS, Azure, GCP and IBM SoftLayer) and across multiple PaaS (Heroku, Azure Store, Cloud Foundry, OpenShift and others) without dealing with clusters and nodes deployments.

Redis Cloud and RLEC provide additional capabilities that extend the abilities of open source Redis to make it even more suitable for caching purposes. These include:

- Memcached compliant protocol for drop-in replacement of legacy cache engines
- Flash storage as slower RAM for managing large datasets in a cost effective manner
- Advanced cross zone/datacenter/cloud replication that provides cache locality for geographically distributed applications
- Application-transparent administration and operations
- Monitoring and alerting for the cache's metrics
- Zero downtime or performance impact while scaling up or down

Summary

The distributed cache is a widely-adopted design pattern for building robust and scalable applications. By serving data from the cache, and instead of fetching it from slower devices or re-computing it, the application's overall performance is improved. The cache also relieves some of the load on external data sources as well as improves the availability of the application during outages.

Redis is the most popular distributed caching engine today. It is production-proven and provides a host of capabilities that make it the ideal distributed caching layer for applications. Redis Cloud and RLEC provide additional capabilities that extend the abilities of open source Redis to make it even more suitable for caching purposes.

To learn how to use Redis with your application contact expert@redislabs.com.