Advanced Caching Architectures and Patterns

# Volatile Caches, Cache Persistence and Rehydration

Cache persistence is the capability of storing a cache’s contents in persistent storage, so that a power failure or other outage does not lose the contents of the cache.

A **volatile cache** is a cache that exists in memory and is subject to erasure when a power outage or system restart occurs. In case of failure or other issue, the contents of a volatile cache cannot be assumed to be available—**the system must always be functional, even if the cache itself is removed**. **The performance of the system may be impacted, but the capabilities and functionality cannot be affected.**

With **cache persistence**, **contents persist even during power outages and system reboots**. An application might rely on an object being stored in the cache forever. A persistent cache may be chosen for performance reasons, as long as it is acceptable for an application to fail and not perform if a value is removed inappropriately (Did not understand the last 2 sentences)

Typically, a volatile cache is implemented in RAM, or some other high-performance**, non-permanent** **memory store**. A persistent cache typically **relies** on a **hard disk**, **SSD**, or **other long-term persistent storage**.

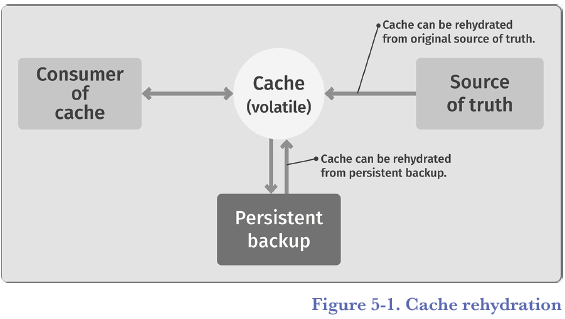
(Do you mean there could be scenario when a cache is entirely on the disk?)

**It is possible to implement a persistent cache in volatile memory**, as long as the cache mechanism makes **appropriate redundant backups of the cached contents** into persistent memory. In this situation, if the cache fails (such as via a power outage or system reboot) such that the volatile memory is wiped clean, **then the**

**redundant cache copy in the persistent memory is used to re-create the contents of the cache in the volatile memory, before the cache comes back online. This process is called cache rehydration.**

A volatile cache can be rehydrated from either **a persistent backup storage medium**, or **from the backing store that provides the reference copy of the required data:**

(I think by the backing store he means the original on-disk database but I guess we will figure it out.)



* **Redis can operate** **as** either a **volatile** **or persistent** cache. It uses **RAM for its** **primary memory storage**, **making it act like a volatile cache**, **yet permanent storage can be used** **to provide the** **persistent backup and rehydration**, **so that Redis can be used as a persistent cache.**

## Redis Persistence

Redis offers a range of persistent options, specifically:

**1.** Append-only files (AOF)

**2.** Point-in-time backups (RDB)

**3.** A combination of both

Together, these provide:

* a variety of options for making data **persistent as needed**
* while maintaining the performance **advantages of being RAM-based.**

### Append-only Files (AOF)

Redis uses a file called the append-only file (AOF), in order to create **a persistent backup of the primary volatile cache** in persistent storage.

**The AOF file stores a real-time log of updates to the cache**. This file is updated continuously, so it represents an accurate, persistent view of the state of the cache when the cache is shut down, **depending on configuration and failure scenarios.(The configuration and failure scenarios needs to be investigated)**

When the cache is restarted and cleared, **the commands recorded in the AOF log file can be replayed** to re-create the state of the Redis cache at the time of the shutdown.

The result? The cache, while implemented primarily in volatile memory, can be used as a reliable, persistent data cache.

* The option APPENDONLY yes enables the AOF log file.
* **All changes to the cache will result in an entry being written to this log** **file**
* **but the log file itself isn’t necessarily stored in persistent storage immediately**.
* **For performance reasons, you can delay the write to persistent storage for a period of time to** improve overall system performance. This is controlled via APPENDFSYNC.
* The following options are available:

• **APPENDFSYNC no**: This allows **the operating system to cache the log file** and wait to persist it to permanent storage when it deems necessary.

• **APPENDFSYNC everysec**: This forces a write of the AOF log file to persistent storage once every second.

• **APPENDFSYNC always**: This forces the AOF file to be written to persistent storage immediately after every log entry is created.

To be completely safe and to guarantee that your cache operates as a persistent cache correctly, you should use the APPENDFSYNC always command, as this is the only way to guarantee that a system crash will not

cause data loss.

However, **if your business can cope with some amount of cache loss during a system crash**, then the everysec and no options can be used to improve performance.

The result of the APPENDONLY command is a continuously growing log file. Redis can, in the background, rebuild the log file by removing no longer necessary entries in the log file. For example, if you:

**1.** Added an entry to the cache

**2.** Changed the entry’s value

**3.** Changed the entry’s value again

**4.** Deleted the entry

Then there would be four entries in the log file. Ultimately, all four of these entries are no longer needed to rehydrate the cache, since the entry is now deleted. **The following command will clean up the log file:**

BGREWRITEAOF

The result is a log file with the shortest sequence of commands needed to rehydrate the current state of the cache in memory. **You can force this command to be executed automatically, which is the recommended best practice, rather than manually**.

### Point-in-Time Backups (RDB -> Redis Database)

Sometimes it is useful to create **a backup copy of the current contents in the cache**. This is what the RDB backup is for.

This command creates a **highly efficient**, **smallest possible**, **point-in-time backup file** of the current contents of

the cache. **It can be executed at any point in time as follows:**

SAVE

This command **creates** a **dump.rdb** file that contains a complete current snapshot of the Redis cache. Alternatively, you can issue the following command:

BGSAVE

**This command returns immediately and creates a background job that creates the snapshot.**

### Comparing RBD to AOF persistence

If your goal is to create a reliable, persistent cache that can survive process crashes, system crashes, and other system failures, then the only reliable way to do that is to use **AOF persistence with APPENDFSYNC set to always.**

**No other method guarantees that the entire state of the cache will be properly stored in persistent storage at all times.**

If your goal is to maintain a series of point-in-time backups **for historical and system-recovery purposes** (such as saving one backup per day for an entire month), then the RDB backup is the proper method to create these backups.

This is because the RDB is a single file providing an accurate snapshot of the database at a given point in time. This snapshot is guaranteed to be consistent.

**However, RDB cannot be used to survive system failures, because any changes made to the system between RDB snapshots will be lost during a system failure.**

So, depending on your requirements, both RDB and AOF can be used to solve your persistent needs.

Used together, they can provide both a **system-tolerant persistent cache**, **along with historical point-in-time snapshot backups.**

**I add:(You need to read more about these options and where each one makes more sense, take this whole Redis discussion as an introduction to available options when it comes to caching)**

*Based on a quick skim over the official Redis document, For big datasets, the RDB option provides a faster restart and apparently, the dump.rdb file is much more light-weight compared to the AO file.*

*It also says that using only the AOF is not a good idea because with the RDB option you will have a back up that I think can be used for remote servers in a cloud environment.*

# Mixed RAM/SSD Caching (available in Redis Enterprise)

Open source Redis requires the entire cache, both the keys and the value of the keys, to be stored only in RAM. However, in Redis Enterprise, you can configure Redis to store **the value** of keys in either RAM or SSD flash memory.

This allows significantly **larger** cache implementations. While caching is not its main use case, this feature, called Redis on Flash (RoF), is part of Redis Enterprise and can be useful in caching environments.

(Do some research on what exactly the intended use case was and when it makes sense to use it for caching purposes. Also did I get it right? It enables you to not evict key-values completely. Instead, you just move the values and hence, a bigger number of cached key-values)

In RoF, **all the data keys are still stored in RAM**, but the value of those keys is intelligently stored in **a mixture of RAM and SSD flash storage**. The value is stored based on a least-recently used (LRU) eviction policy. **More actively used values are stored in RAM and lesser used values are stored in SSD.**

Given that SSD storage is **significantly larger and less expensive** than (**if not quite as fast as**) RAM, using RoF can allow you to build **significantly larger** **caches** more **cost effectively**.

Note that the use of persistent SSD flash memory **does not automatically convert your cache into a persistent cache.** This is because the keys are still stored in RAM, regardless of where your data values are stored, RAM or SSD. Therefore, using RoF with SSD storage does not remove the requirement of creating AOF

and/or RDB backup files to create a true persistent cache.

# In-Cache Function Execution

Redis allows you to execute arbitrary functions **within the cache database itself**. This is useful for a number of in-database application execution processes.

Essentially, you can execute full-fledged Python scripts (**with more languages coming**) **within the execution environment of a Redis instance.**

As a simple example, imagine you have in a Redis database a few Hash-maps that represent user-related information, such as first name, last name, and age.

Then you can use the RG.PYEXECUTE command to execute a Python script to perform data cleanup on this information. Here is a sample script that deletes all users who are younger than 35 years old:

> RG.PYEXECUTE "GearsBuilder()

.filter(lambda x: int(x['value']['age']) > 35)

.foreach(lambda x: execute('del', x['key']))

.run('user:\*')"

(Learn more about these transaction, batch, and event-driven processing use cases)

The **RedisGears** module—a **serverless engine** for transaction, batch, and event-driven data processing—creates a powerful execution environment that **allows you to build complex caching mechanisms**.

For instance:

* you could implement inline or aside caches talking to other backend databases from within RedisGears.
* It can be used to implement the write-through and write-behind caching patterns.

# Microservices Architecture