RedisConf 2018 is near! Join us in San Francisco, April 24 - 26 at Pier 27. NEW: use the code RCredisio for 75% discount.

This page provides a technical description of **Redis persistence**, it is a suggested read for all the **Redis** users. For a wider overview of **Redis persistence** and the durability guarantees it provides you may want to also read **Redis persistence** demystified.

Redis Persistence

Redis provides a different range of **persistence** options:

- The RDB **persistence** performs point-in-time snapshots of your dataset at specified intervals.
- the AOF **persistence** logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset. Commands are logged using the same format as the **Redis** protocol itself, in an append-only fashion. **Redis** is able to rewrite the log on background when it gets too big.
- If you wish, you can disable **persistence** at all, if you want your data to just exist as long as the server is running.
- It is possible to combine both AOF and RDB in the same instance. Notice that, in this case, when **Redis** restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

The most important thing to understand is the different trade-offs between the RDB and AOF **persistence**. Let's start with RDB:

RDB advantages

- RDB is a very compact single-file point-in-time representation of your **Redis** data. RDB files are perfect for backups. For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days. This allows you to easily restore different versions of the data set in case of disasters.

- RDB is very good for disaster recovery, being a single compact file can be transferred to far data centers, or on Amazon S3 (possibly encrypted).
- RDB maximizes **Redis** performances since the only work the **Redis** parent process needs to do in order to persist is forking a child that will do all the rest. The parent instance will never perform disk I/O or alike.
- RDB allows faster restarts with big datasets compared to AOF.

RDB disadvantages

- RDB is NOT good if you need to minimize the chance of data loss in case **Redis** stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of **Redis** stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in **Redis** to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

AOF advantages

- Using AOF **Redis** is much more durable: you can have different fsync policies: no fsync at all, fsync every second, fsync at every query. With the default policy of fsync every second write performances are still great (fsync is performed using a background thread and the main thread will try hard to perform writes when no fsync is in progress.) but you can only lose one second worth of writes.
- The AOF log is an append only log, so there are no seeks, nor corruption problems if there is a power outage. Even if the log ends with an half-written command for some reason (disk full or other reasons) the **redis**-check-aof tool is able to fix it easily.
- **Redis** is able to automatically rewrite the AOF in background when it gets too big. The rewrite is completely safe as while **Redis** continues appending to the old file, a completely new one is produced with the minimal set of operations needed to create the current data set, and once this second file is ready **Redis** switches the two and starts appending to the new one.
- AOF contains a log of all the operations one after the other in an easy to understand and parse format. You can even easily export an AOF file. For instance even if you flushed everything for an error using a FLUSHALL command, if no rewrite of the log was performed in the meantime

you can still save your data set just stopping the server, removing the latest command, and restarting **Redis** again.

AOF disadvantages

- AOF files are usually bigger than the equivalent RDB files for the same dataset.
- AOF can be slower than RDB depending on the exact fsync policy. In general with fsync set to *every second* performances are still very high, and with fsync disabled it should be exactly as fast as RDB even under high load. Still RDB is able to provide more guarantees about the maximum latency even in the case of an huge write load.
- In the past we experienced rare bugs in specific commands (for instance there was one involving blocking commands like BRPOPLPUSH) causing the AOF produced to not reproduce exactly the same dataset on reloading. This bugs are rare and we have tests in the test suite creating random complex datasets automatically and reloading them to check everything is ok, but this kind of bugs are almost impossible with RDB **persistence**. To make this point more clear: the **Redis** AOF works incrementally updating an existing state, like MySQL or MongoDB does, while the RDB snapshotting creates everything from scratch again and again, that is conceptually more robust. However - 1) It should be noted that every time the AOF is rewritten by **Redis** it is recreated from scratch starting from the actual data contained in the data set, making resistance to bugs stronger compared to an always appending AOF file (or one rewritten reading the old AOF instead of reading the data in memory). 2) We never had a single report from users about an AOF corruption that was detected in the real world.

Ok, so what should I use?

The general indication is that you should use both **persistence** methods if you want a degree of data safety comparable to what PostgreSQL can provide you.

If you care a lot about your data, but still can live with a few minutes of data loss in case of disasters, you can simply use RDB alone.

There are many users using AOF alone, but we discourage it since to have an RDB snapshot from time to time is a great idea for doing database backups, for faster restarts, and in the event of bugs in the AOF engine.

Note: for all these reasons we'll likely end up unifying AOF and RDB into a single **persistence** model in the future (long term plan).

The following sections will illustrate a few more details about the two **persistence** models.

Snapshotting

By default **Redis** saves snapshots of the dataset on disk, in a binary file called `dump.rdb`. You can configure **Redis** to have it save the dataset every N seconds if there are at least M changes in the dataset, or you can manually call the SAVE or BGSAVE commands.

For example, this configuration will make **Redis** automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:

```
save 60 1000
```

This strategy is known as *snapshotting*.

How it works

Whenever **Redis** needs to dump the dataset to disk, this is what happens:

- **Redis** forks. We now have a child and a parent process.
- The child starts to write the dataset to a temporary RDB file.
- When the child is done writing the new RDB file, it replaces the old one.

This method allows **Redis** to benefit from copy-on-write semantics.

Append-only file

Snapshotting is not very durable. If your computer running **Redis** stops, your power line fails, or you accidentally `kill -9` your instance, the latest data written on **Redis** will get lost. While this may not be a big deal for some applications, there are use cases for full durability, and in these cases **Redis** was not a viable option.

The *append-only file* is an alternative, fully-durable strategy for **Redis**. It became available in version 1.1.

You can turn on the AOF in your configuration file:

```
appendonly yes
```

From now on, every time **Redis** receives a command that changes the dataset (e.g. SET) it will append it to the AOF. When you restart **Redis** it will re-play the AOF to rebuild the state.

Log rewriting

As you can guess, the AOF gets bigger and bigger as write operations are performed. For example, if you are incrementing a counter 100 times, you'll end up with a single key in your dataset containing the final value, but 100 entries in your AOF. 99 of those entries are not needed to rebuild the current state.

So **Redis** supports an interesting feature: it is able to rebuild the AOF in the background without interrupting service to clients. Whenever you issue a BGREWRITEAOF **Redis** will write the shortest sequence of commands needed to rebuild the current dataset in memory. If you're using the AOF with **Redis** 2.2 you'll need to run BGREWRITEAOF from time to time. **Redis** 2.4 is able to trigger log rewriting automatically (see the 2.4 example configuration file for more information).

How durable is the append only file?

You can configure how many times **Redis** will `fsync` data on disk. There are three options:

- `fsync` every time a new command is appended to the AOF. Very very slow, very safe.
- `fsync` every second. Fast enough (in 2.4 likely to be as fast as snapshotting), and you can lose 1 second of data if there is a disaster.
- Never `fsync`, just put your data in the hands of the Operating System. The faster and less safe method.

The suggested (and default) policy is to `fsync` every second. It is both very fast and pretty safe. The `always` policy is very slow in practice (although it was improved in **Redis** 2.0) – there is no way to make `fsync` faster than it is.

What should I do if my AOF gets corrupted?

It is possible that the server crashes while writing the AOF file (this still should never lead to inconsistencies), corrupting the file in a way that is no longer loadable by **Redis**. When this happens you can fix this problem using the following procedure:

- Make a backup copy of your AOF file.
- Fix the original file using the `redis-check-aof` tool that ships with **Redis**:

  $ **redis**-check-aof --fix
- Optionally use `diff -u` to check what is the difference between two files.
- Restart the server with the fixed file.

How it works

Log rewriting uses the same copy-on-write trick already in use for snapshotting. This is how it works:

- **Redis** forks, so now we have a child and a parent process.
- The child starts writing the new AOF in a temporary file.
- The parent accumulates all the new changes in an in-memory buffer (but at the same time it writes the new changes in the old append-only file, so if the rewriting fails, we are safe).
- When the child is done rewriting the file, the parent gets a signal, and appends the in-memory buffer at the end of the file generated by the child.

- Profit! Now **Redis** atomically renames the old file into the new one, and starts appending new data into the new file.

How I can switch to AOF, if I'm currently using dump.rdb snapshots?

There is a different procedure to do this in **Redis** 2.0 and **Redis** 2.2, as you can guess it's simpler in **Redis** 2.2 and does not require a restart at all.

Redis >= 2.2

- Make a backup of your latest dump.rdb file.
- Transfer this backup into a safe place.
- Issue the following two commands:
- **redis**-cli config set appendonly yes
- **redis**-cli config set save ""
- Make sure that your database contains the same number of keys it contained.
- Make sure that writes are appended to the append only file correctly.

The first CONFIG command enables the Append Only File. In order to do so **Redis will block** to generate the initial dump, then will open the file for writing, and will start appending all the next write queries.

The second CONFIG command is used to turn off snapshotting **persistence**. This is optional, if you wish you can take both the **persistence** methods enabled.

IMPORTANT: remember to edit your **redis**.conf to turn on the AOF, otherwise when you restart the server the configuration changes will be lost and the server will start again with the old configuration.

Redis 2.0

- Make a backup of your latest dump.rdb file.
- Transfer this backup into a safe place.
- Stop all the writes against the database!
- Issue a **redis**-cli bgrewriteaof. This will create the append only file.
- Stop the server when **Redis** finished generating the AOF dump.
- Edit **redis**.conf end enable append only file **persistence**.
- Restart the server.
- Make sure that your database contains the same number of keys it contained.
- Make sure that writes are appended to the append only file correctly.

Interactions between AOF and RDB **persistence**

Redis >= 2.4 makes sure to avoid triggering an AOF rewrite when an RDB snapshotting operation is already in progress, or allowing a BGSAVE while the AOF rewrite is in progress. This prevents two **Redis** background processes from doing heavy disk I/O at the same time.

When snapshotting is in progress and the user explicitly requests a log rewrite operation using BGREWRITEAOF the server will reply with an OK status code telling the user the operation is scheduled, and the rewrite will start once the snapshotting is completed.

In the case both AOF and RDB **persistence** are enabled and **Redis** restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

Backing up **Redis** data

Before starting this section, make sure to read the following sentence: **Make Sure to Backup Your Database**. Disks break, instances in the cloud disappear, and so forth: no backups means huge risk of data disappearing into /dev/null.

Redis is very data backup friendly since you can copy RDB files while the database is running: the RDB is never modified once produced, and while it gets produced it uses a temporary name and is renamed into its final destination atomically using rename(2) only when the new snapshot is complete.

This means that copying the RDB file is completely safe while the server is running. This is what we suggest:

- Create a cron job in your server creating hourly snapshots of the RDB file in one directory, and daily snapshots in a different directory.
- Every time the cron script runs, make sure to call the `find` command to make sure too old snapshots are deleted: for instance you can take hourly snapshots for the latest 48 hours, and daily snapshots for one or two months. Make sure to name the snapshots with data and time information.
- At least one time every day make sure to transfer an RDB snapshot *outside your data center* or at least *outside the physical machine* running your **Redis** instance.

Disaster recovery

Disaster recovery in the context of **Redis** is basically the same story as backups, plus the ability to transfer those backups in many different external data centers. This way data is secured even in the case of some catastrophic event affecting the main data center where **Redis** is running and producing its snapshots.

Since many **Redis** users are in the startup scene and thus don't have plenty of money to spend we'll review the most interesting disaster recovery techniques that don't have too high costs.

- Amazon S3 and other similar services are a good way for mounting your disaster recovery system. Simply transfer your daily or hourly RDB snapshot to S3 in an encrypted form. You can encrypt your data using `gpg -c` (in symmetric encryption mode). Make sure to store your password in many different safe places (for instance give a copy to the most important people

of your organization). It is recommended to use multiple storage services for improved data safety.
- Transfer your snapshots using SCP (part of SSH) to far servers. This is a fairly simple and safe route: get a small VPS in a place that is very far from you, install ssh there, and generate an ssh client key without passphrase, then add it in the authorized_keys file of your small VPS. You are ready to transfer backups in an automated fashion. Get at least two VPS in two different providers for best results.

It is important to understand that this system can easily fail if not coded in the right way. At least make absolutely sure that after the transfer is completed you are able to verify the file size (that should match the one of the file you copied) and possibly the SHA1 digest if you are using a VPS.

You also need some kind of independent alert system if the transfer of fresh backups is not working for some reason.