Redis persistence demystified

Monday, 26 March 12

Part of my work on Redis is reading blog posts, forum messages, and the twitter time line for the "Redis" search. It is very important for a developer to have a feeling about what the community of users, and the community of *non* users, think about the product he is developing. And my feeling is that there is no Redis feature that is as misunderstood as its persistence.

In this blog post I'll do an effort to be truly impartial: no advertising of Redis, no attempt to skip the details that may put Redis in a *bad light*. All I want is simply to provide a clear, understandable picture of how Redis persistence works, how much reliable is, and how it compares to other database systems.

The OS and the disk

The first thing to consider is what we can expect from a database in terms of durability. In order to do so we can visualize what happens during a simple write operation:

- 1: The client sends a write command to the database (data is in client's memory).
- 2: The database receives the write (data is in server's memory).
- 3: The database calls the system call that writes the data on disk (data is in the kernel's buffer).
- 4: The operating system transfers the write buffer to the disk controller (data is in the disk cache).
- 5: The disk controller actually writes the data into a physical media (a magnetic disk, a Nand chip, ...).

> Note: the above **is an oversimplification** in many ways, because there are more levels of caching and buffers than that.
>
> Step 2 is often implemented as a complex caching system inside the database implementation, and sometimes writes are handled by a different thread or process. However soon or later, the database will have to write data to disk, and this is what

matters from our point of view. That is, data from memory has to be transmitted to the kernel (step 3) at some point.

Another big omission of details is about step 3. The reality is more complex since most advanced kernels implement different layers of caching, that usually are the file system level caching (called the *page cache* in Linux) and a smaller *buffer cache* that is a buffer containing the data that waits to be committed to the disk. Using special APIs is possible to bypass both (see for instance O_DIRECT and O_SYNC flags of the open system call on Linux). However from our point of view we can consider this as an unique layer of opaque caching (that is, we don't know the details). It is enough to say that often the page cache is disabled when the database already implements its caching to avoid that both the database and the kernel will try to do the same work at the same time (with bad results). The buffer cache is usually never turned off because this means that every write to the file will result into data committed to the disk that is too slow for almost all the applications.

What databases usually do instead is to call system calls that will commit the buffer cache to the disk, only when absolutely needed, as we'll see later in a more detailed way.

When is our write safe along the line?

If we consider a failure that involves just the database software (the process gets killed by the system administrator or crashes) and does not touch the kernel, the write can be considered safe just after the step 3 is completed with success, that is after the write(2) system call (or any other system call used to transfer data to the kernel) returns successfully. After this step even if the database process crashes, still the kernel will take care of transferring data to the disk controller.

If we consider instead a more catastrophic event like a power outage, we are safe only at step 5 completion, that is, when data is actually transfered to the physical device memorizing it.

We can summarize that the important stages in data safety are the 3, 4, and 5. That is:

- How often the database software will transfer its user-space buffers to the kernel buffers using the write (or equivalent) system call?
- How often the kernel will flush the buffers to the disk controller?
- And how often the disk controller will write data to the physical media?

> Note: when we talk about *disk controller* we actually mean the caching performed by the controller *or* the disk itself. In environments where durability is important system administrators usually disable this layer of caching.
>
> Disk controllers by default only perform a write through caching for most systems (i.e. only reads are cached, not writes). It is safe to enable the write back mode (caching of writes) only when you have batteries or a super-capacitor device protecting the data in case of power shutdown.

POSIX API

From the point of view of the database developer the path that the data follows before being actually written to the physical device is interesting, but even more interesting is *the amount of control* the programming API provides along the path.

Let's start from step 3. We can use the *write* system call to transfer data to the kernel buffers, so from this point of view we have a good control using the POSIX API. However we don't have much control about how much time this system call will take before returning successfully. The kernel write buffer is limited in size, if the disk is not able to cope with the application write bandwidth, the kernel write buffer will reach it's maximum size and the kernel will block our write. When the disk will be able to receive more data, the write system call will finally return. After all the goal is to, eventually, reach the physical media.

Step 4: in this step the kernel transfers data to the disk controller. By default it will try to avoid doing it too often, because it is

faster to transfer it in bigger pieces. For instance Linux by default will actually commit writes after **30 seconds**. This means that if there is a failure, all the data written in the latest 30 seconds can get potentially lost.

The POSIX API provides a family of system calls to force the kernel to write buffers to the disk: the most famous of the family is probably the *fsync* system call (see also *msync* and *fdatasync* for more information). Using fsync the database system has a way to force the kernel to actually commit data on disk, but as you can guess this is a very expensive operation: **fsync will initiate a write operation** every time it is called and there is some data pending on the kernel buffer. Fsync() also blocks the process for all the time needed to complete the write, and if this is not enough, on Linux it will also block all the other threads that are writing against the same file.

What we can't control

So far we learned that we can control step 3 and 4, but what about 5? Well formally speaking we don't have control from this point of view using the POSIX API. Maybe some kernel implementation will try to tell the drive to actually commit data on the physical media, or maybe the controller will instead re-order writes for the sake of speed, and will not really write data on disk ASAP, but will wait a couple of milliseconds more. This is simply out of our control.

In the rest of this article we'll thus simplify our scenario to two data safety levels:

- Data written to kernel buffers using the write(2) system call (or equivalent) that gives us **data safety against process failure**.
- Data committed to the disk using the fsync(2) system call (or equivalent) that gives us, virtually, **data safety against complete system failure** like a power outage. We actually know that there is no guarantee because of the possible disk controller caching, but we'll not consider this aspect because this is an invariant among all the common database systems. Also system administrators often can use specialized tools in order to control the exact behavior of the physical device.

> Note: not all the databases use the POSIX API. Some proprietary database use a kernel module that will allow a more direct

interaction with the hardware. However the main shape of the problem remains the same. You can use user-space buffers, kernel buffers, but soon or later there is to write data on disk to make it safe (and this is a slow operation). A notable example of a database using a kernel module is Oracle.

Data corruption

In the previous paragraphs we analyzed the problem of ensuring data is actually transfered to the disk by the higher level layers: the application and the kernel. However this is not the only concern about durability. Another one is the following: is the database still readable after a catastrophic event, or its internal structure can get corrupted in some way so that it may no longer be read correctly, or requires a recovery step in order to reconstruct a valid representation of data?

For instance many SQL and NoSQL databases implement some form of on-disk tree data structure that is used to store data and indexes. This data structure is manipulated on writes. If the system stops working in the middle of a write operation, is the tree representation still valid?

In general there are three levels of safety against data corruption:

- Databases that write to the disk representation not caring about what happens in the event of failure, asking the user to use a replica for data recovery, and/or providing tools that will try to reconstruct a valid representation *if possible*.
- Database systems that use a log of operations (a journal) so that they'll be able to recover to a consistent state after a failure.
- Database systems that never modify already written data, but only work in *append only* mode, so that **no corruption is possible**.

Now we have all the elements we need to evaluate a database system in terms of reliability of its persistence layer. It's time to check how Redis scores in this regard. Redis provides two different persistence options, so we'll examine both one after the other.

Snapshotting

Redis snapshotting is the simplest Redis persistence mode. It produces point-in-time snapshots of the dataset when specific conditions are met, for instance if the previous snapshot was created more than 2 minutes ago and there are already at least 100 new writes, a new snapshot is created. This conditions can be controlled by the user configuring the Redis instance, and can also be modified at runtime without restarting the server. Snapshots are produced as a single *.rdb* file that contains the whole dataset.

The durability of snapshotting is limited to what the user specified as *save points*. If the dataset is saved every 15 minutes, than in the event of a Redis instance crash or a more catastrophic event, up to 15 minutes of writes can be lost. From the point of view of Redis transactions snapshotting guarantees that either a MULTI/EXEC transaction is fully written into a snapshot, or it is not present at all (as already said RDB files represent exactly *point in time* images of the dataset).

The RDB file can not get corrupted, because it is produced by a child process in an append-only way, starting from the image of data in the Redis memory. A new rdb snapshot is created as a temporary file, and gets renamed into the destination file using the atomic rename(2) system call once it was successfully generated by a child process (and only after it gets synched on disk using the fsync system call).

Redis snapshotting does NOT provide good durability guarantees if up to a few minutes of data loss is not acceptable in case of incidents, so it's usage is limited to applications and environments where losing recent data is not very important.

However even when using the more advanced persistence mode that Redis provides, called "AOF", it is still advisable to also turn snapshotting on, because to have a single compact RDB file containing the whole dataset is extremely useful to perform backups, to send data to remote data centers for disaster recovery, or to easily roll-back to an old version of the dataset in the event of a dramatic software bug that compromised the content of the database in a serious way.

It's worth to note that RDB snapshots are also used by Redis when performing a master -> slave synchronization.

> One of the additional benefits of RDB is the fact for a given database size, the number of I/Os on the system is bound,

whatever the activity on the database is. This is a property that most traditional database systems (and the Redis other persistence, the AOF) do not have.

Append only file

The Append Only File, usually called simply AOF, is the main Redis persistence option. The way it works is extremely simple: every time a write operation that modifies the dataset in memory is performed, the operation gets logged. The log is produced exactly in the same format used by clients to communicate with Redis, so the AOF can be even piped via netcat to another instance, or easily parsed if needed. At restart Redis re-plays all the operations to reconstruct the dataset.

To show how the AOF works in practice I'll do a simple experiment, setting up a new Redis 2.6 instance with append only file enabled:

```
./redis-server --appendonly yes
```

Now it's time to send a few write commands to the instance:

```
redis 127.0.0.1:6379> set key1 Hello
OK
redis 127.0.0.1:6379> append key1 " World!"
(integer) 12
redis 127.0.0.1:6379> del key1
(integer) 1
redis 127.0.0.1:6379> del non_existing_key
(integer) 0
```

The first three operations actually modified the dataset, the fourth did not, because there was no key with the specified name. This is how our append only file looks like:

```
$ cat appendonly.aof
*2
$6
SELECT
$1
0
*3
$3
set
$4
key1
$5
Hello
*3
$6
```

```
append
$4
key1
$7
 World!
*2
$3
del
$4
key1
```

As you can see the final DEL is missing, because it did not produced any modification to the dataset.

It is as simple as that, new commands received will get logged into the AOF, but only if they have some effect on actual data. However not all the commands are logged as they are received. For instance blocking operations on lists are logged for their final effects as normal non blocking commands. Similarly INCRBYFLOAT is logged as SET, using the final value after the increment as payload, so that differences in the way floating points are handled by different architectures will not lead to different results after reloading an AOF file.

So far we know that the Redis AOF is an append only business, so no corruption is possible. However this desirable feature can also be a problem: in the above example after the DEL operation our instance is completely empty, still the AOF is a few bytes worth of data. The AOF is an *always growing file*, so how to deal with it when it gets too big?

AOF rewrite

When an AOF is too big Redis will simply rewrite it from scratch in a temporary file. The rewrite is NOT performed by reading the old one, but directly accessing data in memory, so that Redis can create the shortest AOF that is possible to generate, and will not require read disk access while writing the new one.

Once the rewrite is terminated, the temporary file is synched on disk with fsync and is used to overwrite the old AOF file.

You may wonder what happens to data that is written to the server while the rewrite is in progress. This new data is simply also written to the old (current) AOF file, and at the same time queued into an in-memory buffer, so that when the new AOF is ready we can write this missing part inside it, and finally replace the old AOF file with the

new one.

As you can see still everything is append only, and when we rewrite the AOF we still write everything inside the old AOF file, for all the time needed for the new to be created. This means that for our analysis we can simply avoid considering the fact that the AOF in Redis gets rewritten at all. So the real question is, how often we write(2), and how often we fsync(2).

> AOF rewrites are generated only using sequential I/O operations, so the whole dump process is efficient even with rotational disks (no random I/O is performed). This is also true for RDB snapshots generation. The complete lack of Random I/O accesses is a rare feature among databases, and is possible mostly because Redis serves read operations from memory, so data on disk does not need to be organized for a random access pattern, but just for a sequential loading on restart.

AOF durability

This whole article was written to reach this paragraph. I'm glad I'm here, and I'm even more glad you are *still* here with me.

The Redis AOF uses an user-space buffer that is populated with new data as new commands are executed. The buffer is usually flushed on disk *every time we return back into the event loop*, using a single write(2) call against the AOF file descriptor, but actually there are three different configurations that will change the exact behavior of write(2), and especially, of fsync(2) calls.

This three configurations are controlled by the **appendfsync** configuration directive, that can have three different values: no, everysec, always. This configuration can also be queried or modified at runtime using the CONFIG SET command, so you can alter it every time you want without stopping the Redis instance.

appendfsync no

In this configuration Redis does not perform fsync(2) calls at all. However it will make sure that clients **not using** pipelining, that is, clients that wait to receive the reply of a command before sending the next one, will receive an acknowledge that the command was executed

correctly **only after the change is transfered to the kernel by writing the command to the AOF file descriptor, using the write(2) system call**.

Because in this configuration fsync(2) is not called at all, data will be committed to disk at kernel's wish, that is, every 30 seconds in most Linux systems.

appendfsync everysec

In this configuration data will be both written to the file using write(2) and flushed from the kernel to the disk using fsync(2) **one time every second**. Usually the write(2) call will actually be performed every time we return to the event loop, but this is not guaranteed.

However if the disk can't cope with the write speed, and the background fsync(2) call is taking longer than 1 second, Redis may delay the write up to an additional second (in order to avoid that the write will block the main thread because of an fsync(2) running in the background thread against the same file descriptor). If a total of **two seconds** elapsed without that fsync(2) was able to terminate, Redis finally performs a (likely blocking) write(2) to transfer data to the disk at any cost.

So in this mode Redis guarantees that, in the worst case, within 2 seconds everything you write is going to be committed to the operating system buffers *and* transfered to the disk. In the average case data will be committed every second.

appednfsync always

In this mode, and if the client does not use pipelining but waits for the replies before issuing new commands, data is both written to the file and synched on disk using fsync(2) **before an acknowledge is returned to the client**.

This is the highest level of durability that you can get, but is slower than the other modes.

The default Redis configuration is **appendfsync everysec** that provides a good balance between speed (is almost as fast as **appendfsync no**) and durability.

> What Redis implements when appendfsync is set to **always** is usually called **group commit**. This means that instead of using an fsync call for every write operation performed, Redis is able to *group* this commits in a single write+fsync operation performed before sending the request to the group of clients that issued a write operation during the latest event loop iteration.
>
> In practical terms it means that you can have hundreds of clients performing write operations at the same time: the fsync operations will be factorized - so even in this mode Redis should be able to support a thousand of concurrent transactions per second while a rotational device can only sustain 100-200 write op/s.
>
> This feature is usually hard to implement in a traditional database, but Redis makes it remarkably more simple.

Why is pipelining different?

The reason for handling clients using pipelining in a different way is that clients using pipelining *with writes* are sacrificing the ability to read what happened with a given command, before executing the next one, in order to gain speed. There is no point in committing data before replying to a client that seems not interested in the replies before going forward, the client is asking for speed. However even if a client is using pipelining, writes and fsyncs (depending on the configuration) always happen when returning to the event loop.

AOF and Redis transactions

AOF guarantees a correct MULTI/EXEC transactions semantic, and will refuse to reload a file that contains a broken transaction at the end of the file. An utility shipped with the Redis server can trim the AOF file to remove the partial transaction at the end.

Note: since the AOF file is populated using a single write(2) call at the end of every event loop iteration, an incomplete transaction can only appear if the disk where the AOF resides gets full while Redis is writing.

Comparison with PostrgreSQL

So how durable is Redis, with its main persistence engine (AOF) in its default configuration?

- Worst case: It guarantees that write(2) and fsync(2) are performed within two seconds.
- Normal case: it performs write(2) before replying to client, and performs an fsync(2) every second.

What is interesting is that in this mode Redis is still extremely fast, for a few reasons. One is that fsync is performed on a background thread, the other is that Redis only writes in append only mode, that is a big advantage.

However if you need maximum data safety and your write load is not high, you can still have the best of the durability that is possible to obtain *in any database system* using **fsync always**.

How this compares to PostgreSQL, that is (with good reasons) considered a good and very reliable database?

Let's read some PostgreSQL documentation together (note, I'm only citing the interesting pieces, you can find the full documentation here in the PostgreSQL official site)

> **fsync (boolean)**
>
> If this parameter is on, the PostgreSQL server will try to make sure that updates are physically written to disk, by issuing fsync() system calls or various equivalent methods (see wal_sync_method). This ensures that the database cluster **can recover to a consistent state after an operating system or hardware crash.**
>
> [snip]
>
> In many situations, turning off synchronous_commit for noncritical transactions can provide much of the potential performance benefit of turning off fsync, without the attendant risks of data corruption.

So PostgreSQL needs to fsync data in order to avoid corruptions. Fortunately with Redis AOF we don't have this problem at all, no

corruption is possible. So let's check the next parameter, that is the one that more closely compares with Redis fsync policy, even if the name is different:

> **synchronous_commit (enum)**
>
> Specifies whether transaction commit will wait for WAL records to be written to disk before the command returns a "success" indication to the client. Valid values are on, local, and off. The default, and safe, value is on. When off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times wal_writer_delay.) Unlike fsync, setting this parameter to off does not create any risk of database inconsistency: an operating system or database crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly.

Here we have something much similar to what we can tune with Redis. Basically the PostgreSQL guys are telling you, want speed? Probably it is a good idea to disable synchronous commits. That's like in Redis: want speed? Don't use **appendfsync always**.

Now if you disable synchronous commits in PostgreSQL you are in a very similar affair as with Redis **appendfsync everysec**, because by default *wal_writer_delay* is set to 200 milliseconds, and the documentation states that you need to multiply it by three to get the actual delay of writes, that is thus 600 milliseconds, very near to the 1 second Redis default.

> MySQL InnoDB has similar parameters the user can tune. From the documentation:
>
> If the value of innodb_flush_log_at_trx_commit is 0, the log buffer is written out to the log file once per second and the flush to disk operation is performed on the log file, but nothing is done at a transaction commit. When the value is 1 (the default), the log buffer is written out to the log file at each transaction commit and the flush to disk operation is

performed on the log file. When the value is 2, the log buffer is written out to the file at each commit, but the flush to disk operation is not performed on it. However, the flushing on the log file takes place once per second also when the value is 2. Note that the once-per-second flushing is not 100% guaranteed to happen every second, due to process scheduling issues.

You can read more here.

Long story short: even if Redis is an in memory database it offers good durability compared to other on disk databases.

From a more practical point of view Redis provides both AOF and RDB snapshots, that can be enabled simultaneously (this is the advised setup, when in doubt), offering at the same time easy of operations and data durability.

Everything we said about Redis durability can also be applied not only when Redis is used as a datastore but also when it is used to implement queues that needs to persist on disk with good durability.

Credits

Didier Spezia provided very useful ideas and insights for this blog post. The topic is huge and I'm sure I overlooked a lot of things, but surely thanks to Didier the current post is much better compared to the first draft.

Addendum: a note about restart time

I received a few requests about adding some information about restart time, since when a Redis instance is stopped and gets restarted it has to read the dataset from disk into memory. I think it is a good addition, because there are differences between RDB and AOF persistence, and between Redis 2.6 and Redis 2.4. Also it is interesting to see how Redis compares with PostgreSQL and MySQL in this regard.

First of all it's worth to mention why Redis requires to load the whole dataset in memory before starting to serve request to clients: the reason is not, strictly speaking, that it is an in-memory DB. It

is conceivable to think that a database that is in memory, but *uses the same representation of data in memory and on disk* could start serving data ASAP.

Actually the true reason is that we optimized the different representations for the different scopes they serve: on disk we have a compact append-only representation that is not suitable for random access. On memory we have the best possible representation for fast data fetching and modification. But this forces us to perform a *conversion step* on loading. Redis reads keys one after the other on disk, and encodes the same keys and associated values using the in-memory representation.

With RDB file this process is very fast for a few reasons: the first is that RDB files are usually more compact, binary, and sometimes even encode values in the same format they are in memory (this happens for small aggregate data types that are encoded as *ziplists* or *intsets*).

CPU and disk speed will do a big difference, but as a general rule you can think that a Redis server will load an RDB file at the rate of 10 ~ 20 seconds per gigabyte of memory used, so loading a dataset composed of tens of gigabytes can take even a few minutes.

Loading an AOF file that was just rewritten by the server takes something like twice per gigabyte in Redis 2.6, but of course if a lot of writes reached the AOF file *after* the latest compaction it can take longer (however Redis in the default configuration triggers a rewrite automatically if the AOF size reaches 200% of the initial size).

Restarting an instance is usually not needed however in a setup with a single server it is a better idea to use replication in order to transfer the control to the new Redis instance without service interruption. For instance in the case of an upgrade to a newer Redis version usually the system administrator will setup the Redis instance running the new version as slave of the old instance, then will point all the clients to the new instance, will turn this instance into a master, and will finally shut down the old one.

What about traditional on disk databases? They don't need to load data in memory... or maybe yes? Well basically they do a better job than Redis is this regard, because, when you start a MySQL server it is albe to serve request since the first second, however if the database and index files are no longer in the operating system cache what is happening is a *cold restart*. In this case the database will work since

the start, but will be very slow and may not be able to cope with the speed at which the application is requesting data. I saw this happening multiple times first-hand.

What is happening in a cold restart is that the database is actually *reading* data from disk to memory, very similarly to what Redis does, but incrementally.

Long story short: Redis requires some time to restart if the dataset is big. On disk databases are better in this regard, but you can't expect that they'll perform well in the case of a cold restart, and if they are under load it is easy to see a condition where the whole application is actually blocked for several minutes. On the other hand *once* Redis starts, it starts at full speed.

Edit: want to learn more?

- This article at sqlite.org about atomic commits is very good.
- What every programmer should know about disks
- Disks lie

Posted at 06:08:08 | **permalink** | **55 Comments** | **print**

Comments