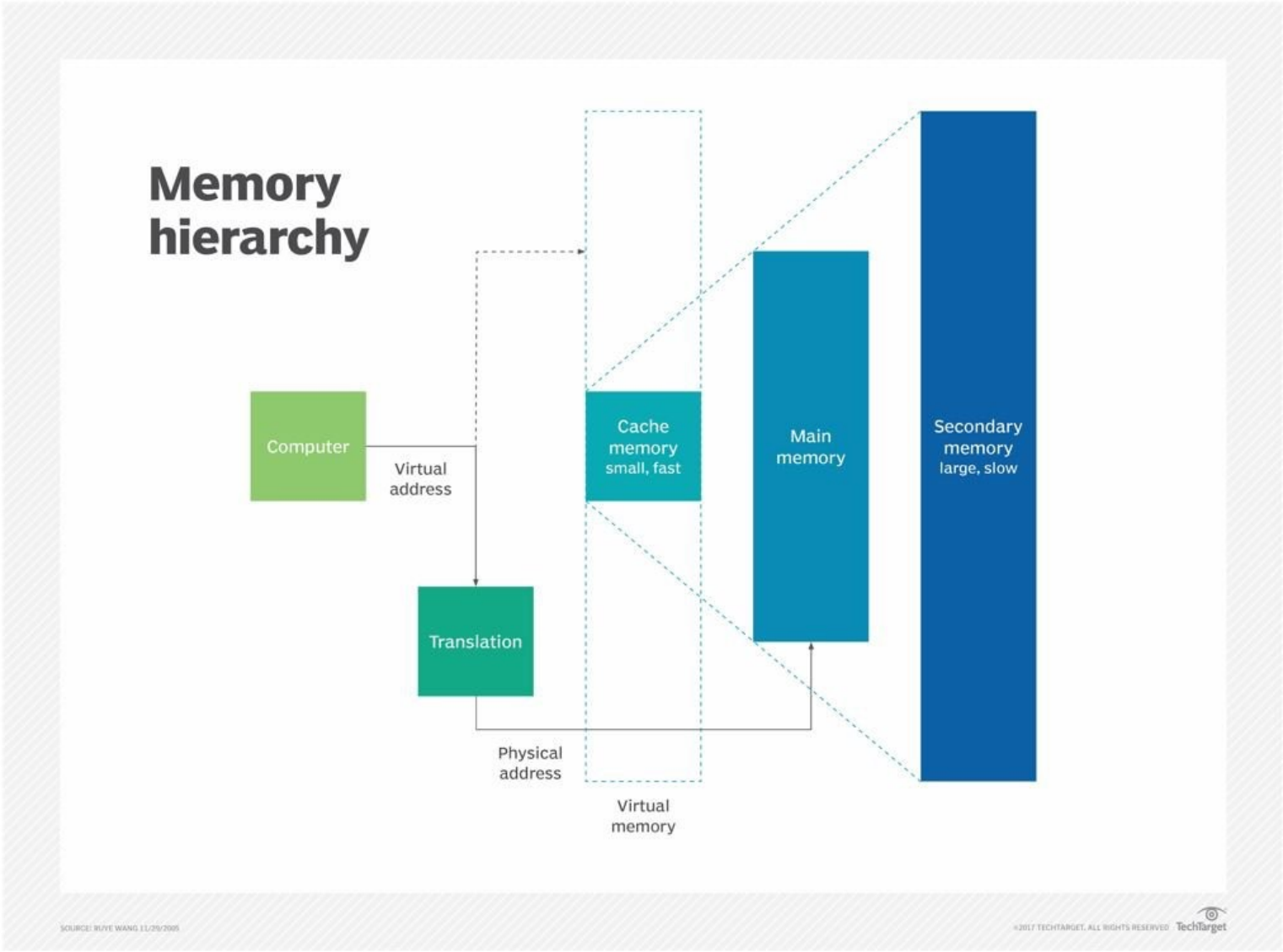
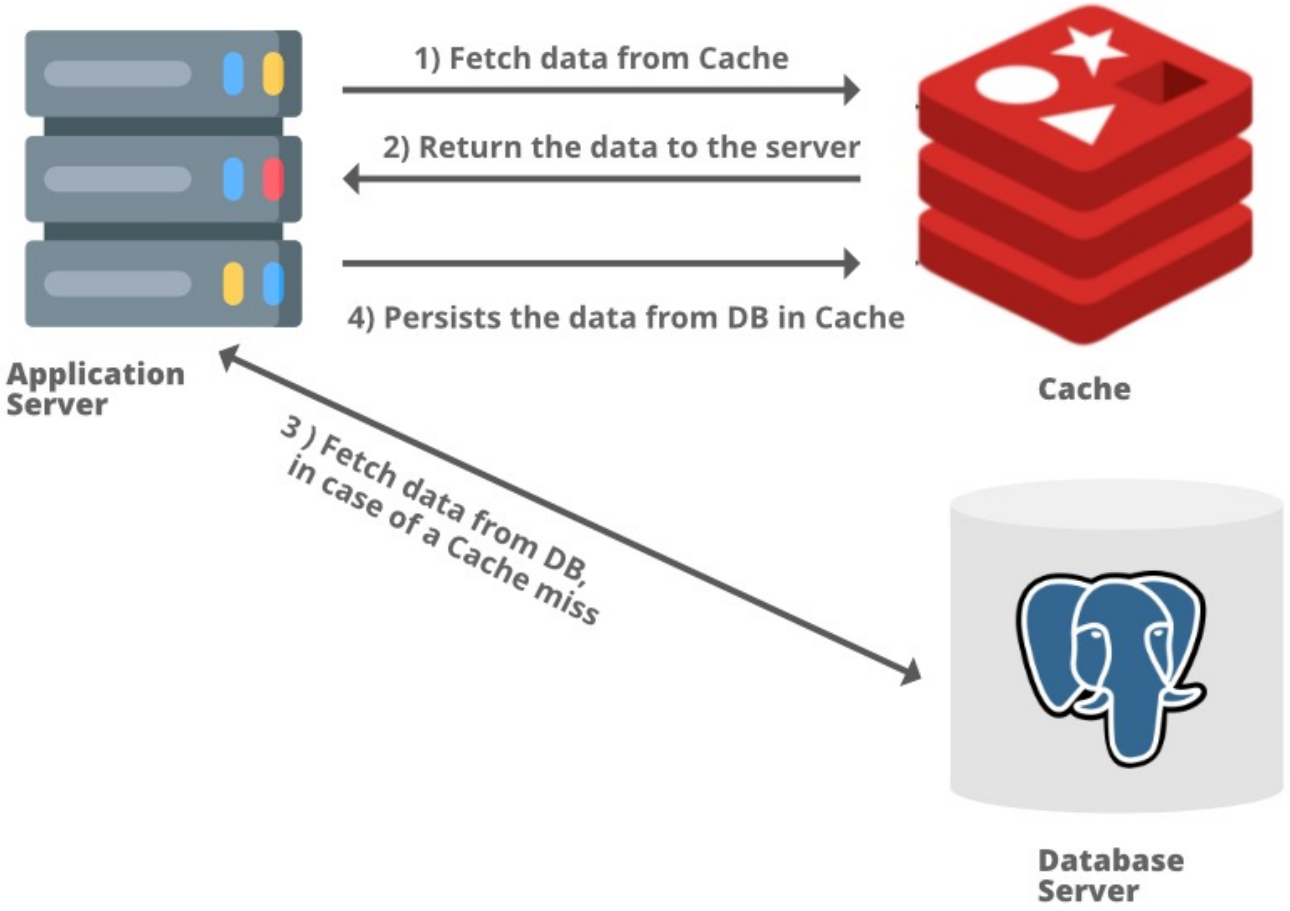
https://www.linkedin.com/in/fenil-2144-shah/



Cache

- A faster and smaller segment of memory whose access time is as close as registers(type of computer memory used by CPU) are known as Cache memory. In a hierarchy of memory, cache memory has access time lesser than primary memory. Generally, cache memory is very smaller and hence is used as a buffer. Cache is commonly used by the central processing unit (CPU), applications, web browsers and operating systems.
- Cache is used because bulk or main storage can't keep up with the demands of clients. Cache decreases data access times, reduces latency and improves input/output (I/O). Because almost all application workloads depend on I/O operations, the caching process improves application performance.
- When a cache client attempts to access data, it first checks the cache. If the data is found there, that is referred to as a cache hit. The percent of attempts that result in a cache hit is called the cache hit rate or ratio.
- Requested data that isn't found in the cache -- referred to as a cache miss -- is pulled from main memory and copied into the cache. How this is done, and what data is ejected from the cache to make room for the new data, depends on the caching algorithm, cache protocols and system policies being used.



Cache

- Caches are used to store temporary files, using hardware and software components. An example of a hardware cache is a CPU cache. This is a small chunk of memory on the computer's processor used to store basic computer instructions that were recently used or are frequently used.
- Many applications and software also have their own cache. This type of cache temporarily stores app-related data, files or instructions for fast retrieval.
- Web browsers are a good example of application caching. As mentioned earlier, browsers have their own cache that store information from previous browsing sessions for use in future sessions. A user wanting to rewatch a YouTube video can load it faster because the browser accesses it from cache where it was saved from the previous session.
- operating systems, where commonly used instructions and files are stored;
- content delivery networks, where information is cached on the server side to deliver websites faster;
- domain name systems, where they can be used to store information used to convert domain names to Internet Protocol addresses; and
- databases, where they can reduce latency in database query

Cache

- **Performance:** Storing data in a cache allows a computer to run faster. For example, a browser cache that stores files from previous browsing sessions speeds up access to follow up sessions. A database cache speeds up data retrieval that would otherwise take a good bit of time and resources to download.
- **Offline work:** Caches also let applications function without an internet connection. Application cache provides quick access to data that has been recently accessed or is frequently used. However, cache may not provide access to all application functions.
- **Resource efficiency:** Besides speed and flexibility, caching helps physical devices conserve resources. For example, fast access to cache conserves battery power.
- **DrawBacks**:
  - **Corruption**: Caches can be corrupted, making stored data no longer useful. Data corruption can cause applications such as browsers to crash or display data incorrectly.
  - **Performance**: Caches are generally small stores of temporary memory. If they get too large, they can cause performance to degrade.

Cache

- **Outdated information:** Sometimes an app cache displays old or outdated information. This can cause an application glitch or return misleading information. This is not a problem for static content but is a problem for dynamic content that changes over sessions or between sessions.
- **Cache algorithms:**
  - **Least Frequently Used** keeps track of how often a cache entry is accessed. The item that has the lowest count gets removed first. In real life, we can take the example of typing some texts on your phone. Your phone suggests multiple words when you type something in the text box. In this case, your phone keeps track of the frequency of each word you type and maintains the cache for it. Later the word with the lowest frequency is discarded from the cache when it's needed. If we find a tie between multiple words then the least recently used word is removed.
  - **Least Recently Used** puts recently accessed items near the top of the cache. When the cache reaches its limit, the least recently accessed items are removed.
  - **Most Recently Used** removes the most recently accessed items first. This approach is best when older items are more likely to be used. Tinder maintains the cache of all the potential matches of a user. It doesn't recommend the same profile to the user when he/she swipes the profile left/right in the application.

Cache policies

- **Write-around** cache writes operations to storage, skipping the cache. This prevents the cache from being flooded when there are large amounts of write I/O. The disadvantage to this approach is that data isn't cached unless it's read from storage. As a result, the read operation is slower because the data hasn't been cached.. So this approach is suitable for applications that don’t frequently re-read the most recent data.
- **Write-through** cache writes data to cache and storage. The advantage of write-through cache is that newly written data is always cached, so it can be read quickly. A drawback is that write operations aren't considered complete until the data is written to both the cache and primary storage. This can introduce latency into write operations. We can use this approach for the applications which have frequent re-read data once it’s persisted in the database. In those applications write latency can be compensated by lower read latency and consistency.
- **Write-back** cache is like write-through in that all the write operations are directed to the cache. But with write-back cache, the write operation is considered complete after the data is cached. Once that happens, the data is copied from the cache to storage. The problem with this approach is that until you schedule your database to be updated, the system is at risk of data loss. If there is a disk failure and the modified data hasn’t been updated into the DB. Since the database is the source of truth, if you read the data from the database you won’t get an accurate result.

Types of caches

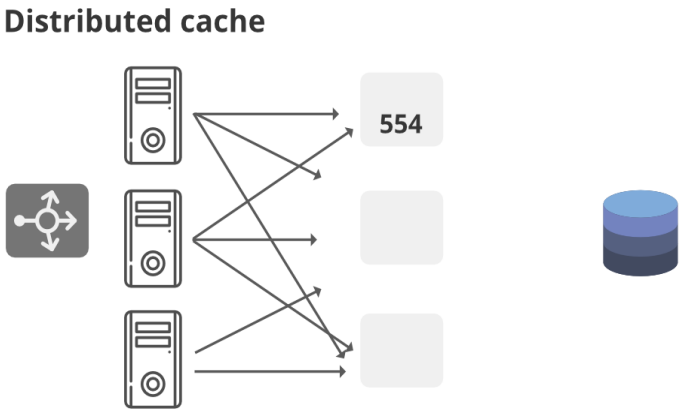
- **Cache memory** is RAM that a microprocessor can access faster than it can access regular RAM. It is often tied directly to the CPU and is used to cache instructions that are accessed a lot. A RAM cache is faster than a disk-based one, but cache memory is faster than a RAM cache because it's close to the CPU
- **Cache server**, sometimes called a proxy cache, is a dedicated network server or service. Cache servers save webpages or other internet content locally.
- **CPU cache** is a bit of memory placed on the CPU. This memory operates at the speed of the CPU rather than at the system bus speed and is much faster than RAM.
- **Disk cache** holds recently read data and, sometimes, adjacent data areas that are likely to be accessed soon. Some disk caches cache data based on how frequently it's read. Frequently read storage blocks are referred to as hot blocks and are automatically sent to the cache.
- **Flash cache**, also known as solid-state drive caching, uses NAND flash memory chips to temporarily store data. Flash cache fulfills data requests faster than if the cache were on a traditional hard disk drive or part of the backing store.

Types of caches

- **Application Server Cache**: Let's say a web server has a single node. A cache can be added in in-memory alongside the application server. The user's request will be stored in this cache and whenever the same request comes again, it will be returned from the cache. For a new request, data will be fetched from the disk and then it will be returned. Once the new request will be returned from the disk, it will be stored in the same cache for the next time request from the user. Placing cache on the request layer node enables local storage.
- Problem arises when you need to scale your system. You add multiple servers in your web application (because one node can not handle a large volume of requests) and you have a load balancer that sends requests to any node. In this scenario, you'll end up with a lot of cache misses because each node will be unaware of the already cached request.

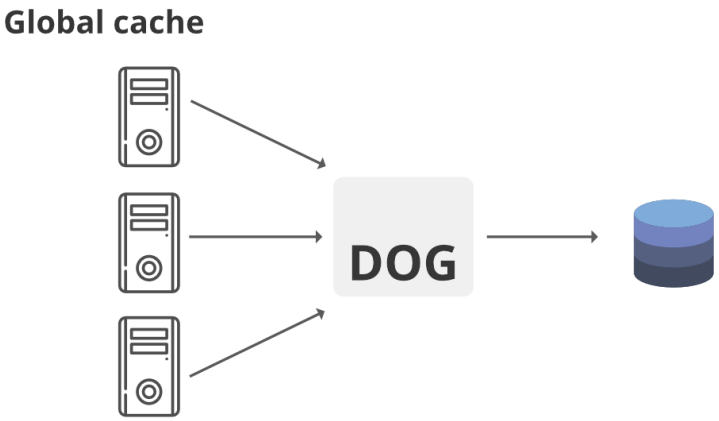
Types of caches

- **Distributed Cache**: In the distributed cache, each node will have a part of the whole cache space, and then using the consistent hashing function each request can be routed to where the cache request could be found. Let's suppose we have 10 nodes in a distributed system, and we are using a load balancer to route the request then
- Each of its nodes will have a small part of the cached data.
- To identify which node has which request the cache is divided up using a consistent hashing function each request can be routed to where the cached request could be found. If a requesting node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to check if the data is available.
- We can easily increase the cache memory by simply adding the new node to the request pool.



Types of caches

- **Global Cache:** You will have a single cache space and all the nodes use this single space. Every request will go to this single cache space. There are two kinds of the global cache
- First, when a cache request is not found in the global cache, it’s the responsibility of the cache to find out the missing piece of data from anywhere underlying the store (database, disk, etc).
- Second, if the request comes and the cache doesn’t find the data then the requesting node will directly communicate with the DB or the server to fetch the requested data.



Cookies

- Cookies are small files that contain information useful to a web site — such as password, preferences, browser, IP Address, date and time of visit, etc. Every time the user loads the website, the browser sends the cookie back to the server to notify the website of the user's previous activity.
- Cookies have a certain life span defined by their creators and it expires after the fixed time span.
- Cookies often track information like how frequently the user visits, what are the times of visits, what banners have been clicked on, what button clicked, user preferences, items in shopping cart, etc. This allows the site to present you with information customized to fit your needs.
- Cookies are usually used to store information needed for shorter periods. Cookies were first introduced by Netscape. In those earlier stages cookies did not receive good acceptance, since rumors said it might hack your personal data. Later people realized that cookies are actually harmless, and now they are highly accepted.

Cookies vs Cache

- Cookies are used to store information to track different characteristics related to user, while cache is used to make the loading of web pages faster.
- Cookies store information such as user preferences, while cache will keep resource files such as audio, video or flash files.
- Typically, cookies expire after some time, but cache is kept in the client's machine until they are removed manually by the user or has expiry.
- Cookies consumes less space in terms of capacity while cache consumes large space in terms of capacity.
- While cookie's contents are stored in both server and browser but Cache's website contents are stored in browser only.
- Eg- you search website to buy shoes, we leave and visit other website, will see ads related to that as browser history is stored as cookies based on which ads are shown, But if we visit same buying site again, it loads quickly because image and content are already stored in it.

SERVER CACHE VS CLIENT CACHE

- A server cache is all about storing the mostly used data in the server. The server holds the pre-assembled version of the various web pages of a website.
- Coming to the browser cache, the required data is stored in the user's hard drive. This is helpful as the user's computer does not have to download certain heavy elements of a web page again and again every time the user visits the webpage. However, a user has the right to clear the browser cache, and it is required when you want the server to serve the updated version of the webpage.
- A server cache can hold content, code, queries, and that too on multiple servers to serve end users faster. On the other hand, the browser cache stores HTML pages, CSS files, JavaScript scripts, and most important images and other types of media objects.
- The media objects take the most time to load as they are large in size and therefore, if they are stored in the user's computer, it can be served instantly rather than bringing them from the server.

Redis Cache

- Redis is an open-source, in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster
- So Redis can be used as a traditional monolithic and can be used as distributed system as a cluster of nodes with sharding. (Link: https://redis.io/). Redis
- All Redis data resides in the server's main memory, in contrast to databases such as PostgreSQL, SQL Server, and others that store most data on disk. In comparison to traditional disk-based databases where most operations require a roundtrip to disk, in-memory data stores such as Redis don't suffer the same penalty. They can therefore support an order of magnitude more operations and faster response times. The result is — blazing fast performance with average read or writes operations taking less than a millisecond and support for millions of operations per second.
- Redis data lives in memory, which makes it is very fast to write to and read from, but in case of server crashes you lose all that's in the memory, for some applications, it's ok to lose these data in case of a crash, but for other app, it's important to be able to reload Redis data after server restarts.

Redis Cache

- Redis is No-Sql Data Store storing in the form of Key-Value Pair. Redis acts as a cache for Real Time Queries. It also persists(saves) the data in the disk using background thread.
- Eg- Deleting Whatsapp message for everyone. Feature is available until its present in cache.
- Redis is single threaded on the frontend . It can connect to only one client at one time .
- Redis can store String , List and Hashes. Doesn’t supports int, float, double as storage level.

Redis Cache

- Redis provides a different range of persistence options:
- **RDB** (Redis Database): The RDB persistence performs point-in-time snapshots of your dataset at specified intervals. 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 NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage).
- RDB is very good for disaster recovery, being a single compact file that can be transferred to far data centers. RDB allows faster restarts with big datasets compared to AOF.
- 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.
- **AOF** (Append Only File): 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 in the background when it gets too big.

Redis Cache

- 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 a half-written command for some reason (disk full or other reasons) the Redis-check-of tool is able to fix it easily.
- 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. AOF can improve the data consistency but does not guarantee so likely you can lose your data but less than RDB mode considering the RDB is faster.
- **No persistence**: If you wish, you can disable persistence completely, if you want your data to just exist as long as the server is running.
- **RDB + AOF**: 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.

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.
- How it works:
  - 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.
- So Redis stores snapshots of your data to disk in a dump.rdb file in the following conditions:
  - Every minute if 1000 keys were changed
  - Every 5 minutes if 10 keys were changed
  - Every 15 minutes if 1 key was changed
- So if you're doing heavy work and changing lots of keys, then a snapshot per minute will be generated for you, in case your changes are not that much then a snapshot every 5 minutes, if it's really not that much then every 15 minutes a snapshot will be taken.

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`*
- **appendfsync always**: fsync every time new commands are appended to the AOF. Very very slow, very safe. Note that the commands are appended to the AOF after a batch of commands from multiple clients or a pipeline are executed, so it means a single write and a single fsync (before sending the replies).
- **appendfsync everysec**: 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.
- **appendfsync no**: Never fsync, just put your data in the hands of the Operating System. The faster and less safe method. Normally Linux will flush data every 30 seconds with this configuration, but it's up to the kernel exact tuning

Append-only file

- 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.
- Redis can not guarantee consistency under any model as the writing to disk is always done async by the engine, and likely you can lose the data if the crash happened before data sync you can reduce this but you can not prevent.