A Crash Course in Redis

BYTEBYTEGO
SEP 21, 2023 · PAID

68 1 1 Share ···

Redis (Remote Dictionary Server) is an open-source (BSD licensed), in-memory database, often used as a cache, message broker, or streaming engine. It has rich support for data structures, including basic data structures like String, List, Set, Hash, SortedSet, and probabilistic data structures like Bloom Filter, and HyperLogLog.

Redis is super fast. We can run Redis benchmarks with its own tool. The throughput can reach nearly ***100k requests*** per second.

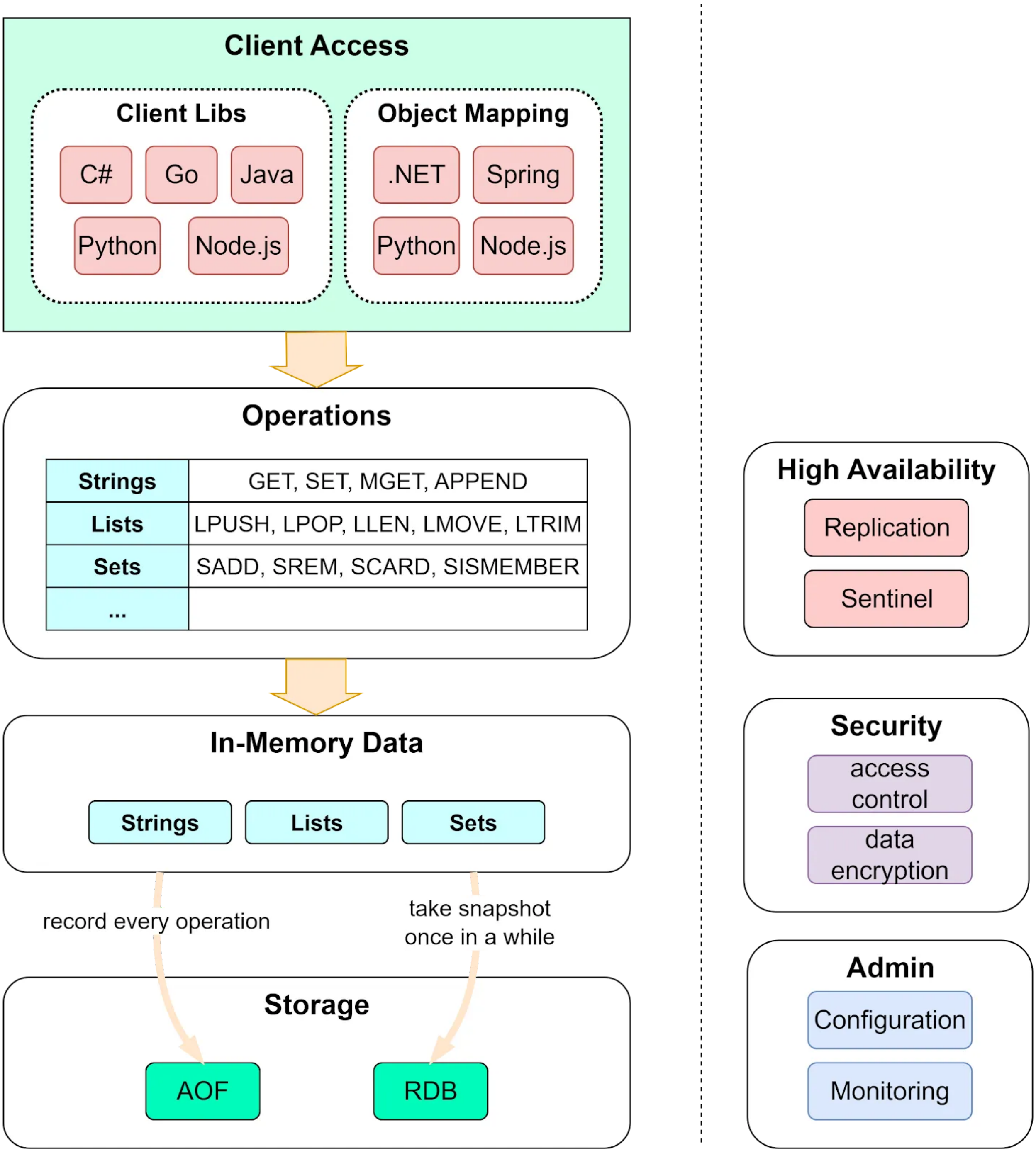
In this issue, we will discuss why Redis is fast in its architectural design.

Redis Architecture

Redis is an in-memory key-value store. There are several important functions:

1. The data structures used for the values
2. The operations allowed on the data structures
3. Data persistence
4. High availability

Below is a high-level diagram of Redis' architecture. Let’s walk through them one by one.



Strings	GET, SET, MGET, APPEND
Lists	LPUSH, LPOP, LLEN, LMOVE, LTRIM
Sets	SADD, SREM, SCARD, SISMEMBER
...	

Client Libraries

There are two types of clients to access Redis: one supports connections to the Redis database, and the other builds on top of the former and supports object mappings.

Redis supports a wide range of languages, allowing it to be used in a variety of applications. In addition, the OM client libraries allow us to model, index, and query documents.

Data Operations

Redis has rich support for value data types, including Strings, Lists, Sets, Hashes, etc. As a result, Redis is suitable for a wide range of business scenarios. Depending on the data types, Redis supports different operations.

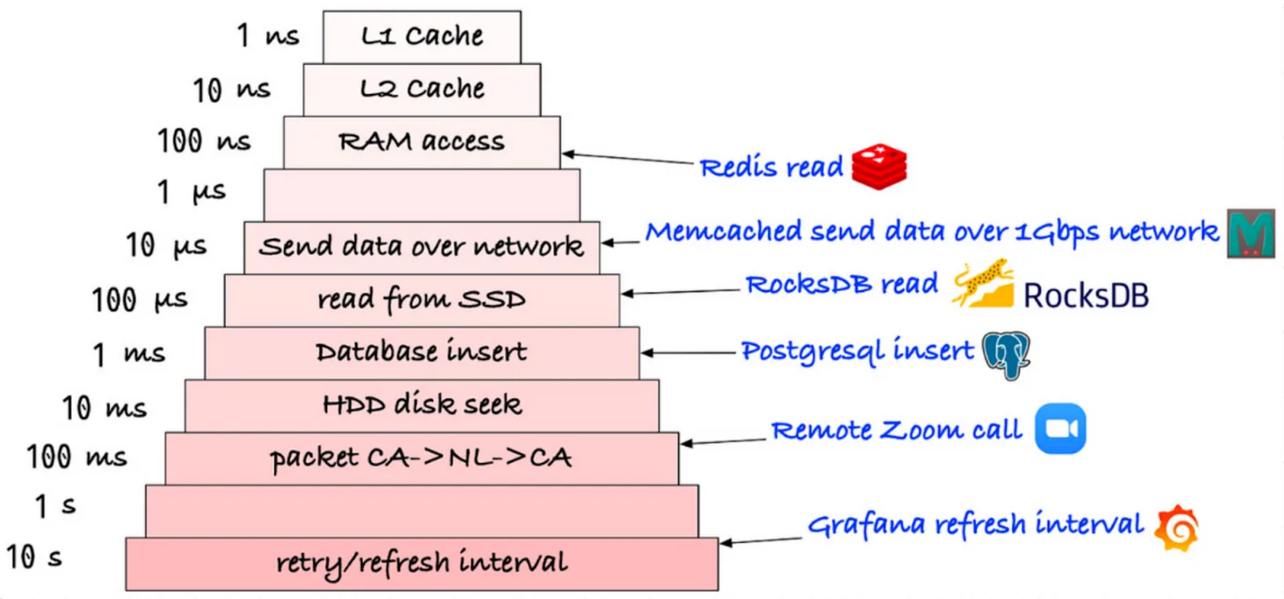
The basic operations are similar to a relational database, which supports CRUD (Create-Read-Update-Delete):

- GET: Retrieve the value of a key
- PUT: Create a new key-value pair or update an existing key
- DELETE: Delete a key-value pair

The data structures and operations are an important reason why Redis is so efficient. We will cover more in later sections.

In-Memory v.s. On-Disk

Redis holds the data in memory. The data reads and writes in memory are generally 1,000X - 10,000X faster than disk reads/writes. See the below diagram for details.



However, if the server is down, all the data will be lost. So Redis designs on-disk persistence as well for fast recovery.

Redis has 4 options for persistence:

1. AOF (Append Only File).

AOF works like a commit log, recording each write operation to Redis. So when the server is restarted, the write operations can be replayed and the dataset can be reconstructed.

2. RDB (Redis Database).

RDB performs point-in-time snapshots at a predefined interval.

3. AOF and RDB.

This persistence method combines the advantages of both AOF and RDB, which we will cover later.

4. No persistence.

Persistence can be entirely disabled in Redis. This is sometimes used when Redis is a cache for smaller datasets,

Clustering

Redis uses a leader-follower replication to achieve high availability. We can configure multiple replicas for reads to handle concurrent read requests. These replicas automatically connect to the master after restarts and hold an exact copy of the leader instance.

When the Redis cluster is not used, Redis Sentinel provides high availability including failover, monitoring, and configuration management.

Security and Administration

Redis is often used as a cache and can hold sensitive data, so it is designed to be accessed via trusted clients inside trusted environments. Redis security module is responsible for managing the access control layer and authorizing the valid operations to be performed on the data.

Redis also provides an admin interface for configuring and managing the cluster. Persistence, replication, and security configurations can all be done via the admin interface.

Now we have covered the basic components of Redis architecture, we will dive into the design details that make Redis fast.

In-Memory Data Structures

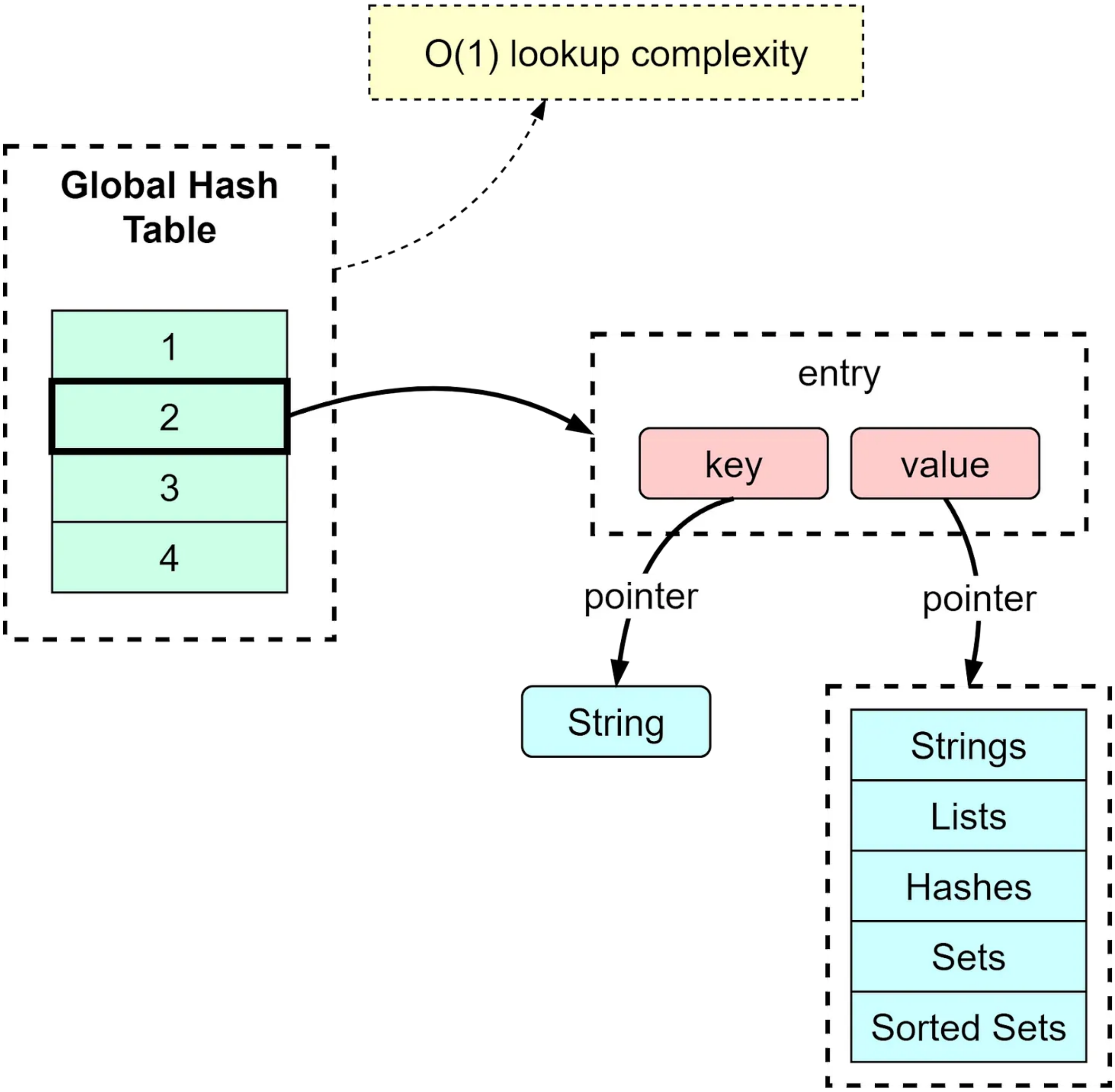
Redis is not the only in-memory database product in the market. But how can it achieve microsecond-level data access latency and become a popular choice for many companies?

One important reason is that storing data in memory allows for more flexible data structures. **These data structures don't need to go through the process of serialization and deserialization** like normal on-disk data structures do, so can be optimized for fast reads and writes.

Key-Value Mappings

Redis uses a hash table to hold all key-value pairs. The elements in the hash table hold the pointers to a key-value pair entry. The diagram below illustrates how the global hash table is structured.

With the hash table, we can look up key-value pairs with O(1) time complexity.

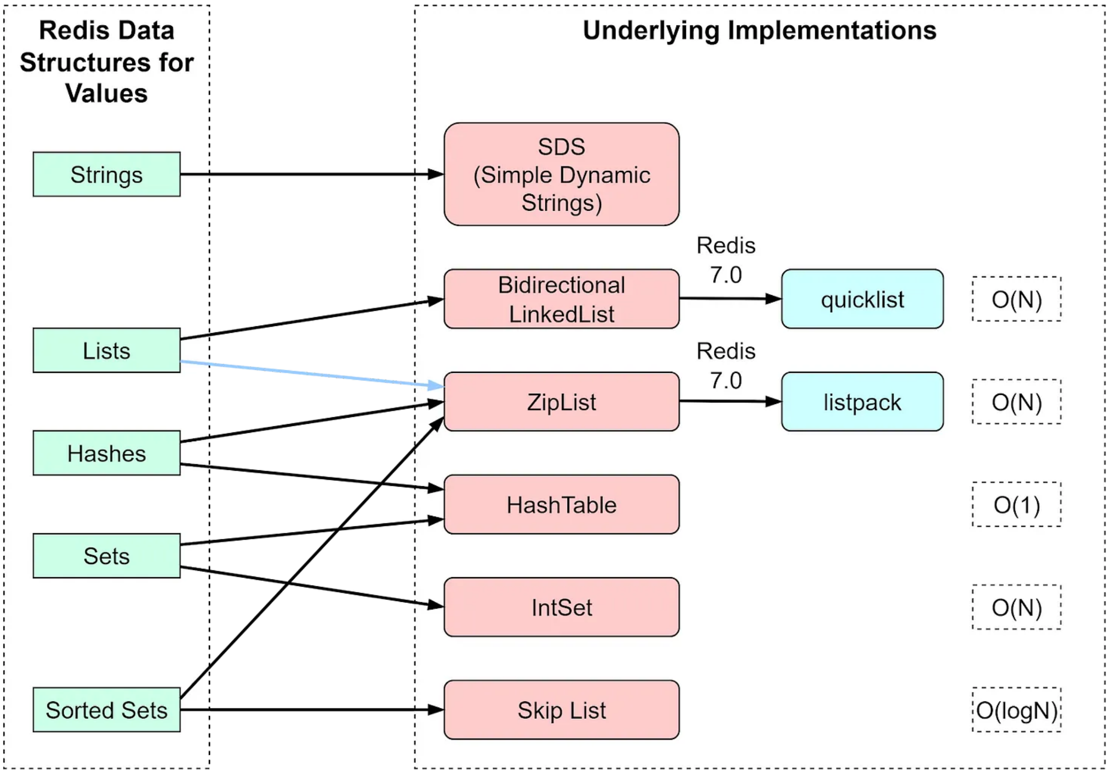


Like all hash tables, when the number of keys keeps growing, there can be hash conflicts, which means different keys fall into the same hash bucket. Redis solves this by chaining the elements in the same hash bucket. When the chains become too long, Redis will perform a rehash by leveraging two global hash tables.

Value Types

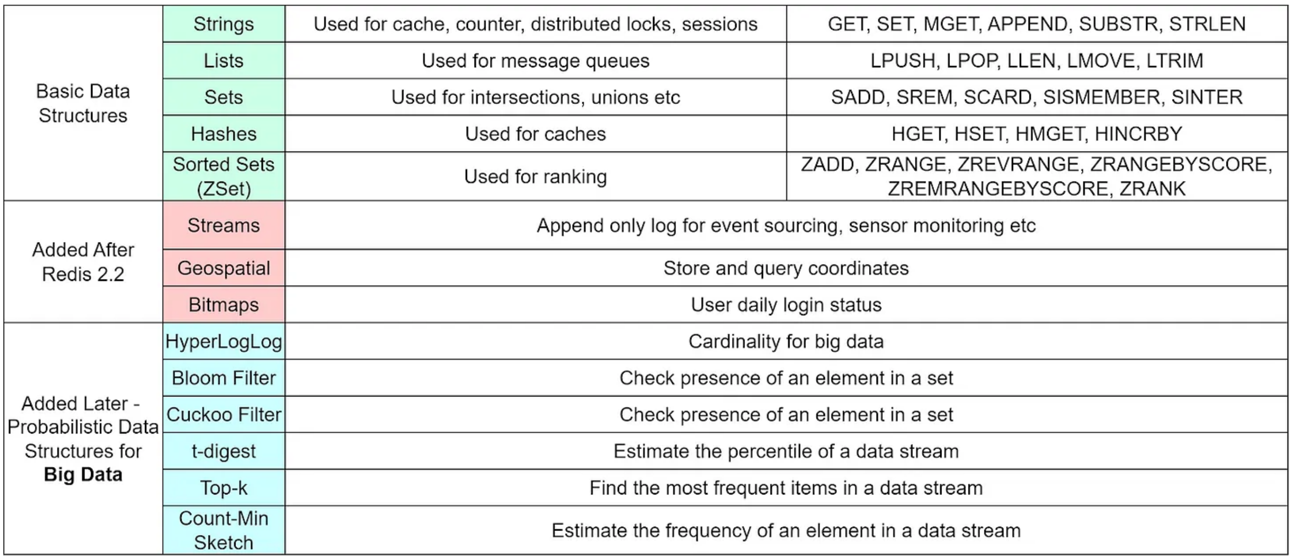
The diagram below shows how Redis implements the common data structures. String type has only one implementation, the SDS (Simple Dynamic Strings). List, Hash, Set, and SortedSet all have two types of implementations.

Note that Redis 7.0 changed List implementation to quicklist, and ZipList was replaced by listpack.



Besides these 5 basic data structures, Redis later added more data structures to support more scenarios. The diagram below lists the operations allowed on basic data structures and the usage scenarios.

These data types cover most of the usage of a website. For example, geospatial data stores coordinates that can be used by a ride-hailing application like Uber; HyperLogLog calculates cardinality for massive amounts of data, suitable for counting unique visitors for a large website; Stream is used for message queues and can compensate the problems with List.



Basic Data Structures	Strings	Used for cache, counter, distributed locks, sessions	GET, SET, MGET, APPEND, SUBSTR, STRLEN
	Lists	Used for message queues	LPUSH, LPOP, LLEN, LMOVE, LTRIM
	Sets	Used for intersections, unions etc	SADD, SREM, SCARD, SISMEMBER, SINTER
	Hashes	Used for caches	HGET, HSET, HMGET, HINCRBY
	Sorted Sets (ZSet)	Used for ranking	ZADD, ZRANGE, ZREVRANGE, ZRANGEBYSCORE, ZREMRANGEBYSCORE, ZRANK
Added After Redis 2.2	Streams	Append only log for event sourcing, sensor monitoring etc	
	Geospatial	Store and query coordinates	
	Bitmaps	User daily login status	
Added Later - Probabilistic Data Structures for **Big Data**	HyperLogLog	Cardinality for big data	
	Bloom Filter	Check presence of an element in a set	
	Cuckoo Filter	Check presence of an element in a set	
	t-digest	Estimate the percentile of a data stream	
	Top-k	Find the most frequent items in a data stream	
	Count-Min Sketch	Estimate the frequency of an element in a data stream	

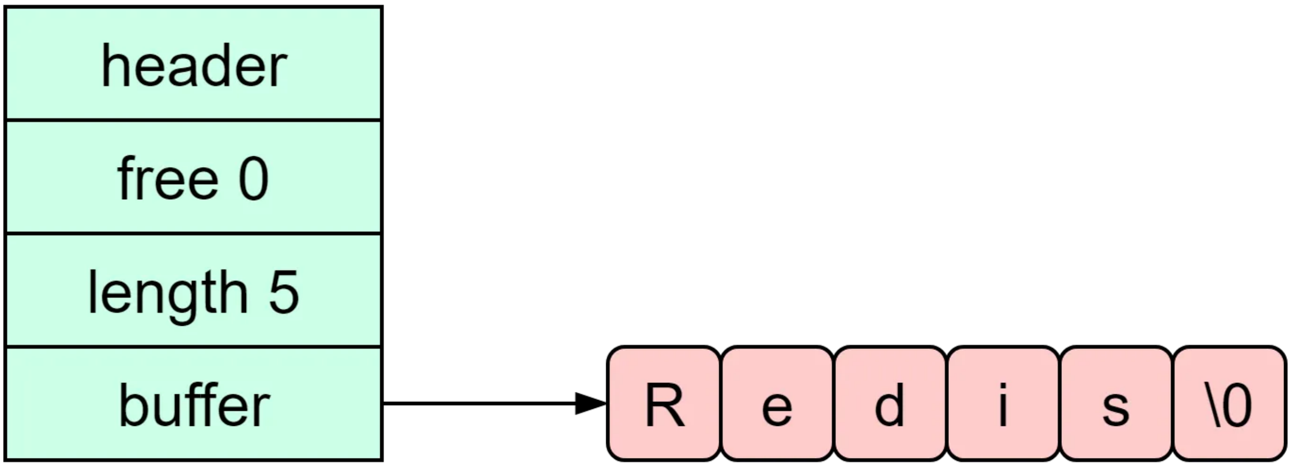
Now let's look at why these underlying implementations are efficient.

SDS

Redis SDS stores sequences of bytes. It operates the data stored in *buf* array in a binary way, so SDS can store not only text but also binary data like audio, video, and images.

The string length operation on an SDS has a time complexity of O(1) because the length is recorded in *len* attribute. The space is pre-allocated for an SDS, with *free* attribute recording the free space for future usage. The SDS API is thus safe, and there is no risk of overflow.

The diagram below shows the attributes of an SDS.



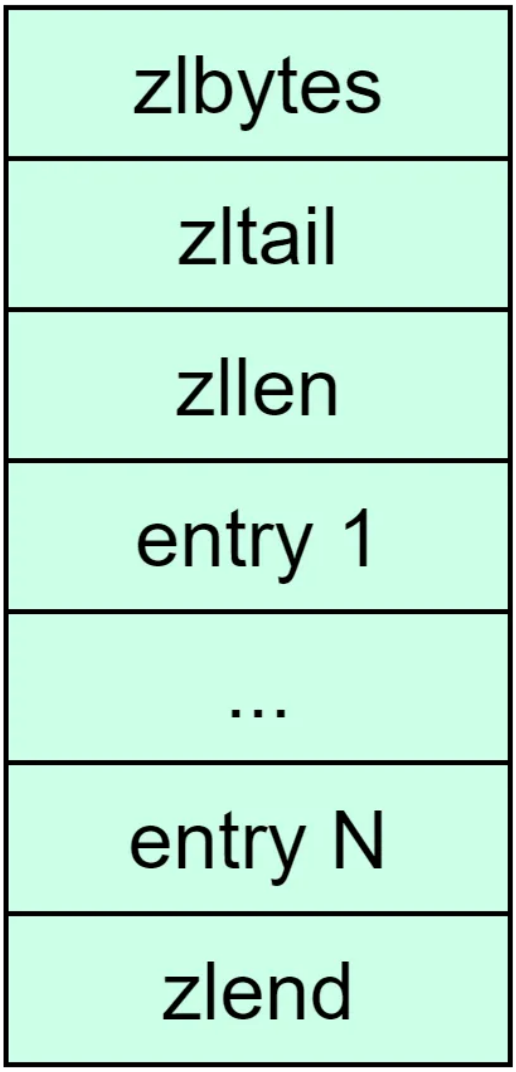
Zip List

A zip list is similar to an array. Each element of the array holds one piece of data. However, unlike an array, a zip list has 3 fields in the header:

- *zlbytes* - the length of the list
- *zltail* - the offset at the end of the list
- *zllen* - the number of entries in the list

The zip list also has a *zlend* at the end, which indicates the end of the list.

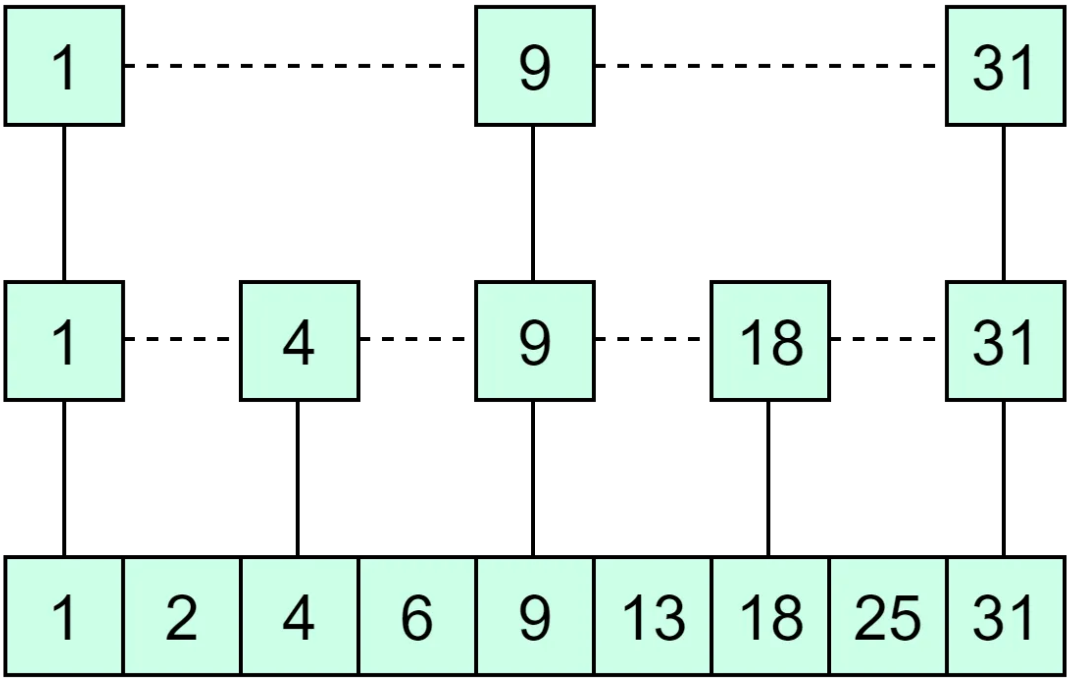
In a zip list, locating the first or the last element is O(1) time complexity because we can directly find them by the fields in the header. Locating other elements needs to go through the elements one by one, and the time complexity is O(N).



Skip List

The skip list adds multi-layer indices on top of a linked table so that we can locate the element faster. The diagram below shows what a skip list looks like.

If the number of elements is less than 128 and the size of each element is less than 64 bytes, Redis uses zip list as the underlying data structure for ZSet. For a larger sorted set, Redis uses skip list for ZSet.



Threading model

When dealing with a large number of concurrent visits, it is recommended to increase the number of threads. Then why is single-threaded Redis so fast?

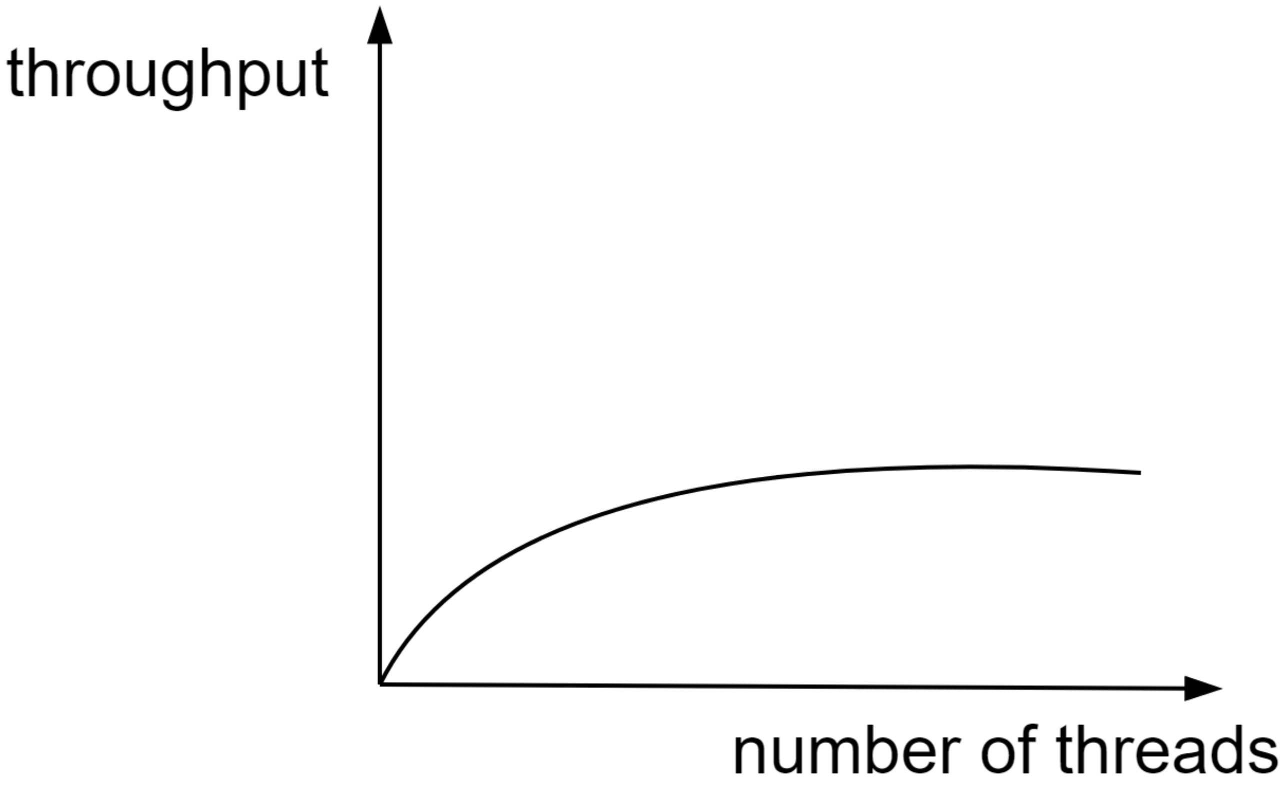
Is Redis really single-threaded?

When we say Redis is single-threaded, it means that all network IO and read/write operations are handled by a single thread. Other Redis functions, such as persistence and data replication, are handled by other threads. So we can say Redis is single-threaded for critical path operations.

After Redis 6.0, it uses multi-threading to handle network I/O requests because single-threaded network processing cannot fully leverage the high-performant network hardware. However, Redis still uses a single-threaded model for read/write operations on data structures. This change greatly improves Redis' performance.

Problems with multi-threading

Multi-threading model has a cost. As we can see in the diagram below, when we increase the number of threads, the throughput increases at the beginning; but when we increase the number of threads further, the throughput does not increase proportionally and sometimes even drops



Why does Redis use a single-threaded model?

Single-threaded Redis cannot fully leverage the capabilities of a modern CPU with multiple cores. What is the reason behind the design?

The official answer by Redis is:

It's not very frequent that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance, when using pipelining a Redis instance running on an average Linux system can deliver 1 million requests per second, so if your application mainly uses O(N) or O(log(N)) commands, it is hardly going to use too much CPU.

So the performance of Redis is bound to the network and memory, not the CPU. The design significantly reduces the cost of context switches.

I/O Multiplexing

The I/O multiplexing mechanism refers to a single thread handling multiple I/O streams. This capability is provided by the *select/epoll* functions. Redis leverages I/O multiplexing to allow multiple listening sockets and connected sockets to exist in the kernel simultaneously. Once requests arrive, they are sent to a queue (event loop) and dispatched to different event handlers.

The diagram below shows the process. Clients 1 and 2 have established connections with Redis, and Client 3 is initiating the connection. The Redis network module invokes the *epoll* mechanism, instructing the kernel to monitor these 3 sockets. When an accept event, read event, or write event comes in, they are put into an event loop. Redis then calls the corresponding event handlers to process the events.

Optimized Persistence

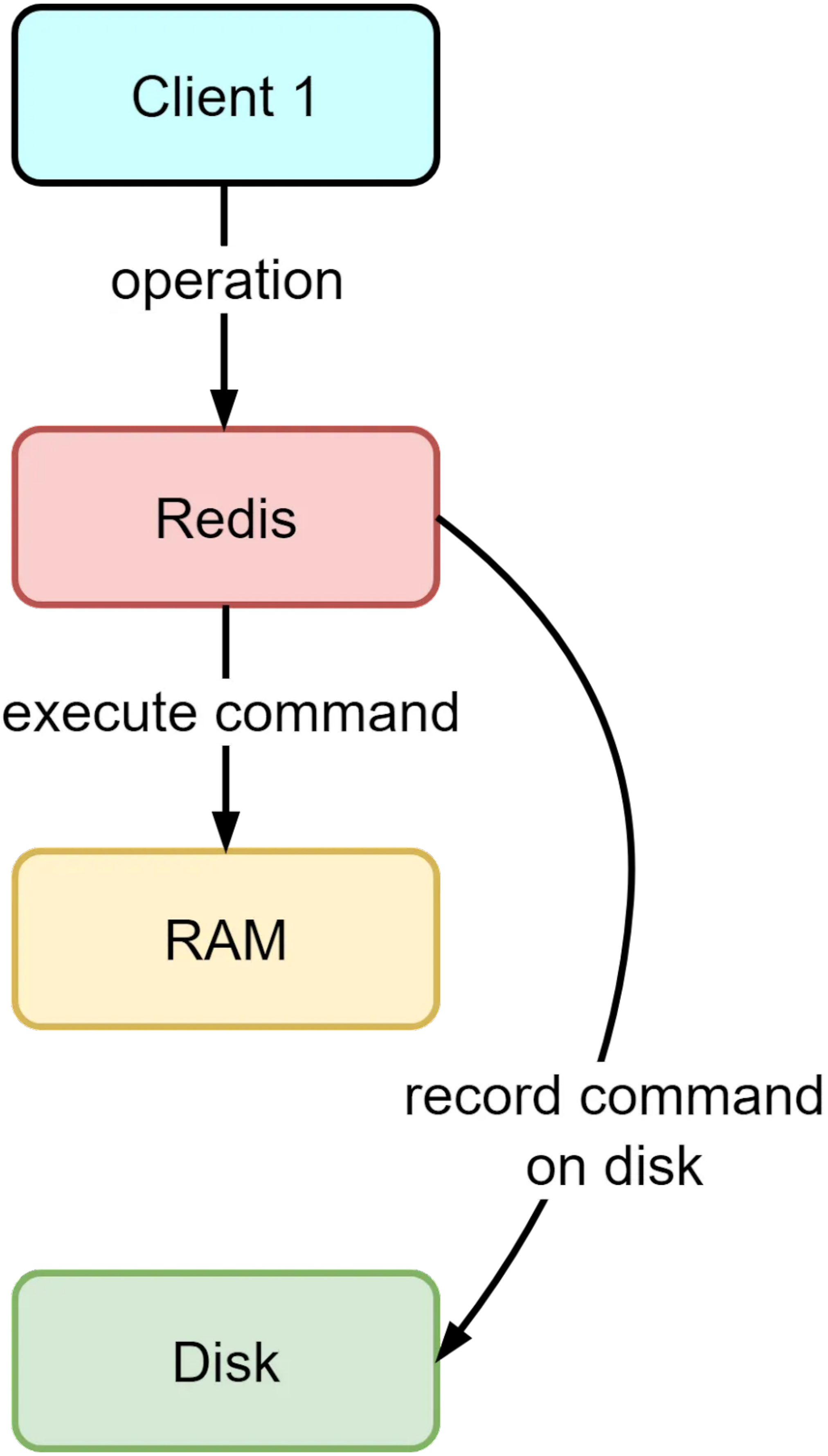
We often use Redis as a cache and keep the data in memory. If the Redis server goes down, all the data is lost. So Redis provides persistence for fast recovery. Note that the data persistence is not done on the critical path thread and can be optimized to reduce I/O pressure on the Redis server.

Redis provides two persistence mechanisms:

1. AOF (Append Only File) log
2. RDB snapshots

AOF

AOF log is different from Write Ahead Log (WAL). The latter means the modified data is written to a log file first before it is modified. The AOF log is the opposite. It is a write-after log, meaning Redis executes commands to modify the data in memory first and then writes it to the log file. The diagram below shows how AOF works.



AOF log records the commands instead of the data. The event-based design simplifies data recovery. Additionally, AOF records commands after the command has been executed in memory, so it does not block the current write operation.

If we record every command on disk after it is executed, the I/O load can be high. So Redis provides three options:

- **Always**. The command is written to the disk immediately after it is executed.
- **Everysec**. After each write command is executed, the command is written to the memory buffer first. The accumulated commands in the memory buffer are written to the disk every second.
- **No**. After each write command is executed, the command is written to the memory buffer first. Later the operating system decides when to flush the buffer to the disk.

RDB

The restriction of AOF is that it persists commands instead of data. When we use the AOF log for recovery, the whole log must be scanned. When the size of the log is large, Redis takes a long time to recover. So Redis provides another way to persist data - RDB.

RDB records snapshots of data at specific points in time. When the server needs to be recovered, the data snapshot can be directly loaded into memory for fast recovery.

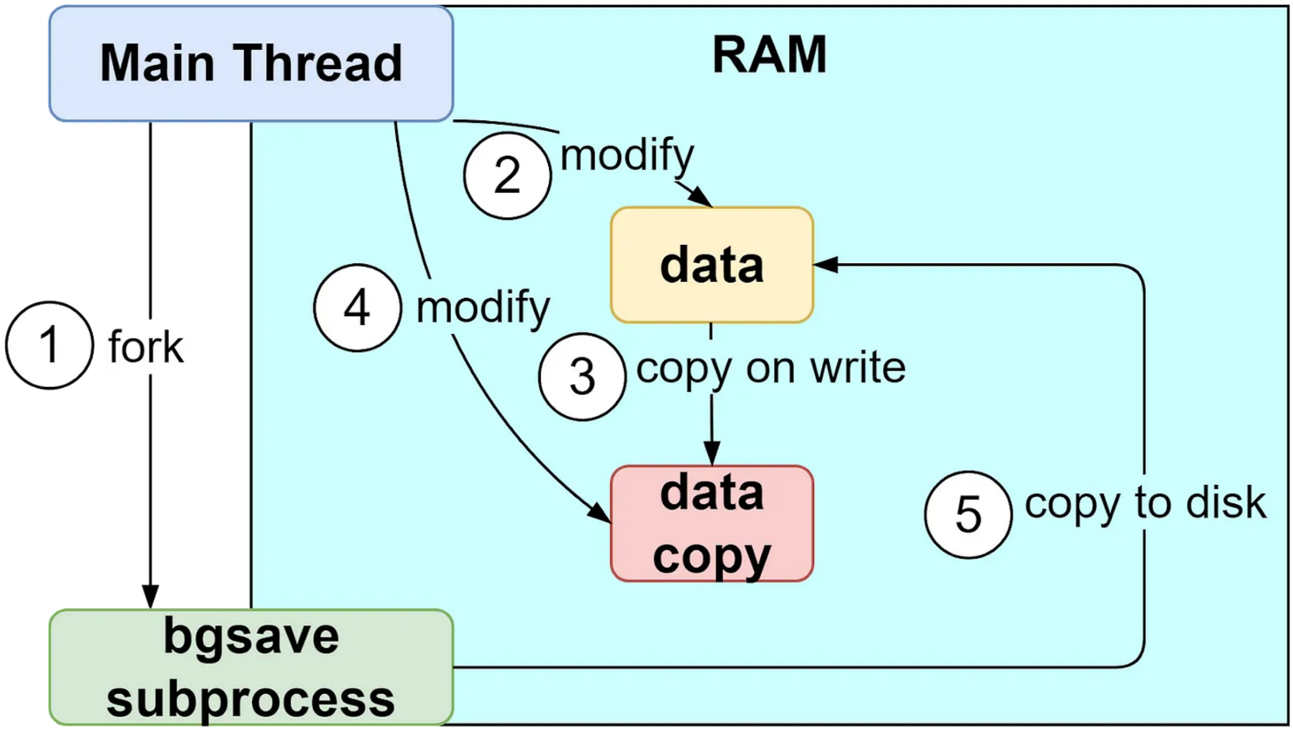
RDB process shouldn't block write operations. The diagram below shows how RDB works with the copy-on-write mechanism.

Step 1: The main thread forks the‘ bgsave’ sub-process, which shares all the in-memory data of the main thread. ‘bgsave’ reads the data from the main thread and writes it to the RDB file.

Steps 2 and 3: If the main thread modifies data, a copy of the data is created.

Steps 4 and 5: The main thread then operates on the data copy. Meanwhile ‘bgsave’ sub-process continues to write data to the RDB file.

As a result, the main thread can continue to modify data while the background copy operation is taking place.



Summary

In this issue, we talk through the components of Redis architecture and then deep dive into why Redis is fast. The in-memory data structures provide more efficiency and flexibility, allowing most scenarios to be done directly in Redis. The single-threaded operational model has fewer race conditions on the data and fewer context switches among the threads. Redis' performance is not CPU-bound, so we don't need to use multiple threads for critical-path operations. The newer version of Redis uses multi-threading to handle network requests, which is an optimization in response to the performance improvement of network hardware. Last but not least, Redis optimized persistence to reduce I/O pressure and avoid blocking on the main thread.

These good design points make Redis an ideal cache choice and an excellent design example for developers.

68 Likes · 1 Restack

1 Comment

Write a comment...

Sunny Writes Sunny’s Substack 3 hrs ago · *edited 3 hrs ago*

I felt this issue isn't like a paid one! Could have been better. I like the writeup though. Thanks!

♡ LIKE REPLY SHARE •••