REDIS

Introduction:-

  REDIS (REmote DIctionary Server) is an in-memory, key-value database, commonly referred to as a data structure server. One of the key differences between Redis and other key-value databases are Redis’s ability to store and manipulate high-level data types. These data types are fundamental data structures (lists, maps, sets, and sorted sets) that most developers are familiar with. Redis’s exceptional performance, simplicity, and atomic manipulation of data structures lend itself to solving problems that are difficult or perform poorly when implemented with traditional relational databases.

  Redis is an open-source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. It supports data structures such as string, hashes, lists, and sets sorted sets with range queries, bitmaps, hyper logs, and geospatial indexes with radius queries.

  Redis is written in ANSI C and works in most POSIX systems like Linux, BSD, and OS X without external dependencies. Linux and OS X are the two operating systems where Redis is developed and more tested.

  Redis is basically used for cache management. It reduces the client's workload and speeds up the application.

Applications:-

  **Caching** – Due to its high performance, developers have turned to Redis when the volume of reading and write operations exceed the capabilities of traditional databases. With Redis’s capability to easily persist the data to disk, it is a superior alternative to the traditional Memcached solution for caching.

  **Publish and Subscribe** – Since version 2.0, Redis provides the capability to distribute data utilizing the Publish/Subscribe messaging paradigm. Some organizations have moved to Redis and away from other message queuing systems (i.e., RabbitMQ, zeromq) due to Redis’s simplicity and performance.

  **Queues** – Projects such as Rescue use Redis as the backend for queueing background jobs.

  **Counters** – Atomic commands such as HINCRBY, allow for a simple and thread-safe implementation of counters. Creating a counter is as simple as determining a name for a key and issuing the HINCRBY command. There is no need to read the data before incrementing, and there are no database schemas to update. Since these are atomic operations, the counters will maintain consistency when accessed from multiple application servers.

Key Features:-

  **High-Level Data Structures** – Provides five possible data types for values: strings, lists, sets, hashes, and sorted sets. Operations that are unique to those data types are provided and come with well-documented time-complexity (Big O notation).

  **High Performance** – Due to its in-memory nature, the project maintainer’s commitment to keeping complexity at a minimum, and an event-based programming model, Redis boasts exceptional performance for reading and write operations.

  **Lightweight With No Dependencies** – Written in ANSI C, and has no external dependencies. Works well in all POSIX environments. Windows is not officially supported, but an experimental build is provided by Microsoft.

  **High Availability** – Built-in support for asynchronous, non-blocking, master/slave replication to ensure high availability of data. There is currently a high-availability of the solution called Redis Sentinel that is currently usable but is still considered a work in progress.

Companies using REDIS:-

  **Twitter** – Deployed a massive Redis cluster to store the timelines for all users. Utilizing the list data structure, Twitter stores the 800 most recent incoming tweets for a given user. View the presentation given by Twitter on how they distribute timelines at scale.

  **Pinterest** – Stores the user follower graphs in a Redis cluster where data is shared across hundreds of instances. Pinterest turned to Redis after finding that their original solution of MySQL and Memcached was reaching its limits. More on how Pinterest is using Redis.

  **Github** – An early adopter of the Redis project, Github has developed and open-sourced the library, Rescue, to facilitate the execution of background jobs that have been placed in a queue. Github developers took advantage of the fact that Redis had solved many of the difficult queueing issues, so the developers could stay focused on the difficult worker scheduling issues. More on how Github uses Redis for their job queueing needs.

What I have implemented in my project:-

I have implemented six of the given goal functions in my project and I have implemented them on IntelliJ IDEA. For implementation, Java language has been used. And then I have uploaded all the files on GitHub as per the guidelines.

The functions that I have implemented in this project:

1)      **GET**: Get the value of key. If the key does not exist the special value nil is returned. An error is returned if the value stored key is not a string, because GET only handles string values

2)      **SET**: Set key to hold the string value. If key already holds a value, it is overwritten, regardless of its type. Any previous time to live associated with the key is discarded on the successful SET operation. OK if SET was executed correctly.  A Null Bulk Reply is returned if the SET operation was not performed.

3)      **EXPIRE**: Set a timeout on key. After the timeout has expired, the key will automatically be deleted. A key with an associated timeout is often said to be *volatile* in Redis terminology.

4)      **ZADD**: Add all the specified members with the specified scores to the sorted set stored at key. It is possible to specify multiple score/member pairs. If a specified member is already a member of the sorted set, the score is updated and the element reinserted in the right position to ensure the correct ordering.

5)      **ZRANGE**: Returns the specified range of elements in the sorted set stored at key. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with equal scores.

6)      **ZRANK**: Returns the rank of member in the sorted set stored at key, with the scores ordered from low to high. The rank (or index) is 0-based, which means that the member with the lowest score has rank 0.

Why did I choose Java language?

Actually, recently I have started learning java and want to explore more in that so I took help from the internet to know more about the libraries and I feel very comfortable with it. Its set of libraries is also very vast that helps the developer to not build everything from scratch.

What are the further improvements that can be made to make it efficient?

Here we are writing to a disk whenever set commands executed, so a number of disk access is directly proportional to a number of the set commands.

So we can reduce the disk access by writing disk only when our client program is going to exit.

What data structures have I used and why?

We have used a Hash map for storing key and value pairs. So that Insertion takes O(1) time and same as Access takes O(1) time

We also use Map<Float, Set<String>> for storing the list. (i.e. Zadd, Zrange,Zrank)

For one key we have many values, but the value should be unique. i.e. mentioned in our requirement. So, we use that DS.

Does your implementation support multi-threaded operations? If No why can’t it be? If yes then how?

Yes, my implementation supports Multithreading, our server can handle multiple clients.

Whenever a new Client comes then the server creates a thread and all the request of that client has been served in that child thread. And the further process takes place.

-----------------------THANK YOU-----------------------