redislabs
HOME OF REDIS

WHITE PAPER

Popular Redis Uses for Beginners

Dave Nielsen

CONTENTS

Introduction to Redis 2
What Does 'Data Structure Store' Mean? 2
Popular Uses of Redis 3
Redis for User Session Management 3
Redis for Real-time Analytics 5
Redis for User Recommendations 6
Redis for Managing Queues of Work 7
Redis for Caching 8
What's next 8

Introduction to Redis

Redis is an in-memory, NoSQL data structures store, frequently used as a database, cache, and message broker. Unlike other in-memory stores, it can persist your data to disk, and is remarkably versatile due to a wide variety of data structures (Sets, Sorted Sets, Hashes, Lists, Strings, Bit Arrays, HyperLogLogs, Geospatial Indexes). Redis commands enable developers to perform highly performant operations on these structures with very little complexity. In other words, Redis is purpose-built for performance and simplicity.

Recent benchmarks have found Redis to be the highest performing NoSQL database with up to 8 times the throughput and up to 80% lower latency than other NoSQL databases. Redis has also been benchmarked at 1.5 million operations/second at sub-millisecond latencies while running on a single, modest cloud instance - the least amount of hardware compared to other NoSQL databases. Read about Redis benchmarks here and here.

This whitepaper provides an overview of some popular uses cases to help you get started with Redis.

What Does 'Data Structure Store' Mean?

Most NoSQL databases store data in one data store without requiring an SQL schema. For example, MongoDB stores most of its data in a document structure, Cassandra in a columnar data-store, Couchbase data is mostly key/value with some document data-store capabilities, and Memcached stores data in a simple string structure. Redis doesn't require an SQL schema either, but what makes Redis different is that you have so many data structures to chose from.

These data structures are highly optimized. Each provides specialized commands that help you execute complex functionality effortlessly and with very little code. They simplify intricate functionality, which would normally have taken dozens or hundreds of lines of code and would have consumed substantial network bandwidth to implement. These data structures make Redis extremely powerful and allow Redis-based applications to handle extreme volumes of operations at very low latency.

Redis data structures are:

- Strings
- Hashes
- Lists
- Sets
- Sorted sets
- Bit Arrays
- HyperLogLogs
- Geospatial Indexes

Redis data structures can be combined like "Lego" building blocks to form more complex data structures. And the Redis commands give them incredible power. Now let's talk about some popular use cases enabled by these data structures.

Popular Uses of Redis

Redis is used often for:

- User Session Data Management
- Real-time analytics such as counters/leaderboards/most viewed/highest–lowest ranks
- Recommendations such purchase or article recommendations based on common profile characteristics
- Message queues for workflow and other jobs
- Caching both static and interactive data

Redis is also used for high speed transactions, geospatial searches, time series data, distributed locking and many other use cases. We will talk about how the data structures enable several of these diverse uses – there is a link to additional resources at the end of this article.

Redis for User Session Management

Imagine your website is used by thousands or millions of users everyday. You need to quickly store and retrieve user and usage information such as first name, profile photo URL, pages visited, items looked at, time spent per page, and other details to personalize each user's experience. You'll need to store this data in multiple session variables and your web application will need to make this information available across multiple servers at low latency to meet the response times that your users expect.

The traditional way to store session information is in the app tier. But this falls short because it requires a session based load-balancer in-front of your web/app-tier in-order to forward the request to the right server. This approach is: 1) complex, requires special configuration of the load-balancer; 2) not efficient, the load-balancer actually stops balancing the load and only forwards requests to the web/app server that stores the session; 3) not reliable as once the web/app server fails the entire session information is lost.

Alternatively, if you keep all your session data in a shared and reliable datastore that is accessible from each web/app server and supports high throughput and low-latency operations, your application can perform safely under a high volume of users and operations.

Using disk based NoSQL databases for session data introduces unnecessary latency, while Memcached, using a simple key value data structure, is not able to provide the interactive possibilities that Redis data structures do. In addition, Memcached can be unreliable. You can lose your entire cache of user sessions if your server fails.

So Redis and specifically Redis' Hash data structure is your best choice. It lets you store all of your information in multiple fields, while giving you the flexibility to use other Redis data structures. And not only is it available in low latency memory, but you can have immediate failover in case your server goes down.

The Redis Hash data structure is a collection of key-value pairs – so you could have a single User ID or Username associated with multiple fields like page view, time on site, current page and more. HGET, HSET commands let you get and update individual values; HMGET/HMSET let you get or update multiple values. Individual fields can be incremented without reading the entire Hash with commands like HINCRBY.

Example 1: Using a Redis HASH to store user session information.

```
HMSET usersession:1 userid 8754 name dave ip 10.20.104.31 hits 1
OK
HMGET usersession:1 userid name ip hits
1) “8754”
2) “dave”
3) “10.20.104.31”
4) “1”
HINCRBY usersession:1 hits 1
(integer) 2
HSET usersession:1 lastpage “home”
(integer) 1
HGET usersession:1 lastpage
“home”
HDEL usersession:1 lastpage
(integer) 1
DEL usersession:1
(integer) 1
```

***Figure 1**: Redis-CLI console output*

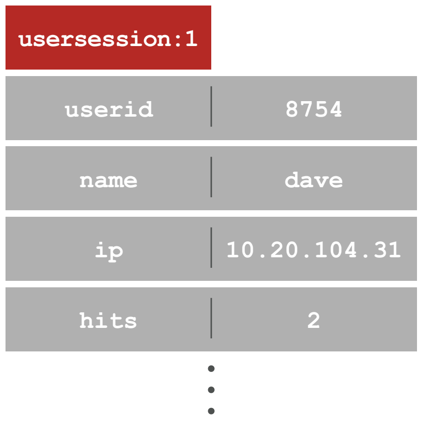

***Figure 2**: Representation of user session data stored in a Redis Hash data structure*

Another beauty of Redis in this scenario is that you can use the Redis EXPIRE command to delete user session data once your user sessions have expired. It sets a Time To Live (TTL) for your Redis Hash key and deletes it automatically after a time you specify, such as 20 minutes.

Redis for Real-time Analytics

Redis is exceptionally good for real-time analytic calculations – top scores, top ranked contributors, top posts, and many more. Imagine you have a distributed application like an auction, a multi-player game, or a real time user poll. Many users are responding or taking action in real time – increasing bids, collecting points or responding with their opinions. Your application needs to instantaneously track and show the top bid, the top score or the top opinion. Here's where Redis SORTED SET plays a simple, yet elegant role.

Redis Sorted Sets automatically keep lists of user scores updated and in order. You use Redis commands like ZADD to keep scores updated. ZRANGE and ZREVRANGE will get you the top or bottom values of the Sorted Set. ZRANK will pull the actual rank of a user in the pile as well. With Sorted Sets you don't have to worry about sorting efficiencies, sorting order and the other myriad of decisions you have to make with traditional databases.

Example 2: Using SORTED SETs for simple real time analytics. In this example, your sorted set key is game:1 and members are being added with scores with the ZADD command. The score for id:3 is being incremented by 44000 by the ZINCRBY command and finally ZREVRANGE is used to find the top scorer.

```
ZADD game:1 10000 id:A
(integer) 1
ZADD game:1 34000 id:B
(integer) 1
ZADD game:1 340 id:3
(integer) 1
ZADD game:1 35000 id:4
(integer) 1
ZINCRBY game:1 44000 id:3
(integer) 0
ZREVRANGE game:1 0 -1
id:3
```

***Figure 3**: Game:1 is automatically sorted by score and ZREVRANGE returns the highest scorer*

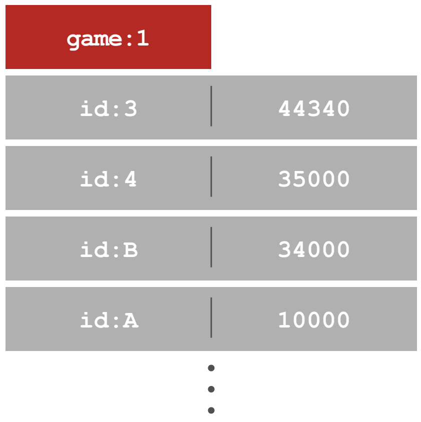

***Figure 4**: Representation of real-time scores stored in a Redis Sorted Set data structure*

Redis for User Recommendations

It’s quite common for retail or ecommerce sites to recommend items purchased by others who have purchased the same item. The same goes for recommending articles that have the same tags of the article you are reading. Once again, Redis can be used to find common purchases or articles with ease and simplicity.

Redis SETs are unordered collections of strings. But their real power comes from Redis Set commands. With most databases, comparing more than 2 tables or collections would create performance issues. With Redis, you can create multiple Sets with the SADD command, then use SINTER to find members of the intersection of all the given sets. You can also use SISMEMBER to determine if a particular value can be found in a Set and SMEMBERS or SSCAN (recommended for large sets) to return all the members of a Set.

In the article recommendation scenario above, you can easily identify articles to recommend using Redis Sets. To prepare your data, you use SADD to add each article to one or more Sets of tags. There might be hundreds of Sets representing hundreds of tags.

Once your Sets are populated, you are ready to calculate a recommendation. For example: If a user is viewing article:1 which has been tagged tag:1, tag:2, tag:3 and you want to recommend new articles. You would perform ‘SINTER tag:1 tag:2 tag:3’ to get a list of articles that have also been tagged with the same 3 tags.

Here is an example of the Redis functions you would use to add tags to articles and then find articles tagged tag:1, tag:2 and tag:3.

Example 3: Use SADD to add tags to each article as they are created, SMEMBERs to find all the tags and article has. If you want to find articles that have common tags, you would create a sorted set per tag and then use SINTER to find articles that have multiple common tags

```
SADD article:3 tag:3 tag:2 tag:1
(integer) 3
SMEMBERS article:3
1) “tag:1”
2) “tag:2”
3) “tag:3
SADD tag:1 article:3 article:1
(integer) 2
SADD tag:2 article:22 article:14 article:3
(integer) 3
SADD tag:3 article:9 article:3 article:2
(integer) 3
SISMEMBER article:3 tag:1
(integer) 1
SINTER tag:1 tag:2 tag:3
1) “article:3”
```

***Figure 5**: Console output for Example 3*

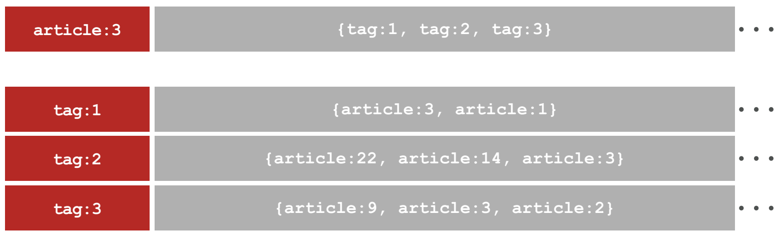
article:3
{tag:1, tag:2, tag:3}
tag:1
{article:3, article:1}
tag:2
{article:22, article:14, article:3}
tag:3
{article:9, article:3, article:2}

***Figure 6**: Representation of articles and tags stored in a Redis Set data structures*

Redis for Managing Queues of Work

In a recent user survey, we found using Redis for queues was a very common use case. Many popular job management systems such as resque, celery and sidekiq use Redis behind the scenes. A workflow, with tasks that need to be processed sequentially, can be managed easily with Redis Lists. For example, your app could send user bids to an auction system that also needs to send process intensive email notifications; or your app could accept an order for multiple products that also needs to process credit card transactions; or your app could send web application events to a remote monitoring system. You may find the most effective way to design your application is to leverage a queue to store these tasks for worker processes to work through them asynchronously.

Redis Lists are collections of strings ordered by order of insertion, or as specified. For example LPUSH will add a value to the 'left' end of a List, usually referred to as the beginning or top of the List. And RPUSH will add the value to the 'right' end of the list, usually referred to as the end or bottom of the List. RPOP will remove the last element of the list, while LPOP will remove the first element. RPOPLPUSH, as fun as it sounds, is an efficient way to 'pop' an element from the right end of one List and 'push' it to the left end of another List.

Example 4: In the below example, we use LPUSH to add "orange", "green" and "blue" tasks to the left side (i.e. the top or beginning) of queue:1. Then we use RPUSH to add the "red" task to the right side (i.e. the bottom or end) of the same queue. Then we use RPOPLPUSH to simultaneously remove the "red" task from the right side of queue:1 and push it to the left side of queue:2.

```
LPUSH queue:1 orange
(integer) 1
LPUSH queue:1 green
(integer) 2
LPUSH queue:1 blue
(integer) 3
RPUSH queue:1 red
(integer) 4
RPOPLPUSH queue:1 queue:2
```

__Figure 7__: Console output for List commands LPUSH, RPUSH and RPOPLPUSH

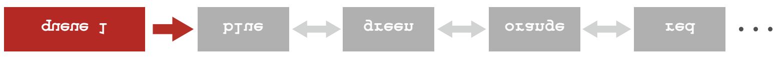

__Figure 8__: Representation of tasks added to queue:1 stored in a Redis Set before RPOPLPUSH

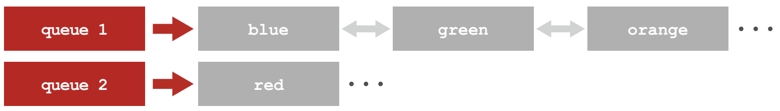

__Figure 9__: Representation of red task after it has moved to queue:2 using RPOPLPUSH

Redis for Caching

Caching is a well known Redis use case. You will find it among the first use cases when getting started with Redis. Caching is a common technique used to enable high performance systems to scale. The principle behind caching is simple: instead of keeping all of your data in one central database, you move a copy of the data you access frequently into Redis server to relieve the #1 bottleneck in your application: the connection to your database.

A caching tier is a distributed system design pattern that offers many advantages. Once you've separated the cache from the application, not only is your application more independent it can actually share cached data between servers and with other applications and (micro)services. Some popular caching tier use cases are:

- Caching static web pages and HTML fragments to reduce serving load from the web and/or application server
- Caching page assets (code snippets, media files) that are used in conjunction with CDNs and fast storage systems
- Caching User Session information, as previously mentioned, for virtually every type of app, SOA and microservices framework
- Object caching for user profiles and hot data (updated and read frequently)
- API calls and feature flags cache and throttling mechanisms
- Caching the results of database queries

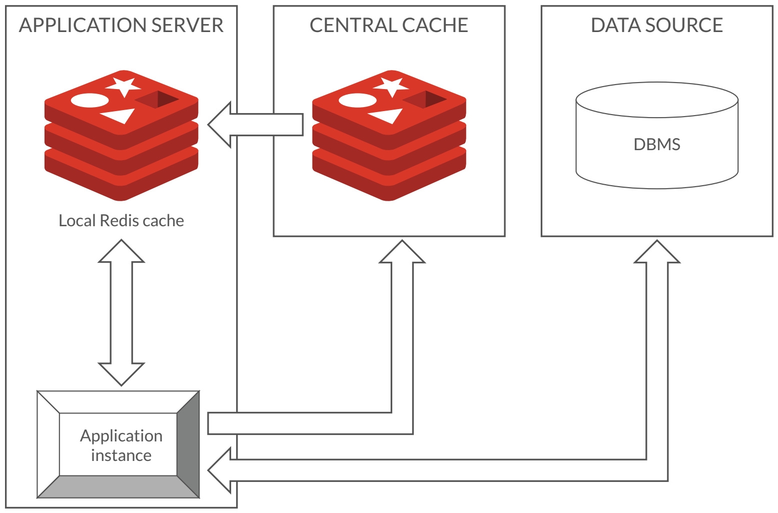

***Figure 10**: Redis can be setup on the same server as your application, or setup as a central cache. A central cache is typically used for high availability.*

What's next

Redis is popularly used as a database, a cache and a message broker. Data structures in Redis are similar to some data models in other NoSQL databases except they are implemented in memory with the simultaneous purposes of high performance, high memory space efficiency and low application network traffic. Data structures offer a matchless flexibility of use cases that make Redis an extremely popular solution for a variety of web scale application scenarios.

For more resources on how to use redis, watch some of our previously recorded webinars or read Dr.Josiah Carlson's popular Redis book, "Redis in Action" available at https://redislabs.com/resources/ebook/.

redislabs
HOME OF REDIS

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redislabs.com