redislabs
home of redis

whitepaper

A High Performance Recommendations Engine with Redis and Go

Table of Contents

Abstract

Next generation user facing applications are expected to include a built-in recommendations engine that tells the user what he or she's likely to "like," "purchase", "read" or "listen to" next. Redis, the popular open source, in-memory database known for its in-database analytics capability, and Go, an open source programming language that makes it easy to build reliable and efficient software, combine to deliver a simple, high performance recommendations engine that doesn't require many system resources. This paper outlines the algorithm and code necessary to implement a collaborative filtering approach to generating recommendations.

The supporting code can be found at https://github.com/RedisLabs/redis-recommend

What is a Recommendation Engine?

A recommendations engine is an application or micro-service that presents users with the choices they are most likely to make next. Recommendations could include the next music track a user is likely to want to hear, the next movie that they might watch or the next step they'll choose while making a reservation.

At a system level, recommendations engines match users with items they are most likely to be interested in. By pushing relevant, tailor-made items to users, application developers can encourage users to purchase relevant items, increase their time spent on a site or in the app, or click on the right ads – ultimately helping maximize revenues, usage or eyeballs.

Effective recommendation engines need to meet the following criteria:

1. Generate the right and relevant choices for their users (this usually depends on the algorithm chosen)
2. Provide high performance, with choices presented to users in real-time
3. Be efficient with system resources, as with any well-written application

Approaches to Building a Recommendations Engine

There are two basic approaches for building recommendation engines:

Content-based Classification:

This approach relies on classification by a large number of item and user attributes, assigning each user to possible classes of items.

- Pros:
  - Can be very targeted
  - Provides detailed control to the system owner
  - Does not require the user`s history.

- Cons:
  - Requires deep knowledge of items
  - Complicated data model
  - A lot of manual work to enter the items
  - Usually requires the user to enter a lot of details
- Best for: dating, restaurant recommendations, etc.

Collaborative Filtering:

This approach taps into user behavior and makes recommendations based on actions made by other users with similar behavior.

- Pros:
  - Very generic, content of the item is irrelevant
  - Can generate surprisingly interesting results
- Cons:
  - Requires a significant level of user history before recommendations are viable
  - Can be computationally heavy
- Best for: movie and music recommendations

Both options are easily implemented with Redis, but we choose the collaborative filtering approach because it is more popular and represents the modern way of implementing recommendation engines.

A Simple Redis Recommendation Engine Written in Go: redis-recommend

This project demonstrates how to build a recommendation engine with Redis, using code written in Go and the Redigo client library.

Redis is an open source (BSD licensed), in-memory database platform store, which can be used as a database, cache and message broker. Redis data structures are like “Lego” building blocks – they simplify the implementation of complex functionality, and are extremely efficient because data operations are performed in-memory, right next to where the data is stored, which conserves cpu and network resources.

We will demonstrate how some Redis data structures can tremendously reduce application complexity, while delivering very high performance at high scale. For this engine, we mainly use Redis sorted sets and the associated operations.

Why Redis for Recommendations:

If you look at the approaches above, both choices need set operations and sorting, and require that each be done very quickly. With Redis data structures like sorted sets, a solution is extremely easy to implement. Also, with Redis running extremely efficiently in-memory, you don’t have to worry about

performance under any load conditions. Compared to a disk-based or RDBMS solution, Redis provides orders of magnitude higher throughput at much lower latencies (<1ms), with very little hardware.

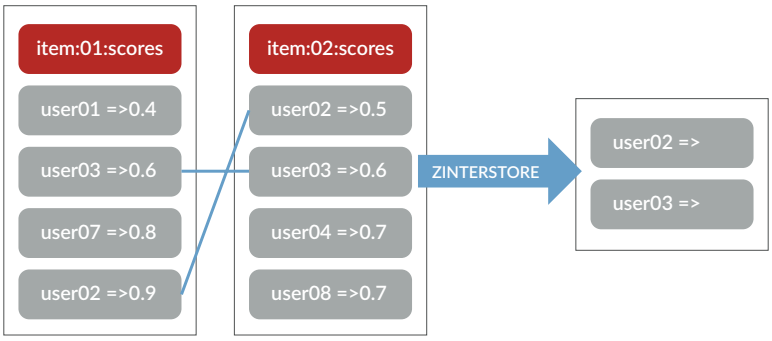
The Chosen Approach

With my collaborative filtering example, the algorithm is simple:

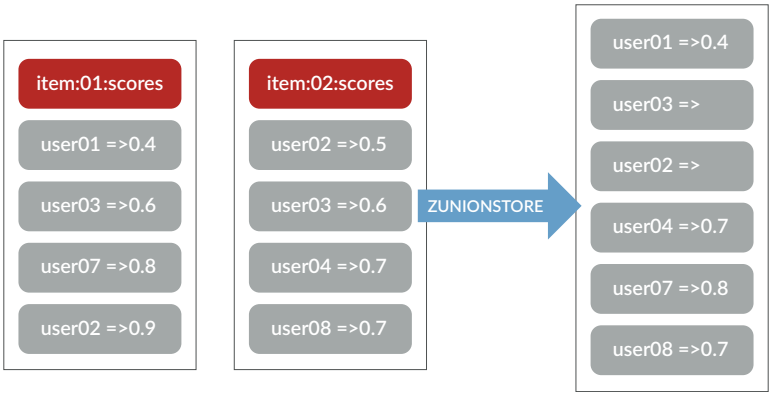
- For a given user, find the top similar users by:
  1. Find all users that rated at least one (or N) common items as the user, and use them as candidates
  2. For each candidate, calculate a score using the Root Mean Square (RMS) of the difference between their mutual item ratings
  3. Store the top similar users for each individual user
- Now find the top item recommendations by:
  1. Find all the items that were rated by the user's top similars, but *have not* yet been rated by the individual user
  2. Calculate the average rating for each item
  3. Store the top items

The Redis Implementation

The main Redis objects in use will be sorted sets. For example, intersect functionality will let us easily find users who rated the same items (ZINTERSTORE):



And if we want all the users who rated a group of items (ZUNIONSTORE):



Let's follow the logic step by step:

Step 1 -Insert rating events:

Each rating event of a user (U) for a giving item (I), will produce a score (R) to be stored in two sorted sets (the user's and the item's):

```
ZADD user:U:items R I

ZADD item:I:scores R U
```

Note: We stored the union in a temporary sorted set, "ztmp".

Now let's use ZRANGE to fetch:

```
ZRANGE user:U:items 0 -1
```

Step 2 - Get candidates for similarity:

In order to get the similarity candidates for user (U), we need the union of all the users that have mutually rated items with U. Let's assume U rated items I1, I2, I3:

```
ZUNIONSTORE ztmp 3 item:I1:scores item:I2:scores item:I3:scores
```

NOTE: We stored the union in a temporary sorted set, "ztmp".

Now let's use ZRANGE to fetch:

```
ZRANGE ztmp 0 -1
```

Step 3 - Calculate similarity for each candidate:

Now we need to calculate the similarity for each of the candidates. Assuming users U1 and U2, we want the RMS of all the differences in the ratings of the items rated by both users. Redis gives us ZINTERSTORE, so we can get the intersection between U1 and U2 items.

In order to calculate the rating difference, we can use weights. Multiplying U1's ratings by -1 and U2's ratings by 1 will give us:

```
ZINTERSTORE ztmp 2 user:U1:items user:U2:items WEIGHTS 1 -1
```

After calculating the RMS on the client side, the results will be stored in the sorted set user:U1:similars.

Step 4 - Getting the candidate items:

Now that we have a sorted set of users similar to U1, we can extract the items that the similar users rated. We'll do this with ZUNIONSTORE with all U1's similar users, but then we need to make sure we exclude all the items U1 has already rated.

We'll use weights again, this time with the AGGREGATE option and ZRANGEBYSCORE command. Multiplying U1's items by -1 and all the others by 1, and specifying the AGGREGATE MIN option will yield a sorted set that is easy to cut: All U1's item scores will be negative, while the other user's item scores will be positive. With ZRANGEBYSCORE, we can fetch the items with a score greater than 0, giving us just what we wanted.

Assuming U1 with similar users U3,U5,U6:

```
ZUNIONSTORE ztmp 4 user:U1:items user:U3:items user:U5:items
user:U6:items WEIGHTS -1 1 1 1 \ AGGREGATE MIN

ZRANGEBYSCORE ztmp 0 inf
```

Step 5 - Calculate score for each candidate item:

The last step is to calculate a score for each of the candidate items, which is the average rating given by U1's similar users.

To get all the ratings of an item (I) given by U1's similars, we intersect the two sets and take only the item scores by using WEIGHTS:

```
ZINTERSTORE ztmp 2 user:U1:similars item:I:scores WEIGHTS 0 1
```

The average score given by the similar users will be calculated on the client side. The results will then be stored in a sorted set named user:U1:suggestions.

Installation

Download and install Go and Redis.

Install Redigo:

```
go get github.com/garyburd/redigo/redis
```

Install redis-recommend:

```
go get github.com/RedisLabs/redis-recommend

cd $GOPATH/src/github.com/RedisLabs/redis-recommend/

go build
```

How to Use the Engine

Rate an item:

```
./redis-recommend rate <user> <item> <score>
```

Find (n) similar users for all users:

```
./redis-recommend batch-update [--results=<n>]
```

Get (n) suggested items for a user:

```
./redis-recommend suggest <user> [--results=<n>]
```

Get the probable score a user would give to an item:

```
./redis-recommend get-probability <user> <item>
```

The Code

The project contains two packages, *main* and *redrec*. *main* parses the input args, instantiates a *redrec* object and calls the relevant *redrec* functions. Package *redrec* implements the logic explained above using redigo as a Redis connector.

Example - the function **getSuggestCandidates**

getSuggestCandidates returns an array of strings containing the items rated by the input user similars:

```
func (rr *Redrec) getSuggestCandidates(user string, max int) ([]
string, error)
```

The user`s similars are fetched using ZRANGE:

```
similarUsers, err := redis.Strings(rr.rconn.Do("ZRANGE", fmt.
Sprintf("user:%s:similars", user), 0, max))
```

Then we can build the argument list for a ZUNIONSTORE command to store all the items that similar users rated, except the ones the input user already rated. To achieve this, we add the "WEIGHTS" option with a -1 multiplier to the input user, along with the "AGGREGATE MIN" option:

```
args := []interface{}{}

args = append(args, "ztmp", float64(max+1), fmt.
Sprintf("user:%s:items", user))

weights := []interface{}{}

weights = append(weights, "WEIGHTS", -1.0)

for _, simuser := range similarUsers {

    args = append(args, fmt.Sprintf("user:%s:items", simuser))

    weights = append(weights, 1.0)

}


args = append(args, weights...)

args = append(args, "AGGREGATE", "MIN")

_, err = rr.rconn.Do("ZUNIONSTORE", args...)
```

We then filter only the positive results with ZRANGEBYSCORE:

```
candidates, err := redis.Strings(rr.rconn.Do("ZRANGEBYSCORE",
"ztmp", 0, "inf"))
```

Finally we delete the temporary set and return the result:

```
    _, err = rr.rconn.Do("DEL", "ztmp")

    return candidates, nil
```

Conclusion and Next Steps

As you can see from the above, with Redis sorted sets, it becomes extremely easy to implement functionality for a recommendations engine. You can even generate similarities using location, demographics, time or other parameters. Redis makes it simple and extensible to do so, due to its variety of data structures.

Redis Labs, the home of open source Redis, provides Redis Cloud, a fully managed on-demand Redis service that runs with seamless scaling and high availability in the cloud of your choice, right next to your application. If your application is to be deployed on-premises, Redis Labs Enterprise Cluster (RLEC) provides the same effortless scaling, stable high performance and high availability with instant automatic failover in the environment of your choice.