# Recommendations Using Redis

## How to Develop a Simple Recommendations Engine Using Redis

BY **ROSHAN KUMAR**

CONTENTS

Recommendation systems need not always involve complex machine learning techniques. With enough data in hand, one can develop a recommendation system with little effort. One of the simplest recommendation systems is just a lookup table based on user interests. When you have data for many users and their behaviors, collaborative filtering is an easy solution to make recommendations. For example, for an e-commerce site with collaborative filtering, you can determine which users that purchased a sleeping bag have also purchased a flashlight, lantern, and bug repellent. Content-based recommendation systems go a step further and incorporate greater sophistication in predicting what a user is likely to want, based on that user's interactions. This article demonstrates how to develop simple recommendation systems in Redis based on user-indicated interests and collaborative filtering.

## WHAT IS REDIS?

Redis is an in-memory, NoSQL data store, frequently used as a database, cache, and message broker. Unlike other in-memory data stores, it can persist your data to a disk, and can accommodate a wide variety of data structures (Sets, Sorted Sets, Hashes, Lists, Strings, Bit Arrays, HyperLogLogs, and 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.

## DATA STRUCTURES IN REDIS

Data structures are like Lego building blocks; helping developers achieve specific functionality with the least amount of complexity, least overhead on the network, and lowest latency because operations are executed extremely efficiently in memory, right next to where the data is stored.

Data structures set Redis apart from other key/value stores and NoSQL databases in terms of versatility and flexibility. Redis data structures include:

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

## WHAT IS A RECOMMENDATION ENGINE?

A recommendation engine is an application or microservice that presents users with the choices they are most likely to make next. Recommendations could include the next song a user is likely to want to hear, the next movie that they might watch, or another action that they may choose to take after completing a reservation.

At a system level, recommendation engines match users with items in which they are most likely to be interested. By pushing relevant, personalized recommendations to users, applications 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.

At a system level, recommendation engines match users with items in which they are most likely to be interested. By pushing

*Continued on next page*

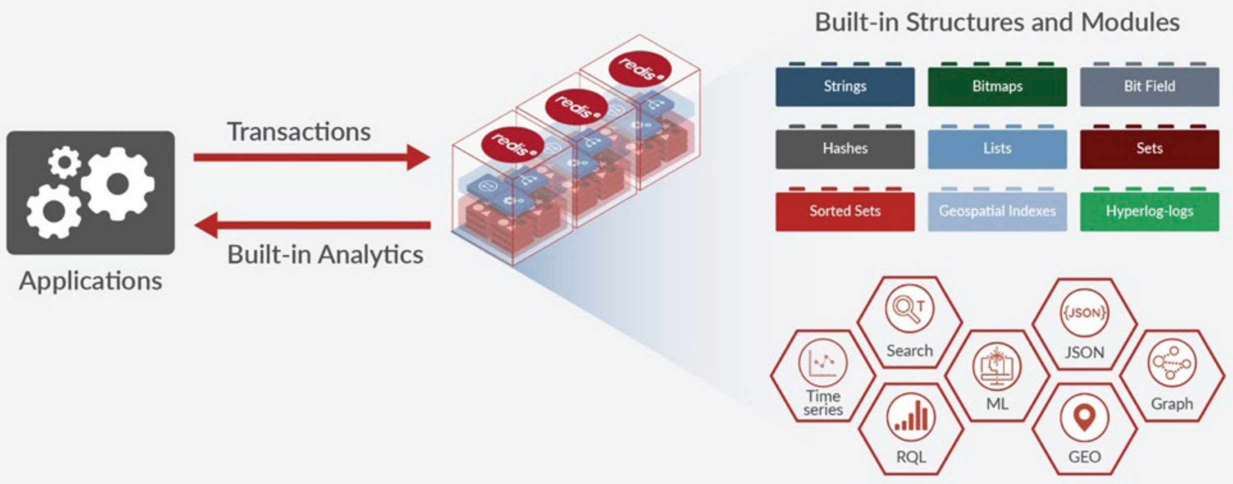
redislabs
home of redis

MOST LOVED DATABASE
BY DEVELOPERS. ENTERPRISE-READY

LEARN MORE ABOUT REAL-TIME ANALYTICS

Personalization | Machine Learning | Fraud Detection | Interactive Reporting

## Redis for Real-Time Analytics



Redis is the **#1**

- most loved database *Stack Overflow*
- database technology on AWS *Sunlogic*
- database on Docker *Datadog*
- in Growth among top 3 NoSQL databases *DB-Engineers.com*
- database used by Node.js developers *Node.js*
- NoSQL in user satisfaction *G2Crowd*

relevant, personalized recommendations to users, applications 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.

## COMMON TYPES OF RECOMMENDATION ENGINES

The most common recommendation engines are based on user-chosen profile settings, collaborative filtering, and content-based recommendations.

Recommendations based on user-chosen profile settings are the simplest to implement. However, they are static; they don't take user behavior into account or try to understand what's being recommended.

Collaborative filtering works well when you have many users and have collected enough information to understand and categorize your users based on their behavior. Collaborative filtering is quite effective and can generate surprisingly interesting results. On the flip side, they can be computationally heavy.

Content-based recommendations rely on Machine Learning and an understanding of the dimensions of user attributes and the attributes of the items that are being recommended. Preparing the right data model is often a tricky and avery lengthy process. However, with the right data model, content-based recommendations can produce great results with little historical data or only a few users in the system.

## REDIS DATA STRUCTURES AND COMMANDS FOR RECOMMENDATIONS

Redis data structures can tremendously reduce application complexity while delivering very high performance at high scale. Recommendation solutions often need Set operations such as intersection, union, and set difference to be executed very quickly. Redis data structures such as Strings, Sets, and Sorted Sets come in handy while implementing a recommendation solution. Also, being an in-memory database platform, Redis delivers very high performance with sub-millisecond latency with very little computational resources.

Before we begin setting up the recommendation system, let's get acquainted with some of the Redis data structures and their commands:

| DATA STRUCTURE | COMMAND | DESCRIPTION |
|---|---|---|
| Strings | `GET key` | Get the value of the key. |
| | `SET key value` | Set key to hold the string value. |
| Sets | `SADD key member [member …]` | Add one or more members to a set. |
| | `SCARD key` | Get the number of members in a set. |
| | `SDIFF key [key …]` | Subtract multiple sets. |
| | **`SDIFFSTORE`** `destination key [key …]` | Subtract multiple sets and store the resulting set in a key. |
| | **`SINTER`** `key [key …]` | Intersect multiple sets. |
| | **`SINTERSTORE`** `destination key [key …]` | Intersect multiple sets and store the resulting set in a key. |
| | **`SISMEMBER`** `key member` | Determine if a given value is a member of a set. |
| | **`SMEMBERS`** `key` | Get all the members of a set. |
| | **`SREM`** `key member [member …]` | Remove one or more members from a set. |
| | **`SUNION`** `key [key …]` | Add multiple sets. |
| | **`SUNIONSTORE`** `destination key [key …]` | Add multiple sets and store the resulting set in a key. |
| | **`SSCAN`** `key cursor` | Incrementally iterate Set elements. |
| Sorted Sets | **`ZADD`** `key score member [score member…]` | Add one or more members to a sorted set, or update its score if it already exists. |
| | **`ZCARD`** `key` | Get the number of members in a sorted set. |
| | **`ZCOUNT`** `key min max` | Count the members in a sorted set with scores within the given values. |
| | **`ZINCRBY`** `key increment member` | Increment the score of a member in a sorted set. |
| | **`ZINTERSTORE`** `destination numkeys key [key …]` | Intersect multiple sorted sets and store the resulting set in a key. |
| | **`ZRANGE`** `key start stop [WITHSCORES]` | Return a range of members in a sorted set by index. |
| | **`ZRANGEBYSCORE`** `key min max [WITHSCORES]` | Return a range of members in a sorted set by scores. |
| | **`ZRANK`** `key member` | Determine the index of a member in a sorted set. |
| | **`ZREM`** `key member [member …]` | Remove one or more members from a sorted set. |
| | **`ZREVRANGE`** `key start stop [WITHSCORES]` | Return a range of members in a sorted set by index with scores ordered from high to low. |
| | **`ZREVRANGEBYSCORE`** `key max min [WITHSCORES]` | Return a range of members in a sorted set by score, with scores ordered from high to low. |

*Continued on next page*

| DATA STRUCTURE | COMMAND | DESCRIPTION |
|---|---|---|
| | **ZREVRANK** key member | Determine the index of a member in a sorted set with scores ordered from high to low. |
| | **ZSCORE** key member | Get the score associated with the given member in a sorted set. |
| | **ZUNIONSTORE** destination numkeys key [key …] | Add multiple sorted sets and store the resulting set in a new key. |
| | **ZSCAN** key cursor | Incrementally iterate sorted sets, elements, and associated scores. |

## RECOMMENDATIONS BASED ON USER INTERESTS

This is a simple recommendation system based on interests that are identified by users. In this method, we let the users select the categories in which they are interested. We also classify the items by their categories. Then we match user interests to the items based on the categories.

The algorithm:

1. Find all the categories a user, U, is interested in. Let's call the set Categories$_{userU}$
2. Get all the items associated with Categories$_{userU}$

### STEP 1

Set the categories each user is interested in.

```
SADD user:<user id>:categories <category>
```

### STEP 2

Maintain a set for each category such that the set contains all the items in that category.

```
SADD category:<category>:items <item id>
```

### STEP 3

Get all the categories a user is interested in (assuming this is a small set. Use SSCAN for a large dataset).

```
SMEMBERS user:<user id>:categories
```

### STEP 4

Get all the items that belong to the categories in which the user is interested.

```
SUNION category:<category 1>:items category:<category
2>:items category:<category 3>:items …
```

For large data sets it's a good idea to use `SUNIONSTORE`.

### SAMPLE SCENARIO: The local grocery store's mobile app

The local grocery shop has just released a new mobile application where it allows its customers to select the product categories they are interested in. The backend of the application tracks all the products on sale for each category. When a customer walks into the store and opens the application, that customer will receive personalized, targeted coupons. The data structures are shown in this example:

```
categories = {organic, dairy,… }
category:organic:items = {milk, carrots, tomatoes, …}
category:dairy:items = {milk, butter, cheese, …}
user:U1:categories = {organic, dairy}
user:U2:categories = {dairy}
```

When user U1 opens her application, she will receive promotions related to the following items:

```
SUNION category:organic:items category:dairy:items
= {milk, carrots, tomatoes, butter, cheese, …}
```

## COLLABORATIVE FILTERING BASED ON USER-ITEM ASSOCIATIONS

In this approach, we tap into user behavior and make recommendations based on the actions made by other users with similar behavior.

The algorithm:

1. Find other users (U1, U2, U3…) who are associated with the same set of items as user U.
2. Get all the items associated with users U1, U2, U3…
3. Remove the items that are already associated with U, so that only the non-associated items are recommended to U.

### STEP 1

Maintain a set of all items associated with a user, e.g. all items items purchased via an e-commerce application.

```
SADD user:<user id>:items <item id>
```

### STEP 2

For each user-to-item association, maintain a reverse mapping of items to users.

```
SADD item:<item id>:users <count> <user id>
```

### STEP 3

Get all the items associated with the user (assuming this is a small set. Use SSCAN for a large dataset).

```
SMEMBERS user:<user id>:items
```

### STEP 4

Get all the users that belong to the categories in which the user is interested.

```
SUNIONSTORE user:<user id>:all_recommended user:<user id
1>:items user:<user id 2>:items user:<user id 3>:items …
```

The final set computed above will contain all the items associated with other users who have the same item associations.

### STEP 6

Get the list of items that aren't yet associated with the user, but are associated with other users with similar behavior.

```
SDIFF user:<user id>:all_recommended user:<user id>:items
```

### SAMPLE SCENARIO: The local grocery store's mobile app for recommendations

The concept of personalized promotions by the local grocery store becomes a grand success. The store then decides to up its level by promoting things based on user behavior. The store wants to tell the customers, "The customers who purchased X also purchased Y." The data structures for this example would look like:

```
userid:U1:items = {milk, bananas}
userid:U2:items = {milk, carrots, bananas}
userid:U3:items = {milk}
item:milk:users = {U1, U2, U3}
item:bananas:users = {U1, U2}
item:carrots:users = {U2}
```

What items should we recommend to U1?

```
SMEMBERS user:U1:items
= {milk, banana}
SUNION item:milk:users items:banana:users
= {U1, U2, U3}
SUNIONSTORE user:U1:all_recommended user:U1:items
user:U2:items user:U3:items
= {milk, bananas, carrots}
SDIFF user:U1:all_recommended user:U1:items
= {milk, bananas, carrots} - {milk, bananas}
= {carrots}
```

The grocery store will recommend carrots to U1.

## COLLABORATIVE FILTERING BASED ON USER-ITEM ASSOCIATIONS AND THEIR RATINGS

This approach not only computes the common behavior based on the same set of items associated with different users, but also on how each user rates those items. For a given user, U, this technique finds all the users who have rated the items similar to U. Then, it recommends items based on the items rated by the users that displayed similar behavior.

The Algorithm:

For a given user, find the top similar users by:

1. Find all users that rated at least one (or N) common item(s) 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.

Find the top recommended item:

1. Find all the items that were rated by users similar to the original user, but that have not yet been rated by the individual user.
2. Calculate the average rating for each item.
3. Store the top items.

### STEP 1 Insert rating events

Maintain a Sorted Set for each user to store all the items rated by that user.

```
ZADD user:<user id>:items <rating> <item id>
```

Have a Sorted Set for each item; track all the users who rated that item.

```
ZADD item:<item id>:scores <rating> <user id>
```

### STEP 2 Get candidates with the same item ratings

This is a two-step process. In the first step, fetch all the users who have rated the same items. Secondly, find how similar each user computed in the previous step is with respect to user U, whom we need to recommend.

1. Find items rated by <user id>

   ```
   ZRANGE user:<user id>:items 0 -1
   ```

2. Find users who have rated the same items

   ```
   ZUNIONSTORE user:<user id>:same_items 3
   item:I1:scores item:I2:scores item:I3:scores
   ```

This is a two-step process. In the first step, fetch all the users who have rated the same items. Secondly, find how similar each user computed in the previous step is with respect to user U, whom we need to recommend.

### STEP 3 Calculate similarity for each candidate

Find the difference between <user id> and others in the list. This example uses ZMEMBERS assuming a small dataset. Use ZSCAN when working with a large dataset.

```
ZRANGE user:<user id>:same_items 0 -1
ZINTERSTORE rms:<user id1>:<user id2> 2 user:<user
id1>:items user:<user id2>:items WEIGHTS 1 -1
```

The absolute value gives the root mean square between two users. After this step, implement your own logic to identify who is close enough to a given user based on the root mean square between the users.

### STEP 4 Getting the candidate items

Now that we have a sorted set of users similar to U1, we can extract the items associated with those users and their ratings. 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, returning only those items that U1 has not rated.

Assuming <user id 1> with similar users <user id 3>, <user id 5>, <user id 6>:

```
ZUNIONSTORE recommendations:<user id 1> 4 user:<user id
1>:items user:<user id 3>:items user:<user id 5>:items
user:<user id 6>:items WEIGHTS -1 1 1 1 AGGREGATE MIN
```

**SAMPLE SCENARIO: The local grocery store's mobile app for recommendations**

The grocery chain now decides to add yet another feature within its application. It allows the customers to rate the items on a scale from 1 to 5. The customers who purchase similar items and rate them in a similar fashion would be more closely related as the store starts promoting items based not just based on their purchasing behavior, but also on how they rate those items.

The data structures would look like:

```
userid:U1:items = {(milk, 4), (bananas, 5)}
userid:U2:items = {(milk, 3), (carrots, 4), (bananas, 5)}
userid:U3:items = {(milk, 5)}
item:milk:scores = {(U1, 4), (U2, 3), (U3, 5)}
item:bananas:scores = {(U1, 5), (U2, 5)}
item:carrots:scores = {(U2, 4)}

ZRANGE user:U1:items 0 -1
= {(milk, 4), (bananas, 5)}

ZUNIONSTORE user:U1:same_items 2 item:milk:scores
item:bananas:scores
user:U1:same_items = {(U1, 9), (U2, 8), (U3, 5)}

ZINTERSTORE rms:U1:U2 2 user:U1:items user:U2:items
WEIGHTS 1 -1
ZINTERSTORE rms:U1:U3 2 user:U1:items user:U3:items
WEIGHTS 1 -1
rms:U1:U2 = {(bananas, 0), (milk, 1)};
rms:U1:U3 = {(milk, -1)};
RMS of rms:U1:U2 = 0.7
RMS of rms:U1:U3 = 1
```

From the above calculation, we can conclude that U2 is closer to U1, than U3 is to U1. However, for our calculations, we will choose RMS values less than or equal to 1. Therefore, we will consider the ratings of both U2 and U3.

```
ZUNIONSTORE recommendations:U1 3 user:U1:items
user:U2:items user:U3:items WEIGHTS -1 1 1  AGGREGATE MIN
recommendations:U1 = {(bananas, -5), (milk, -4),
(carrots, 4)}
```

The item that has the highest score is recommended to U1. In our example, the store recommends carrots to U1.

## ADVANCED RECOMMENDATIONS

Collaborative filtering is a good technique when you have a large dataset pertaining to user behavior. Collaborative filtering is generic, and doesn't dig into the content of the item being recommended. This technique works fine when many users share common interests. Content-based recommendations, on the other hand, are tedious. They are most effective when incorporating predictive analytics and machine learning techniques. Redis-ML offers categorizing techniques using tree ensembles such as Random Forest.

The pseudo-code below illustrates how we can use the Redis-ML module for recommendations. The code assumes you have already generated a model on Apache Spark and loaded the model into Redis. Apache Spark provides you the necessary tools to create and train a Machine Learning (ML) module. When you load an Apache Spark ML model into Redis, Redis-ML automatically translates the Spark ML model into Redis data structures and makes it available for serving immediately.

The idea in the code is to:

1. Get the user profile from Redis.
2. Fetch the user's interest categories. We can allow the user to select the categories they are interested in, or compute the categories based on their purchase history, or both.
3. Retrieve all the items that belong to the interested categories.
4. For each item, calculate the score in the Random Forest classifier (RedisRandomForestClassfy).
5. Sort the items based on the rating, and recommend the highest item with the highest rating.

```
void setRecommendationsByInterests(String userid){
        // Sample data: “age:31”, “sex:male”,
        “food_1:pizza”, “food_2:sriracha”
        String[] featureVector = redis.call(“hget”,
        userid+”:profile”);


        Category[] userInterestCategories = redis.
        call(“smembers”,
        “interest_categories:”+userid);

        // For each category we have a machine learning
        model that will recommend
        // the most suitable items according to the users
        feature vector.
        // The models are trained on Spark and stored on
        Redis ML.
        for(category in userInterestCategories){
                // Get all items of this category
                String[] items = redis.call(“smembers”,
```

*Code continued on next page*

```
                    "item_to_categories:"+category);
                    // for each category get a score from the
                    random forest classifier
                    for(item in items){
                        category.itemScores[item] =
                        RedisRandomForestClassify(forestId =
                        "category:item:"+item, featureVector)

                    }

                    // Sort the classification results and get
                    the top
                    // results to render recommendations
                    results[category] = category.itemScores.
                    sort()[0:n_items]
                    // add recommended items for this user
                    under each category
                    redis.call("sadd","reco_items_by_
                        category:"
                        +category+":user:"+userid,
                        results[category]);
            }
    }
```

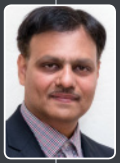
For more information about Redis-ML, visit redismodules.com/modules/redis-ml.

## OPTIMIZING RECOMMENDATIONS FOR REAL-TIME SERVING IN PRODUCTION

The Set and Sorted Set operations take time and resources, especially with a large dataset. For real-time recommendations, all we need is the final product: a Set or a Sorted Set of recommended items for each user. As a high-performance, low-latency, in-memory data store, Redis can usually perform all the computation required for recommendations. However, we recommend that you prepare the final recommended product in advance for each user in order to (1) deliver recommendations with sub-millisecond latency, and (2) make the solution resource-efficient. All the temporary Sets and Sorted Sets used for computations can be discarded once a final recommendation Set is generated for a user.

Should recommendations be created as a batch job, or as an on-going process while users update their profiles or activity? This really depends on numerous factors, such as how frequently users access an application, how frequently their behavior changes, the volume of transactions, and business goals. For example, if the solution designer is creating recommendations in a retirement planning application (one used infrequently by users), it may not matter if recommendations are updated in real-time. On the other hand, if the solution designer is creating recommendations for day traders, the recommendations need to best reflect market conditions to be useful. Solutions designers must study their data, the user behavior, the recommendation goals, etc. to choose the right level of responsiveness.

## ABOUT THE AUTHOR



**ROSHAN KUMAR** is a Senior Product Marketing Manager at Redis Labs, Inc. He has extensive experience in software development and marketing in the technology sector. In the past, Roshan has worked at Hewlett-Packard, and many successful Silicon Valley startups – ZillionTV, Salorix, Alopa, and ActiveVideo to name a few. As an enthusiastic programmer, he designed and developed mindzeal.com, an online platform hosting computer programming courses for young students. Roshan holds a Bachelor's degree in Computer Science, and MBA from Santa Clara University, California, USA.