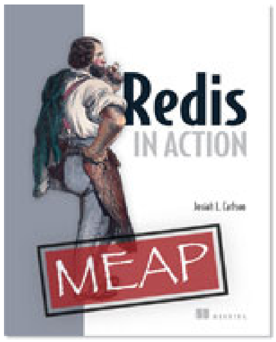
MANNING PUBLICATIONS



Redis application components offer usable code and solutions for solving specific problems. But each component contains techniques that can be used for other problems, and each specific solution can be applied to a variety of personal, public, or commercial projects. In this article based on chapter 6 of Redis in Action, you'll see how two different types of autocomplete quickly find users in short and long lists of items.

Autocomplete with Redis

In the web world, autocomplete is a method that allows you to quickly look up things that you want to find without searching. Generally, it works by taking the word that you have started to type and finding all words that start with that word.

Some autocomplete tools will even let you type the beginning of a phrase and finish the phrase for you. As an example, autocomplete in Google's search shows us that Betty White's SNL appearance is still popular, even years later (which is no surprise; she's a firecracker). It shows us the URLs we recently visited and want to revisit when we type in the address bar, and it helps us remember login names. All of these and other options are built to help us get to information faster. Some of them, like Google's search box, are backed by terabytes of remote information. Others, like our browser history and login boxes, are backed by much smaller local databases.

But, they all get us what we want with less work.

We are going to be building two different types of autocomplete in this article. The first uses lists to remember the most recent 100 contacts that a user has communicated with, trying to minimize memory use. Our second autocomplete offers better performance and scalability for larger lists but uses more memory per list. They differ in their structure use, the methods that are used on them, and the time it takes for the operations to complete. Let's first start with the recent contacts autocomplete.

Recent contacts autocomplete

The purpose of this autocomplete is to keep a list of the most recent users that each player has been in contact with. To increase the social aspect of the game and to allow people to quickly find and remember good players, FAKE GAME COMPANY is looking to create a contact list for their chat, which remembers the most recent 100 people that each user has chatted with. On the client side, when someone is trying to start a chat, they can start typing the name of the person they want to chat with, and it will show the list of users whose screen names start with the name they have partially typed. Figure 1 shows an example of this kind of autocompletion.

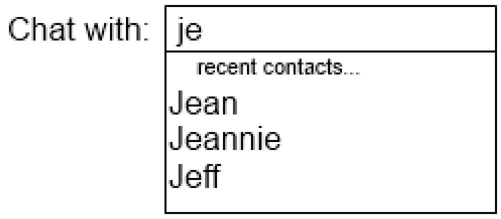


Figure 1 A recent contacts autocomplete showing users with names starting with "je"

Because each of the millions of users on the server will have their own list of their most recent 100 contacts, we need to try to minimize memory use, while still offering the ability to quickly add and remove users from the list. Because Redis LISTs keep the order of items consistent, and because LISTs use minimal memory compared to some other structures, we will use them to store our autocomplete lists. Unfortunately, LISTs don't offer enough functionality to actually perform the autocompletion inside Redis, so we are going to perform the actual autocomplete outside of Redis, but inside of Python. This lets us use Redis to store and update these lists in a minimal amount of memory, leaving the relatively easy filtering to Python.

Generally, there are three operations that need to be performed against Redis in order to deal with the recent contacts autocomplete lists. The first operation is to add or update a contact to make them the most recent user contacted. To perform this operation, we need to remove the contact from the list if it exists, add it to the beginning of the list, and trim the list if the list is now more than 100 items. We can do these operations with `LREM`, `LPUSH`, and `LTRIM`, in that order. To make sure that we don't have any race conditions, we are going to use a MULTI/EXEC transaction around our commands. We can see the complete function as listing 1.

Listing 1 The `add_update_contact()` function

```
def add_update_contact(conn, user, contact):
    ac_list = 'recent:' + user
    pipeline = conn.pipeline(True)                                  #1
    pipeline.lrem(ac_list, contact)                                 #2
    pipeline.lpush(ac_list, contact)                                #3
    pipeline.ltrim(ac_list, 0, 99)                                  #4
    pipeline.execute()                                              #5
```

#1 Set up the atomic operation
#2 Remove the contact from the list if it exists
#3 Push the item onto the front of the list
#4 Remove anything beyond the 100th item
#5 Actually execute everything

The second operation that we are going to perform is to remove a contact, if the user doesn't want to be able to find them anymore. This is a quick `LREM` call, which can be seen as follows.

```
def remove_contact(conn, user, contact):
conn.lrem('recent:' + user, 1, contact)
```

The final operation that we need to perform is to fetch the autocomplete list itself to find the matching users. Again, because we are going to be performing the actual autocomplete processing in Python, we are going to fetch the whole LIST, then process the list in Python. We can see this in listing 2.

Listing 2 The `fetch_autocomplete_list()` function

```
def fetch_autocomplete_list(conn, user, prefix):
    candidates = conn.lrange('recent:' + user, 0, -1)              #1
    matches = []
    for candidate in candidates:                                   #2
        if candidate.lower().startswith(prefix):                   #2
            matches.append(candidate)                              #3
    return matches                                                 #4
```

#1 Fetch the autocomplete list
#2 Check each candidate

#3 We found a match
#4 Return all of the matches

This autocomplete will work just fine for our specific example. It won't work as well if the lists grow significantly larger because removing an item takes time proportional to the length of the list. But, because we were concerned about space, and have explicitly limited our lists to 100 users, it will be fast enough. If you find yourself in need of much larger "most or least recently used list," you can use ZSETs with timestamps instead.

Address book autocomplete

In the earlier recent contacts autocomplete, Redis was used primarily to keep track of the contact list, not to actually perform the autocomplete. This is okay for short lists, but, for longer lists, fetching thousands or millions of items to find just a handful would be a waste. Instead, for autocomplete lists with many items, we must find matches inside Redis.

Going back to FAKE GAME COMPANY, the recent contacts chat autocomplete has become one of the most used social features of our game. Our number two feature, in-game mailing, has been gaining momentum. To keep the momentum going, we are going to be adding an autocomplete for mailing. But, in our game, we only allow users to send mail to other users that are in the same in-game social group as they are, which we call guilds. This helps to prevent abusive and unsolicited messages between users.

Guilds can grow to thousands of members, so we can't use our old LIST-based autocomplete method. But, because we only need one autocomplete list per guild, we can use a bit more space per member. And, to minimize the amount of data to be transferred to clients who are autocompleting, we are going to be performing the autocomplete prefix calculation inside Redis using ZSETs.

To store each autocomplete list is going to be a little different from other ZSET uses. Mostly, we are going to be using ZSETs for its ability to tell us whether an item is in the ZSET quickly, what position they are in, and to quickly pull ranges of items from anywhere inside the ZSET. What makes this use different is that all of our scores are going to be zero. By setting our scores to zero, we use a secondary feature of ZSETs: ZSETs sort by member names when scores are equal. When all scores are zero, all members are sorted based on the binary ordering of the strings. In order to actually perform the autocomplete, we are going to be inserting lowercased contact names. Conveniently enough, we also only ever allowed users to have letters in their names, so we don't need to worry about numbers or symbols.

What do we do? First, we calculate the range of strings that start with the prefix. As an example, for all alpha-numeric strings that start with `abc`, we are looking for strings that compare at least as large as `abc` but less than `abd`. To have an easy ending point for our range, we can concatenate a `{` (curly brace) character to the end of our prefix, which is the next character after `z` in ASCII. This gives us an ending point of `abc{`. Even though we have a convenient starting point `abc`, in order to perform our autocomplete, we need to be able to arbitrarily add and remove members to the ZSET without affecting other items, so we need to find the predecessor of `abc`. The simplest method is to just look at the last character and find its predecessor. In this case, it is `b`, so we can use `abb` and again add the `{` to the end, giving us `abb{`.

But, what if we want to find users with the prefix `aba`? If we try to find the predecessor of `a`, we end up having to borrow from the `b`. Rather than having to (in the worst case) implement base-26 subtraction by 1, the character immediately before `a` in ASCII is `` ` `` (a backquote), which we will use instead. Once again keeping it consistent and adding a `{` gets us ``ab`{`` as the start for `aba`. The function that performs all of this can be seen in listing 3.

Listing 3 The `find prefix range()` function

```
valid_characters = '`abcdefghijklmnopqrstuvwxyz{'                  #1

def find_prefix_range(prefix):
    posn = bisect.bisect_left(valid_characters, prefix[-1:])        #2
    suffix = valid_characters[(posn or 1) - 1]                      #3
    return start + suffix + '{', prefix + '{'                       #4
```

#1 Sets up our list of characters that we know about
#2 Finds the position of prefix character in our list of characters
#3 Finds the predecessor character
#4 Returns the range

Once we have the range of values that we are looking for, we need to insert our ending points into the ZSET, find the indices of those newly added items, pull some number of items between them (we'll fetch at most 10 to avoid overwhelming the user who's trying to find their message recipient), and then remove our added items. To ensure that we are not adding and removing the same items, which would be the case if two members of the same guild were trying to message the same user, we will also concatenate a randomly generated 128-bit UUID to our start and end points.

To make sure that the ZSET is not being changed when we try to find and fetch our ranges, we are going to use WATCH with MULTI and EXEC after we have inserted our endpoints. The full autocomplete function can be seen in listing 4.

Listing 4 The `autocomplete_on_prefix()` function

```
def autocomplete_on_prefix(conn, guild, prefix):
    start, end = find_prefix_range(prefix)
    identifier = str(uuid.uuid4())
    start += identifier
    end += identifier
    zset_name = 'members:' + guild

    conn.zadd(zset_name, start, 0, end, 0)                          #2
    pipeline = conn.pipeline(True)
    while 1:
        try:
            pipeline.watch(zset_name)
            sindex = pipeline.zrank(zset_name, start)               #3
            eindex = pipeline.zrank(zset_name, end)                 #3
            erange = min(sindex + 11, eindex - 1)                   #3
            pipeline.multi()
            pipeline.zrange(zset_name, sindex+1, erange)            #4
            pipeline.zrem(zset_name, start, end)                    #4
            items = pipeline.execute()[0]                           #4
            break
        except redis.exceptions.WatchError:                         #5
            continue                                                #5

    return [item for item in items if '{' not in item]              #6
```

#1 Find the start/end range for the prefix
#2 Add the start/end range items to the ZSET
#3 Find the ranks of our end points
#4 Get the values inside our range, and clean up
#5 Retry if someone modified our autocomplete zset
#6 Remove start/end entries if an autocomplete was in progress

To add and remove members from a guild is very straightforward, we only need to ZADD and ZREM the user from the guild's ZSET. We can see both of these functions in listing 5.

Listing 5 The `join_guild() and leave_guild()` functions

```
def join_guild(conn, guild, user):
    conn.zadd('members:' + guild, user, 0)
def leave_guild(conn, guild, user):
    conn.zrem('members:' + guild, user)
```

This method of adding items to a ZSET to create a range, fetching items in the range, and then removing those added items can be useful. Here we use it for autocomplete, but this technique can also be used for arbitrary sorted indexes.

Summary

In this article, we built two different types of autocomplete to quickly find users in short and long lists of items. The first one used lists to remember the most recent 100 contacts that a user has communicated with, trying to minimize memory use. Our second autocomplete offered better performance and scalability for larger lists but used more memory per list.