antirez weblog

Auto Complete with Redis

Monday, 13 September 10

Hello! This isn't just a blog post, it is actually the Redis weekly update number eight, but we can't tell this to Google: naming posts like *Redis Weekly update #...* will not play well with Google indexing, so I'm starting to use blog post titles more predictive of what people will actually type into a search engine, in this specific case **autocomplete redis**, probably.

Too bad that I'm forced to play the game of the SE0 L00z3r, but it's for a good reason. I guess this will also work well when articles are submitted in places like Hacker News, Programming Reddit and so forth. Readers will have a simpler way to tell if they want to actually click or not on the link. Ok... end of the off topic.

I hope the topic of this post will be very interesting for many Redis users, and for many non Redis users that experimented the pain of modeling auto complete for web or mobile applications using a database that is not designed to make this a simple task.

Auto complete is a cool and useful feature for users, but it can be non trivial to implement it with good performances when the set of completions is big.

Simple auto complete

In the simplest auto completion schema you have:

- A set of N words, that are in some way notable words that you want to auto complete. This can be for instance the names of the biggest cities in the world in a subscription form, or the most frequent search terms in a search service. We'll try to complete a list of female names. You can find the raw file name here: female-names.txt.
- A prefix string, that is, what your user is writing in the input field.

Given the prefix, you want to extract a few words that start with the specified prefix.

But in what order? In many applications (like the list of cities in a web form) this order can just be lexicographic, and this is more simple and memory efficient to model in Redis, so we'll start with it, trying to evolve our approach to reach more interesting results.

So for instance if the prefix is **mar**, what I want to see in the completion list is **mara**, **marabel**, **marcela**, and so forth.

Since we want a scalable schema, we also require that this entries are returned in O(log(N)) time in the average case. A O(N) approach is not viable for us. We'll see the space complexity, that is another very important meter, in a moment.

A little known feature of sorted sets

The order of elements into a sorted set is, of course, given by the score. However this is not the only ordering parameter since sorted set elements may have the same score. Elements with the same score are sorted lexicographically, as you can see in this example:

```
redis> zadd zset 0 foo
(integer) 1
redis> zadd zset 0 bar
(integer) 1
redis> zadd zset 0 x
(integer) 1
redis> zadd zset 0 z
(integer) 1
redis> zadd zset 0 a
(integer) 1
redis> zrange zset 0 -1
1. "a"
2. "bar"
3. "foo"
4. "x"
5. "z"
```

This is a very important feature of sorted sets, for instance it guarantees O(log(N)) operations even when large clusters of elements have the same score. And this will be the foundation for our code completion strategy.

Well... that's not enough, but fortunately there is another cool feature of sorted sets, that is the ZRANK command. With this command given an

element you can know the position (that is, the index) of an element in the sorted set. So what should I do if I want to know what words follow "bar" in my sorted set? (note that indexes are zero based)

```
redis> zrank zset bar
(integer) 1
redis> zrange zset 2 -1
1. "foo"
2. "x"
3. "z"
```

Hey that seems pretty close to what we want in our first completion strategy... but we need a final trick.

Completion in lexicographical ordering

So what we are able to do is looking up a word in the sorted set and obtain a list of words lexicographically following this word. This is a bit different from what we actually need, as we want to complete words just having a prefix of the word (or sentence, and in general any string).

We can have all this paying a price, that is: more space.

What we do is the following: instead of adding only full words to the sorted set, we add also all the possible prefixes of any given word. We also need a way to distinguish actual words from prefixes. So this is what we do:

- For every word, like "bar", we add all the prefixes: "b", "ba", "bar".
- For every word we finally add the word itself but with a trailing "*" character,

so that we can tell this is an actual word (of course you can use a different marker, even binary to avoid collisions). So we also add "bar*". If I add the word "foo", "bar", and "foobar" using the above rules, this is what I obtain:

```
redis> zrange zset 0 -1
1. "b"
2. "ba"
3. "bar"
4. "bar*"
5. "f"
6. "fo"
7. "foo"
8. "foo*"
```

```
9. "foob"
10. "fooba"
11. "foobar"
12. "foobar*"
```

Now we are very close to perform the completion!

If the user is writing "fo" we do the following:

```
redis> zrank zset fo
(integer) 5
redis> zrange zset 6 -1
1. "foo"
2. "foo*"
3. "foob"
4. "fooba"
5. "foobar"
6. "foobar*"
```

We just need to get all the items marked with the trailing "*', so we'll be able to show "foo" and "foobar" as possible completions.

Let's try it with a bigger dictionary

We'll use a list of 4960 female names, female-names.txt and a Ruby script to check how well our approach is working.

```ruby
# compl1.rb - Redis autocomplete example
# download female-names.txt from http://antirez.com/misc/female-names.txt

require 'rubygems'
require 'redis'

r = Redis.new

# Create the completion sorted set
if !r.exists(:compl)
    puts "Loading entries in the Redis DB\n"
    File.new('female-names.txt').each_line{|n|
        n.strip!
        (1..(n.length)).each{|l|
            prefix = n[0...l]
            r.zadd(:compl,0,prefix)
```

```
17          }
18          r.zadd(:compl,0,n+"*")
19      }
20  else
21      puts "NOT loading entries, there is already a 'compl' key\n"
22  end
23
24  # Complete the string "mar"
25
26  def complete(r,prefix,count)
27      results = []
28      rangelen = 50 # This is not random, try to get replies < MTU size
29      start = r.zrank(:compl,prefix)
30      return [] if !start
31      while results.length != count
32          range = r.zrange(:compl,start,start+rangelen-1)
33          start += rangelen
34          break if !range or range.length == 0
35          range.each {|entry|
36              minlen = [entry.length,prefix.length].min
37              if entry[0...minlen] != prefix[0...minlen]
38                  count = results.count
39                  break
40              end
41              if entry[-1..-1] == "*" and results.length != count
42                  results << entry[0...-1]
43              end
44          }
45      end
46      return results
47  end
48
49  complete(r,"marcell",50).each{|res|
50      puts res
51  }
```

compl1.rb hosted with ❤ by **GitHub** **view raw**

What we do in the script is adding all the words using our set of rules, then using ZRANGE in small ranges of 50 elements to get as much words as we like. This seems all cool, but what is the space and time complexity of our approach?

The big O

Both ZRANK and ZRANGE (with a fixed range of 50 elements) are O(log(N)) operations. So we really don't have problems about handling huge word lists composed of millions of elements.

Let's check the memory requirements: in the worst case, for every word, we need to add M+1 elements in the sorted set, where M is the length of the word. So for N words, we need N*(Ma+1) elements where Ma is the average length of our words. Fortunately in the practice there are a lot of collisions among prefixes, so this is going to be better. For the 4960 female names we needed 14798 elements in the sorted set. Still it's better to do your math using the N*(Ma+1) figure just to be safe. It seems like that the memory requirement is perfectly viable even with millions of words to complete.

Query prediciton

Our current schema completed things in lexicographically order. Sometimes this is what you want, like in words of an online dictionary. Other times what you want to do is, instead, much similar to what Google is doing in its *Instant Search*. Given a prefix you want to find the first N words having this prefix **ordered by frequency**, that is, if I'm typing "ne" I want to see things like "netflix", "news", "new york times", that is what people searched most in the latest weeks, out of all the strings starting by "ne".

This can't be done easily with a simple "range query" of the kind we did earlier. A given prefix can have sub prefixes with different "top strings" and we need to be able to update our data at runtime in a non blocking way. Our approach is using a different sorted set for every prefix.

We also need to create a schema where it is simple to build and update our sorted sets incrementally. The direct solution to this problem using Redis is the following:

- Every time an user enters a query, like "news", we compute all the prefixes: "n", "ne", "new", "news". We use every prefix as the key name for a sorted set, executing the a **ZINCRBY <prefix> 1 news** for each key.
- In order to complete a given prefix, we just need to perform a ZRANGE <prefix> 0 4, to show the first 5 items.

There is a problem with this approach: most completion systems will only show the top five or so items, but in order to compute what this top items are, we need to take a longer list. In theory we would need to take the *whole* list of all the words searched for every possible prefix...

Fortunately stream algorithms can help us. We know that statistically even taking just, for instance, 300 items per prefix, we'll be able to have a very good approximation of the top five items, since if a query is frequent enough, it will eventually win over the other queries. So this is what we actually need to do every time we receive a search for e given string. For every prefix of the search string:

- If our sorted set for this prefix is still not at the max number of elements (300 in the example), just do ZINCRBY, that is, add the element with score 1.
- Instead if the max number of items was already reached, remove the element with lower score, and add this new one with a score that is the lower score remaining, plus 1.

If you feel like this is too complex, you can use a variation of this algorithm that will have more or less the same performances, that is:

- Just ZINCRBY the current element, then sample three random elements and remove the one with lower score.

I guarantee you that this works well for completion ;) But actually this algorithms have different guarantees that are related to the distribution of the input. For instance if all the strings have very very close frequencies it

will have an hard time to output reliable data, but this is not the case, we know that the search is the kind of problem where a small subset of search strings accounts for a very big percentage of all the searches.

But if you are curious, make sure to take at look at this google search.

Clean up stage

Clearly because of the long tail nature of searches, we'll end with a lot of sorted sets related to prefix of words very rarely used. This sorted sets will have maybe just a few items, all with score 1. There is no interest for us in this data... and it's a good idea to clean up this keys, otherwise our approach will use too much (wasted) memory.

There are two ways to do this. If you are using Redis 2.0 the simplest thing to do is to use an associated key for every sorted set with the time of last update. Then using RANDOMKEY you can get a random element from time to time with a background worker, and check if it's better to delete it since it was updated a few days ago the last time.

If you are using 2.2-alpha (Redis master branch) you have a better option, that is, every time you update a sorted set just set a new EXPIRE. Redis will care to remove expired keys for you. Hint: in Redis 2.2 it is now possible to both write against expiring keys AND to update expire times without troubles.

I'm sure it's possible to do much better than what I described in this comments, but I hope this will get you started in the right direction if you plan to use Redis to implement completion. Please use the comments if you want to share your experience or ideas. Thanks for reading!

Comments

pablete writes: **1**
13 Sep 10, 12:53:07

Would it make sense to have Ternary Search Trees (http://www.cs.princeton.edu/~rs/strings/) implemented in redis as a basic data type like t_set.c t_hash.c etc ? TST combine the time efficiency of digital tries with the space efficiency of binary search trees.

Salman writes: **2**
13 Sep 10, 13:40:13

It would also be interesting to add some sort of dynamic ordering based on historical searches.

Pedro Melo writes: **3**
13 Sep 10, 14:07:36

I think your code for def complete is wrong. You can end up with words in results that lack the prefix you want.

Inside the range.each you also need to break and return if the prefix of the current element is no longer the same as the prefix argument.

teleo writes: **4**
13 Sep 10, 14:10:33

Hi Salvatore,

Thanks for the interesting post.

Most real-life cases require case-insensitive lookup, though (while preserving the original case of the results). In addition, sometimes a particular locale needs to be used.

How would you implement these cases in Redis? Also, is it possible to control the specific implementation of lexicographical sort used by Redis?

Thanks,
Te.

antirez writes: **5**
13 Sep 10, 14:21:30

@Pablete: the problem is not in the data structure, adding a new "COMPLETE" command would work without adding prefixes. But our goal is trying to solve a lot of problems with a reasonably small API!

@Pedro: thanks you, fixed.

@Salman: it's done in the second part of the article, using multiple sorted sets.

@teleo: we should use strcoll() or something like that, and a version that is case insensitive probably too, before releasing Redis 2.2 (this way you can control this

using environment vars). This is a known limitation indeed... also the same problem applies to SORT.

Thanks for the comments!

Dhruv writes: **6**
13 Sep 10, 17:32:11

Hello Antirez,
I didn't quite get how you can compute the top 'k' items by prefix and frequency as you have explaine above. Please could you explain it with an example or something more easier?

TobyM writes: **7**
20 Sep 10, 07:21:10

@antirez
I think the command to suggest a completion for a query would be ZREVRANGE <prefix>, 0, 4 since the zset is sorted from low to high. ZRANGE would give the _least_ frequent queries.

@Dhruv
Each prefix becomes a key pointing to a zset; each zset contains the fully query (and the score reflects how common that query is). To see the top 'k' items for a prefix 'p', just look up the 'k' highest ranked elements of the zset at the key 'p'.

Subhranath Chunder writes: **8**
26 Jan 11, 07:35:12

Already implemented a very basic version of it using Python and Django.

It was basically my first tryout on Redis, and it seems good.

---

lior writes: **9**
13 Feb 11, 22:18:57

Nice post! I think I will try it since I need rapid development and I have allot of auto completion scenarios
you can control the range of the set by adding another zrank command :

redis> zrank zset fo --> start Index
redis> zrank zset g --> end Index

Even though the lexicography sort is not perfect
you can get a range of results and then sort them in your local app machine by another logic

comments closed

PROGRAMMING AND WEB

Welcome, this blog is about programming, web, open source projects I develop, and rants I love to share from time

HOT ARTICLES

- Redis persistence demystified
- How to take advantage of Redis just adding it to your stack

NEWSLETTER

It's possible to receive new posts in your mailbox writing your email address and hitting the *subscribe* button:

to time. From the point of view of a programmer that loves to define himself a craftsman.

- Everything about Redis 2.4
- Auto Complete with Redis
- Redis Manifesto
- Redis 2.6 is near, and a few more updates
- Redis Sentinel beta released
- A different take on sexism in IT
- Redis for win32 and the Microsoft patch
- How my todo list works

» full listing

Subscribe

Delivered by FeedBurner

Send me a feedback: I wish to see an article about

send