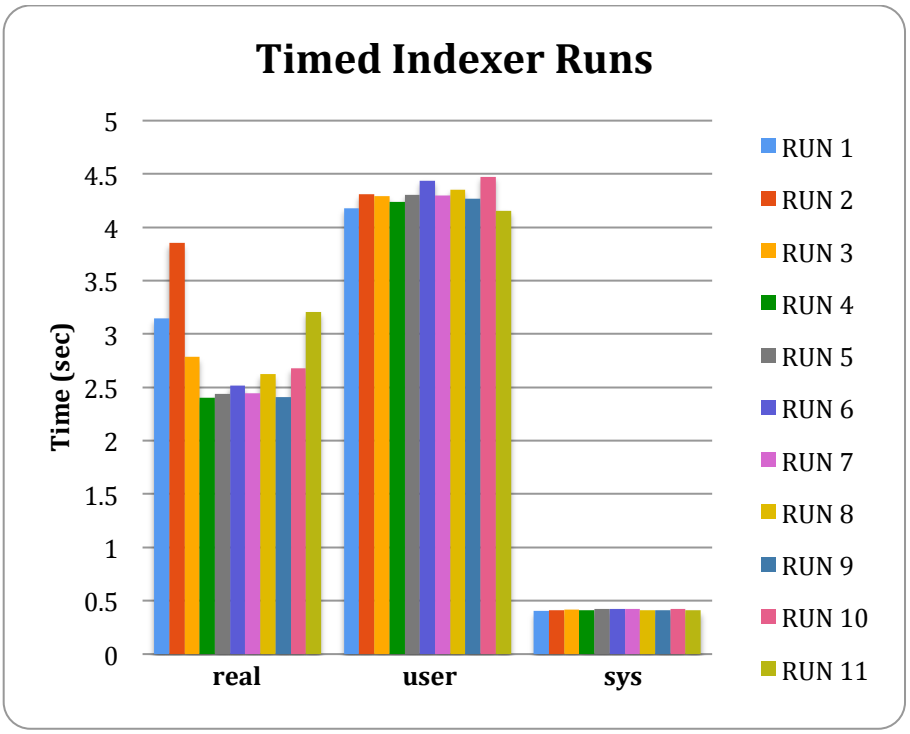
Scrabble-Indexer time(1) results:




The scrabble-indexer part of my program runs in an average of 2.77 real seconds, 4.30 user seconds, and .415 system seconds. That is, from a stopwatch perspective, it took an average 2.77 seconds. The .415 system seconds are time that the program spent in the kernel doing system calls and so forth. The fact that it took 4.3 user seconds shows that at least part of the program is multithreaded. All of the threads individually took less than the 2.77 real seconds to run, but cumulatively took up an average of 4.3 seconds of cpu time. Since I didn't write any multithreaded code, odds are that it was either the Sort that I used from Java Collections (which is more than likely), and/or some sort of compiler optimization voodoo that parallelized my existing code or the program setup/teardown.

I did some adhoc timing inside of my program, getting a "real start time" benchmark by assigning the value of System.currentTimeMillis() to a member variable during the constructor, and getting the difference between the current time and that member variable at several major junctures. I found that the parsing of the initial file and scoring of the words contained in that file by score was done in ~0.4 seconds after the constructor was initially called, sorting those words was done by ~ 0.6 seconds, and that the whole process was done ~1.25 seconds after the constructor was called. Given an overall (real) run time of ~2.77 seconds, this means that the program spent ~ 1.52 seconds prior to running any of the actual program logic I wrote.

Of the ~1.25 seconds spent after entering into the meat of the program:

- ~0.4 seconds were spent reading in words from the file and computing their scrabble scores;
- ~0.2 seconds were spent sorting the ~100k words gathered from the file,
- ~0.5 seconds were spent populating the contains-letter hash table, and
- ~0.15 seconds were spent deduplicating the contains-letter lists in the hash table and writing them to their respective index files.

I postulate that most of the multi-threaded behavior happened in the sorting stage, and was facilitated by the library calls. I say this, because the writing of the words in the list to various files happened in near-linear time, and the sort would be at best, O(nlogn). Given that, when n = ~100k, nlogn is ~11.5 times larger than n, we'd expect the sorting to take at least 11 times more than the write at the end. The sort actually takes around ~0.2 real seconds, which is considerably less than 11.5 times the ~0.15 the write-to-disk step took. I'd say that, since 11.5 times 0.15 is around 1.7, it's reasonable to say that the sorting should have taken ~1.7 seconds in user time.

Add that 1.7 seconds of user time to:

- the ~1.52 seconds of pre-constructor real runtime
- the ~0.4 seconds of reading in and scoring words
- the ~0.5 seconds of populating the contains-letter hash tables
- the ~0.15 seconds of deduplicating and writing words to their respective contains-letter index files.

And you get around ~4.27 seconds of user run time, which is within .03 seconds of the actual average user run time. There's a large margin for error in these calculations, but it does, in my estimation, establish that a great deal of threading must be going on in the sorting, even if the exact timing is off.

These timed tests are of the final version of my scrabble-indexer program. The first version deduplicated the words being written to the index files in O(m) time inside a for loop over n, making that part of the program run in quadratic time (m*n) instead of the current program's O(1) deduplication time (I was able to achieve this constant time deduplication, because the word lists were already sorted by score and alphabetical order, so any duplicate words would be next to each other when being written to the files). The previous version, instead of taking an average of 2.77 seconds to run like the current version, took approximately 2.5 MINUTES to run. So, I'm happy with my 2.77 seconds.

Suggester Analysis:

I put together a series of test cases for the scrabble-suggester:

**(0)** QUERY = "" K = 5, testing the program's BASE run time using invalid arguments (this test case only prints the USAGE message).
**(1)** QUERY = "et" K = 10, the provided example test case in the spec
**(2)** QUERY = "embezzlements" K = 10, testing the program's reaction to being asked for the top ten occurrences of a query that will only be found once
**(3)** QUERY= "an" K = 300, testing a common string and asking for a large (K) # of occurrences.
**(4)** QUERY = "zygl" K= 5, testing for the top 5 occurrences of a query that has no occurrences
**(5)** QUERY = "e" K=76170, testing for the worst: query consists of the most common letter, and K is equal to two more than the total number of words that contain that letter in the dictionary.

**Test (0)** is used to establish a base runtime--that is, how fast the program runs if no actual program logic is executed, and it only spits out a usage message before exiting.

**Test (1)** is used as a "normal" case, to establish how long a non-edge-case would take to run.

**Tests (2),** and to an extent **(5),** are used to test how the program runs when asked to find more occurrences of a string than actually exist.

**Test (3)** is used to test how the program behaves when given a moderate load, where K is indeed less than the total number of occurrences of QUERY.

**Test (4)** is used to test how the program behaves when given a QUERY that doesn't exist in the source word file at all.

**Test (5)** is used to test for the worst possible case available given the provided dictionary and the program structure.

Results from time(1):

**TEST O:**
*QUERY = "" K = 5*

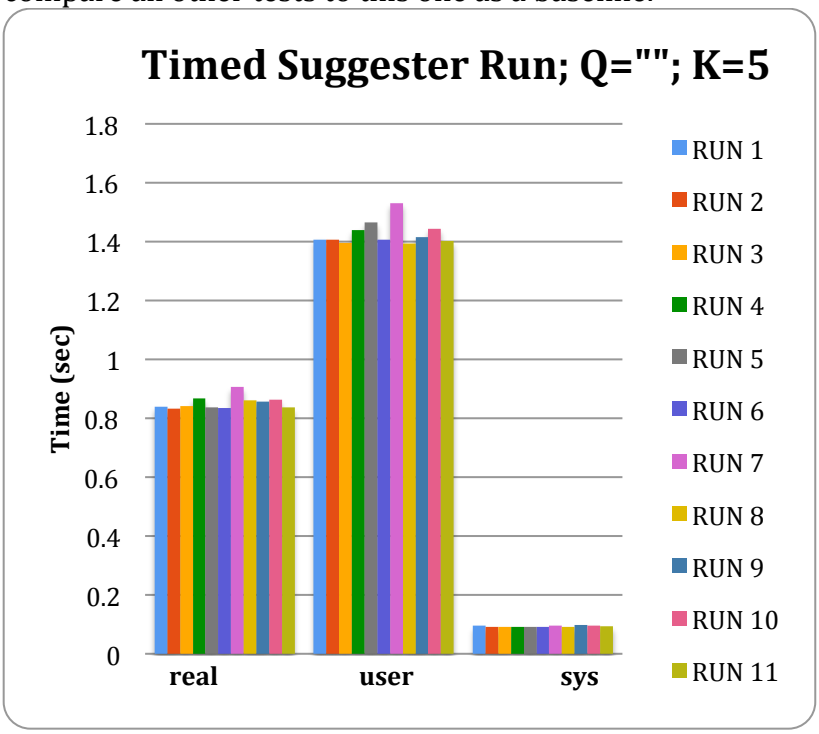
This test case causes the following output:

```
USAGE: ./scrabble-suggester <QueryStr> <K>
```

The program checks to see if the number of args is greater than 1, and since the empty string doesn't count as an argument, this test fails on usage grounds. This time trials shows, then, the minimum amount of time it takes for the program to start up and then near-immediately exit.

By the median values of all the runs, this test case runs for ~0.84 real seconds, ~1.40 user seconds, and ~0.093 system seconds. Since the user time is greater than the real time, we know some sort of parallelization is going on during program setup and teardown.

Regardless, this test gives us the lower-bound on program run time, and we can compare all other tests to this one as a baseline.

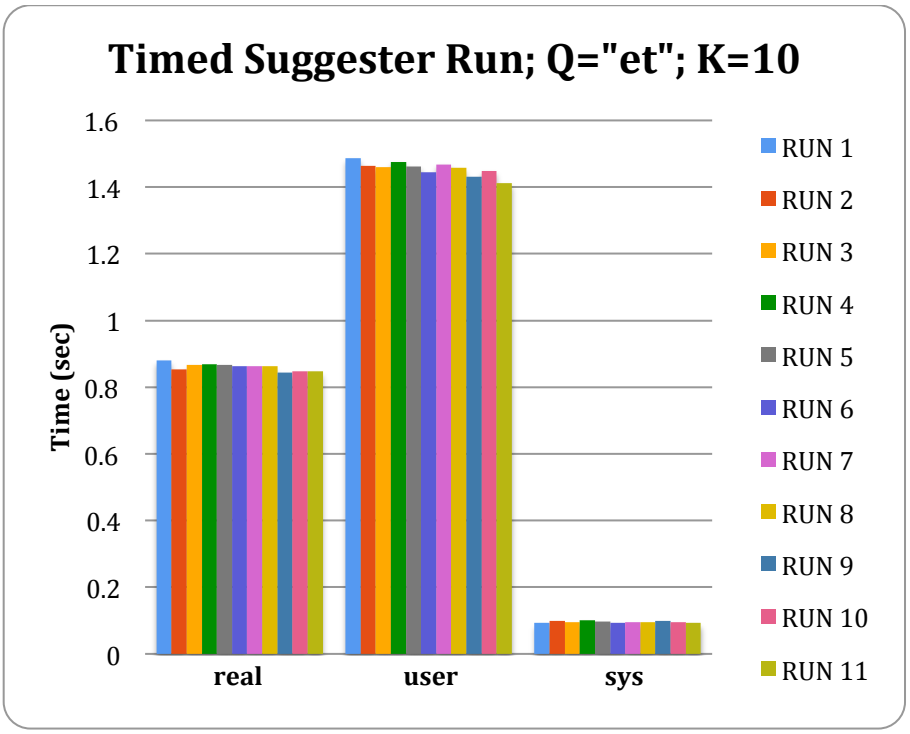


**TEST 1:**
*QUERY = "et" K = 10*

This was the given test case in the problem description. The output from running this is as follows:

```
Top 10 words containing et:
alphabetizing
hypothetically
sympathetically
alphabetized
alphabetizers
appetizingly
catheterization
mozzettas
alphabetizer
alphabetizes
```

The median run time for this case was ~0.86 real seconds, ~1.46 user seconds, and ~.095. In all three methods of measurement, the run time is only hundredths or even thousandths of a second slower than the "print the usage message" test case 0.

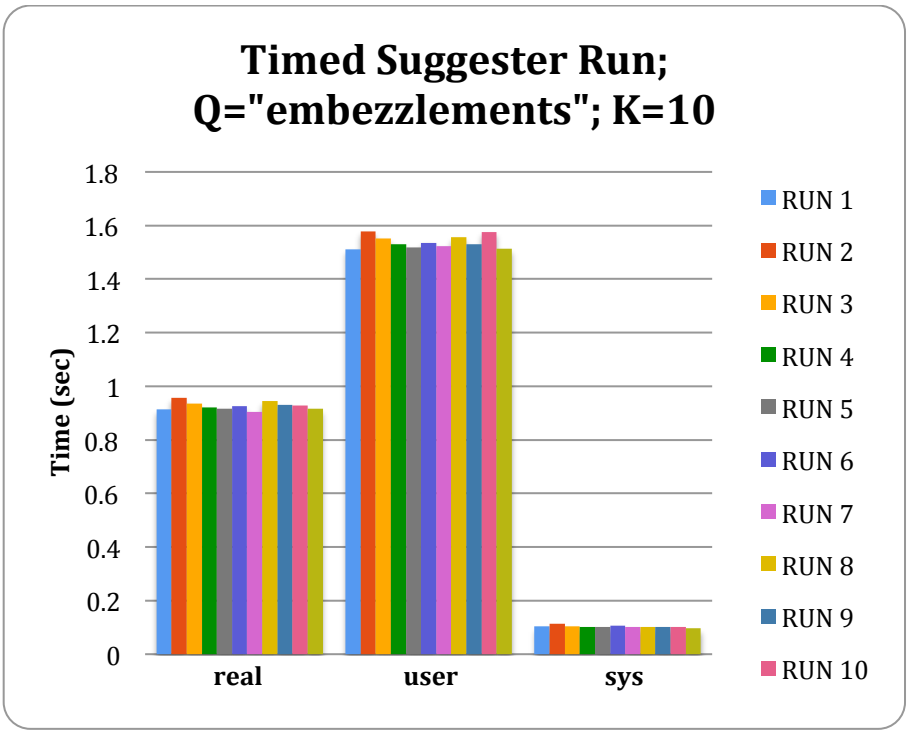


**TEST 2:**
*QUERY = "embezzlements" K = 10*

This query was chosen because it is only contained in one word in the given dictionary file: itself. The K value was chosen to be higher than the number of occurrences to check for bounds errors or issues. There are none, as seen by the output:

```
Top 1 words containing embezzlements:
embezzlements
```

This case takes has median runtime of ~0.92 user seconds, ~1.53 user seconds, ~0.10 system seconds. In this time, it goes through every word from the original dictionary that contains 'z', the least-common letter in "embezzlements".



**TEST 3:**
*QUERY = "an" K = 300*

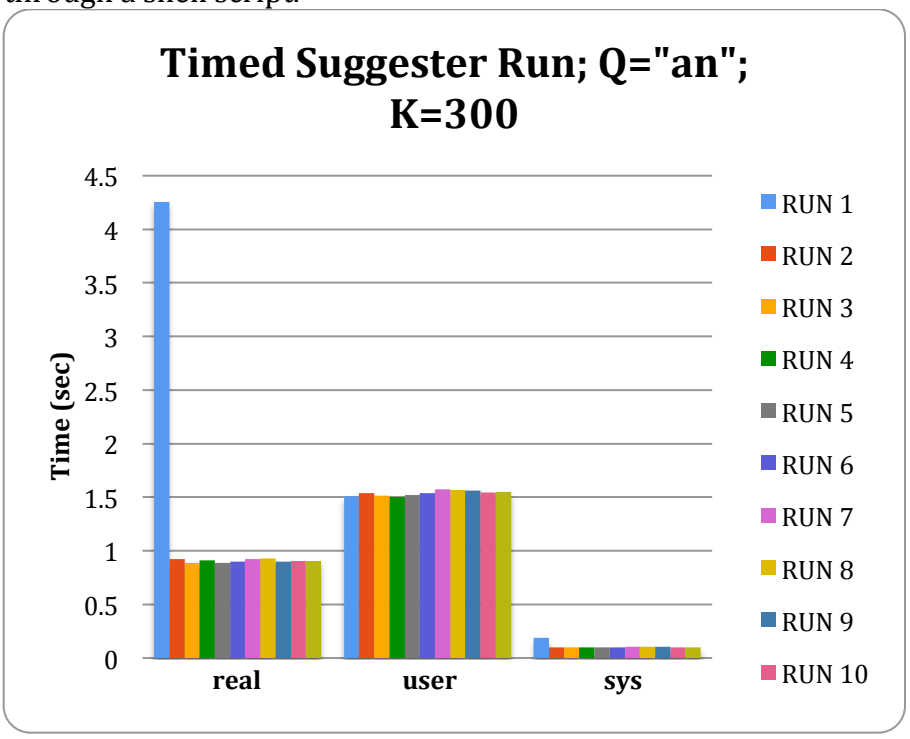
This was chosen because it's a heavy use case. It's heavy because 'a' and 'n' are the most common letters found in words from the original dictionary file, and because 300 is a relatively large number, and try as I might, I still like nice round numbers that are divisible by ten as opposed to numbers like 2^8 (256), even though I know that the latter makes you look smarter (and can provide performance boosts in some cases).

The output is lengthy, but upon spot checks, correct.

In this test case, the median runtime is ~0.91 real seconds, ~1.54 user seconds, and ~0.1 system seconds. The average runtime is ~1.21 real seconds, with the user and system runtimes being roughly equal to the medians. The standard deviation is ~1.01 real seconds, which is mostly caused by the 3.5 second disparity between the first test run and every other subsequent run.

As we will also observe in test case (5), larger K values have a peculiar effect on the first run of a test case in a series of runs in a shell script. Namely the first run has a user runtime that is significantly longer than any immediate subsequent run. Because of that, the median and average values of the test runs have larger disparities than in other test cases.

I believe that the reason the first run takes so long has to do with some sort of shell script setup, since the effect seems to disappear when not running a program through a shell script.

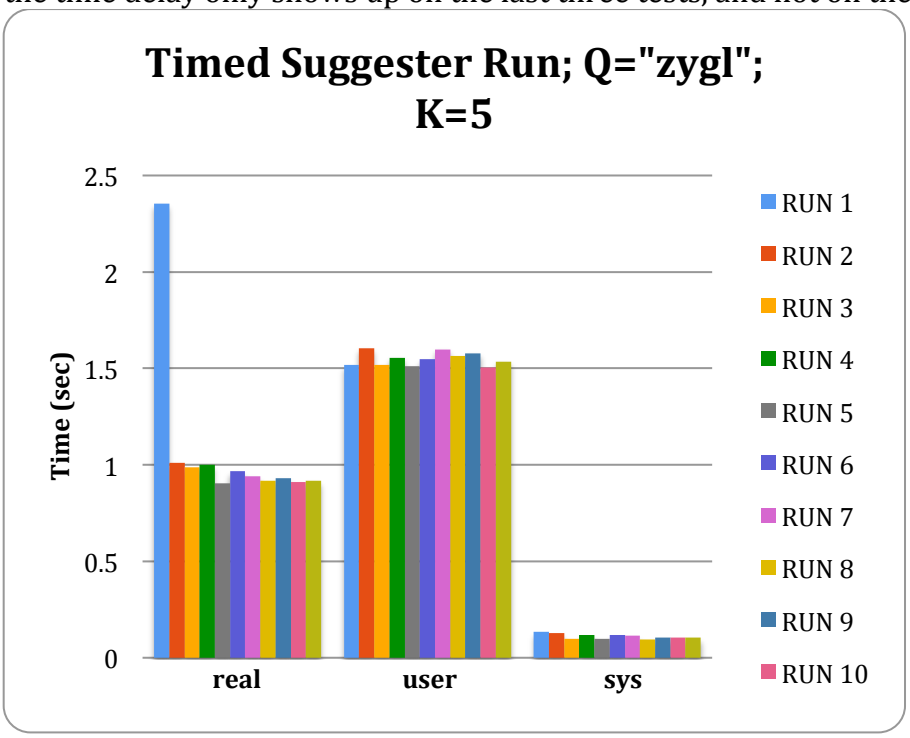


**TEST 4:**
*QUERY = "zygl" K = 5*

This test case tests for the top 5 occurrences of a query that has no occurrences. The output from running this is as follows:

```
Top 0 words containing zygl:
```

The median runtime for this is 0.94 real seconds, 1.55 user seconds, and .11 system seconds. The first run in this takes a while to run from the shell script, but this delay is an artifact of the testing method, and not of the program itself. We can see this, because the user and sys times should be greater than or equal to the real time, and in this case, there's obviously a third party contributing to the "real" time. Since the problem was to solve the suggester, and not to create a shell script testing it, I'm just disregarding the first run, though I have spent quite a while trying to figure out why the time delay only shows up on the last three tests, and not on the first four.



**TEST 5:**
*QUERY = "e" K = 76170*

This test case represents the "worst case" possible, given the provided input and spec. Since e is the letter most commonly found in the dictionary's words, the contains-letter index file for E is the largest file, with 76168 lines/words. By picking 'e' as our query, we guarantee that all lines in e's contains-letter index file will match the query. By picking a K that is two larger than the number of words in that file, we test for index-out-of-bounds type errors, and we make sure we go through the largest possible index file, line-by-line, until we hit the end.

The output is lengthy, but upon diffs with the contains-letter file for e, fully correct.

We observe, in this test run suite, that the first test run's real time was longer than any subsequent run. This discrepancy only appears when running the test commands immediately one after the other in a shell script. When running them one-by-one by hand, outside a shell script, no initial spike occurs. So, in all probability, it's just an artifact of running so many commands in a shell script. But since that's outside the scope of this project, I'm going to set it aside for now.

The median runtime of this test case is 1.75 real seconds, 2.64 user seconds, and .39 system seconds. This is around double our minimum run time benchmark from test 0, so it's not too shabby.