ECS 116 Assignment 2 Report

Nisha Thiagaraj, Connor Young, Cindy Chen

May 2024

1 Teammates

Nisha Thiagaraj, Connor Young, Cindy Chen

2 Statement about ChatGPT

I, Connor Young, use the GitHub Copilot extension in VS Code. I primarily use Copilot as a better auto-complete when writing my code. I find it most helpful when I want to explicitly write repetitive code instead of using a for loop when making lots of graphs for example. In this situation, Copilot provides me better results since there is more context for it to use when making suggestions. I do not rely on Copilot when writing specific functions because the suggestions are often not what I want or completely wrong. There are occasions when Copilot gives a suggestion that gives me new ideas for my code, but it is very rare that I do not change the code Copilot suggests. Overall, I've found that GitHub Copilot helps to speed up my coding, but it has not changed the way I code nor do I rely on it.

I, Nisha Thiagaraj, used ChatGPT occasionally to help with some of my errors when I got them while compiling the functions to create my code for step 3a. ChatGPT helped point out errors and gave me ways to rewrite the code so that I did not get those errors anymore. Sometimes, ChatGPT was not very helpful and my errors would not get solved, so I would have to figure it out myself. Overall, I was able to successfully run all my code after using ChatGPT as an occasion assistant. I did not use any other LLMs.

I, Cindy Chen, did not use any LLMs for my code.

3 Statement about distributed work

We each worked on part 1 and 2 of the assignment independently, with some collaboration for specific functions in the util.py file. When we reached step 3, we divided the work for each subpart. Nisha worked on 3a, creating the code for the `listings_join_reviews` json file. Cindy worked on 3b, which was creating the `text_search_query` json file. Finally, Connor worked on 3c, which was creating the `update_datetimes_query` json file. All the associated util functions created for these subparts were also written by the team member in charge of that section. For part 4, which is creating the plots, we utilized some of the TA's discussion code, and Connor took the lead on writing code for the plots which was later utilized by the full team. Finally, for part 5 and the report, all team members contributed. Specifically, Nisha question 1, Cindy did question 2, and Connor did questions 3 and 4.

4 Visualizations

4.1 Nisha

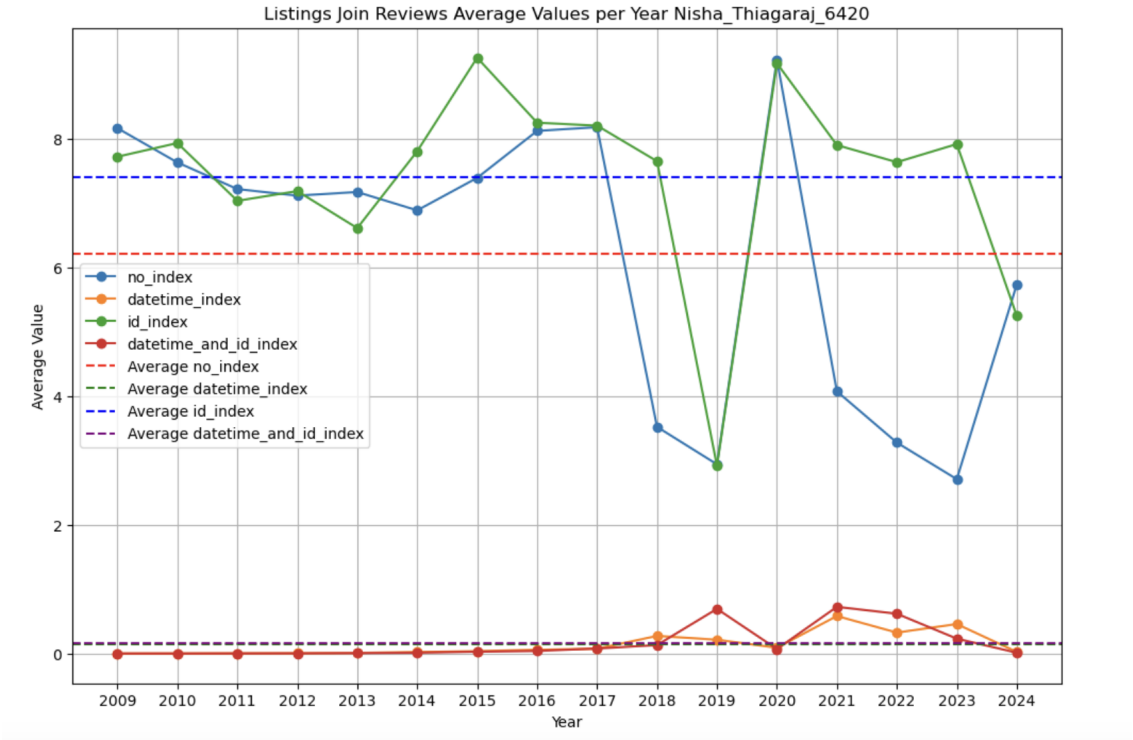

Figure 1: Listings Join Reviews Average Values per Year.

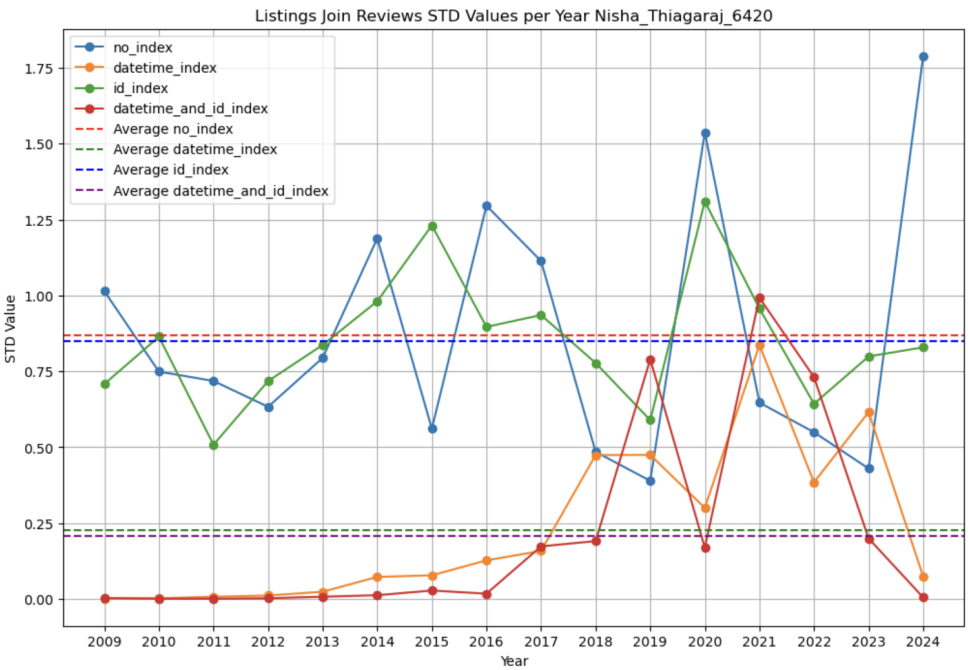

Figure 2: Listings Join Reviews STD Values per Year.

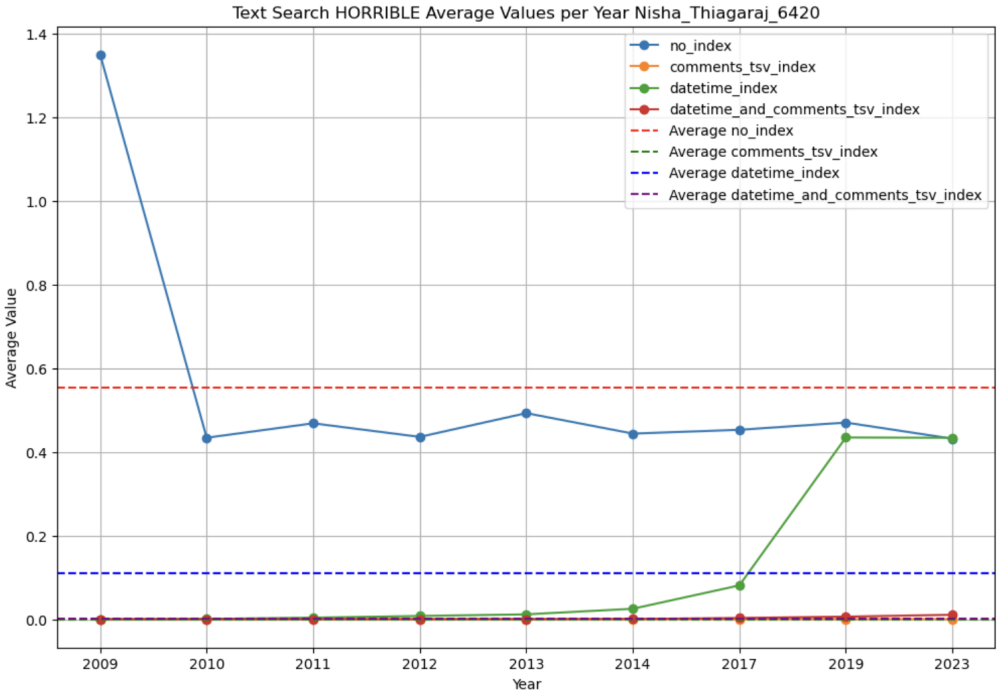

Figure 3: Text Search HORRIBLE Average Values per Year.

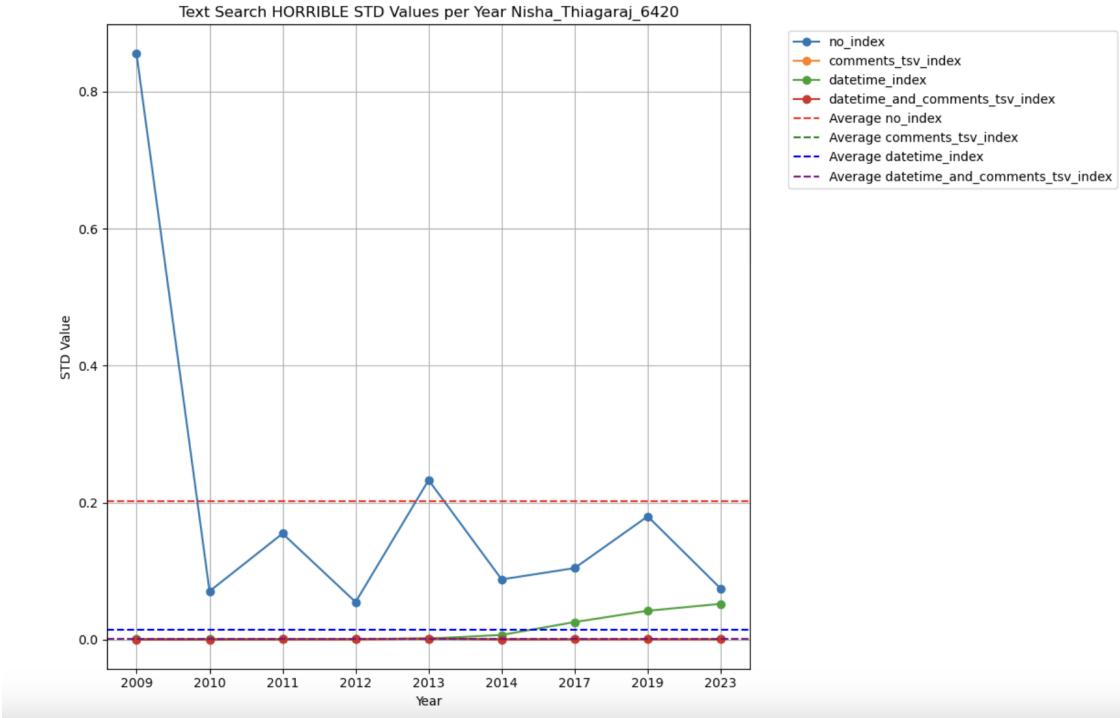

Figure 4: Text Search HORRIBLE STD Values per Year.

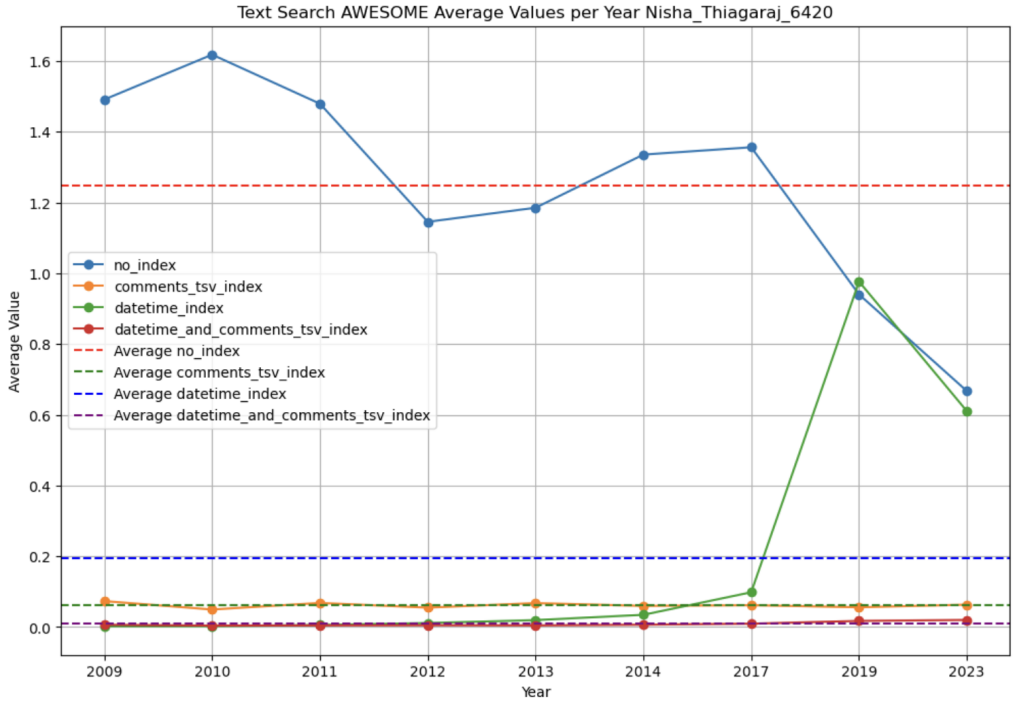

Figure 5: Text Search AWESOME Average Values per Year.

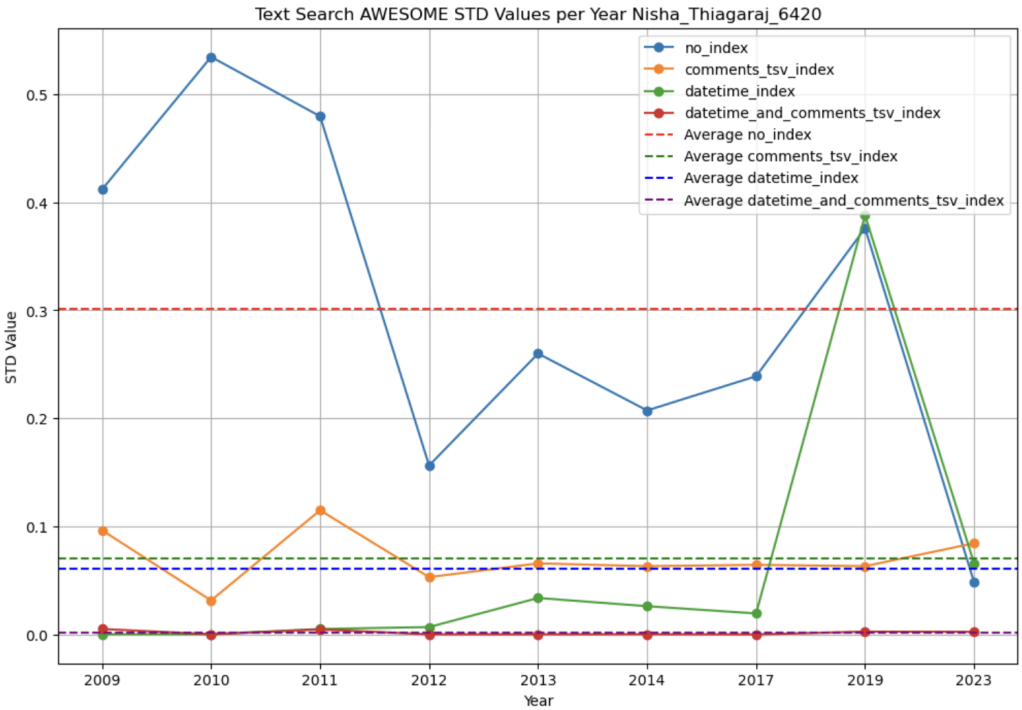

Figure 6: Text Search AWESOME STD Values per Year.

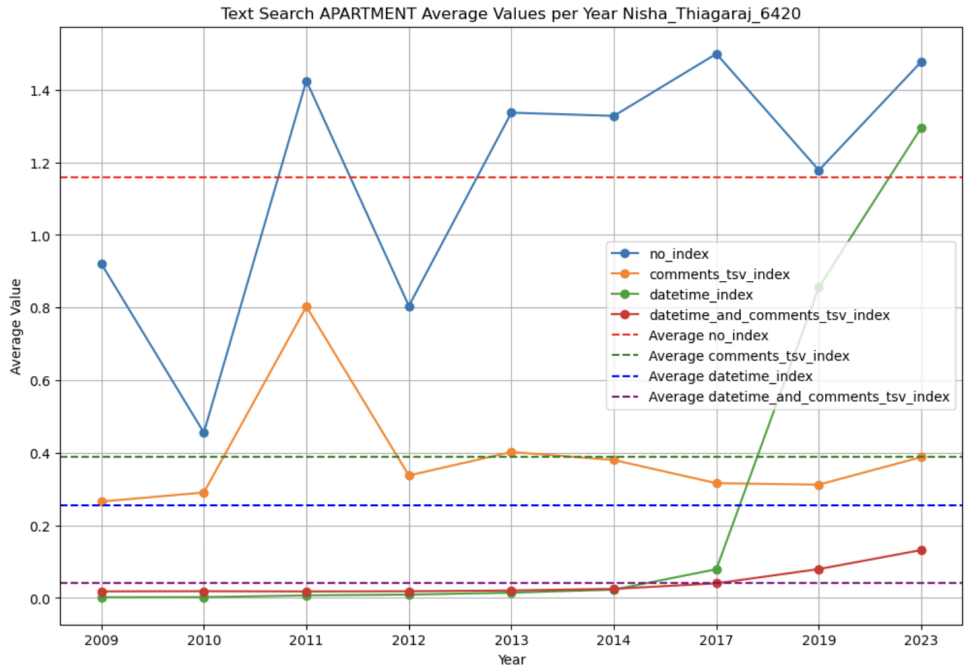

Figure 7: Text Search APARTMENT Average Values per Year.

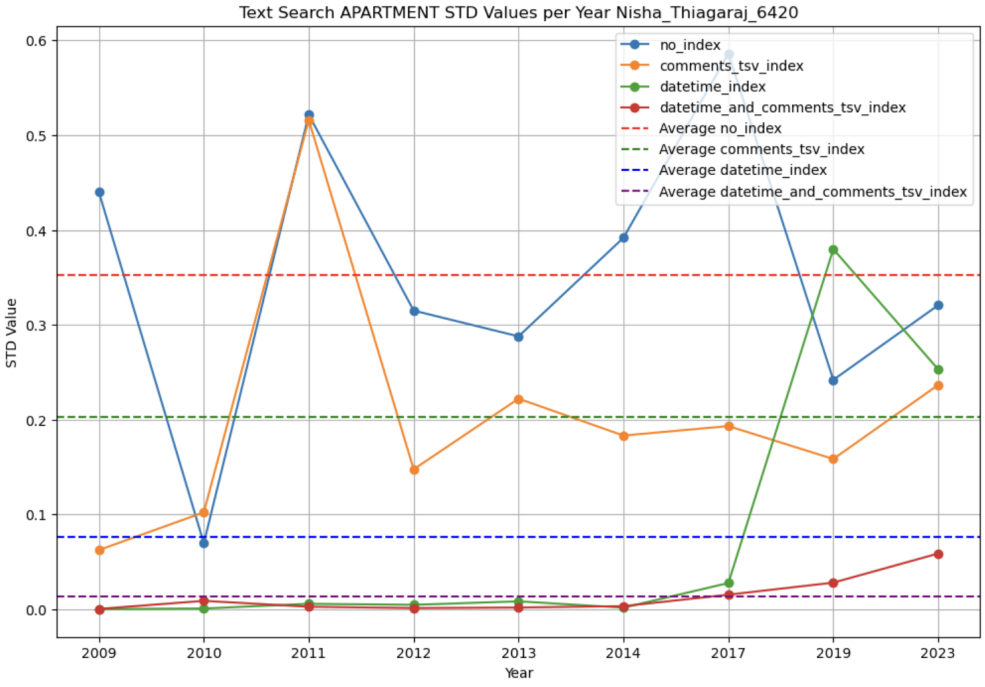

Figure 8: Text Search APARTMENT STD Values per Year.

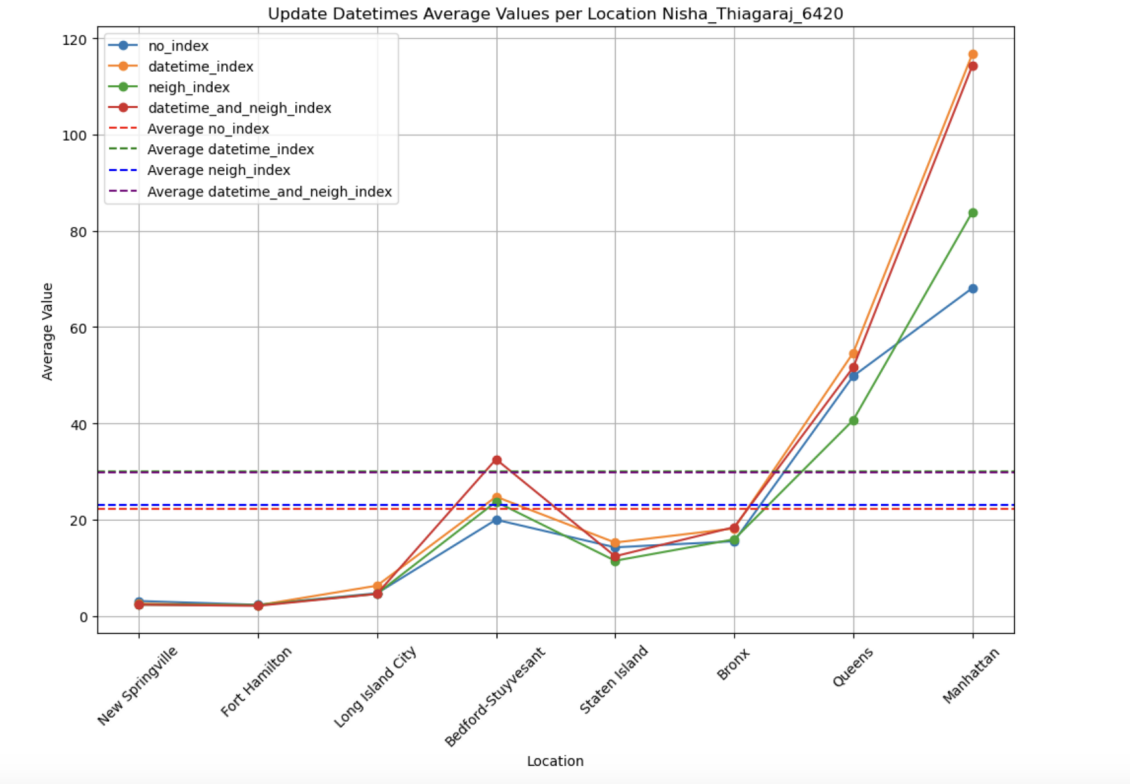

Figure 9: Update Datetimes Query Average Values per Location.

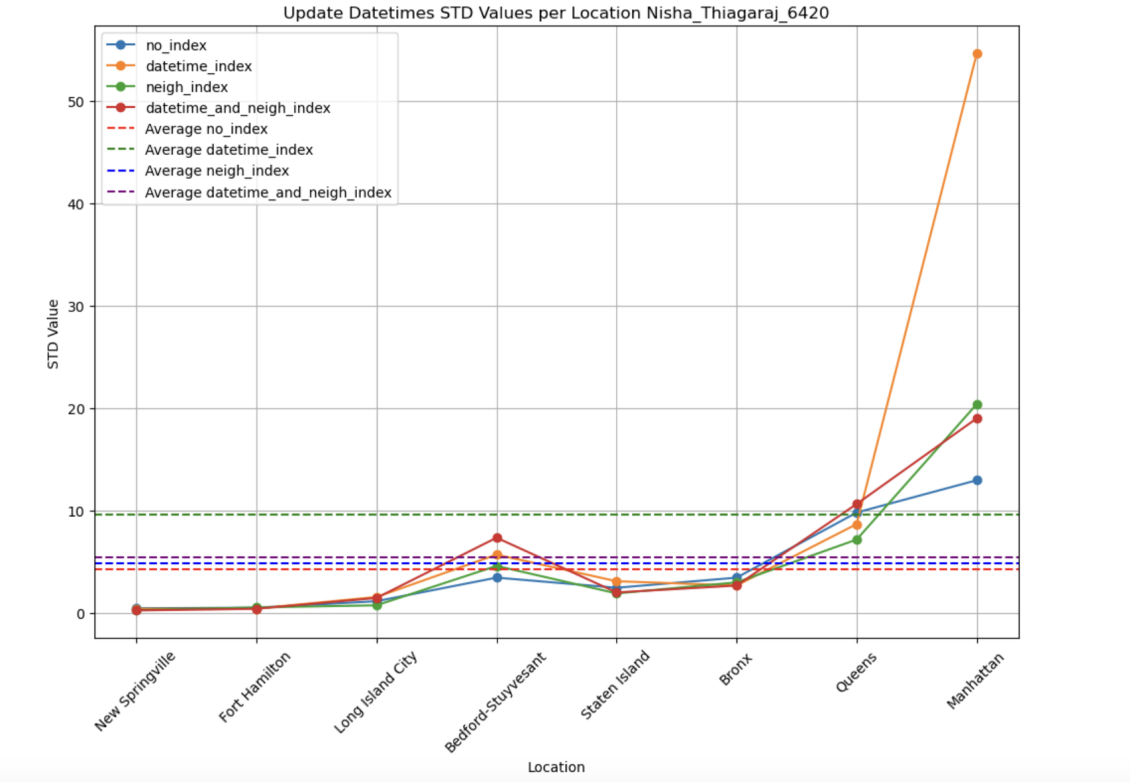

Figure 10: Update Datetimes Query STD Values per Location.

4.2 Connor

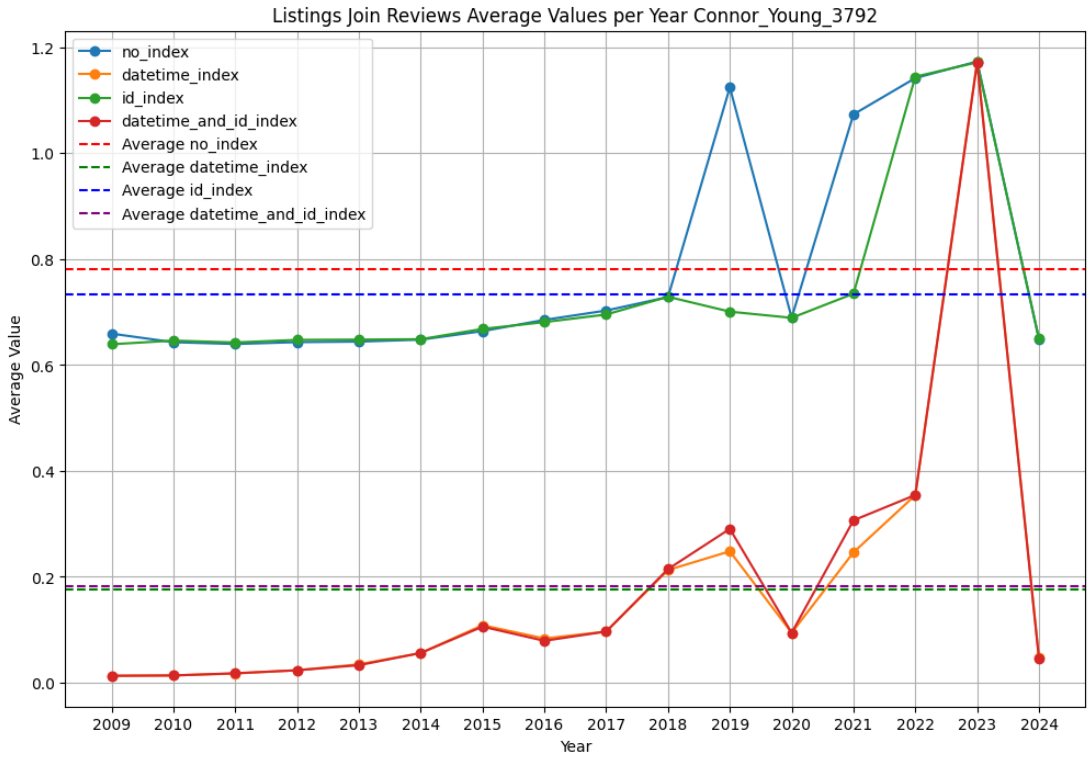

Figure 11: Listings Join Reviews Average Values per Year.

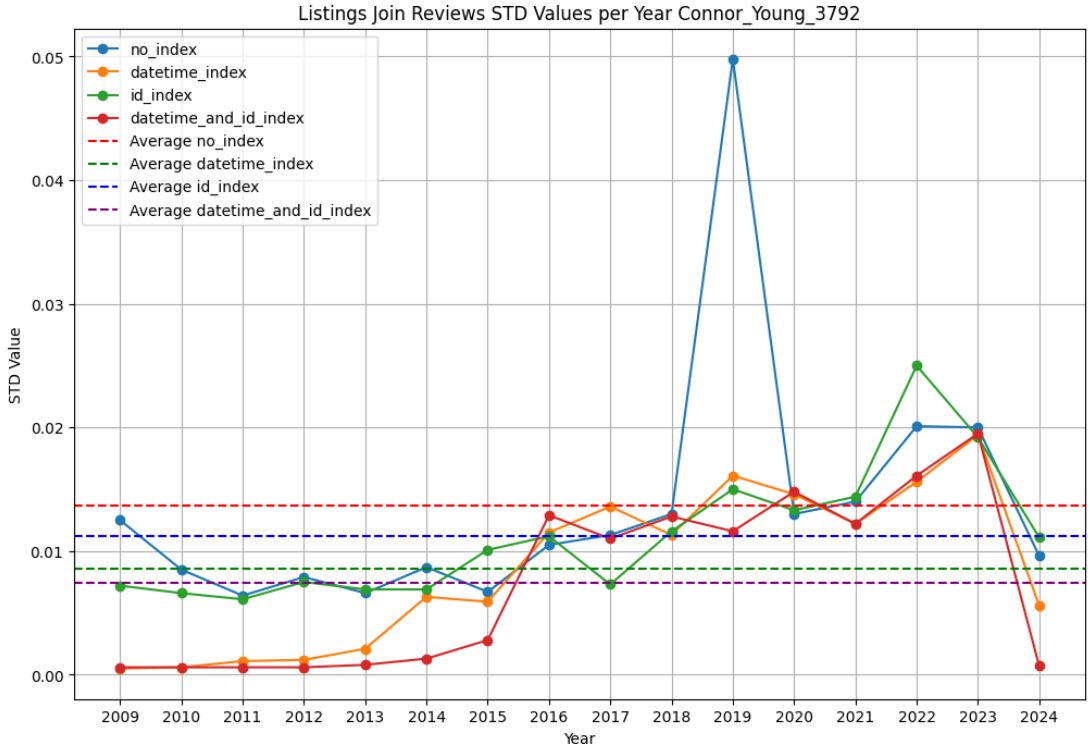

Figure 12: Listings Join Reviews STD Values per Year.

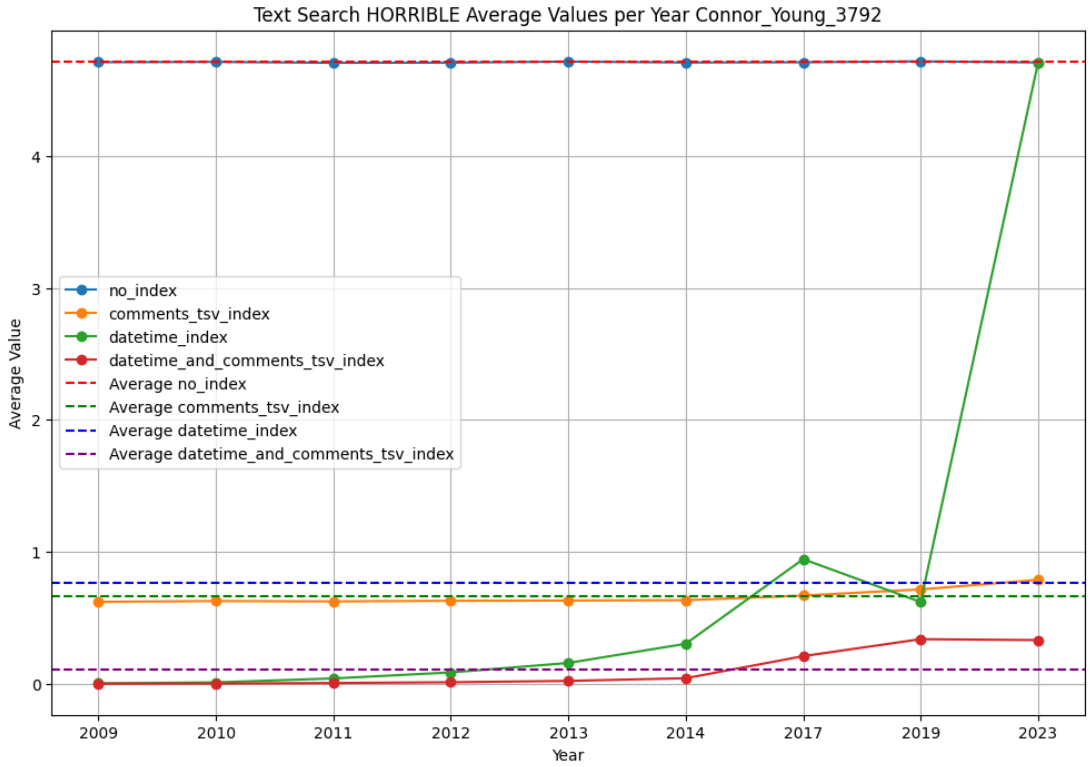

Figure 13: Text Search HORRIBLE Average Values per Year.

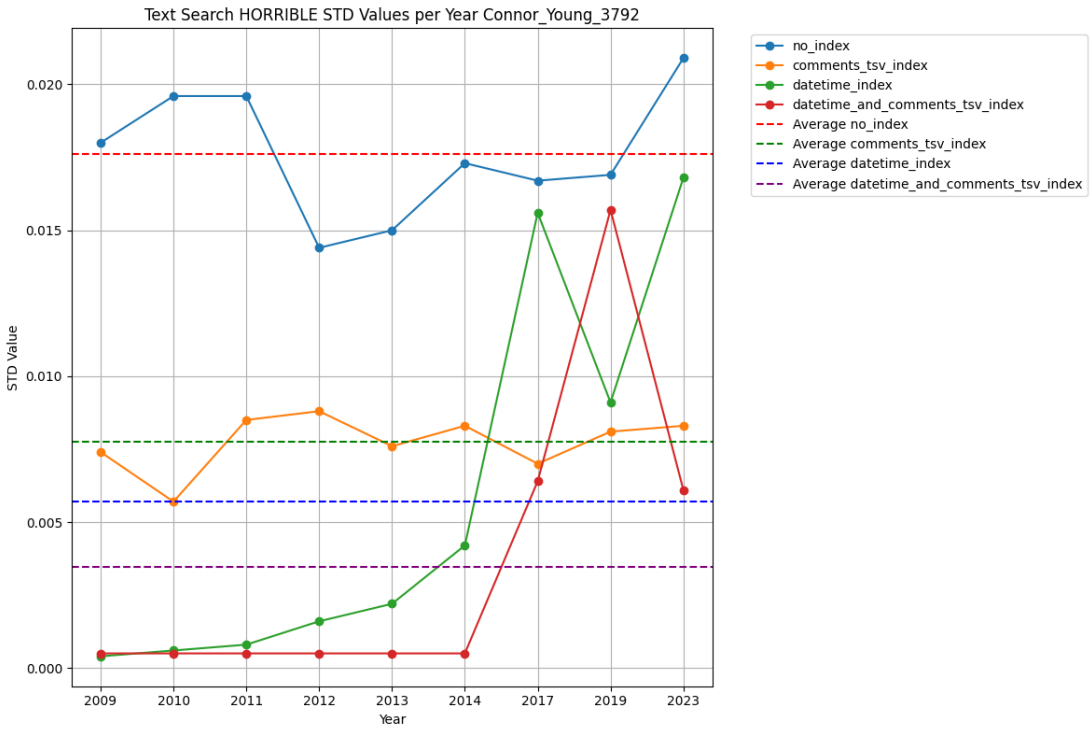

Figure 14: Text Search HORRIBLE STD Values per Year.

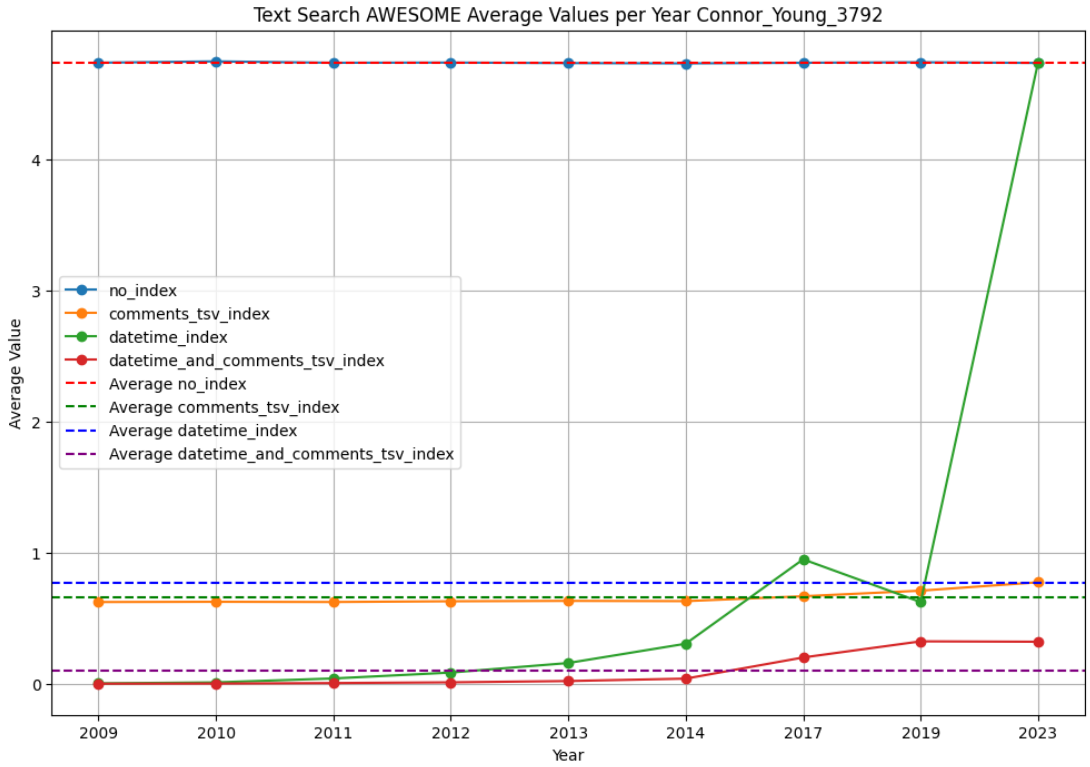

Figure 15: Text Search AWESOME Average Values per Year.

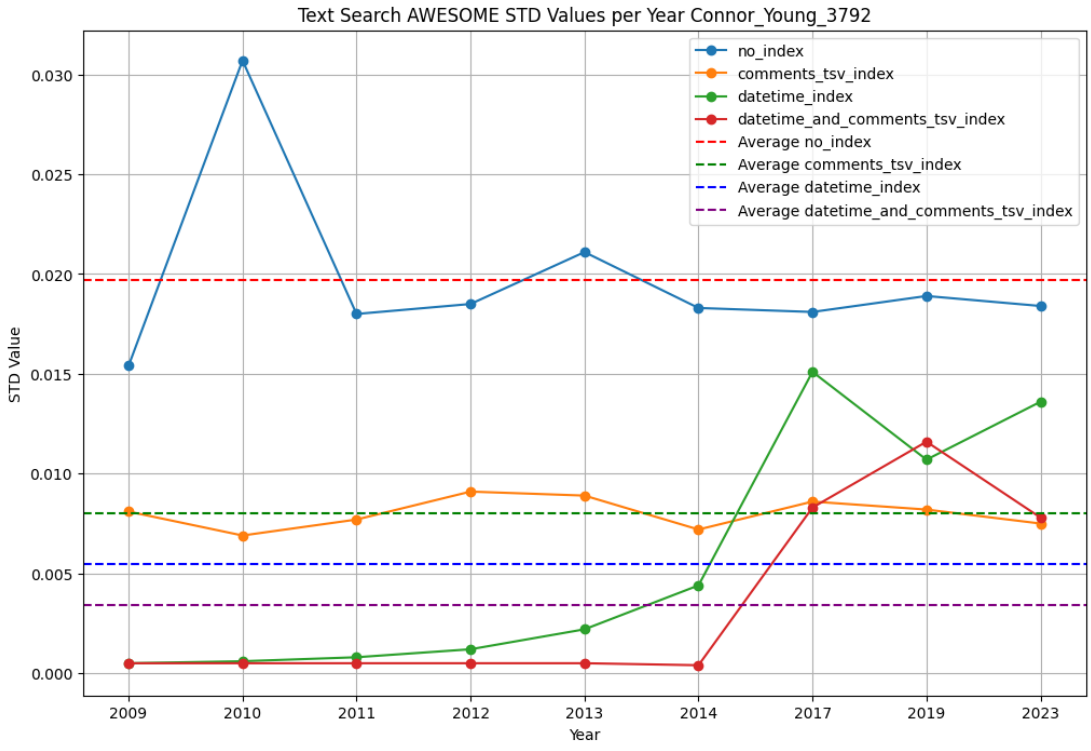

Figure 16: Text Search AWESOME STD Values per Year.

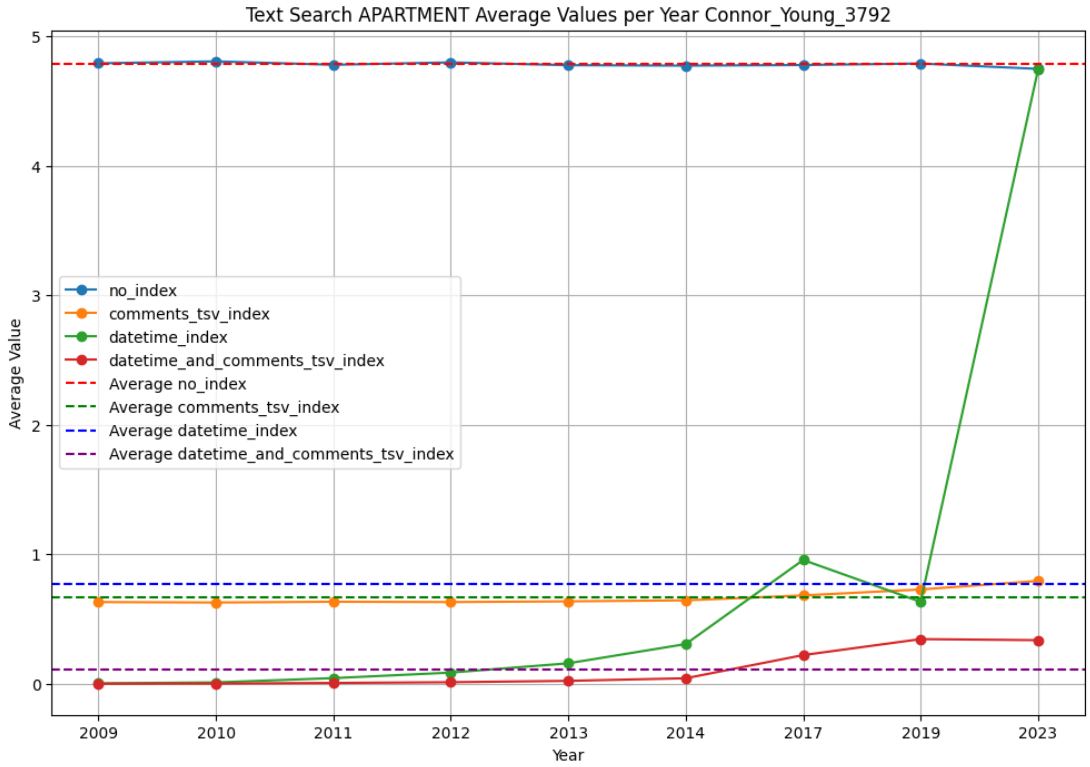

Figure 17: Text Search APARTMENT Average Values per Year.

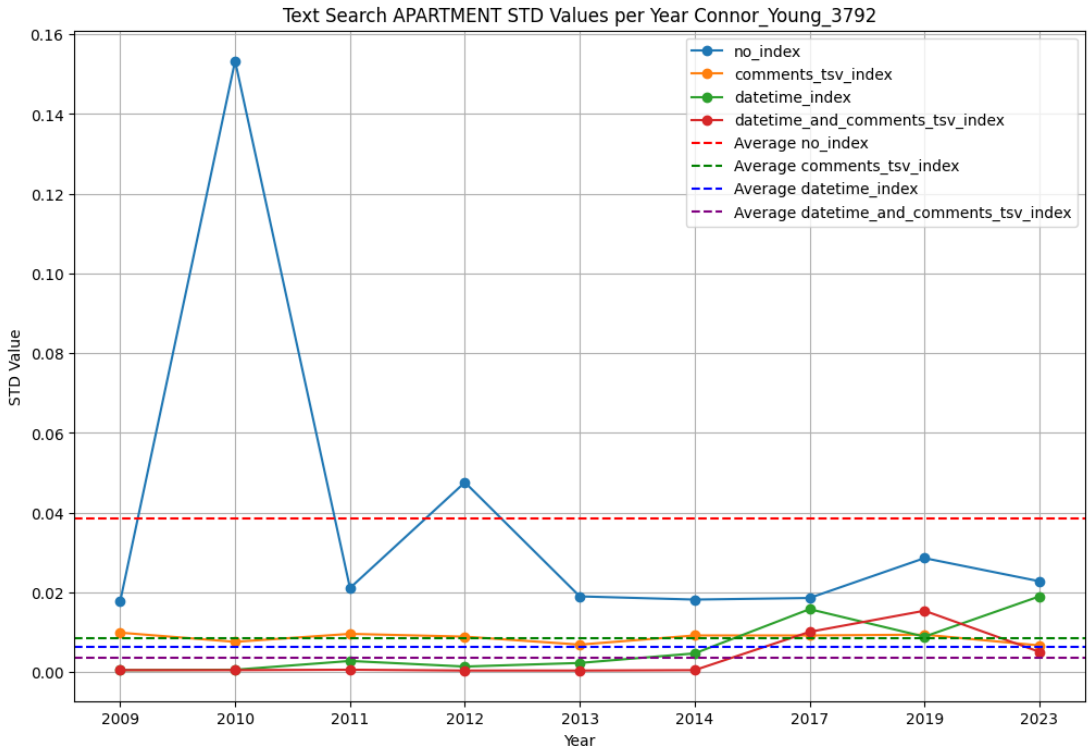

Figure 18: Text Search APARTMENT STD Values per Year.

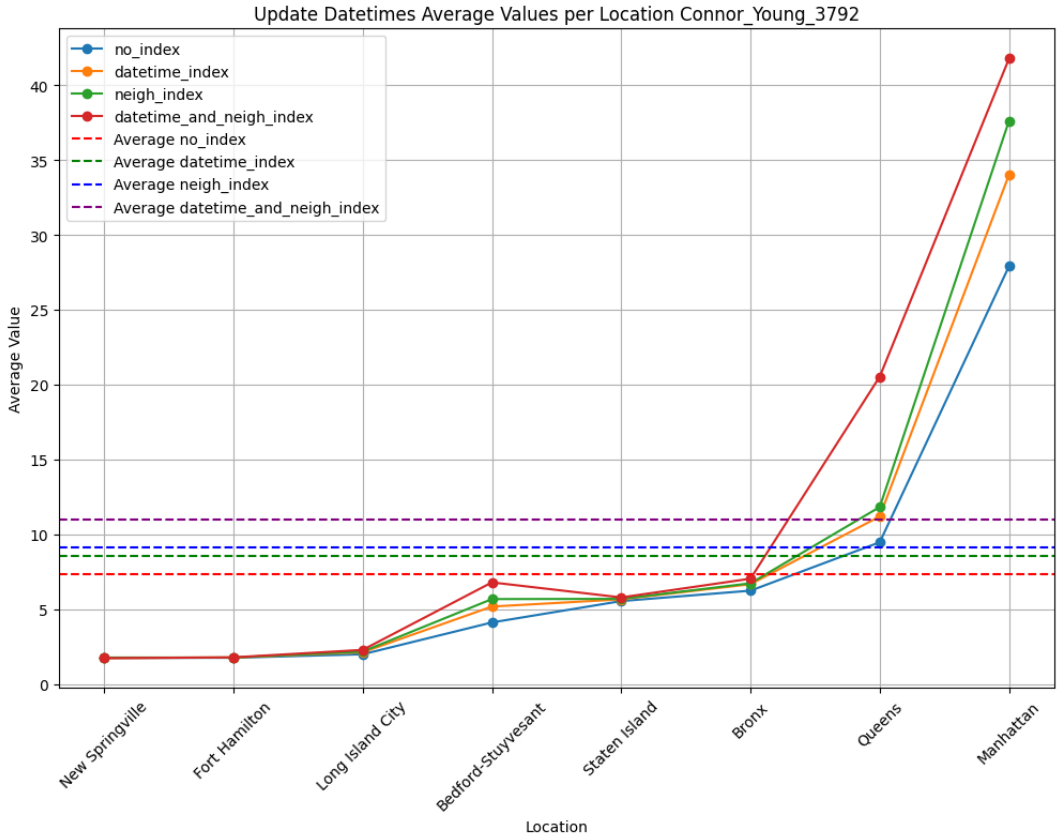

Figure 19: Update Datetimes Query Average Values per Location.

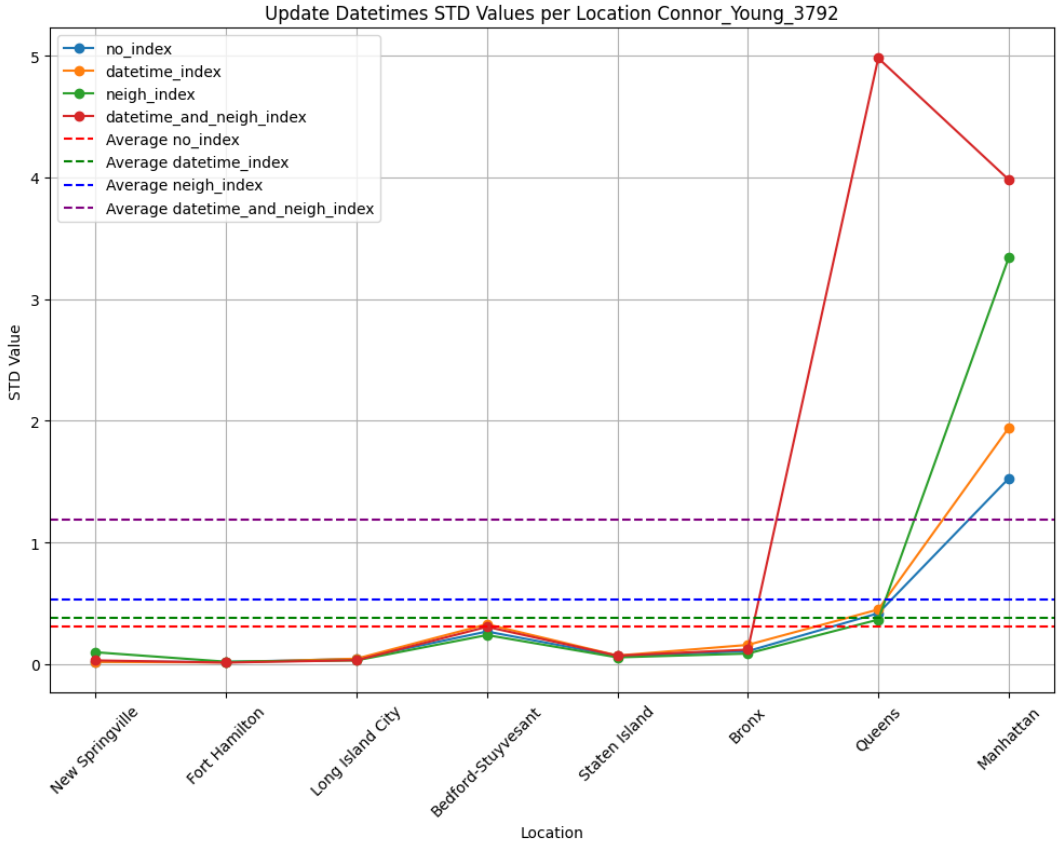

Figure 20: Update Datetimes Query STD Values per Location.

4.3 Cindy

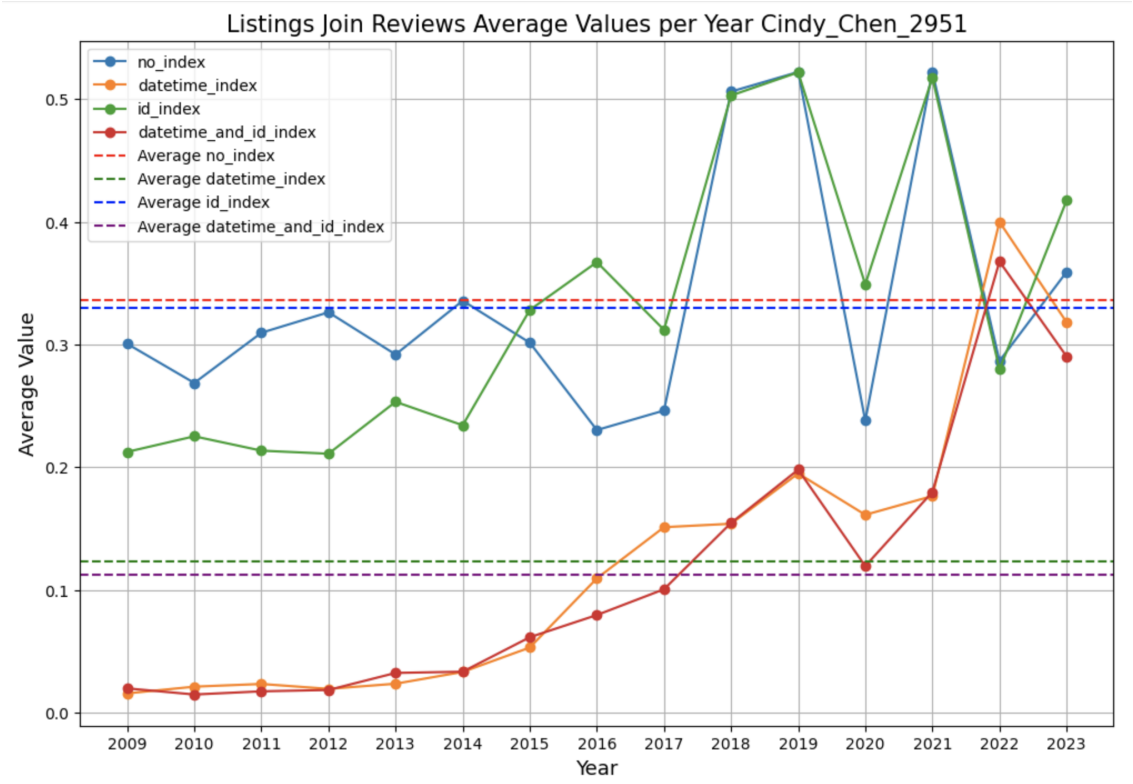

Figure 21: Listings Join Reviews Average Values per Year.

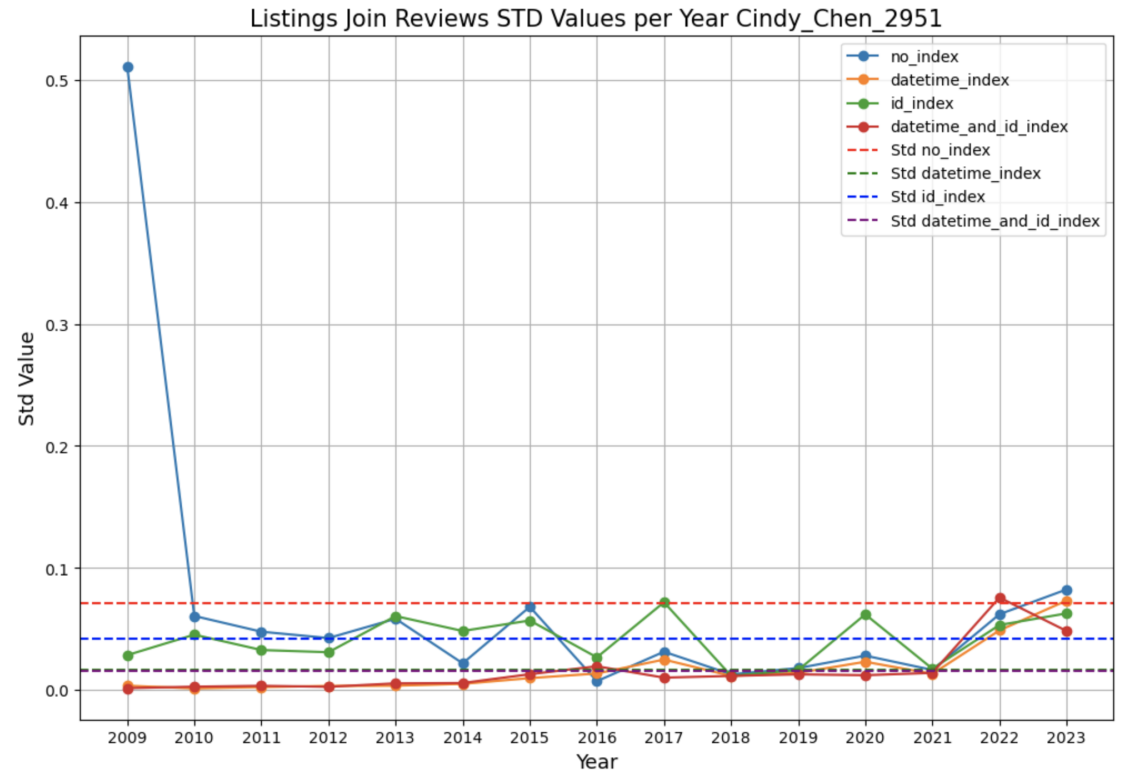

Figure 22: Listings Join Reviews STD Values per Year.

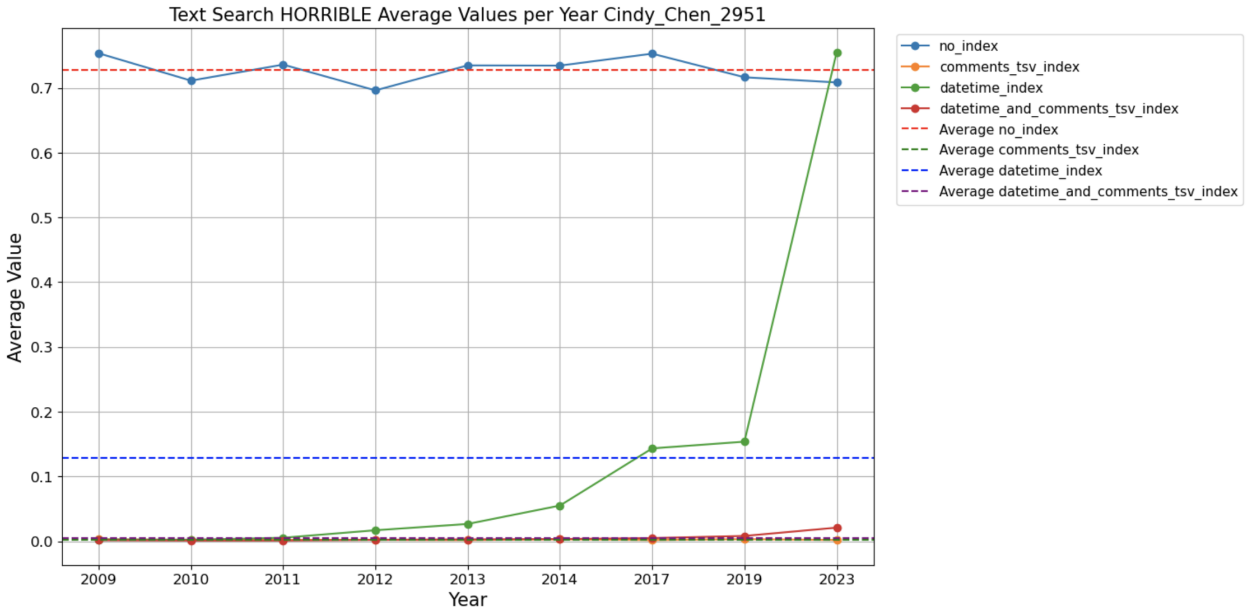

Figure 23: Text Search HORRIBLE Average Values per Year.

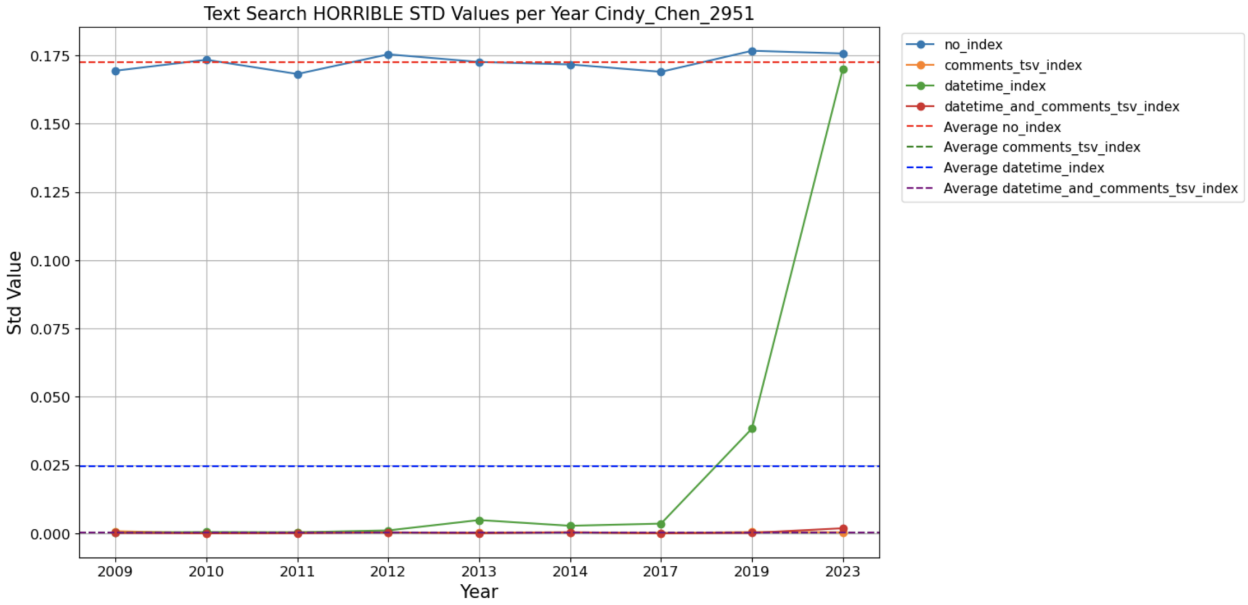

Figure 24: Text Search HORRIBLE STD Values per Year.

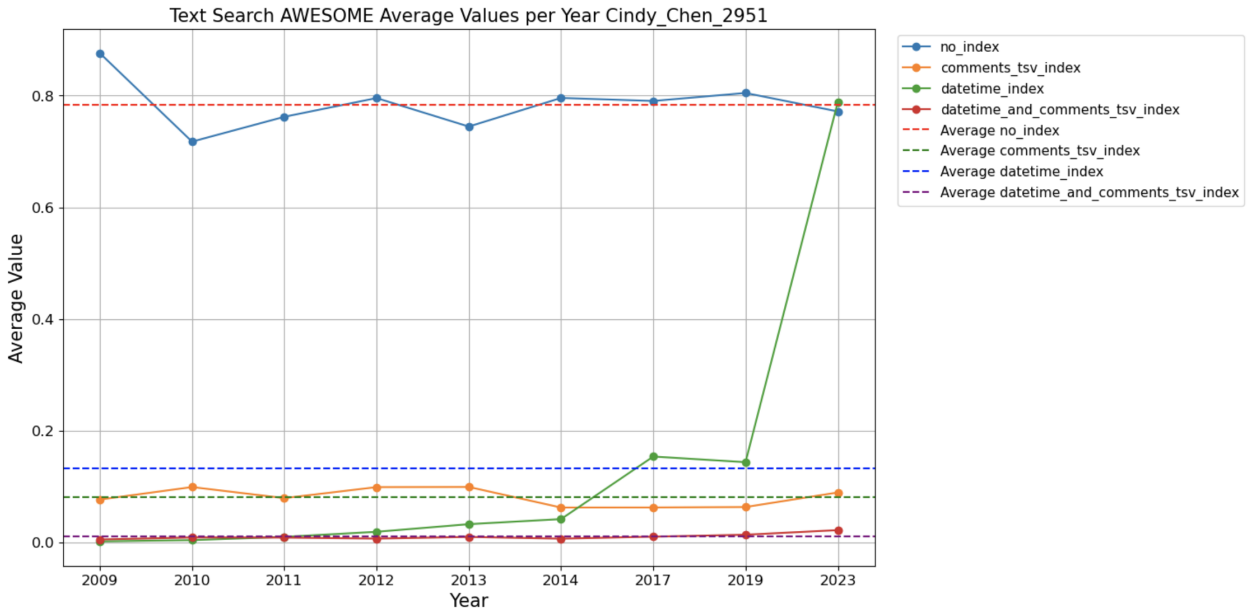

Figure 25: Text Search AWESOME Average Values per Year.

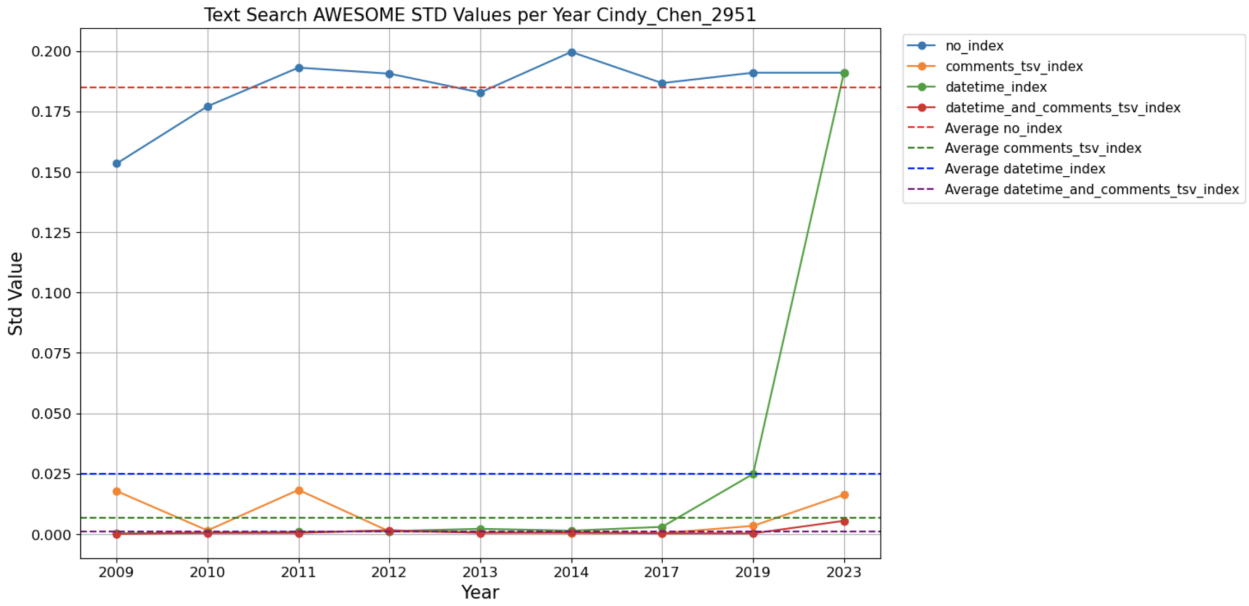

Figure 26: Text Search AWESOME STD Values per Year.

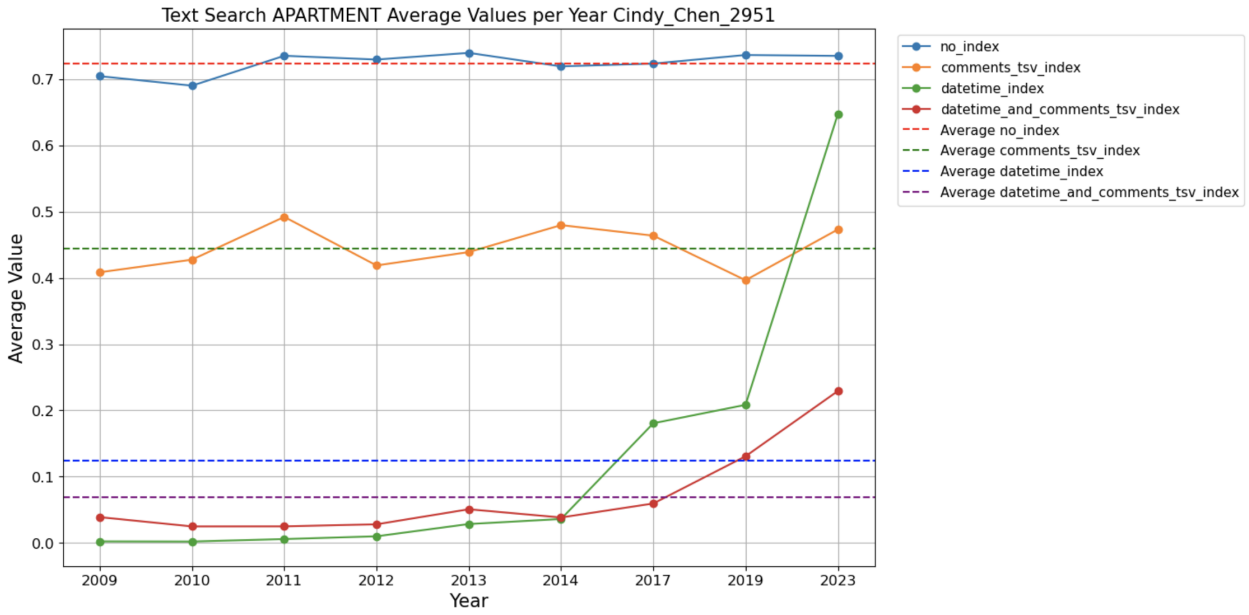

Figure 27: Text Search APARTMENT Average Values per Year.

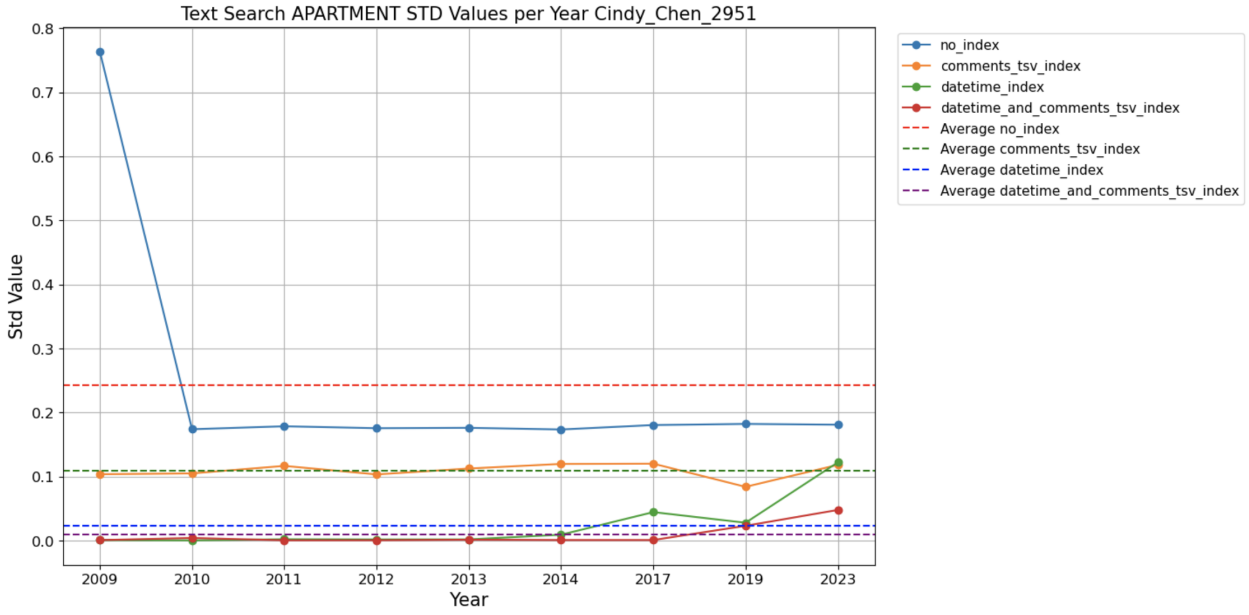

Figure 28: Text Search APARTMENT STD Values per Year.

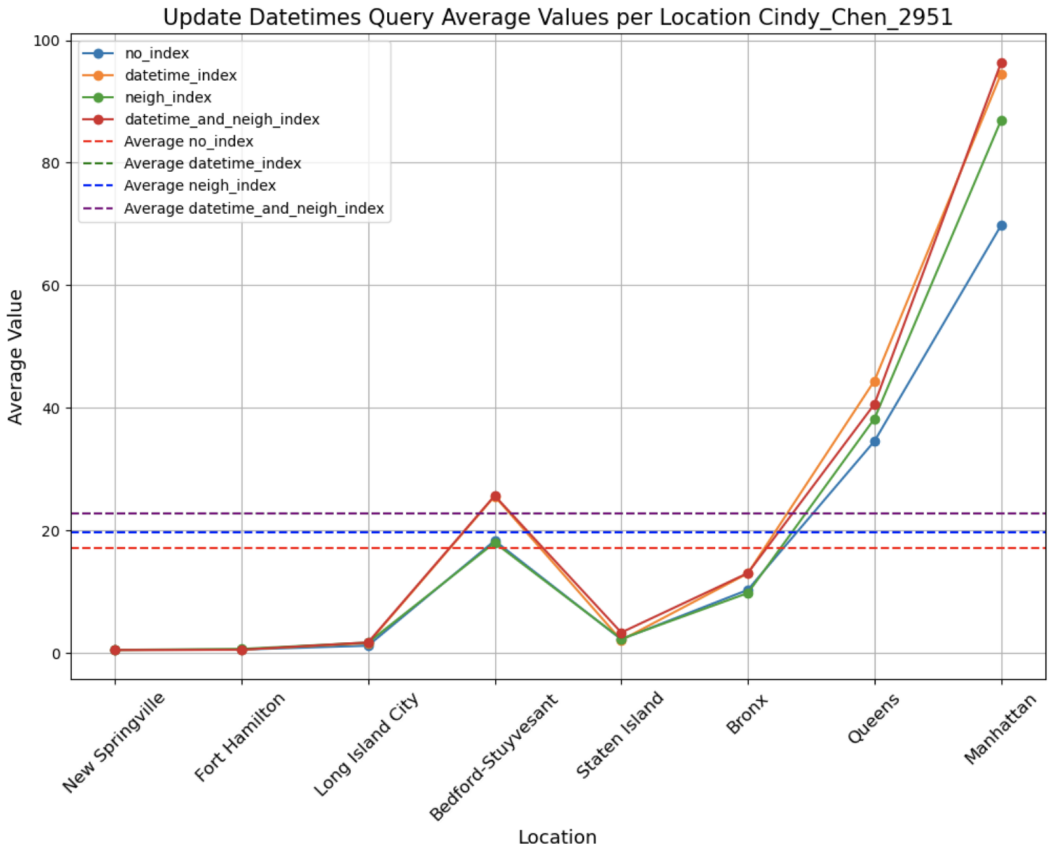

Figure 29: Update Datetimes Query Average Values per Location.

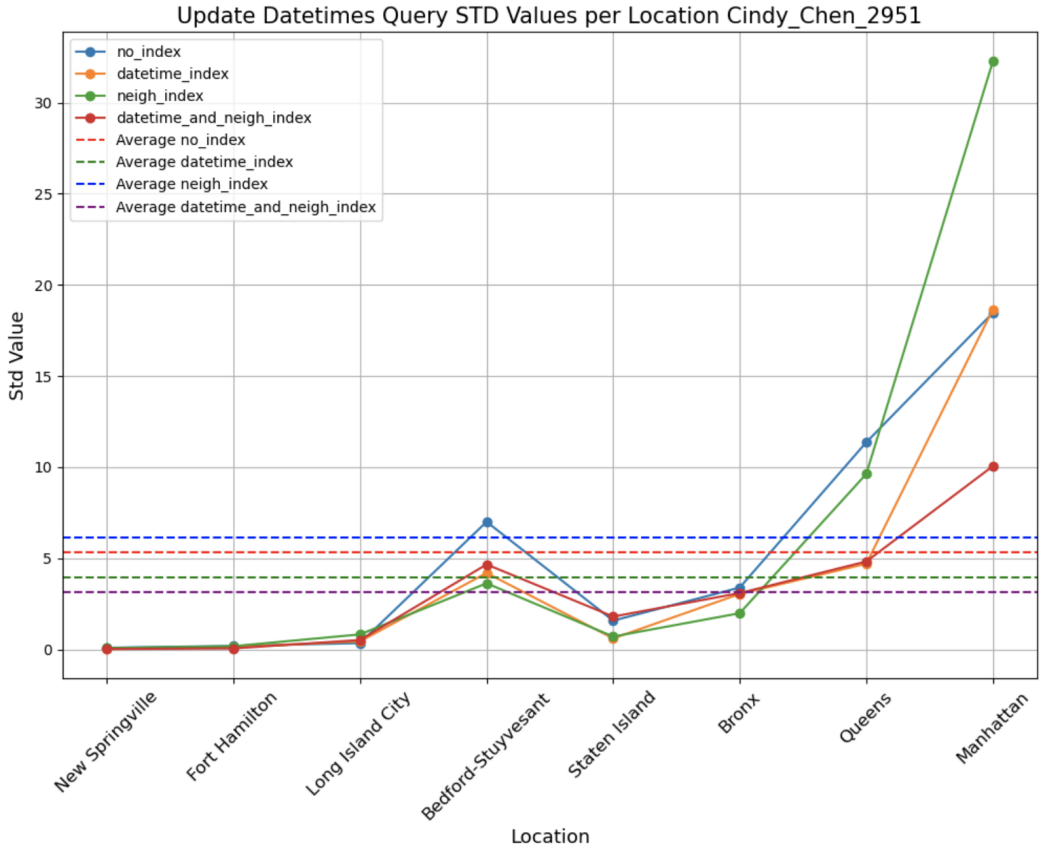

Figure 30: Update Datetimes Query STD Values per Location.

5 Explanations of Performance Results

5.1 Question 1

For Step 3a, the best index was datetime index. We can see from Nisha's and Connor's graph datetime index had the lowest average values per year, the average for the datetime index was also the lowest out of all 4 indexes. However, for Cindy's graph the the datetime and id index were a bit more effective than the datetime index, but they were still very close regarding average values per year. So overall, datetime index seems to be the most effective. The datetime index seemed to be the best because dates are easier to filter based on years since it is simple to extract that information. However, with the id index and no index, it is much more difficult to filter without the year since every row needs to be examined before the next one can be looked at.

5.2 Question 2

For the text search queries, the slowest runtime by far was from the queries using no index at all. With the text search vector query, run time improved. This is because the text search vector makes it a lot easier to identify certain lexemes compared to the ILIKE approach that needs to search through every word in the content of the reviews. Our group had significantly faster runtimes once we added in the datetime index. The run times per query were near 0, which was even faster than we expected, but we were able to confirm that the query was returning the correct count for each word and year. We think that the datetime index improved the runtime a lot because the datetime index can be sorted by year, and thus when we are looking for a specific year, such as 2009, it is much easier to focus on the section of reviews from 2009 and searching solely within those reviews for the word of interest. If we do not have the datetime index, the query has to go through every row to check the value of the datetime column, versus having an index that immediately identifies the year each row is associated with.

5.3 Question 3

Based on the average run times of updating the reviews table for each index configuration, we can see that no index consistently had the fastest run times and having the datetime and neigh index consistently had the longest run times. The comparison between just the datetime and just the neigh index is less consistent because the faster index seemed to vary between our machines. Regardless, our results indicate that having the datetime index did have an expense when updating the reviews table because it resulted in longer run times compared to having no index.

From our results, we can also see that this expense varies based on the location used in each update. The updates for New Springville, Fort Hamilton, Long Island City, Staten Island, and Bronx had relatively smaller expenses compared to the other other locations. This variance seems to be related to the number of records associated with each location, with fewer records being associated to a smaller expense to having the datetime index. For example, using Connor's results, if you take the difference in average run time between the updates using no index and the updates using both the datetime and neigh index and sort by this difference, the locations will be in almost exactly the same order as when you sort the locations by number of records. This is shown in the table on the next page.

Location	Records	Difference	Order
New Springville	104	-0.0078	1
Fort Hamilton	1000	0.0258	2
Long Island City	10859	0.2935	4
Staten Island	13726	0.2422	3
Bronx	35296	0.8004	5
Bedford-Stuyvesant	99705	2.6757	6
Queens	173392	11.0634	7
Manhattan	341287	13.8692	8

From this table, we can see that except for Long Island City and Staten Island, the difference between using no index and using both the datetime and neigh index scales with the number of records associated with each location. Long Island City and Staten Island have a relatively similar amount of records, so the association not holding with these results is not too surprising.

5.4 Question 4

For this question, we will be using Connor's data.

The average time to run the `listings_join_reviews_2019` query is 1.1242 seconds using no index and 0.2482 seconds using the `datetime_in_reviews` index. Thus, you would save 0.876 seconds if you ran the query 1 time and you would save 14 minutes and 36 seconds if you ran the query 1000 times.

The average time to run the `update_datetimes_query_Manhattan` query is 27.9098 seconds using no index and 34.0193 seconds using the `datetime_in_reviews` index. Thus, you would lose 6.1095 seconds if you ran the query 1 time and you would lose 1 hours, 41 minutes and 49.5 seconds if you ran the query 1000 times.

Let S denote the time saved by using the index for 1 execution of the query and let L denote the time lost by using the index for 1 execution of the update.

$$\begin{aligned} xS &= (1-x)L \\ xS &= L - Lx \\ xS + xL &= L \\ x(S+L) &= L \\ x &= \frac{L}{S+L} \\ x &= \frac{6.1095}{0.876 + 6.1095} \\ x &= \frac{6.1095}{6.9855} \\ x &= 0.8746 \end{aligned}$$

Thus, if we were to do more than 87.46% queries and less than 12.54% updates, then we would save time by using the index. Conversely, if we were to do less than 87.46% queries and more than 12.54% updates, then we would save time by not using the index.

6 References

We did not use any outside sources.