Build your own IR system

Terence Beijloos
Radboud University
Nijmegen, Netherlands

Mourya Bandaru
Radboud University
Nijmegen, Netherlands

Lukas Hamstra
Radboud University
Nijmegen, Netherlands

ABSTRACT

In this project, we covered two ways to build an IR system from scratch-one using inverted index and one bucketed inverted index index version both using BM25. We compared both versions in terms of speed, memory usage, and accuracy. Additionally, we compared our IR system, using the same metrics, to a similar system namely Pyterrier.

Our "standard" IR system uses Inverted Index and the BM25 algorithm in order to retrieve the documents and rank them. The Bucketed version creates buckets of similar documents by using the Locality-Sensitive Hashing method. Following this, the buckets are sorted into an inverted index structure, ranking buckets for relevant query terms when executing a query.

We used these methods on the NFCorpus dataset, which was developed by Boteva et al. (2016), representing a dataset made for the ranking system evaluation regarding medical information retrieval. Its structure also fits our goals well, since it maps text queries to relevant sources rather than directly comparing text-to-text, making it compatible with our indexing methods. The project is more about trying out these indexing methods ourselves to gain hands-on experience and insights rather than aiming for advanced results.

CCS CONCEPTS

• **Information systems → Retrieval models and ranking**.

KEYWORDS

Information Retrieval, Indexing, BM25, Sparse Models, LSH

1 INTRODUCTION

Information Retrieval systems help us find the information we need quickly, whether we are searching the web or looking through large datasets, by allowing quick access. At the core of these systems is indexing methods that organize data in order to retrieve it faster. While modern IR systems use advanced techniques in their construction, doing so from scratch will reveal how they work and what some of the trade-offs involved are.

In this project, we compare the performance of two indexing methods: Inverted Index and Bucketed Inverted Index. Each of these methods is combined with BM25 ranking algorithm to rank documents for relevance. Our main question is: How do these two indexing methods compare in terms of speed, memory usage, and accuracy?

We are using the NFCorpus dataset to test our systems with real-world queries. This is not a project about something groundbreaking but about learning through hands-on implementation. By comparing these two methods, we hope to see which one works better in different situations and understand more about how indexing impacts IR system performance.

In this project will try these techniques ourselves, so we understand the strengths and weaknesses of each, by directly comparing the performance of these techniques regarding practical metrics such as execution time, memory consumption, and accuracy. To determine how reasonable our IR systems function we will compare them to an existing solution, Pyterrier.

2 RESEARCH QUESTIONS

(1) **How does the Normal Inverted Index compare to the Bucketed Inverted Index in terms of speed during query execution?**
(2) **What is the difference in memory usage between the Normal Inverted Index and the Bucketed Inverted Index?**
(3) **How do the indexing methods impact the accuracy of document retrieval, as measured by Normalized Discounted Cumulative Gain (NDCG)?**

3 BACKGROUND AND RELATED WORK

3.1 Inverted Indexing Techniques

The inverted index is a standard data structure in information retrieval systems, mapping terms to the documents containing them [5]. While efficient for many tasks, its performance can degrade with large datasets and complex queries.

3.2 Locality-Sensitive Hashing (LSH) for Bucketing

Locality-Sensitive Hashing (LSH) groups similar documents in buckets to reduce the search space while retrieval. Gionis et al. [2] introduced LSH for approximate nearest-neighbor search, proving its effectiveness in high-dimensional spaces.

3.3 Relevance Scoring and Ranking

BM25 is a ranking function that utilizes term frequency and document length normalization for document scoring in a query-dependent manner [4]. Normalized Discounted Cumulative Gain, on the other hand, is used to assess ranking quality by considering both relevance and order [3].

3.4 Related Works

Zobel and Moffat [6] addressed some challenges of the traditional inverted indexes regarding scalability and memory efficiency. Andoni and Indyk [1] developed a method of combining LSH with ranking methods in order to improve query performance. Direct comparisons among indexing methods with respect to NDCG, memory usage, and query speed are still few in number.

3.5 Contributions of This Work

This work compares the performance of traditional inverted indexing with LSH-based bucketed indexing based on time, memory usage, and NDCG. The work also studies the integration of LSH with BM25 on large datasets for retrieval optimization.

REFERENCES

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–468, 2006.

[2] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, 1999.

[3] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[4] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[5] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[6] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):1–56, 2006.

4 EXPERIMENTAL DESIGN

- **Dataset:** This project uses the NFCorpus dataset, containing natural language queries and pairs of documents. Since the dataset would very likely be appropriate for assessing retrieval models for a realistic query-document matching problem, it should also be small enough to perform feasible experiments.
- **Evaluation Metrics:** Indexing methods performance would be measured based on:
  - **Speed:** Determined based on the execution time of query execution by a profiling tool called `cProfile`.
  - **Memory Usage:** Measured in the processes of indexing and execution of queries by using `memory-profiler`.
  - **Accuracy:** Measured by using Normalized Discounted Cumulative Gain (NDCG), which is the ranking relevance of the documents.
- **Tools and Frameworks:** Implementation and analysis are based on the following tools and frameworks:
  - **Python:** This is the major implementation language used in building and testing indexing systems.
  - **BM25 Algorithm:** An algorithm to score and rank documents based on their relevance to the query.
  - **Inverted Index:** Data structure that maps every term to the documents in which it occurs. Allowing for very fast and efficient retrieval of documents.
  - **Locality-Sensitive Hashing (LSH):**This technique has been employed in the Bucketed Inverted Index to bucket similar documents under the same buckets.
  - **Profiling Tools:** Like `cProfile` a performance profiling tool, `memory profiler` for understanding memory consumptions.
- **Workflows:**
  - **Normal Inverted Index:** Tokenizes the documents, directly indexes and scores the documents using the BM25 Algorithm. Retrieved documents are ranked by their BM25 scores.
  - **Bucketed Inverted:** Documents are first divided into buckets using LSH. An inverted index is built inside each bucket. Queries fetch the appropriate buckets, and documents in these buckets are scored and ranked with the help of BM25.
- **Implementation Phases:**
  (1) **Preprocessing:** The work includes tokenization and then normalizing the dataset by employing lowercasing and removal of stopwords, amongst others.
  (2) **Index Construction:** Building the Normal Inverted Index and the Bucketed Inverted Index.
  (3) **Query Execution:** Running sample queries on both indexing methods to retrieve documents.
  (4) **Metric Analysis:** Recording and comparing the results across the defined metrics (speed, memory usage, and NDCG).

5 IMPLEMENTATION DETAILS

5.1 Normal Inverted Index

The following procedures are used to implement the Normal Inverted Index in the `inverted_index.py` file:

(1) **Tokenization:** In order to extract tokens from document text, the dataset is preprocessed using a lexer. Stopword elimination is one of the preprocessing methods.
(2) **Index Construction:** A posting list with document IDs, term frequencies, and document durations is transferred to each token (term). When creating an index, the average and total document lengths are updated.
(3) **Serialization:** For use in further tasks, the created index is stored to disk in either text or binary format (using `pickle`).
(4) **Query Execution:** Terms from a query are searched the index during retrieval, and the BM25 algorithm is used to rate relevant items.

Python's `SortedList` was used to maintain posting lists in sorted order, optimizing query execution.

5.2 Bucketed Inverted Index with Locality-Sensitive Hashing (LSH)

By putting related documents into buckets, the Bucketed Inverted Index with Locality-Sensitive Hashing (LSH) increases scalability. Here's how this approach is put into practice:

(1) **Document Bucketing:** LSH reduces the search space for queries by bucketing documents with similar queries.
(2) **Index Construction:** A list of buckets is kept for every query. Document references and their term frequencies are included in each bin. After that, these buckets are arranged and sorted into a structure resembling an inverted index.
(3) **Handling Bucket Scoring:** When a query is executed, documents from all relevant buckets for the query terms are retrieved. Documents that do not contain the query term are assigned a value of 0.5 to ensure they contribute to BM25 scoring.
(4) **Query Execution:**After utilizing LSH to retrieve the most relevant buckets, a query uses the BM25 algorithm to score and rank the documents within those buckets.

This approach ensures scalability by narrowing the search space using LSH while still leveraging the inverted index for efficient scoring.

5.3 Evaluation and Comparison

To compare these two indexing methods, we use the following metrics to evaluate performance on the NFCorpus dataset:

- **Speed:** The time it takes for execution measured using the `cProfile` profiling tool.
- **Memory Usage:** Assessed during both indexing and query execution by `memory-profiler`.
- **Accuracy:** - Using Normalized Discounted Cumulative Gain (NDCG), a measure of relevance of retrieved documents to query

The comparison highlights the trade-offs between computational speed, memory efficiency, and retrieval accuracy.

6 OUTPUTS

The following results were obtained for the Normal Inverted Index and LSH-based retrieval methods using the BM25 algorithm. Evaluations were conducted with varying values of k, where k refers to the number of top-ranked documents retrieved per query. This parameter is dynamically adjusted using the `top_k` variable.

6.1 Inverted Index

Results:

Top-k	Mean NDCG	Mean Precision	Mean Recall
2	0.5148	0.4136	0.0752
5	0.5103	0.3206	0.1112
10	0.4876	0.2473	0.1442
25	0.4433	0.1611	0.1891

Table 1: Performance metrics for different Top-k values using the inverted index.

Discussion: The results highlight how the system's performance changes as k increases:

- For smaller values of k, such as $k = 2$, the system achieves higher NDCG and precision, indicating that the top-ranked documents are highly relevant.
- As k increases, recall improves since more relevant documents are included in the results. However, precision and NDCG decline, reflecting reduced relevance at higher ranks.
- Additionally, we observed that at certain k values, both recall and precision were low simultaneously. This suggests that while a larger set of documents is retrieved, many of these may be irrelevant, leading to inefficiencies in performance across both metrics.

cProfile Results: The results of computational performance are as follows:

Function	Details
`bm25.py:41(rank)`	2590 calls, cumulative time: 8.687s
`bm25.py:14(compute_score)`	1,281,899 calls, cumulative time: 6.341s
`<frozen _collections_abc>:771(get)`	1,281,899 calls, cumulative time: 0.884s
`inverted_index.py:33(__getitem__)`	2,577,175 calls, cumulative time: 0.650s
`built-in sorted`	2590 calls, cumulative time: 0.554s

Table 2: Profiling results for computational performance.

The profiling results highlight the computational cost of the BM25 score calculation and frequent data retrieval from the inverted index.

The memory vs speed comparison for Inverted Index is shown in **Figure 1**

6.2 Bucketed Inverted Index with Local-Sensitive Hashing (LSH):

Results:

Top-k	Mean NDCG	Mean Precision	Mean Recall
2	0.3927	0.2969	0.0567
5	0.4231	0.2412	0.0897
10	0.4214	0.1951	0.1196
25	0.3998	0.1343	0.1642

Table 3: Performance metrics for different Top-k values using LSH.

Discussion: The results show the performance of the system for an increasing value of k.

- For smaller values of k, for example, k=2: The system gets relatively higher NDCG and precision, which indicates that the top-ranked documents are more likely to be relevant. This shows that LSH is very effective in choosing a few highly relevant documents at the top ranks.
- With increasing k: Recall improves significantly as more relevant documents are included in the retrieved set. However, precision and NDCG go down, reflecting a trade-off in performance. Lower precision indicates that more irrelevant documents are retrieved while the decline in NDCG reflects reduced relevance among the top-ranked results
- Observations at intermediate k values: On the other hand, in certain values, such as k=10 and k=25, recall and precision comparatively became lower. This represents that with an increase in the set of retrieved documents, the inclusion of some of the irrelevant results decreased the overall efficiency of the system. The inefficiency has possibly resulted from the approximation of LSH or certain limitations in the process of bucket selection.

cProfile Results: The results of computational performance are as follows: The memory vs speed comparison for LSH is shown in

Function	Details
`bm25.py:48(rank)`	2590 calls, cumulative time: 15.013s
`bm25.py:15(compute_score)`	1,984,071 calls, cumulative time: 9.958s
`bm25.py:40(concat)`	6464 calls, cumulative time: 1.513s
`<frozen _collections_abc>:778(__contains__)`	1,990,984 calls, cumulative time: 1.387s
`{method 'get' of 'dict' objects}`	1,984,071 calls, cumulative time: 1.074s
`lsh_index.py:35(__getitem__)`	3,975,055 calls, cumulative time: 0.960s
`{built-in method builtins.sorted}`	2590 calls, cumulative time: 0.659s
`{built-in method math.log}`	1,984,071 calls, cumulative time: 0.537s
`bm25.py:65(<listcomp>)`	2463 calls, cumulative time: 0.329s
`{method 'add' of 'set' objects}`	1,984,071 calls, cumulative time: 0.272s

Table 4: Profiling results of the LSH implementation.

Figure 2

7 DISCUSSION

7.1 Local-Sensitive Hashing

The LSH system works by shingling the given documents and creating one-hot encoded arrays for each specific document, indicating whether or not a certain shingle is present in the document. Next,

through minhashing, the one-hot encoded array is turned into a document signature of variable length. This signature is then separated into a variable amount of bands, and these bands are compared across documents to determine which documents end up in the same bucket.
By adjusting the signature size and band amount, the average similarity between documents in a bucket can be impacted. After some testing, the length of the signature was decided upon to be 128, and the amount of bands to compare to be 8. This means that every signature of 128 numbers has 8 bands, each 16 numbers long. If any one of these bands matches with a band from another document, the documents are added to the same bucket.
A flaw in LSH is the very present possibility of false positives in the buckets. Normally these would be filtered out. However, it was decided not to do so for this system. This decision was taken because, when a query is executed, it documents collected from the buckets relevant to the query terms will still be scored and ranked, and thus the false positives will be filtered out in that stage by being relegated to the bottom of the rankings.

7.2 Comparing results for Inverted Index and LSH Index

Table 5: Comparison of NDCG, Precision and Recall results for Inverted Index and LSH

Top-k	Metric	Inverted Index	LSH Index	Difference
2	Mean NDCG	0.5148	0.3927	+0.1221
	Mean Precision	0.4136	0.2969	+0.1167
	Mean Recall	0.0752	0.0567	+0.0185
5	Mean NDCG	0.5103	0.4231	+0.0872
	Mean Precision	0.3206	0.2412	+0.0794
	Mean Recall	0.1112	0.0897	+0.0215
10	Mean NDCG	0.4876	0.4214	+0.0662
	Mean Precision	0.2473	0.1951	+0.0522
	Mean Recall	0.1442	0.1196	+0.0246
25	Mean NDCG	0.4433	0.3998	+0.0435
	Mean Precision	0.1611	0.1343	+0.0268
	Mean Recall	0.1891	0.1642	+0.0249

Insights:

- **Mean NDCG Performance:** From the above, it is observed that Inverted Index has better NDCG over LSH for all values of Top-k, having the maximum difference for Top-2 (+0.1221), suggesting better ranking relevance for Inverted Index against LSH.
- **Mean Precision:**Inverted Index gives better precision values consistently and, as Top-k increases, the difference narrows (for Top-2: +0.1167 with Top-25: +0.0268).
- **Mean Recall:** Recall favors Inverted Index for all Top-k values, implying better retrieval coverage.

Insight conclusion: Looking at the results in **Table 5** the obvious difference is the relatively constant, slightly worse scoring of the LSH index system. Because of the bucketed setup of the LSH index it will collect more documents in total to rank when executing a query. As mentioned before: documents that are collected for an index term that do not actually contain the index term (due to being sorted into a bucket that does include the index term) get a value of 0.5. This means that, during scoring, these documents are treated as more relevant than those with a value of 0. With accurate bucketing this would in most cases ring true. However, we believe that our bucketing was not robust enough, and hence lowered the scores of the LSH index system.

cProfile speed comparision

- **BM25 Rank Function:**
  - For the inverted index, `bm25.py:41(rank)` was called 2590 times with a cumulative time of 8.687s.
  - For the LSH implementation, `bm25.py:48(rank)` was called 2590 times with a cumulative time of 15.013s, almost double the time of the inverted index.
- **BM25 Compute Score:**
  - For the inverted index, `bm25.py:14(compute_score)` had 1,281,899 calls with a cumulative time of 6.341s.
  - For the LSH implementation, `bm25.py:15(compute_score)` had 1,984,071 calls, taking 9.958s, showing increased computational overhead.
- **Index Access Time:**
  - `inverted_index.py:33(__getitem__)` in the inverted index had 2,577,175 calls with a cumulative time of 0.650s.
  - `lsh_index.py:35(__getitem__)` in LSH had 3,975,055 calls, taking 0.960s, reflecting increased index access frequency and time.
- **Other Computations:**
  - LSH introduced additional computations, such as `math.log` (1,984,071 calls, 0.537s) and `bm25.py:40(concat)` (6464 calls, 1.513s), indicating more complex operations.

Memory Comparison:
As expected, we see that LSH index consumes more memory than the Inverted Index in **Table 4**. This is due to the nature of the index structure: where the inverted index only stores documents containing at least one instance of the index term a LSH index term can also contain documents that do not explicitly contain the index term. This is because any bucket containing at least one document with the index term mentioned is stored for that index term, regardless of the frequency in the other documents in the bucket.

7.3 Post-Construction Improvements

After building the model, following updates were made:

(1) **Better usage of posting lists:** Term Frequencies have been computed during the first scan of the posting list to speed up the process. This would eliminate any need to double look at the list, hence improving the speed of query processing.
(2) **No BM25 Normalization:** The scores from BM25 are left without dividing by the highest, to avoid confusion and follow standards, which will give a better picture of the ranking of the documents done by the model.

(3) **Baseline Performance Comparison:** The system was then tested against PyTerrier, comparing results for accuracy, that is, `BM25` and `NDCG` scores, and speed. The results are presented below. The memory vs speed comparison for PyTerrier is shown in **Figure 3**

Top-k	Mean NDCG	Mean Precision	Mean Recall
2	0.5134	0.4126	0.0742
5	0.5098	0.3232	0.1127
10	0.4850	0.2514	0.1453
25	0.4454	0.1637	0.1901

Table 6: Performance metrics for different Top-k values using PyTerrier.

Compared to PyTerrier, the inverted index is very similar in performance in both systems by all metrics. For example, **at k = 2 the inverted index reaches a Mean NDCG of 0.5148, slightly higher than PyTerrier's 0.5134**, while their Mean Precision and Mean Recall are almost identical. For increasing values of k, both systems show regular decreases in Precision and NDCG but an improvement in Recall.

It is important to note that while our inverted index takes advantage of lexical matching, PyTerrier in its current version does not include lexing. It may be that PyTerrier with lexical matching would achieve better performance, especially for recall measures. Even then, results suggest that our model performed well and stood close to PyTerrier with the settings used.

8 CONCLUSION

We were able to implement both the inverted index and the LSH (Locality Sensitive Hashing) techniques and achieved presentable results with their performance. While the inverted index performed robustly, the LSH exhibited a little lower performance. This is probably because of the suboptimal bucket selection that could have impacted its efficiency.

This project gave us valuable experience in implementing different indexing techniques and in developing a presentable IR model. We also realize now that the dataset used was perhaps not ideal for the type of comparison we were making; rather, it was suited more towards learning-to-rank algorithms—a realization a bit too late in the process.

Despite these challenges, we were able to achieve our key objective of comparing the two indexing approaches and drawing valuable insights from the same. On the whole, the project met our initial goals and gave us a deeper understanding of Indexing techniques.

ACKNOWLEDGMENTS

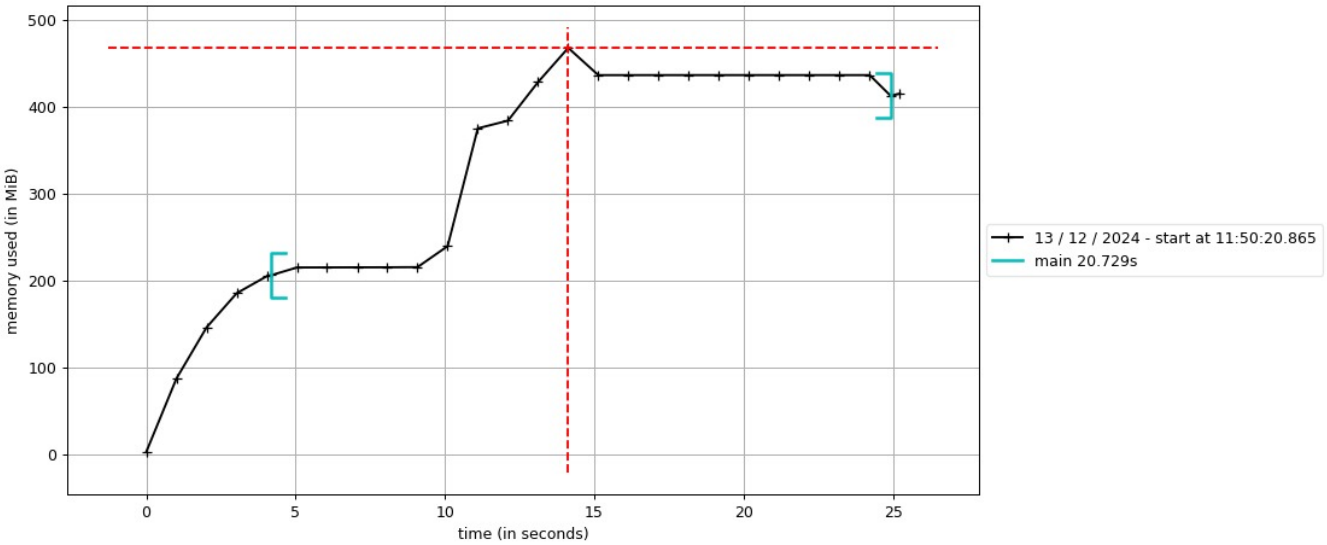

Figure 1: Memory usage vs. Execution time during the evaluation process for Inverted Index.

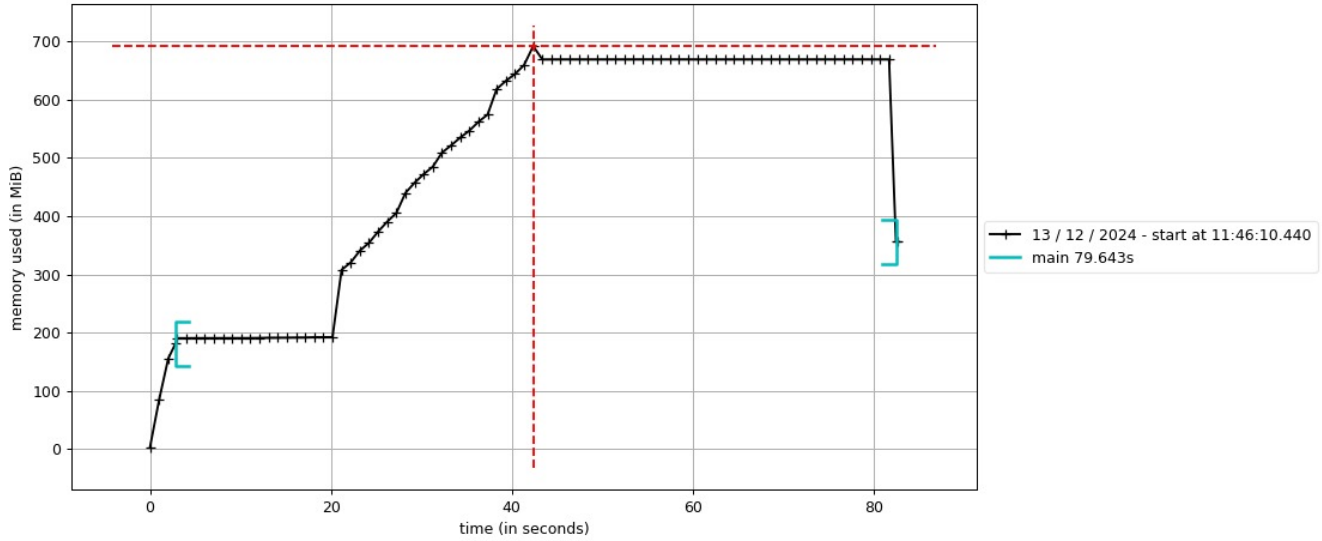

Figure 2: Memory usage vs. Execution time during the evaluation process for LSH Index.

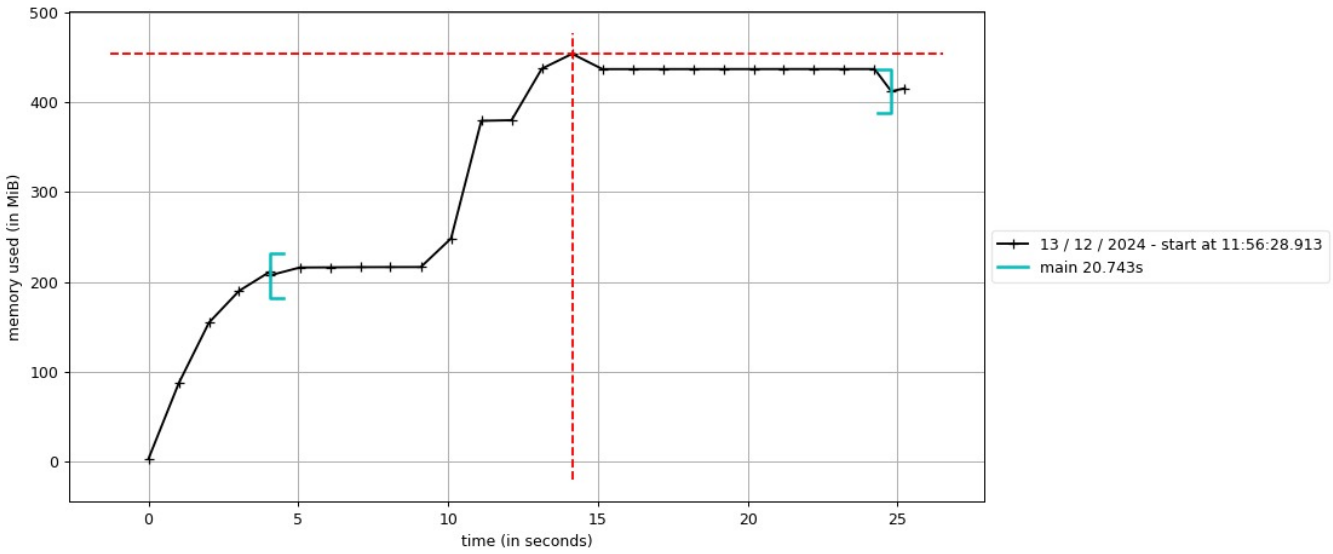

Figure 3: Memory usage vs. Execution time during the evaluation process for PyTerrier.