# Parametric Search using In-memory Auxiliary Index

Nishant Verman and Jaideep Ravela
Stanford University, Stanford, CA
{nishant, ravela}@stanford.edu

**Abstract**

In this paper we analyze the performance of a traditional parametric search system and compare it to a system using an in memory auxiliary index. An analysis shows traditional database-based parametric systems incur a huge time hit due to disk accesses. We show that using an in-memory index in such scenarios results in huge time savings.

## 1. Introduction

Parametric searches deal with effectively combining search over a traditional corpus with a query over data in a relational database. Such a capability can extremely useful in domains consisting of text documents and some associated metadata. Each paper has metadata such as the year of publication, name of author(s), journal of publication etc. associated with it. A typical query in such a system might involve a search over some words in the text as well as a subset of the metadata. Traditional systems divide the search text into two distinct parts – over the text, stored in an inverted index and the metadata stored in a relational database. A query received by the system is then divided appropriately and sent to the two components. The results obtained are combined and the final results presented back to the user. However such a setup can severely affect query performance. Since the metadata includes textual data, searches over such fields involve performing wildcard string matching. E.g. a query such as title = "data streams" will get transformed to either title = "%data stream%" or title = "%data%" AND title = "%stream%" to get the matching documents. Relational databases performance is extremely poor for such queries. In such a case the database accesses increases the total query time, severely affecting system performance.

In this paper, we present an effective alternative to the above mentioned setup. Since the total size of the metadata is small as compared to the actual text, we argue that it should be stored in an in-memory auxiliary index. Since memory is cheaper nowadays, large amounts of metadata can be effectively indexed. This approach results in significant savings since all the metadata gets searched within memory. This avoids any disk accesses which are much more expensive than memory reads. Hence now the querying is done over the inverted index (as before) and an in-memory metadata index. The auxiliary index can be further optimized by using appropriate data structures to provide optimal performance. We provide comparative results to show the system performance in both cases.

The organization of the rest of the paper is as follows:

- Section 2 describes previous work done in the field
- Section 3 describes our architecture and tools used
- In Section 4 we define the methods used to collect the corpus
- Section 5 describes evaluation methods used to compare system performance
- In Section 6 we discuss the results obtained.
- Section 7 details the conclusions.

- Finally, Section 8 describes possible extensions to the system and future work

## 2. Previous work

We found that the Lucene system provides the capability to index data in main memory using the RAMDirectory class. However due to the overhead of additional Lucene features, we decided to implement our own in-memory index. This enabled us to use highly efficient and compact data structure and hence improved query performance significantly.

## 3. System Architecture and Tools:

The Parametric Search System has two primary components:

a) The Indexing System that takes as input a PS or a text file with associated Meta data

b) The Search System that given queries executes them and prints the results.

The data that is indexed for the Parametric Search project is in the form of two text files: 1. a text file with all the free text of a research paper. 2. A text file with all the associated Meta data for a particular research document. (The process of getting PostScript files from the Citeseer website and their associated Meta data has been automated. See tools)

### a) Indexing Component

The Indexing component takes care of the following tasks:

- Add a document's text into the main Lucene index called as "mainindex" in this project.
- Add a document's Meta data (in the form of another text file with the same name but a ".meta" file extension) into the MySQL database.
- Furthermore, the indexing component stores a document's Meta data into a different Lucene index (called "metaindex")
- Iterate over each and every document in a directory and do the above.

The ParametricIndexer class acts as the interface to the user. It iterates over each and every paper (text document) in our repository and passes that file as an input into our IndexFiles class. This class takes care of inserting the document into the "mainindex".

Every document that is indexed is tagged with a unique document id. This document id is also added to every record in the database and every Meta data file so that the Searching component can easily correlate the free text document to the Meta data.

Finally, there is an AuxillaryIndex class that takes care of loading the Meta data in the "metaindex" into memory for the Search process (This class is actually used in the Searching component but is conceptually a part of the indexing process). The AuxillaryIndexerLoader class gets all the Meta data stored as fields in the "metaindex" and puts them into data structures in memory. Each Meta data item has a hash of keys and an associated index into the Vector of document id sets.

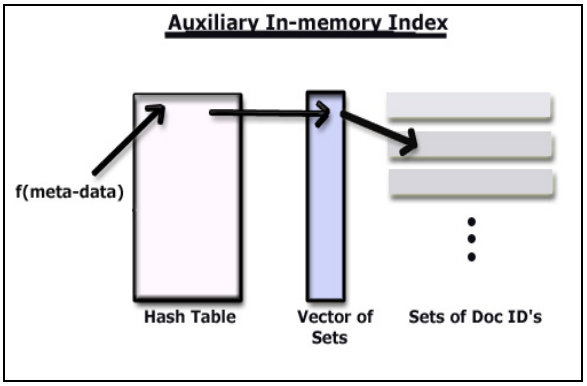



**Fig. 1**

**b) Searching Component**

The searching component takes care of the following tasks:

- Load the Auxiliary Index into memory at the start of a search session.
- Provide a Parser for user queries.
- Send the parsed query in appropriate data structures to the Individual Search Components (the Lucene text index Searcher, the Database searcher and the Auxiliary Index Searcher).
- Intersect the results of the individual searches and print the time taken by each component

A typical query is of the form:
"text=$txt_1$, $txt_2$ ,$txt_n$;
author=$author_1$,$author_2$,$author_n$;
title=$title_1$, $title_2$, $title_n$;
year=$year_1$, $year_2$, $year_n$"

In the above query, the entry for the text field is searched in the mainindex. All the other parameters are searched in the Database and the memory index. The QueryParser class can handle an arbitrary number of parameters (including 0) for each field.

Each individual search component (SearchFiles to search Lucene mainindex, DBquery to search MySQL database, AuxiliaryIndexLoader to search the in-memory index) takes in a Vector of the input query parameters. The component then finds a union of search results on each of these components and returns the results.

The results of the "text" query in mainindex and those of metadata queries through the database are intersected for the final results. Similarly, the results of the "text" query and the in-memory index are intersected to find the results.
(see Appendices for a diagram of this process)

**Tools**

The following tools were coded or downloaded to aid in data collection:

1) Lucene: The installation in the *cs276/software* was used to index the corpus. Features such as stemming and stop words were used in generation of inverted index.
2) MySQL: We installed and ran the MySQL database. Appropriate indices were created to speed up query performance.
3) ParseHTML.java was written to take in seed pages and extract the entire postscript file from that page and parse the metadata from the same page.
4) Pstotext: This C program was downloaded from http://research.compaq.com/SRC/virtualpaper/pstotext.html to convert postscript files into text files.
5) QueryGen.pl: Perl script to generate queries that contain a random number of fields (1 to 3) with random number of values for each field (1 to 9). The queries themselves are generated from random records from the database.

**4. Corpus collection**

The corpus consists of a set of a set of five thousand PostScript (.ps) documents along with their metadata fields. The documents are obtained from the http://www.citeseer.nj.nec.com website primarily from the Computer Science field. The following metadata fields are also stored for each document:

- Title of research paper
- Author(s)
- Journal in which paper was published
- Year of publication
- Keywords

It is possible that some metadata fields have some metadata missing. Additionally it is also possible for a document not to have any associated metadata – such a situation may arise if the data is simply not available or there is an error with the html page. However a set of metadata fields cannot exist without their associated document i.e. if the document is not present, the metadata fields will not be indexed in the database.

To obtain the documents, we used a web crawler and some seed pages of document links. Due to space constraints, we had to limit the document size of 6 MB – those over this were simply skipped. They system first tries to download a document and if it is successful, the associated metadata is also obtained and stored. The first four metadata fields (title, author, journal, and year) are present on the website and are copied verbatim. To obtain keywords for a document the following procedure is used:

- A list of important keywords is compiled for a given document collection – this includes search words that were used to find the documents.
- The "abstract" field is obtained from the document – this is a short description of the research paper.
- Each keyword (from the previously generated list) is searched in the abstract. In case of match, it gets included in the "Keywords" metadata field of the document. A maximum of five such keywords are searched and included.
- If less than five keywords are found, a simple heuristic is used – any word in the abstract of length 6 or greater is included in the "Keywords" list till five words have been obtained.

It is worthwhile to mention that the above heuristic to collect keywords works surprisingly well – we observed that most words obtained in this way were a good reflection of the corpus. This might be because most commonly used words are shorter in length and hence this heuristic filtering gives good results.

## 5. Evaluation Methods

To compare the performance of the database to that of the in-memory index, we indexed the available metadata in both MySQL and an auxiliary index. Similar queries were run against both sources and the time taken to return the result was measured. We did not measure the quality of the results at this time; however some basic observations are described in the next section.

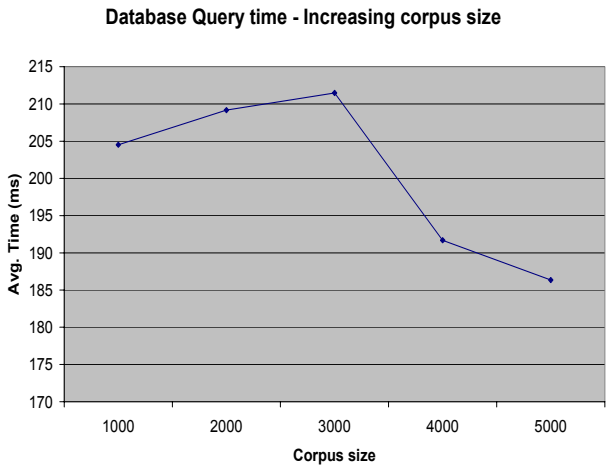
The time was measured for the two systems for two types of queries:

- The number of parameters that are searched on is varied
- The number of values per parameter is varied

We also measured the effect of varying the corpus size from 1000 documents and incrementing in steps of a 1000.

## 6. Results

Fig. 2 illustrates the effects of varying the corpus size on the total query time for the database. Initially we did not create any indices on the database.



However indices were created on all fields at a corpus size of 3000 documents. This explains the sharp drop in query time as show in the figure above.

Fig. 3 shows the results for same queries on the in-memory index.

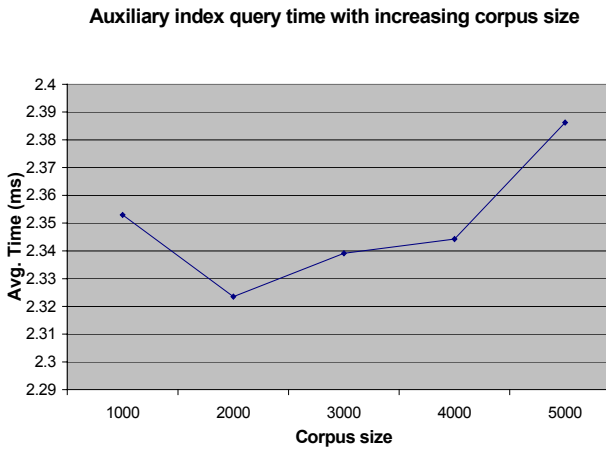



**Fig. 3**

As shown above, the query performance improves by a factor of 100-fold compared to the database query time. As expected, the query time increases with size of corpus. The query time for corpus size of 1000 is unexpectedly higher than that for 2000 probably due to an uneven system load. Since all test results were performed on public machines, we were unable to ensure constant system load.

Next we evaluated the effects of varying the number of values searched per field e.g. the search query included two year values. The searcher gets the results from the database and in-memory index for all matching documents containing at least one of the search terms. This provides a good emulation of range queries over the year field.

Fig. 4 shows the results obtained for running 100 range queries over the database. Each range query had an arbitrary number of parameters (1 – 3) and up to 9 different values per parameter. We used the random number generator package to ensure a random distribution.

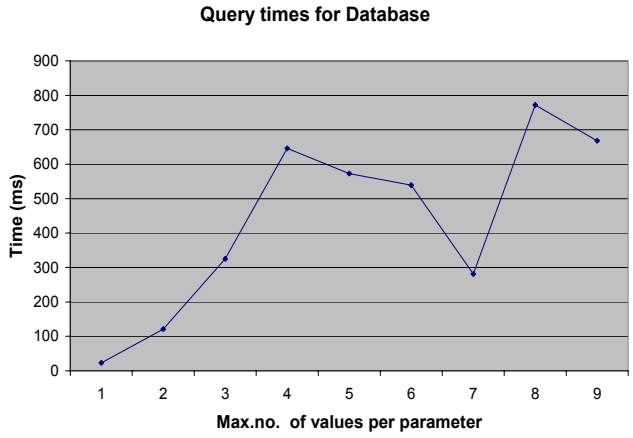



**Fig. 4**

The general trend is that the query time increases as the number of values per parameter increases. This is expected since the database has to perform increasing number of queries.

Similar results are obtained for the auxiliary index as shown in Fig. 5.

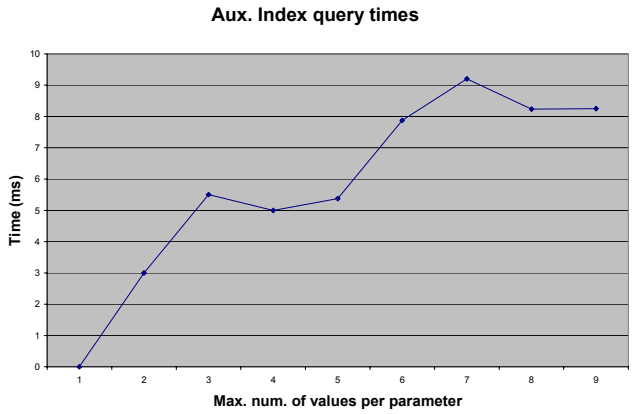



**Fig. 5**

Finally we show a comparison of the time taken by Lucene to search over the free text and that taken by the database to search the metadata terms. The result is show in Table 1 below. The table represents a small sample of all the queries run.

| Time taken by Lucene (ms) | Time taken by the database (ms) |
|---|---|
| 126 | 132 |
| 158 | 664 |
| 40 | 390 |
| 114 | 79 |
| 93 | 1144 |
| 190 | 1425 |
| 106 | 1640 |

**Table 1**

The table conclusively shows that the database contributes a significant chunk of time to the total query time.

## 7. Conclusion

As shown previously, the in-memory index results in a significant decrease in the time spent on searching the metadata. Table 1 shows that the database is a bottleneck in the entire search process. Even in the most pessimistic scenario where the time taken by Lucene is similar to that of the database, an auxiliary index would cut down the total search time by half. Hence we advocate that the database be replaced by an in-memory auxiliary index. We feel that with the decreasing costs of main memory, an in-memory auxiliary index is a viable option.

## 8. Limitations and Future Work

The auxiliary index can result in a significant saving in time if sufficient memory is available. In our case, the entire auxiliary index fit in main memory, hence the search process did not involve any time-consuming disk accesses. However, even if main memory is limited, data compression techniques can be used to optimize the search.

Similarly the database access can be sped up by maintaining a cache of previous 'n' searches. This can prove to be especially useful in the case of a corpus consisting of research documents since many keywords are repeatedly searched.

We used a single Lucene index to store all the free text. This process can be slow and time consuming. We implemented a naïve distributed indexing scheme using Lucene – this had to abandoned due to data corruption issues. However we feel this can potentially lead to a significant reduction in total indexing time.

## References

[1] Citeseer: An automatic citation indexing system

*C.Lee Giles, Kurt D.bollacker, Steve Lawrence*

[2] Using a Relational Database for an Inverted Text Index

*Steve Putz*
*Xerox Palo Alto Research Center*

**High Level System Implementation**