NERSENG: Query Analysis and Indexing

Carsten Kropf, Bertram Schlecht
Institute of Information Systems
D-95028, Hof, Germany
{carsten.kropf, bertram.schlecht}@iisys.de

Abstract

This article describes the general setup of NERSENG, a search engine for named entity related web documents. The search engine is, in this case, mainly adopted towards analyzing the documents searching for person names occurring inside the textual parts of crawled documents. We explain the general search engine architecture as well as the occurrence and distribution of entities (person names) in queries and documents. The two major contributions of our work are on the one hand methods to automatically extract entities from unstructured queries and on the other hand an efficient indexing strategy for being able to deliver the search results fast to a query issuer.

1 Introduction

Search engines gained increasing interest over the last years. Major providers, like Google, Bing or Yahoo get billions of requests every day. These standard textual retrieval engines have to efficiently handle access to the fulltexts crawled and indexed before. Therefore, established technologies, like the inverted index can be used to efficiently explore the underlying data space.
These techniques have been investigated for a long time, now, and users can, nowadays, use a sophisticated set of methods to retrieve the desired results. Not only the deep web analysis, also the storage, retrieval and ranking parts of the search engines are very sophisticated, today.
However, in most cases, the standard search engines only search for occurrence of certain terms inside texts. These texts may also be searched for phrases. Yet, most queries are retrieved solely from an inverted index. Structured information, derived in advance, cannot be searched by most search engines, whereas there also exist approaches (like Google Knowledge Graph). However, these structured searches mostly rely on structured information and cannot intermix the data with searches for keywords. Thus, there is the lack of a possibility to search for, e.g., a person name and keywords related to this name.
There already exist methods for extracting structured information from unstructured texts, like named entity recognition (NER) technologies which are able to extract, e.g. a person or company name, from a fulltext part. Likewise, those information might also come from data stored as semantic annotations inside the texts. Based on this, there also exists the possibility to enhance the retrieval process by taking into account these portions of information also for searches.
Searches (except phrase searches) always focus a certain topic. This topic is then intermixed with certain keywords closely related to it. As an example, people searching for "Bill Cutting" (a role of the film Gangs of New York) might also want to know how much money the actor got for this role. In this case, the entity (person name) is mixed together with a query keyword to filter the articles about the person after this keyword, e.g. "Bill Cutting money". Most fulltext search engines would then retrieve data containing the keywords and not necessarily detect the entity name inside this query. For our example the user mostly gets results, which describe how to save money by reducing bills. Only a few of these results is linked to the entity "Bill Cutting".
This paper results from NERSENG (**N**amed **E**ntity **R**etrieval **S**earch **ENG**ine), a web search engine focussed towards exactly these objects of investigation is given. We try to build a search engine supporting simultaneous searches for named entities together with query keywords to construct a more sophisticated searching experience for users.
This paper describes current work carried out with focus on the following parts:

- Statistics of entities in search queries and documents (section 3)
- Description of the search engine (section 4)
- Detection of entities in search queries (section 5)
- Database and indexing scheme used for storing/retrieval of the documents (section 6)

2 Related Work

The index structure used for storing the data for enabling fast search operations is based on a B-Tree (B+-Tree) [Bayer and McCreight, 1972]. Hybrid index structures extending the functionality of base structures to enable fast access to heterogeneous data types have already been proposed for geo-textual application domains. Most of the structures focus towards Geographic Information Retrieval Systems. Examples for these structures are the (M)IR^2-Tree [Felipe *et al.*, 2008] or the bR*-Tree [Zhang *et al.*, 2009]. An overview of currently available and used techniques in the research area of spatial keyword query processing can be found in [Chen *et al.*, 2013]. Retrieval techniques which build the basics of the hybrid index structure, used here, can be found in [Göbel *et al.*, 2009] or [Göbel and Kropf, 2010] which also use Zipf's Law [Zipf, 1949] to distinguish between high and low frequently used terms. However, all of these structures are focussed on spatial in

combination with textual searches whereas this paper focusses on a combination between texts and entities occurring inside the textual parts extracted in advance.
A compressed trie [Morrison, 1968] is used for extracting candidates from the particular search queries.
[Cheng *et al.*, 2007] deals with query construction and ranking of entities in an entity-based search engine, though we do not want a user to learn a new syntax for search queries. [Guo *et al.*, 2009] describes a probabilistic approach for finding named entities in queries. However, we are of the opinion that within a search engine, a statistical approach is slower than using a trie. Within [Kumar and Tomkins, 2009] the behaviour of online search queries is discussed and it is shown that queries can be divided into different classes, e.g. URL-queries.

3 Entities in Queries and Documents

In order to show that an entity-based search engine adds some value compared to a traditional one we have to show that a certain percentage of queries contains an entity, in our case a person. [Kumar and Tomkins, 2009] show that 52.9% of all web queries contain a structured object, e.g. a product, a location or a person. But for our work we are temporarily only interested in the amount of queries containing a person. Since no entity-annotated corpus for search queries is, to our knowledge, freely avaible, we try to approximate this number, with different algorithms explained in the following.

3.1 Approximation of Search Queries with Entities

Named Entity Recognition usually considers the semantic structure of a text. However search queries have in most cases no semantic structure, e.g. from the query "Bill Cutting money" no conclusion can be made whether "Bill Cutting" refers to a person or not. For the approximation of search queries with entites we use an approach which disregards the semantics of the queries. It will be described in the following.
As a data base for the approximation we use the AOL Query Log[1], which includes a total of $\sim$ 36M search queries. Since the corpus contains many duplicate queries and this would influence our measurement results we removed redundant elements and created a list which contains $\sim$ 11M unique queries. Our approach to find persons in queries is a lookup in a name list. We first split queries at whitespaces into single words. After that we check if one of the words is included in the name list. At this point we ignore the capitalization of the single words and names, because web search queries mostly consist of lowercase letters. The list itself was created from the data of the 1990 U.S. Census list of surnames and first names[2]. This method shows the result that 50.64% of the queries contain a name. This outcome is based on the fact that some elements of the name list have ambiguous meanings, e.g. the list includes the last name 'in' which is one of the most common words in english. Due to this we tried to filter the list, removing all words which have ambiguous meanings. Our first approach uses WordNet (see [Miller, 1995]), a lexical database for the English language. We make the assumption that an element of the name list, which is also included in WordNet, has other meanings than just the name and remove the element from the list. Our result with this filtered list is that 13.75% of all queries contain a name. However WordNet includes also names, mostly of famous or historical persons, so our first assumption isn't quite correct and we removed words from the list which are actual names and have no other meaning. Consequently we implemented a second algorithm for filtering the name list (see algorithm 1).

Algorithm 1: filterList(names)

```
   // Generate the filtered name list
   FN from the given census name list
1  for $n \in names$ do
       // Generate similary word list
       from WordNet
2      S.add(n)
3      S.add(similarWords(name, $\alpha$))
       // Check the shape of each element
       of S. If it's a potetial name add
       it to C
4      for $s \in S$ do
5          if $hasNameShape(s)$ then
6              C.add(s)
       // Sum up the probability of every
       word in S and C
7      PS = sumProb(S)
8      PC = sumProb(C)
       // Check if the relation between
       PC and PS ist greater then a given
       factor $\beta$.  If yes add it to the
       filtered name list
9      if PS != 0 then
10         if PC / PS >= $\beta$ then
11             FN.add(n)
```

Thereby the method "similarWords" generates a list of similar words from the original name using WordNet. Initially, all synsets (a set of synonyms) of the "name" parameter are loaded from WordNet. Every word of a single synset is checked whether the Levenshtein distance (see [Levenshtein, 1966]) is less than the passed paramter α. In this case, the word is added to a return list which forms the set S together with the original name. A small α parameter allows only words which differ in a few letters from the original word, whereas a high α parameter, e.g. ten, allows completely different words. Thus, a set C is generated, which contains all the potential names of S. A word in S is a potential name if it passes the "hasNameShape" method, which checks if the first character the given word is uppercase and the following characters are lowercase, with the return value true. Then the summed word probabilities of the sets C and S are calculated and stored in the variables PS and PC. For this the word distribution, within the two corpora Reuters TRC2 and RCV1[3], serves as a data base. If the ratio of these probabilities is greater than β, the word is added to the filtered name list. In figure 1 the number of queries that contain entities, for different values of α and β is shown.
We see that the higher the selected values of α and β

[1]http://www.gregsadetsky.com/aol-data/, accessed 2013-06-20

[2]http://www.census.gov/genealogy/www/data/1990surnames/names_files.html, accessed 2013-06-20

[3]http://about.reuters.com/researchandstandards/corpus/, accessed 2013-06-18

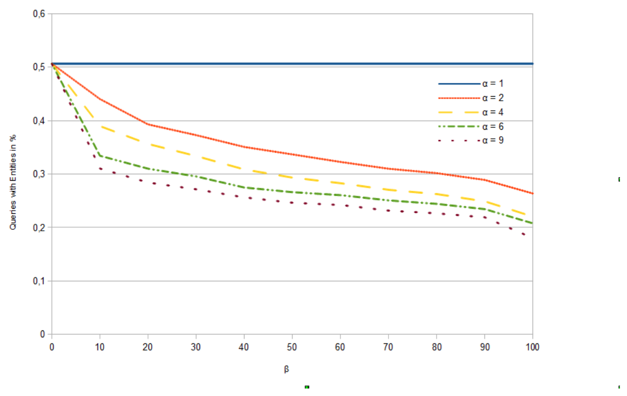

Figure 1: Queries with entities for different α and β values

are, the more names are filtered out of the list. From a certain level of the α-value the number of filtered names does not increase any longer , because every word from every synset is added to S. With a β value of 100% only names are accepted, for which the Sets C and S are identical. We cannot make definitive statements about accurate values of alpha and beta, at the moment. However, we have manually checked 2000 queries and 326 of them contain a person (16.3%). So we imply that a α value of nine and a β value of 95% brings the best results.

3.2 Distribution of Entities to Documents

To determine the distribution of entities to documents the Reuters RCV1 corpus was automatically annotated using the Stanford NER system (see [Finkel *et al.*, 2005]) which is part of the Stanford Core NLP[4]. As models we used the "english-left3words-distsim.tagger" for the part of speech tagger and "english.all.3class.distsim.crf.ser.gz" for the ner system. The corpus contains a total of 806K documents from those 491K include entities, in our case persons. Overall the corpus contains $\sim$ 486K entities. As $\frac{491000}{806000} \approx 61\%$ of the total document corpus contains person names as entities and our work focusses on crawling web sites containing news, we are optimistic that also in a realistic data environment a large percentage of the crawled documents will contain entities.

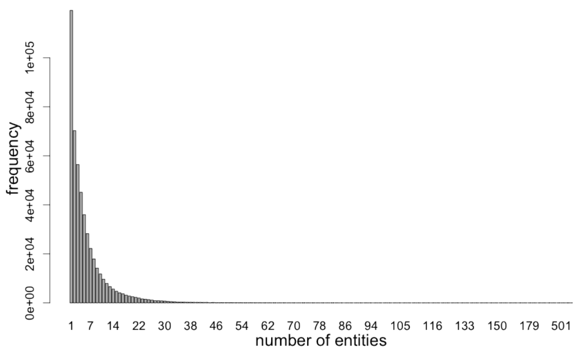

Figure 2: Distribution of Entities to Documents

[4]http://nlp.stanford.edu/software/corenlp.shtml, accessed 2013-06-20

The distribution of entities to documents of Reuters RCV1 dump can be seen in figure 2. The maximum number of entities inside documents is 1064. All documents from Reuters RCV1 that contain persons have an average size of 289.69 words and 5.73 entities. The distribution shows an extreme positive skewness which means that most of the documents contain less than or exactly 7 entities (third quartile). Therefore, we are optimistic that on the one hand building the hybrid index, described later, and (re-)building the trie is sufficiently fast.

4 Search Engine Architecture

As the main contributions of this work, entity detection and hybrid indexing, are embedded in a search engine application, we will describe shortly the process to get documents from the web, analyze the data and store them in a database for retrieval via a search engine.

Figure 3 displays an overview over the entire search engine

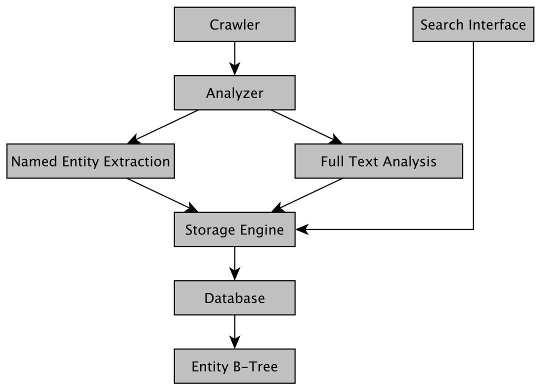

Figure 3: Overview over the Search Engine Architecture

architecture with the particular main contributions of our research work embedded inside. The particular individual parts are described in a little more detail in the following subsections.

4.1 Crawling and Analyzing

The first two parts in the document generation are crawling and analysis of the crawled documents. Therefore, two possibilities are given to retrieve documents from the web. On the one hand, there exists the possibility to crawl the web documents just by seeding a list of uniform resource identifiers (URIs) which is then input into a deep web analysis. Then, the list of URIs to be crawled from each initial URI is generated sequentially and queued to be crawled in a later step. On the other hand, there exists a module to crawl based on RSS feeds. There again, an initial seed of sites is generated (manually) whose RSS information are retrieved repeatedly based on the time restrictions given by each individual site which offers the RSS feed information. The crawler may, based on the given information, therefore download portions of information from the web which are analyzed in a further step. The boilerpipe[5] library is thereafter used to extract the boilerplate free fulltext. This approach serves to derive the "really" relevant information from the crawled sites to remove, e.g. advertisements or link lists from the web page content which is then used for

[5]https://code.google.com/p/boilerpipe/, accessed 2013-06-10

further analysis steps.
The analysis needs to take into account the particular features to be extracted from the texts. These should be prepared in order to store them in the database environment, we will describe in the following. The fulltext part is split into individual terms using normalization (e.g. character normalization and stemming) and thus pre-processed for the use of the hybrid index structure. This process is executed using the Apache Lucene[6] library. The other part, besides the fulltext to be extracted from the crawled web documents are the entities. As described in section 3, the named entities in our case with focus on person names are extracted from the documents using the Stanford NLP library modules with appropriate models to analyze the web articles.

4.2 Indexing Environment

For indexing the data crawled and analyzed in advanced, we use a relational database system. The h2 database[7] is used as the database server, in this case. This section details the setup and adaptions made for the h2 database to run properly for storage of the search engine.
The database itself was extended to allow custom index structures to be set up on tables which are loaded from external places. As the database is implemented in Java, there exists a mechanism to load the index structures from jar files and instantiate them as access methods for certain columns.
For enabling the h2 database to load external index structures from jar files, two tables are introduced which store information about the access structures:

- *INDICES*, storing the information about the index structures available for the database and
- *OPCLASSES*, storing information about so called operator classes which make the index structures, which are generalized, work for specific data types.

Table	Column	Meaning
INDICES		
	ID	Primary Key
	NAME	Name of the index structure
	FILE	File name to find the index in
OPCLASSES		
	ID	Primary Key
	NAME	Name of the operator class
	FILE	File name to find the operator class in
	INDEX	Index reference (Foreign Key)

Table 1: Table Definition of the custom tables for index structure dynamic loading

The indices table stores information about the index name and the file name which contains the index structure. The table *OPCLASSES* stores information about a certain operator class which is linked to an index structure. The operator class serves as concrete implementation of certain functionality required by a particular index structure to be able to deal with a certain kind of data stored inside the database table. An operator class for a B-Tree implementation could thus provide functions for comparing or ordering several values.
Besides the two database tables, also the SQL command for creating index structures was extended inside the h2 database so that the names of the index structure (as seen in table 1) and the name of the respective operator class may be passed to this command. The syntax of the modified

```
CREATE { [ UNIQUE ] [ HASH ] [ indexType ]
 INDEX [ [ IF NOT EXISTS ] newIndexName ]
 | PRIMARY KEY [ HASH ] } ON
 tableName ( indexColumn [,...] )
 [ USING opclassType ]
```

Figure 4: Syntax of the modified Create Index Command

command can be seen in figure 4. The fields *“indexType”* and *“USING opclassType”* are the basic extensions done to the command. The index loader then looks inside the indices table for the name specified by *“indexType”* and the associated operator class from opclasses given in *“USING opclassType”*. Therefore, for setting up an index called “bitlistbtreeindex” on a table called “documents” using the column “doc” and the associated operator class “docopclass” can be done as in the following statement:
`CREATE bitlistbtreeindex INDEX ON documents (doc) USING docopclass.`
These are the general adaptions we have done to the h2 database to enable it to extend the index structures currently available and to create new index structures based on the h2 basic definitions.

4.3 Query Interface

The query interface we present to the user does not differ from standard query interfaces from retrieval engines. Our goal is not to let the user select a certain type of entity to search for based on a pre-defined input field but to parse the query components directly from the query. Therefore, we decided only to have one field to enter the keywords in and not distinguish between fulltext keyword part and named entity (person name) part. The query interface is directly connected to the storage engine which serves the data stored inside the database using the hybrid index structure, described in section 6.

5 Entity Detection in Search Queries

Extracting the existing entities from the unstructured search queries is one of the most important aspects in this search engine. We try to achieve this without the need of specifying additional input fields. Therefore, we need methods to separate the query keywords from entity search candidates in unstructured queries.
Hypothetically, each of the used query keywords might be a candidate for an entity to be found. Initially, every sequence of query keywords may be regarded as a candidate for the entities. Therefore, a mechanism has to exist to filter candidates in order to leave the search effort low when communicating with the database. We chose a compressed trie variant stored in main memory for entity

[6]http://lucene.apache.org/core/, accessed 2013-06-19
[7]http://www.h2database.com/html/main.html, accessed 2013-06-07

candidate filtering.
This compressed trie is set up on top of the entities extracted from the already stored documents inside the existing database. This structure might also be used for auto-completion functionalities in the future. Based on the document statistics used in our test setup, we built the compressed trie to determine the resource allocation inside a real world scenario.
For approximating the performance of the compressed trie, we choose to evaluate the annotated RCV1 corpus (see 3.2).
The different candidates are generated based on the assumption that entities containing more than one word are always written in sequence. Consider, e.g., the query phrase $S = (\texttt{Barack}, \texttt{Obama}, \texttt{election})$ as input to the search. Therefore, we generate a set of a set of n-tuples $Q = \{T_1, T_2, \ldots, T_n\}$ with cardinalities $|Q| = n$ and $|T_i| = n - (i-1), 0 < i \leq n$ from the initial n-tuple of keywords $S = (w_1, w_2, \ldots, w_n)$, where w_k is the keyword at position k and n is the length of the search query. For each $T_i = \{E_{1,i}, \ldots, E_{n-(i-1),i}\} \in Q$ applies that $E_{k,i} = (w_k, w_{k+1}, \ldots, w_{k+i}), k \leq n - i$ is a tuple of length i. That means, that from an initial query, we generate a set of all candidates $C = T_1 \cup T_2 \cup \ldots \cup T_i$ with the cardinality $|C| = \sum_{x=1}^{n} x$ which are subsequently checked for being entities. From a human point of view, it is probably obvious that "Barack Obama" is the entity meant by the query issuer. The approach, described above, then generates the following set of tuples: $Q = \{T_1 = \{(\texttt{Barack}), (\texttt{Obama}), (\texttt{election})\}, T_2 = \{(\texttt{Barack Obama}), (\texttt{Obama election})\}, T_3 = \{(\texttt{Barack Obama election})\}\}$.
As there exist only $\sim$ 486K entities, generated by the approach, described above, in total (inside RCV1) and, based on our measurements, the compressed trie consists of $\sim$ 618K nodes, storing this structure in main memory results in $\sim$ 200 MB. The memory measurement is carried out in Java and thus can only serve as an approximate value. Storing this structure in main memory should not result in any problems on currently used server machines.
The check for candidates is performed using the previously generated compressed trie. Each of the generated tuples is looked up inside the trie and if it is found there, it may be considered a final candidate entity. For being able to retrieve the candidates as described before , they are also stored in the database like this while extracting the information from the documents. These candidate entities are then sent to the database, currently using an *"OR"* conjunction, and retrieve the data using the hybrid index structure, described in section 6. Additional ranking procedures might, in future, take the presence of multiple individual entities into account. The ranking procedure and final retrieval process is, however, not yet implemented and still subject of discussion.
The average query size, determined from AOL Query Log, is ~ 3.014 words per query. Therefore, if we ceil the value to 4, we get a total average amount of entities to be checked first in the trie and afterwards inside the hybrid index of $\sum_{x=1}^{4} x = 10$.
We executed short performance measures on the compressed trie. The most important property, here, is the insertion time as it affects the entire process during crawling and the search effort of a trie is well known as $O(1)$ because it is only related to the length of the input. Therefore, we instantiated the compressed trie from the database to create it freshly (e.g. in case of a data loss, when it is simply contained in main memory). We used, again the prepared Reuters data and imported all known entities into the trie implementation. Importing the existing data of $\sim$ 486K entities extracted as described before, the insertion procedure took in average $\sim 7500ms$. The entire process of querying the data and inserting them into the trie took $\sim 33s$ in average. These numbers show that inserting the data into the trie on the fly while writing them to the database does not cost much as inserting one element into the trie results in $\frac{7500ms}{486000} \approx 0.015ms$. For an average document of $\sim$ 5.73 entities this makes a total average effort of $0.086ms$ for handling the trie per document. This seems to be reasonable as the remaining operations take much longer (e.g. analysis of the documents or inserting them into the database).

6 Indexing and Database Storage

Besides other tables for the search engine architecture described in section 4, there exists one table storing the documents in a denormalized form. This table stores the data to be handled by the index structure. As a full text search and a search for an entity is supposed to be done simultaneously, both parts of data have to be stored inside the table to compute a document representation from these which is thereafter indexed by the specialized index structure, described in the next subsection. Therefore, the table consists of a fulltext part where the text, pre-processed by the application using textual normalization, and an entity part storing an array of named entities associated with the particular textual document are stored. An overview over the main table to put the index on can be seen in table 2. The index itself is constructed on top of the computed column "doc" which is a composition of the two column "words" and "entities". The entities are stored inside an array of string values (varchar).

Table	Column	Meaning
DOCUMENTS		
	ID	Primary Key
	WORDS	TXT object (normalized words)
	ENTITIES	ARRAY of entities (varchar)
	DOC	computed column from words and entities

Table 2: Table Definition of the document table

The queries are supposed to retrieve documents in which the queried entities occur and the textual part is also present. To support this type of queries efficiently, a new hybrid indexing method is introduced enabling efficient retrieval of this kind of data. The index structure used here supports the retrieval of combined data of entities and textual content.
In this case, we use a hybrid index supporting storage of the heterogeneous data types directly inside one structure. This technique can be used to support searches of the given types efficiently. Therefore, there is no need to search in two different tables or access structures and generate intermediate result sets which are intersected at the end but to directly navigate to search results fulfilling both types of search criteria (entities and keywords).

6.1 Architecture

The index structure, used here, is similar to the one described in [Göbel *et al.*, 2009]. The main changes are based on the fact that it is implemented in a real world database system and the change of the augmented structure. In our case, a base index structure whose elements will be augmented with the bitlist has to be able to handle entities (in string representation) efficiently. Therefore, as the main hybrid structure, we chose a B-Tree (or B+-Tree, more precisely) whose elements are augmented with a bitlist which represents the sets of terms valid inside the subtree pointed to by a specific B+-Tree element.

Figure 5 shows a conceptual overview of the components

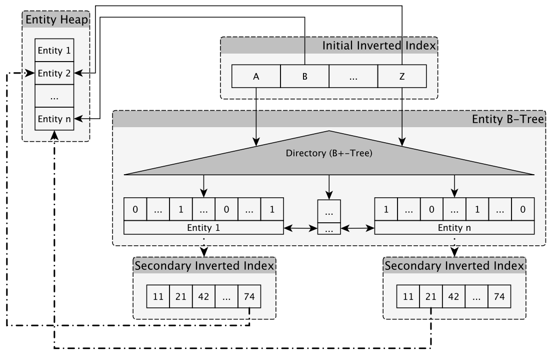

Figure 5: Conceptual Overview of the Entity B+-Tree

of the hybrid B+-Tree.

There are some additional structures used for administration of the index which are omitted in this graphic for simplicity reasons.

The central component of the index structure is the entity B-Tree. It stores combined keys of entity references and the bitlists which represent the presence or absence of terms from the initial inverted index. The initial inverted index itself stores all terms contained in the words column of the database table. If a term has a frequency higher than a pre-defined limit, it is moved into the hybrid part of the entity B-Tree and gets assigned a certain term index. This term index can then be used as identifier inside the bitlist. Primarily, the entities are not inserted into the entity B-Tree. They are stored in the so called "entity heap" where sequential comparisons of items which do not yet exceed the previously mentioned limit with respect to the absolute term frequency. If the limit is exceeded by one term, all entities referred to by each document the particular term points to are inserted into the entity B-Tree and the respective bits are set referring to the term identified by the assigned term index.

6.2 Algorithms

The insertion is done as described in algorithm 2. First, the items are added to the entity heap. After this operation, the references always point to the entity heap, which stores the document to entity assignment, and not the direct references to the documents any more. After that, the insertion operation continues to insert the full text terms into the initial inverted index (line 2). This operation also generates the list of terms and assignment to the entity heap references which exceed the artificial limit. If the references of one particular term does not exceed the artificial limit, the references are stored directly inside the initial inverted index. After that, assignments are generated which point from one entity to all term indices referring to lists of entity heap references to pre-calculate the operations to be performed at the secondary inverted index. The elements of this assignment list are then distributed into the entity B-Tree where first each entity element is inserted (or already found) and then the term index to document list entries from the particular assignment are inserted at the respective secondary inverted index. After that, the "adjustTree" method of the B-Tree is executed. This method performs the standard B-Tree operations. Additionally, it is extended to perform adjustments on bitlists to update the B-Tree, correctly for being able to descend to a subtree which has a certain term code set and additionally fulfill a search condition focussed towards an entity.

Algorithm 2: addDoc(doc)

```
  // add the entities to the entity heap
1 ref = entityHeap.add(doc.getEntities())
  // add the terms to the initial inverted index using the reference from the entityHeap
2 overflowTerms = initialInvInd.add(doc.getTerms(), ref)
  // generate assignments of entities to terms and documents
3 assignments = generateEntityTermAssignment(overflowTerms)
4 for $entry \in assignments$ do
      // insert the entity into the B-Tree
5     leaf = insertEntity(entry.entity)
      // add the term code to documents assignment at the secondary inverted index
6     appendCodesDocs(leaf, entry)
      // adjust the B-Tree, update the keys, nodes and bitlists
7     adjustTree(leaf)
```

Searches are executed as described in algorithm 3. It starts using a list of terms and a set of entities to be contained in the documents inside the result set. First, the initial inverted index is searched for the terms and entities. If one term does not exceed the artificial limit regarding its absolute frequency in the document collection, it is filtered sequentially for containment of the set of entities. If the frequency is higher than the artificial limit the term code is returned. If the set of term codes is empty, either nothing has been found or all terms were found inside the initial inverted index and the intersections are already calculated. Otherwise, the search is continued inside the hybrid index. In each element there, checks are performed if the element satisfies the search condition for an entity and the term index obtained from the initial inverted index, simultaneously (line 5). Afterwards, the final result set is built by searching in each secondary inverted index for the set of term codes obtained from the initial inverted index. In this step, the entities to retrieve can be simply ignored as the hybrid search part only delivers valid leaf nodes inside the entity B-Tree which already have to satisfy the entity search condition. We also know that there exist entries satisfying both search conditions as the bitlist represents exactly this behaviour.

Algorithm 3: search(terms, entities)

```
   // search references in initial
   inverted index
1  entries = initialInvInd.search(terms, entities)
2  if $entries.termCodes = \emptyset$ then
       // No term codes (term indices)
       available, so the final list is
       already built by sequential
       filtering
3      return entries.documents
4  else
       // Search for entities/bits in
       B-Tree (obtain leaf nodes)
5      leafEntries = hybridSearch(entries.termCodes,
       entities)
       // Search attached secondary
       inverted index structures for
       elements that satisfy all
       predicates
6      resultSet = searchSecondary(leafEntries,
       entries.termCodes)
7      return resultSet
```

7 Conclusions and Future Work

In this paper, we presented a search engine for person names and full texts intermixed. The approach used here, may be extended to the use of named entities, in general. The main focus was on the extraction of named entities from unstructured queries as well as database indexing. As this is project still continues, there are still open questions to be answered in future. A subject of investigation in the future will be the proper ranking which could also be integrated directly inside the retrieval process of the index structure. Another possible subject of future investigation is the real distribution of entities inside search queries as, currently, we take already present queries from AOL log. So, in future, when the system is finally running, it is probably more meaningful to investigate "real" queries put to the search engine in order to be able to analyze the real performance of our approach. We also want to compare the current trie approach for entity extraction with probabilistic methods.

References

[Bayer and McCreight, 1972] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173 – 189, 1972. 10.1007/BF00288683.

[Chen *et al.*, 2013] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: an experimental evaluation. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 217–228. VLDB Endowment, 2013.

[Cheng *et al.*, 2007] Tao Cheng, Xifeng Yan, and Kevin Chen chuan Chang. Entityrank: Searching entities directly and holistically. In *In VLDB*, pages 387–398, 2007.

[Felipe *et al.*, 2008] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. *International Conference on Data Engineering*, 0:656–665, 2008.

[Finkel *et al.*, 2005] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[Göbel and Kropf, 2010] Richard Göbel and Carsten Kropf. Towards hybrid index structures for multi-media search criteria. In *DMS'10*, pages 143–148, 2010.

[Göbel *et al.*, 2009] Richard Göbel, Andreas Henrich, Raik Niemann, and Daniel Blank. A hybrid index structure for geo-textual searches. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1625–1628, New York, NY, USA, 2009. ACM.

[Guo *et al.*, 2009] Jiafeng Guo, Gu Xu, Xueqi Cheng, and Hang Li. Named entity recognition in query. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, pages 267–274, New York, NY, USA, 2009. ACM.

[Kumar and Tomkins, 2009] Ravi Kumar and Andrew Tomkins. A characterization of online search behavior. *IEEE DATA ENGINEERING BULLETIN*, 32(2):3–11, 2009.

[Levenshtein, 1966] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[Miller, 1995] George A. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

[Morrison, 1968] Donald R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 10 1968.

[Zhang *et al.*, 2009] Dongxiang Zhang, Yeow Meng Chee, Anirban Mondal, Anthony K. H. Tung, and Masaru Kitsuregawa. Keyword search in spatial databases: Towards searching by document. *Data Engineering, International Conference on*, 0:688–699, 2009.

[Zipf, 1949] George Kingsley Zipf. *Human Behaviour and the Principle of Least Effort: an Introduction to Human Ecology*. Addison-Wesley, 1949.