Architecture

Objectives

- Students should understand the architectural requirements of a search engine.
- They should be able to describe a search-engine architecture at a high level.
- The framework for the SimpleIR engine described throughout this book is also described.

1 High-Level Requirements

Before describing the detailed implementation of our system, this chapter discusses the architecture of a search engine. The architecture is designed to accommodate the following high-level requirements:

- Scalability: The architecture must be scalable to an arbitrarily large document collection. It cannot be limited in any way by the size of the collection.
- Index Efficiency: Indexes must be built in a "reasonable" time frame. A minimum goal of one gigabyte an hour is required.
- Query Efficiency: Users must be able to specify the amount of time they are willing to wait for a query. There is always a tradeoff between how long a query takes and how many relevant documents can be found. Given infinite time, perhaps a search engine could find the perfect documents. By making this a user-driven parameter, we allow users to indicate how much time they are willing to spend to find the perfect document.
- Query Effectiveness: Users must be able to find documents relevant to a query. A high effectiveness level is crucial.

The architecture is designed around these requirements. We first describe the architecture and then identify precisely how the requirements are to accommodate. The following key objects are needed for this architecture.

2 Key Architectural Components

To achieve these requirements two key pieces of software are needed: an indexer and a query processor. The indexer builds secondary access structures that prepare documents for efficient queries and the query processor takes a user query and retrieves a ranked list of documents that are deemed relevant to the query.

2.1 Index Builder

No search engine has time to do a sequential scan of a document collection. Instead, an inverted index is built. This is a mapping from each distinct term to the list of documents that contain the term. Typically, building the inverted index is an computationally intensive process. Many detailed algorithms exist to reduce the time requirements; we describe the key approaches in Chapter 7. Essentially, these algorithms use a fixed amount of memory and build as much of the inverted index as possible in memory and then write to disk when memory is full. Ultimately, a set of temporary files is produced by this technique. A merge process then occurs whereby all the temporary files are merged into a single inverted index.

Additional performance improvements during the indexing process occur in the parsing stage. Frequently suffixes like "ing" and "ed" are removed via stemming algorithms. Common words with little retrieval value such as "a", "an", and "the" are removed as well. By reducing the number of distinct terms, the inverted index process becomes less cumbersome. Finally, numerous compression algorithms exist to compress an inverted index into one-tenth the size of the original text. This is not done so much to reduce storage overhead (space is relatively cheap now), but to reduce the number of

I/O's required to build and query the index. In Chapter 3, we describe the parsing process in detail, and in Chapter 6, we describe more advanced parsing techniques such as light natural language parsing, part-of-speech tagging, and information extraction. In Chapter 4, we describe the basic inverted index construction process, and in Chapter 7 we describe more sophisticated index construction algorithms.

2.2 Query Processing

The query processor includes a simple GUI that allows users to enter a query as a list of terms without any Boolean operators. The query processor takes the query terms and retrieves a ranked list of documents that are deemed relevant to the query. Many scoring functions or similarity measures exist which, for a given query-document combination, compute the relevance of the query to the document.

Identifying good documents is called *relevance ranking* and numerous approaches (e.g.; vector space, probabilistic models, Latent Semantic Indexing, language models, etc.) exist to provide a measure of relevance. These measures are described in Chapter 5. The SimpleIR architecture allows developers to easily change the similarity measure without fundamental changes to the inverted index. These measures are identified as configuration parameters to the search engine. Additionally, the user may choose between from a variety of similarity measures at query time.

The real reason that relevant documents are not "found" by a simple string match with the query is that the "language" of the author of a relevant document does not match the "language" of the user issuing a query. Consider the example of "find documents about red cars" and a document that describes a "rose-colored automobile". The author

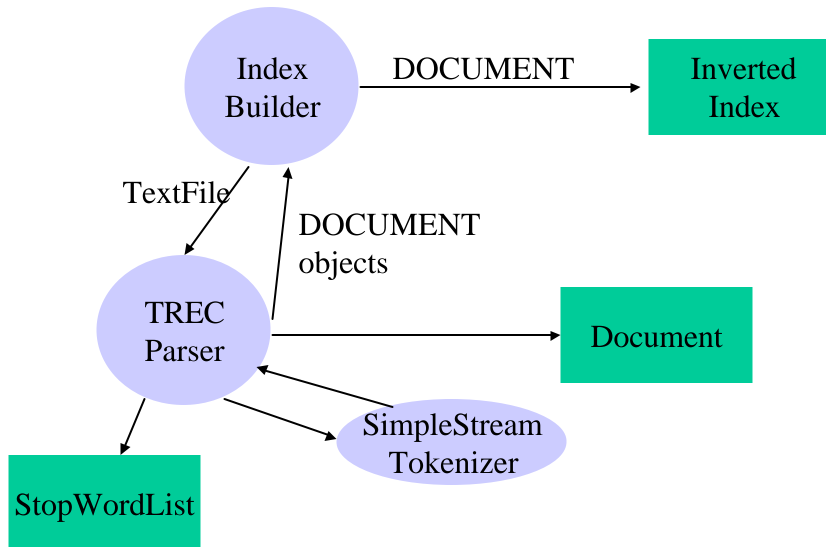
of the query does not use longer words like “automobile”, but despite this difference in tone, is certainly interested in documents about “automobiles”. People use the English language in a variety of ways – simply matching on tokens is doomed to match only a fraction of documents relevant to users. A standard search engine evaluation for the last ten years has repeatedly found that the best search algorithms find only 25-35 percent of the relevant documents. It is widely believed that a human will only find about 70-80 percent of the relevant documents (humans make plenty of mistakes when searching as well). The difference in accuracy is because a person can easily shift between the language of the query and the language of the document and do this translation far more accurately than a ranking algorithm that simply matches the words in the query to the words in the document.

To combat this language-mismatch problem, query expansion techniques find ways to add words to the query in the hopes that the new words can then be applied to the same ranking algorithm and more relevant documents will be found. The hope is that, for the term “red”, we will be able to expand it to include “rose-colored” and find the document that we would have missed had we simply looked for the term “red.” The main query expansion techniques are thesauri, semantic networks, and relevance feedback. These are discussed in greater detail in Chapter 8.

3 Architecture of the Indexer

We first describe the *indexer* architecture as seen in the Figure 5. We discuss the inverted index data structure and how it is typically used and implemented in information retrieval systems in Chapter 4. The indexer contains the following key objects:

Figure 5: Indexing Architecture



- **Document:** Stores information about a single document. The parser creates document objects. The key components of each of these objects are the unique document identifier and the distinct list of terms contained within the document. For each term, the number of occurrence of the particular in the given document is likewise noted.

- **InvertedIndex:** This object contains the entire secondary access structure built to efficiently find documents. Essentially, the inverted index consists of a term dictionary, with an entry for each distinct term in the document collection. For each term, a pointer exists to a linked list. The linked list contains nodes that store the document identifier for an occurrence of a term in a document, and the term frequency – the number of occurrences of the term in the document. The term dictionary is called the *index*, and the linked list is called a *posting list*.

- **IndexBuilder:** This is a worker object that drives the index construction process.

- **PostingListNode:** An object containing an individual entry in a posting list.

- **StopWordList:** An object that stores the list of frequently used words that are ignored by the indexer.

- **Parser:** This drives the parsing process. The parser is called and passed an input file of text; it responds by generating a list of document objects. Each document object has only the tokens that are to be stored in the inverted index.
- **Tokenizer:** This object handles the gory details of token recognition. How we handle things like "F-16" and other special characters is all done here. Additionally, the tokenizer indicates what *type* of token is found (e.g.; number, date, acronym, etc.). The indexer, to determine how the token is processed, uses this *type*.

The storage objects are the Document, InvertedIndex, PostingListNode, and StopWordList. These are just data containers that are used by the two key driver objects. The first is the Parser with assistance from the Tokenizer and the second is the IndexBuilder. We now describe the Parser and IndexBuilder in greater detail.

3.1 Parser

The Parser scans the input files and produces a set of Document objects. A Document object contains all the structured information unique to a document (e.g.; dateline, headline, author, date written) as well as useful information that helps locate the document when a user would like to view it. This includes the length of the document and start position or offset of the document in the input file.

The Parser uses the Tokenizer to identify unique tokens in the document and the *term frequency* of these tokens. This is needed for relevance ranking. Hence, each document contains a mapping of distinct terms to their corresponding *term frequency*. A document "big cat and big dog" contains a mapping of one for each term except for "big" which has a term frequency of two. The parser is a crucial aspect of the system as it determines what is or is not a word. We discuss the parser in much more detail in Chapter 3.

We note that language-specific grammars are often used for parsing (as done in compilers). We developed a grammar for SimpleIR, but our experience indicates that grammars are too slow for the rigors of a search engine. They are good for initial prototyping, but when millions of documents must be processed, their overhead becomes too expensive. Hence, we include a hand-written *tokenizer* instead of using one generated from a grammar. For completeness, we provide a grammar suitable for a search engine for small to medium sized document collection in Chapter 3.

3.2 ***IndexBuilder***

The IndexBuilder drives construction of the index. Input files are obtained from parameters passed to the index builder and files are sent to the parser for parsing. The architecture is designed such that the index builder can asynchronously send a file or files to different processors. Each processor can be parsing documents in parallel. When documents are parsed, a parsing instance returns a list of document objects to the parser. As these document objects asynchronously arrive, the index builder sends a list of document objects to the *add* method in the InvertedIndex. The index builder itself knows very little about the internal operations of the inverted index. It serves as a traffic directory between one or more instantiated parsers and the inverted index.

The InvertedIndex is described in detail in Chapter 3; compression techniques for reducing storage and access time from the inverted index are discussed in Chapter 7. Finally, when no more documents exist, the index builder calls the *write* method in the InvertedIndex that writes the completed inverted index to disk (For a large enough document collection, this might result in the merging of several temporary files build during numerous calls to *add* a document).

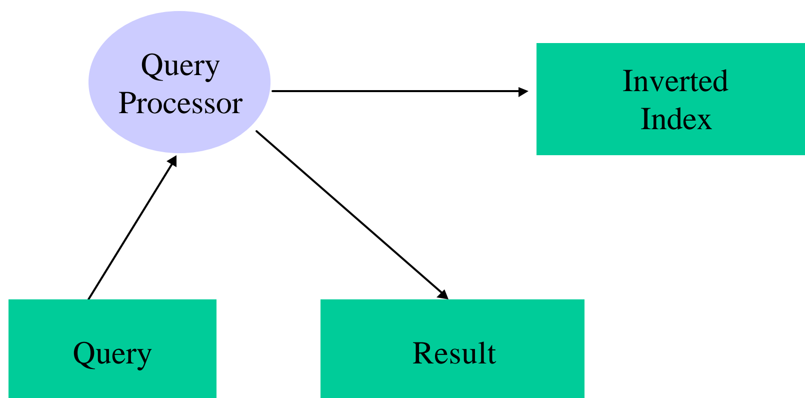
Building an inverted index is straightforward if we have enough main memory to hold the entire structure during the index construction process. Realistically, this is not a reasonable assumption. Typical servers may now have several gigabytes of memory, but it is quite possible that a user would want to index far more documents than what will fit into main memory.

Construction of the inverted index cannot assume there is sufficient memory. For clarity, we describe a simple *add* method in Chapter 3 that does assume the index can fit entire in main memory. In Chapter 7, we relax this constraint and describe more realistic algorithms.

4 Query Processing Architecture

In Figure 6, we depict the query processing architecture. The query processor contains the following key objects:

Figure 6: Query Processing



Query: Contains the parsed terms in the query.

Result: Contains a result document that responds to the query.

QueryProcessor: Takes a query and an InvertedIndex object as input and uses the InvertedIndex to obtain results. The Query object creates Result objects and a sorted list of these is created when query processing terminates. These results can then be sent to a GUI or to a file for output.

The query processor is the only object doing any work during query processing. A parser object is launched just as with document processing, and the query object is produced with the list of distinct terms in the query. For each distinct term in the query, the posting list for that term is retrieved via a request to the InvertedIndex. A similarity measure is computed and result objects are produced.

For instance a query "find all cats and dogs and animals like dogs" will result in a query object with a mapping of each term to its *term frequency*, namely the number of times the term occurs in the query. Many ranking algorithms use the query term frequency; so we need to store it as well.

After building the query object, we check to see if the index is already loaded into memory. If not, we load it. Again, we are assuming that we have enough memory for the index. In Chapter 6, we describe the more realistic scenario of not having sufficient available main memory. The query processor then computes a similarity measure between the query and the documents in the document collection. Finally, results are displayed to the user.

5 Example of SimpleIR Processing

The following four documents were taken from a well-known sample document collection. This document collection is part of the NIST TREC Collection.

```
<DOC>
<DOCNO> FT911-3 </DOCNO>
<HEADLINE>
FT  14 MAY 91 / International Company News: Contigas plans DM900m east
German project
</HEADLINE>
<DATELINE> BONN </DATELINE>
<TEXT>
CONTIGAS, the German gas group 81 per cent owned by the utility Bayernwerk,
said yesterday that it intends to invest DM900m in the next
four years to build a new gas distribution system in  the east German state
of Thuringia.
Reporting on its results for 1989-1990 the company said that the dividend
would remain unchanged at DM8.
Sales rose 9.4 per cent to DM3.37bn, but post-tax profit fell slightly from
DM31.3m to DM30.7m.
</TEXT>
</DOC>
<DOC>
<DOCNO> FT911-10 </DOCNO>
<HEADLINE>
FT  14 MAY 91 / World News in Brief: Brussels rioting
</HEADLINE>
<DATELINE> BRUSSELS </DATELINE>
<TEXT>
Almost 200 North African immigrants were arrested in Brussels at the weekend
during the worst racial rioting ever seen in the normally placid Belgian
capital.
</TEXT>
</DOC>
<DOC>
<DOCNO> FT911-11 </DOCNO>
<HEADLINE>
FT  14 MAY 91 / World News in Brief: Soviet deadlock
</HEADLINE>
<DATELINE> MOSCOW </DATELINE>
<TEXT>
The Soviet parliament was deadlocked over a bill to guarantee the right to
travel abroad.
</TEXT>
</DOC>
<DOC>
<DOCNO> FT911-13 </DOCNO>
<HEADLINE>
FT  14 MAY 91 / London Stock Exchange: Equity Futures and Options Trading
</HEADLINE>
<TEXT>
LONDON derivatives continued to trade under the shadow of Wall Street,
opening weaker following the decline in the US in the previous session and
then slipping still lower as New York extended its slide in London hours
yesterday.

The UK equity futures market began weakly and helped to pull the stock market down. The premium of the June FT-SE 100 index
over the spot index narrowed from 20 points to around 10 on selling by independent traders.
At one stage, the futures market attempted to rally and succeeded in widening the gap over the share market to 20 points. But with
on the defensive, prices were again marked lower and by the close June was back to less than a 10-point premium to the spot market.
The June FT-SE closed at 2,494, down 42 points on the day. Its premium finished at 6 points, against fair value of 15.
In traded options, the stock market's inability to break out of the range of the past fortnight prompted investors to sell calls against
stock holdings. Elsewhere, there was a buyer of 1,000 Asda July 130 calls; a buyer of 500
GrandMet October 650 puts; and a buyer of 250 Barclays June 460 calls.
The Euro FT-SE was boosted by a seller of 1,000 July 2,625 calls and a buyer
of 1,000 July 2,525 puts.
</TEXT>
</DOC>
<DOC>
```

Parsing these four documents produces the following document objects. We turned on some debugging statements to watch the processing of these four documents; the inverted index is printed after each one.

```
Adding document 0
about to add these terms to index:
        slightly        1
        unchanged       1
        invest          1
        intends         1
        3m              1
        gas             2
        81              1
        dm8             1
        dm31            1
        yesterday       1
        dm30            1
        remain          1
        company         1
        dm3             1
        reporting       1
        rose            1
        contigas        1
        9               1
        results         1
        group           1
        37bn            1
        4               1
        utility         1
        east            1
        sales           1
        dividend        1
        years           1
        owned           1
        build           1
        post-tax        1
        fell            1
        dm900m          1
        cent            2
        distribution    1
        profit          1
        german          2
        thuringia       1
        7m              1
        system          1
        bayernwerk      1
        1989-1990       1
        state           1
```

```
Adding document 1
about to add these terms to index:
        belgian         1
        racial          1
        200             1
        capital         1
        weekend         1
        rioting         1
        placid          1
        north           1
        arrested        1
        worst           1
        immigrants      1
        african         1
        brussels        1
```

```
Adding document 2
about to add these terms to index:
        guarantee       1
        parliament      1
        soviet          1
        deadlocked      1
        abroad          1
        bill            1
        travel          1
```

Adding document 3
about to add these terms to index:

break	1
barclays	1
selling	1
stage	1
prompted	1
london	2
finished	1
slide	1
traded	1
previous	1
shadow	1
traders	1
pull	1
asda	1
continued	1
market's	1
ft-se	3
stock	3
grandmet	1
500	1
boosted	1
42	1
trade	1
market	5
460	1
range	1
seller	1
calls	4
10-point	1
points	4
100	1
decline	1
marked	1
gap	1
650	1
000	3
uk	1
closed	1
day	1
close	1
rally	1
attempted	1
hours	1
widening	1
weakly	1
extended	1
250	1
helped	1
october	1
derivatives	1
options	1
past	1
began	1
york	1
narrowed	1
20	2
prices	1
independent	1
futures	2
fortnight	1
6	1
june	4
session	1
weaker	1
holdings	1
2	3
1	3
sell	1
equity	1
15	1
premium	3
inability	1
10	1
buyer	4
opening	1
yesterday	1
lower	2
puts	2
succeeded	1
494	1
defensive	1
spot	2
euro	1
back	1
street	1
slipping	1
625	1
index	2
130	1
july	3
525	1
investors	1
share	1
wall	1
fair	1

puts : (3 2)
460 : (3 1)
profit : (0 1)
100 : (3 1)
owned : (0 1)
grandmet : (3 1)
10-point : (3 1)
brussels : (1 1)
opening : (3 1)
calls : (3 4)
650 : (3 1)
contigas : (0 1)
seller : (3 1)
dm8 : (0 1)
closed : (3 1)
immigrants : (1 1)
asda : (3 1)
belgian : (1 1)
shadow : (3 1)
250 : (3 1)
german : (0 2)
traders : (3 1)
dm3 : (0 1)
sales : (0 1)
market's : (3 1)
slide : (3 1)
african : (1 1)
london : (3 2)
narrowed : (3 1)
york : (3 1)
yesterday : (0 1) (3 1)
share : (3 1)
session : (3 1)
june : (3 4)
travel : (2 1)
uk : (3 1)
trade : (3 1)
build : (0 1)
helped : (3 1)
weaker : (3 1)
7m : (0 1)
placid : (1 1)
slipping : (3 1)
range : (3 1)
past : (3 1)
parliament : (2 1)
company : (0 1)
investors : (3 1)
continued : (3 1)
defensive : (3 1)
derivatives : (3 1)
625 : (3 1)
north : (1 1)
81 : (0 1)
previous : (3 1)
july : (3 3)
equity : (3 1)
remain : (0 1)
unchanged : (0 1)
wall : (3 1)
group : (0 1)
close : (3 1)
gas : (0 2)
gap : (3 1)
traded : (3 1)
abroad : (2 1)
weekend : (1 1)
sell : (3 1)
boosted : (3 1)
market : (3 5)
break : (3 1)
marked : (3 1)
independent : (3 1)
200 : (1 1)

The inverted index after parsing is completed is shown below:

Issuing a query on “yesterday market” initially obtains the posting list for *yesterday,* namely posting list (0 1), (3 1) and for *market,* namely (3 5). This indicates that *yesterday* occurs once in document zero and once more in document three. Similarly, *market* occurs five times in document three. By running this query, we obtain the following output

Step 1: Query Parsing

First the query is parsed; each token is matched against the stop word list.

Processing Query Token --> yesterday
Processing Query Token --> market

Step 2: Retrieve the Posting Lists

First the posting list for *yesterday* is obtained:

Current Posting List --> [ir.PostingListNode@15732be, ir.PostingListNode@15732ac]

For each entry in this list we update a score vector that keeps the score for each document. More on this topic is discussed in Chapter 4.

The first posting list entry is encountered:

tf --> 1
qtf --> 1
idfSquared --> 0.48
score[0] ==> 0.48

The first entry is document zero; the term frequency is one; the query term frequency (qtf) is also one. This is the number of times this query occurs in the document. The weight of the term is its *idf (more on this in Chapter 4, but for now, think of it as an automatically assigned weight).* The score vector for element zero (corresponding to document zero) is then updated.

Now, the second posting list entry is encountered:

tf --> 1
qtf --> 1
idfSquared --> 0.48
score[3] ==> 0.48

The term frequency of *yesterday* in document three is also one, so the computation is the same and the score vector at element three is updated. At this point, the score vector contains all zero entries except for 0.48 for document zero and document three. This makes sense. At this point, we have only encountered a single term that has matched the

query, and it has occurred only one time in document zero and another time in document three. Thus, currently,, document zero and document three are tied at 0.48.

We continue by processing the posting list for *market.*

Current Posting List --> [ir.PostingListNode@1572d63]
tf --> 5
qtf --> 1
idfSquared --> 1.92
score[3] ==> 10.08

Here we find the term frequency is five as *market* has occurred in document three five times. The query term frequency is one as *market* has only occurred once in the query. The score vector is updated for element three as the combination of term frequency, query term frequency and inverse document frequency: (5)(1)(1.92) = 9.6. The previous score value was 0.48; so our new score value is 10.08.

Step 3: Sort the non-zero Scores

Now we sort the non-zero scores and obtain:

Finished running query
Score[3] ==> 10.08
Score[0] ==> 0.48

6 Summary

We described the high-level architecture of SimpleIR with a running example. Essentially, we built an inverted index and stored it on disk. Queries are run and are able to quickly access the inverted index, and compute a measure of relevance to the query. Many algorithms exist for building and compressing the inverted index, computing the score, and quickly identifying the top scores. More details on these are provided in future chapters.

Exercises

1. Get SimpleIR loaded off of the book website. Compile it and set it up to index a test document collection found in test.doc.

2. Run several queries and verify that they are correct.

3. Try the query "to be or not to be". Why does it respond the way it does?

4. Try this same query on five of your favorite web search engines. What does this tell you about these search engines?