FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Exploiting User Behavior and Markup Structures to Improve Search Result Rankings

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Constantin-Claudiu Gavrilete
Matrikelnummer 0725195

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Mag. Dr. Philipp Leitner

Wien, 07.10.2013

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Exploiting User Behavior and Markup Structures to Improve Search Result Rankings

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Constantin-Claudiu Gavrilete
Registration Number 0725195

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.-Prof. Mag. Dr. Schahram Dustdar
Assistance: Mag. Dr. Philipp Leitner

Vienna, 07.10.2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Constantin-Claudiu Gavrilete
Lilienfelderstrasse 50 4/4, 3150 Wilhelmsburg

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Abstract

Popular web search engines, such as *Google*, rely on traditional search result ranking methods such as the *vector space model* or *probabilistic models* in combination with the famous *PageRank* algorithm. In the last couple of years personal document relevance got considered when ranking search results, as it improves the result quality significantly.

This thesis extends related work in the area of result personalization, by providing the concept and implementation of a keyword based personal search engine. Compared to other work, the personal relevance is measured by the user's activity with keywords of one document, such as clicking or hovering. A ranking algorithm is introduced, which considers keyword and document frequencies in the vector space model, combined with the interaction of those keywords to compute the document score.

The concepts are implemented as an HTML5 browser extension for Google Chrome, which actively measures the user's interaction with the visited content, without interfering with the normal surfing behavior. Querying for visited content retrieves stored documents and orders them according to their personal relevance.

An evaluation is conducted to test whether the behavioral ranking factors are significant enough for personal relevance. It is shown, that the interaction of the user with the document's content correlates with its relevance. Furthermore, a benchmark of *WebSQL* and *IndexedDB* as HTML5 data storage structures for *insert*, *update* and *search* operations reveals that the latter technology outperforms the former in almost every configuration.

Kurzfassung

Populäre Web Suchmaschinen, wie *Google*, basieren auf traditionelle Rankingmethoden der Suchresultate, wie z.B. das *Vektorraum Modell* oder *probabilistische Modelle* in Kombination mit dem berühmten *PageRank* Algorithmus. In den letzten Jahren wurde die persönliche Relevanz von einem Dokument auch für Rangbestimmung hinzugezogen, da dieser Faktor die Qualität der Ergebnisse signifikant verbessert.
Diese Diplomarbeit baut auf den Konzepten, der im Zusammenhang mit Resultatpersonalisierung stehender wissenschaftlicher Arbeiten auf und erweitert diese um das Design und Implementierung einer Keyword-basierenden persönlichen Suchmaschine. Im Vergleich zu anderen Arbeiten wird die persönliche Relevanz durch die Interaktion des Benutzer mit den Keywords eines Dokuments bestimmt, wie z.B. durch Klicken oder Bewegungen der Maus über das Keyword. Ein Rankingalgorithmus wird vorgestellt, der neben den traditionellen Keyword- und Dokumenthäufigkeiten im Vektorraum Modell auch die Interaktionen mit Keywords berücksichtigt, um die Dokumentwichtigkeit zu berechnen.
Diese Konzepte wurden als HTML5 Browsererweiterung für Google Chrome implementiert. Diese Erweiterung zeichnet die Interaktionen des Benutzers mit dem sichtbaren Inhalt eines Dokuments auf, ohne dabei seinem normalen Surfverhalten in die Quere zu kommen. Das Abfragen nach bereits gesehenem Inhalt liefert jene gespeicherten Dokumente in einer nach persönlicher Relevanz geordneten Reihenfolge zurück.
Eine Evaluierung wurde durchgeführt, die Aussage über die Signifikanz der gemessenen Faktoren für die persönliche Relevanz treffen soll. Es wird gezeigt, dass die Interaktionen des Benutzers in einem Dokument mit dessen Relevanz korreliert. Weiters wurde ein Performance-Benchmark zwischen den HTML5 Speicherstrukturen *WebSQL* und *IndexedDB* für *insert*, *update* und *search* Operationen durchgeführt. Im Test übertrifft IndexedDB seinen Kontrahenten in fast allen Konfigurationen.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

We live in a time where we are intentionally and unintentionally surrounded by information. Whether it is a billboard ad presented on a highway, the latest articles stuffed together on the front page of a newspaper or content we daily consume when we surf the internet. According to a study from the University of California, San Diego conducted in 2009 the average American consumes up to 34GB worth of content every single day [5].

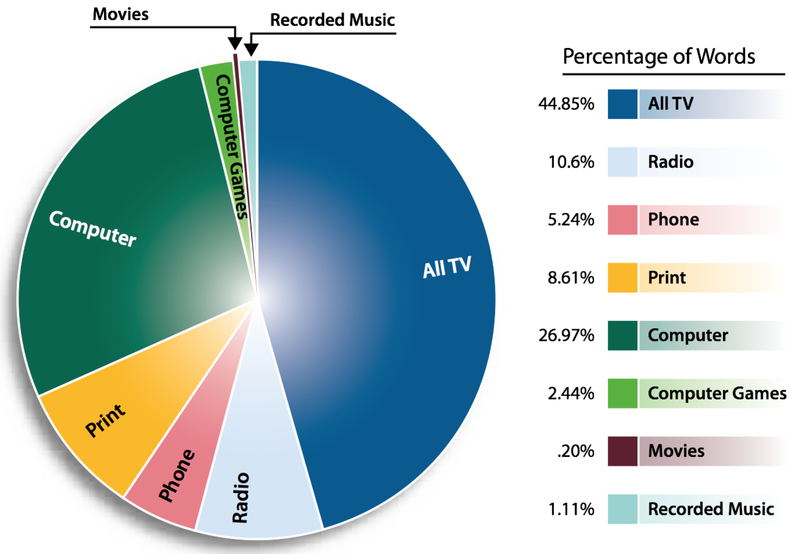


Figure 1.1: Relative information consumption in words [5]

This includes around 100.000 words of information. While the report states that not all of those words are processed consciously it claim that this information crosses our eyes and ears via multiple channels, as shown in Figure 1.1. Although TV still takes up the biggest piece of the pie, computers also represent a notable quarter of the overall information sources. According to an infographic crafted by Go-Gulf.com [24] the average US citizen spends 32 hours a month solely surfing on the internet. Figure 1.2 summarizes how that time is spent online.

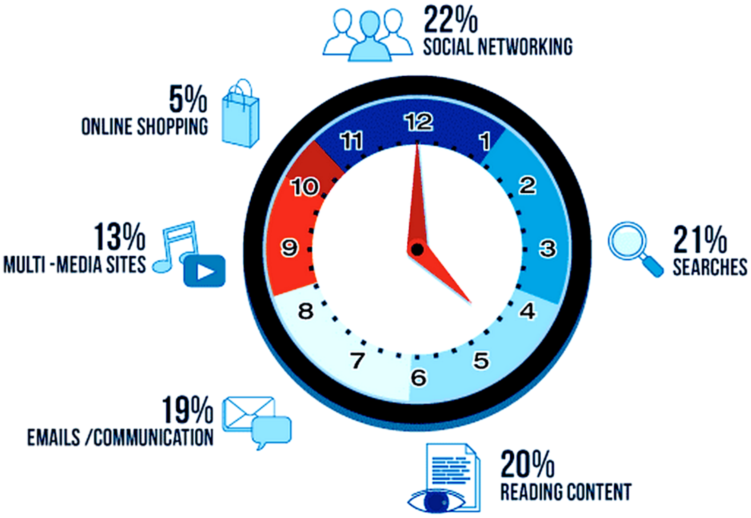


Figure 1.2: How people spend their time online. Cropped out of an infographic from [24].

They furthermore state that out of those 21% spent on online search only Google handles more than 1 billion search queries a day. Out of which, according to [77], 40% of all queries are about content the user has already seen in the past. Although the average internet user in the United States consumes a tremendous amount of information on daily basis they still use a substantial amount of time just to refind that content which has already been seen before.

While major web search engine vendors such as Google provide sophisticated algorithms to improve the web search experience they still lack the possibility to narrow the result list down to personal web history. Even if such a feature was implemented by Google or Yahoo!, they still would not be able to find web sites which were not visited through their search engine. A more promising approach is to search through the web history stored on each browser, but as of September 2013, no major browser vendor stores more content than the web page's title, its URL and some access statistics. A content based history search is simply not possible using conventional means.

1.1 Motivation

The presented statistics above and the lack of solutions for this problem motivated the design and implementation of a personal web search engine, which helps the user refinding already seen content, while not interfering with her normal web surfing behavior. In more detail, when searching for visited web pages, the results have to be ordered according to the user's personal relevance.

While many search engines query the web, for this solution it is indifferent whether the content was perceived from a web site on a desktop computer or from a web browser on a smartphone or from a GUI application like *Adobe Reader* [56].

The core concept of this search engine is to measure several factors, such as the duration one paragraph or image was displayed on the screen or which part of the displayed content the user clicks. Those metrics give insights about whether the user has actively perceived the currently displayed content on the her screen and therefore it has a higher personal relevance to her (see [28]). When the user later on searches for content, those query-matching bits of information are ranked higher in the result list which have a higher importance to the user. Content she has actively interacted with gets a higher ranking rather than content she might have only seen for a fraction of a second while scrolling through the document.

While some systems require some sort of structured data, modern search engines like *Google's web search engine* or *Apple's MacOS X Spotlight* do not have that restriction. They merely take the unstructured nature of documents and build an index around them, which allows to efficiently add entries and query for information. Although natural written language always follow a defined grammar, a developer does not have to predefine the structure of the document in order to use its content. Other search engines exploit the fact that their content uses markup to distinguish between headings and paragraphs. A web searcher could query for content which has the terms *"Pulp Fiction"* in its title and *"tasty burger"* in its body. While those queries could lead to fine grained results, the majority of users prefer to use a search engine that does not take separate queries for each possible markup, and rather searches through the whole document instead.

1.2 Problem Statement

The goal mentioned above implies that the user has to be able to quickly refind the content she is looking for. Therefore, an adequate ranking algorithm should order the search result list of documents according to the user's personal relevance for a specific query. This thesis proves the hypothesis, that the degree of personal relevance correlates with the user's interactivity with the document's content. More specifically, that relevance can be expressed by interactions like seeing, clicking or hovering over keywords of one document. E.g., when querying for *"Big Kahuna Burger"*, documents, in which those keywords have high interaction, signalize a high personal relevance.

To prove this hypothesis an empirical test is conducted, in which users are asked to visit different web sites related to one topic. Each web site has to be rated by the user, according to their information content for that specific topic. While examining those sites, their interactions with the page's content is measured. The hypothesis is considered proven, if the web site rankings obtained by the degree of measured interactions corresponds with the ranking created by the human judgments. Section 2.4 introduces a method to compare two different ranking orders, which is used to determine the degree of similarity between human and measured rankings. Figure 6.1 depicts the design of the experiment, which helps to prove the hypothesis.

1.3 Contribution

Current state-of-the-art search engines still rely heavily on traditional keyword / documents distributions, such as *Term Frequency Inverse Document Frequency* in the *vector space model* or *probabilistic models* to compute the rank of a document which matches a search query. Additionally, they utilize the the degree of incoming and outgoing links of one document to correct its rank, such as the famous *PageRank* [7].

Research has shown, that including user's personal relevance of one document individually, improves the overall result quality significantly [29], [58], [1]. While current rank algorithms of major search engine vendors are kept a secret, it is assumed that personal relevance is partly already considered. This statement can easily be tested by entering the same query on Google on two different computers, which are operated by two different users. The result ranking will differ in many cases.

This thesis picks up the approach of Guo and Agichtein [29] of utilizing mouse interactions on a web page as an indicator of personal relevance. While that method treats a document as the smallest entity of relevance, this thesis extends their approach to a keyword based level, declaring the keyword as the smallest entity. Documents are considered as relevant, if and only if the keywords in that document are relevant to the user. Additionally, not only the interactions with keywords is examined, but also their markups and how it contributes to the overall relevance.

As a side effect, with this method the most relevant keywords of one document can be extracted. This approach paves the way for other research in automatic meta-data extraction, based on the user's relevance.

1.4 Thesis Organization

The remainder of this section gives an overview how this thesis is organized:

- Chapter 2 gives an introduction about how state-of-the-art information retrieval systems work. In detail, it shows how documents can be stored so that a search by keywords can be conducted efficiently. Is also demonstrates concepts to find similarities between a search query and documents in the corpus, which is used to rank those documents. Additionally a method to compare ranking algorithms is presented.

- Chapter 3 shows related work presented by other research in the field of information retrieval, especially in search result personalization. The presented papers are compared with methods and concepts of this thesis.
- In Chapter 4, the design of the document storage structure is discussed, and also what factors are tracked while the user is consuming content. Based on those factors, a ranking algorithm calculates a score for each query-matching document, which uses the traditional *vector space model* [84], but also those tracked behavioral factors.
- Chapter 5 shows an implementation of the design concepts presented in Chapter 4. The search engine is realized as a browser extension for Google Chrome written in CoffeeScript, and utilizing *IndexedDB* as a HTML5 storage structure. Furthermore, this chapter discusses the anatomy of the extension with respect to fast information retrieval.
- In Chapter 6, the results of an empirical tests are analyzed. The focus of this evaluation is to find out how terms in a search query influence the relevance of documents containing those terms. Furthermore the goal is to extract a factor weight setting which ranks documents according to their captured relevance.
- Chapter 7 concludes the work of this thesis and gives a future outlook how to improve the relevance measurement.

CHAPTER 2

State of the Art Review

2.1 Introduction to Information Retrieval

Manning et al. define *Information Retrieval* in [57] as follows:

> *Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).*

The expression of a *document* as used in the definition above denotes an entity the user can search for. In the case of a web search engine, a document typically is a web site, or a file in the file search domain on the local desktop computer. Several documents form a *collection*, which can range from a humanly assessable size until large distributed data centers managed by dozens or hundreds of servers. One document consists of several *words*, which are also often referred to as *terms*. A term is the smallest unit users can form their search queries of. Only query matching documents should be retrieved by an IR system and ranked according to satisfy the user's information need.

Furthermore, Manning defines the goal of an IR system to *"Retrieve documents with information that is relevant to the user's information need and helps the user complete a task"* [15]. This definition yields that the correctness of an IR system cannot be determined objectively and always depends on the user's opinion and conception. Figure 2.1 depicts the concept of a simplified search engine helping to solve the user's tasks, as based on [15].

To demonstrate the collaboration of each component of the IR system, the user task at ① is set to *"Making a Quentin Tarantino movie night with his best movies"*. Solving this task does not necessarily require the usage of a search engine, but let us consider the user does not know already everything there is to know about the Tarantino universe. The information need ② for this task is to get information which movies of Tarantino are his best. This information need

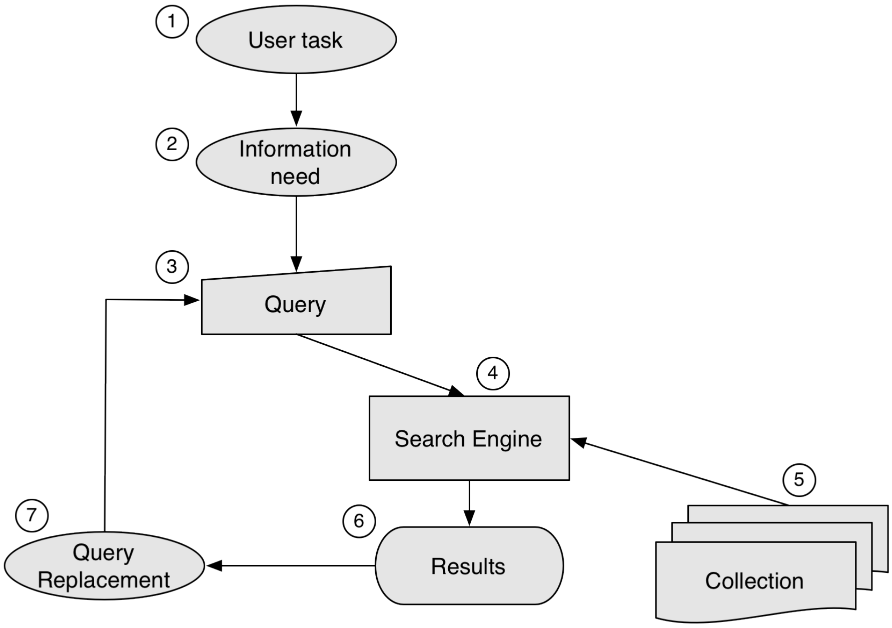


Figure 2.1: Simplified information retrieval system satisfying the information need to solve a task.

is formulated in a query ③, e.g. *"Best Quentin Tarantino movies"*. The entered query gets evaluated by the search engine ④, by methods and concepts shown in Section 2.2 and 2.3. The search engine takes all the documents relevant to the search query from the collection ⑤ and returns them ⑥. At this point, the user's information need can be satisfied with the retrieved documents, but in this example she is not and wants to refine the query ⑦ by *"Best movies directed by Quentin Tarantino."*. The loop ③ - ④ - ⑥ - ⑦ continues until the user has found the information needed to solve the task, or until she quits the process. Due to many reasons, such as distraction or frustration about poor search results.

Manning also identifies a problem of *misformulation* between step ② and ③, where the user knows exactly what she wants, but is unable to form a query which returns relevant results. An extended goal many researchers try to achieve is to reduce the query refining loop by improving components ③, ④, ⑤ and ⑥. A more formal way of measuring the quality of search results are the values *precision* and *recall*, rather than the amount of iterations needed to solve the user task. According to [57], precision is the fraction of the returned results that are relevant to the user's information need, and recall is the fraction of relevant documents in the collection that are retrieved.

As mentioned above, results can be more relevant to user A trying to solve a task than to user B trying to solve the same task. Therefore, the values of precision and recall cannot be determined objectively. While precision and recall were used since the very beginning of scientific information retrieval to measure the effectiveness of a search engine, they do not consider result ranking. A more advanced approach is the usage of *Discounted Cumulative Gain (DCG)*, as discussed in Section 2.4.

Manning et al. define in [57] three scales of information retrieval systems: *web search*, performs searches in a corpus of billions of documents, which are distributed over millions of computers; *personal information retrieval* is focusing on a smaller collection of heterogenous documents, suchs as web sites, e-mails, files etc.; and *enterprise, institutional, and domain-specific search* which is in between the space of the former two. This thesis primarily focuses on the second scale, where only already seen web sites have to be considered for search.

2.2 Indexing

When examining Figure 2.1, the process of *indexing* can be placed at step ⑤. Indexing stores the documents in one collection for the specific purpose of cheap and fast document retrieval by terms. The remainder of this sub section introduces two different approaches to build information retrieval indexes designed for different purposes.

Incidence Matrix

One way to store a collection of documents in an efficiently searchable index is to use an incidence matrix. This structure is a binary matrix which uses terms as row and documents as column labels. An 1 in the matrix at position $(row, column)$ indicates that the term t_{row} occurs in document d_{column}. Table 2.1 depicts an example of such an incidence matrix. When issuing a query like *"Parks Tarantino"*, the user wants to find documents which contain both of those terms. Therefore, we take the row vector of *Parks* and the row vector of *Tarantino* and apply the binary *AND* operation: $1101\ AND\ 0100 = 0100$. Modern search engines use the OR operator to query for terms. Although documents which contain both terms rather than only one are usually ranked higher. The result vector contains 1 in those documents where both *Parks* and *Tarantino* occur.

The incidence matrix can be implemented as a dictionary, where the term t is used as key and the row vector of document occurrences as value. To satisfy the query *Parks Tarantino*, each term has to be looked up in the dictionary, which has a computational complexity of $O(1)$. Depending on how the row vector is stored, the binary ADD operation can also be completed in $O(n)$ with respect to the size of the document collection.

While querying a term-document incidence matrix can be quite fast, this structure has several notable disadvantages. Firstly, it consumes a lot of storage space. For every possible $(term, document)$ pair there has to be an entry in the matrix. This can result in a large sparse matrix, as only a faction of all possible terms occur in one document. Secondly, the insertion of new documents will result into appending additional values in the row vectors of each term. Thirdly, sometimes it is not only important to know in which documents a term occurs in, but

	Django Unchained	Inglourious Basterds	Pulp Fiction	Reservoir Dogs
Buscemi	0	0	1	1
Keitel	0	0	1	1
Madsen	0	0	0	1
Parks	1	0	0	0
Roth	0	0	1	1
Tarantino	1	0	1	1
Waltz	1	1	0	0
...	...	...	...	...

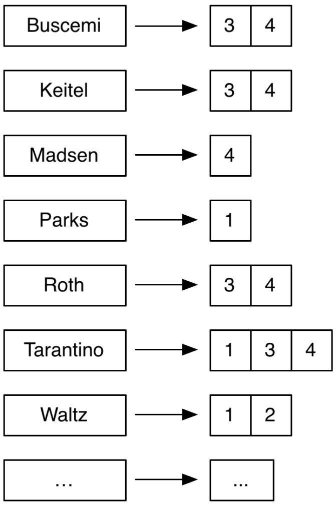
Table 2.1: A term-document incidence matrix. If element (t, d) is 1 the document d contains the term t, 0 otherwise. Taken and adapted from [57].

also on which position in the document. The query *"Parks Tarantino"* yields Quentin Tarantino's movie script *Django Unchained*, which consists over 9000 lines [17]. An additional search has to be conducted only to retrieve the locations of those terms in the document. The incidence matrix performs well, when storing a small corpus of small documents, which barely changes over time.

Inverted Index

One more advanced data structure to store an index of a document collection is the so-called *inverted index*. Manning et al. describe the inverted index in [57] to be *"the most efficient structure for supporting ad hoc text search without rivals"*, where *ad hoc information retrieval* is the standard retrieval task in which the user describes her information need in a query and the information system returns results which satisfy the user's query need, see [4]. The inverted index, similarly to the incidence matrix, is realized as a dictionary, which has the term t as key and an array of document IDs, in which the key occurs, as value. The value of the dictionary is often called *postings list*, and all values grouped together are referred to as *postings*. Furthermore an additional dictionary to lookup which document belongs to which document ID is needed. Figure 2.2a and Figure 2.2b display a simple inverted index example from the Quentin Tarantino movie script's corpus. For example, *Buscemi* occurs in documents 3 and 4 which are labeled *"Pulp Fiction"* and *"Reservoir Dogs"* respectively.

When querying for *"Parks Tarantino"* each term has be looked up separately, as it was done with the incidence matrix. The intersection algorithm, see Algorithm 1, finds all document IDs which contain *Parks* and *Tarantino* with computational complexity of $O(N_t + N_d)$, where N_t denotes the amount of query terms and N_d the size of the document collection. The algorithm assumes that all postings lists are sorted by their document IDs in ascending order. [57] introduces an improved intersection algorithm, which uses skip pointer to faster iterate through the postings list. The algorithm terminates only in the worst case in $O(N_t + N_d)$, and heavily depends on a heuristic for a good skip pointer length.



(a) Inverted index

Document ID	Document name
1	Django Unchained
2	Inglourious Basterds
3	Pulp Fiction
4	Reservoir Dogs
5	...

(b) Document mappings

Figure 2.2: Example of an inverted index in the Tarantino movie domain.

```
input  : $p_1$ and $p_2$, pointers to postings lists of term 1 and 2.
output: intersection result $R$

$R \leftarrow []$;
while $p_1 \neq NULL \wedge p_2 \neq NULL$ do
    if $p_1.docID = p_2.docID$ then
        append $p_1.docID$ to $R$;
        $p_1 \leftarrow p_1.next()$;
        $p_2 \leftarrow p_2.next()$;
    else if $p_1.docID < p_2.docID$ then
        $p_1 \leftarrow p_1.next()$;
    else
        $p_2 \leftarrow p_2.next()$;
    end
end
return $R$;
```

Algorithm 1: Finding the intersection of two arrays. Taken and adapted from [57].

To efficiently build the inverted index from scratch, the following steps have to be accomplished:

1. Collecting all documents in the corpus (*crawling*).

2. Splitting the text of each documents into tokens.

3. Apply linguistic preprocessing of tokens (to retrieve terms).

4. Store the documents and terms in the index (*indexing*).

In the scope of creating a search engine for already seen content, as this thesis aims for, step 1 has to be reapplied every single time the user sees new content. Additionally, repeated visits of the same content have to be considered, i.e. the website `http://www.imdb.com` changes its front page on almost daily basis, so each version has to be treated as a new document.

For the sake of simplicity, all documents in the corpus are written in the English language, so encoding issues or the handling of special characters, which are present in other languages, do not have to be taken care of. Furthermore, it is assumed that the tokens created in step 2 consist only of alphanumerical characters. Any other characters like white spaces, commas, dashes etc. are considered as delimiters and therefore are stripped out. There are special cases where it is not desirable to have that splitting effect, i.e. the expression *"U.S.A."* should not be split into *"U"*, *"S"* and *"A"*. The algorithm should treat this as one token. Manning et al. discuss several simple and complex methods for term preprocessing in [57]. The remainder of this section assumes the documents are split into their corresponding preprocessed terms.

Algorithm 2 demonstrates how to build the inverted index I out of a set of documents D. Each document d in D consists of plain text. Special formatting and markup tags have been removed. This plain text gets split into an array of tokens W, which will be preprocessed. This process returns the terms array T, which will be stored in the index. In line 11, the algorithm stores in which document the term occurs. Furthermore, it saves also the location the occurrence of t in d (line 13). In this simple version, only an array of locations in $I[t][d]$ is stored, but any kind of information which might be need later on for the ranking procedure can be stored here, as described in Section 4.1. For example, the occurrences of the term *Parks* in a heading markup or plain text could be encoded here. Additionally the amount of time the user has seen the term on her screen could be captured, which can affect the ranking of documents tremendously ([60], [29]).

In addition, some terms must be considered which are not meant to be stored in the index, as they do not provide any additional value for the search result. For example, the terms *"and"*, *"for"*, *"an"* occur in the majority of all documents in the corpus. According to Google Ngram Viewer for Books [39], the term *"and"* occupies $2\% - 3\%$ of all terms used in the whole corpus of books published from 1800 till 2000. On the other side of the spectrum, the term *"house"* only occupies around $0.02\% - 0.04\%$. Searching for those high frequent terms does not lead to better results and, because of their ubiquity, they only consume up space in the index. Before placing a term in the index, it has to be checked against a list of so called *stop words*, which include all those terms with no or very little information value.

```
input  : Set of all documents $D$
output: Inverted index $I$
$I \leftarrow \{\}$;
forall the $d$ of $D$ do
    split $d$ into tokens $W$;
    transform tokens W into terms T by preprocessing;
    $c \leftarrow 0$ ;
    forall the $t$ of $T$ do
        if $t \notin I$ then
            $I[t] \leftarrow \{\}$;
        end
        if $d \notin I[t]$ then
            $I[t][d] \leftarrow []$;
        end
        append $c$ to $I[t][p]$;
        $c \leftarrow c + 1$;
    end
end
```

Algorithm 2: Creation of inverted index

2.3 Querying and Ranking

In this subsection, the inverted index consists of a big enough collection of documents and terms, which contain information the user wants to retrieve to satisfy her information need. The user formulates her information need in a query, and the search engine returns those results, which are the most relevant ones.

A simple dictionary lookup in the inverted index can confirm whether it contains a query term and in which documents it occurs. In case of a small document corpus, the result list is also small, and the user can easily examine all entries to find the relevant ones. Nowadays, a web search engine provider such as Google cannot expect its users to open every single result the engine returns. Google's index contains about 40 billion websites according to [18]. The key concept here is proper ranking, so the relevant document is within the top results. The longer the user scrolls down the result list, the more likely she will refine the search query to retrieve the needed information. This was observed among others by Agichtein et al. in [1]. According to their measurements the click-through rate of the second result drops to 50% compared to the first position in the result list..

Consider a document corpus consisting of two documents d_1: *"Yeah, in a basement. You know, fighting in a basement offers a lot of difficulties. Number one being, you're fighting in* *a basement!"*[1] and d_2: *"Some men can live up to their loftiest ideals without ever going higher than a basement."*[2]. When querying for the term *"basement"* document d_1 is intuitively more important than d_2, as the term occurs 3 times instead of only once. This ranking criteria is applied by the *term frequency ($tf_{t,d}$)*, it counts the occurrences of term t in document d and ranks documents accordingly to their term occurrences.

In this model, a document is represented by a vector of its term occurrences, e.g., the document *"Tarantino likes Tarantino movies"* is encoded in alphabetical term order as $[1, 1, 2]$ (*"likes"*, *"movies"* occur only once while *"Tarantino"* has 2 occurrences). Manning et al. refer to this as the *bag of words model* ([57]) as the semantical order of the terms is lost. For instance, the documents *"Rodriguez is better than Tarantino"* and *"Tarantino is better than Rodiguez"* have the same representation of $[1, 1, 1, 1]$ but the position of the terms cannot be reconstructed and, therefore, the semantical meaning is sacrificed.

According to the *"term frequency only"* ranking criteria, a document containing a query term 10 times more often than a document which contains it only once would be 10 times more relevant. If another document contains the search term 1000 times, it means the other document is 100 times more important. The linearity of *tf* does not represent the relevance factor properly, thus the term frequency needs to be dampened by the *log* operator. The base of the logarithm does not matter. For demonstrational purposes the base of 10 is used in the examples.

In another use case the document could be extended by appending its content at the end, so each term frequency is doubled. Long documents are generally better rated than short ones [85]. While the *tf* value increases, the information conveyed is not higher. Therefore, the $tf_{t,d}$ value is normalized, as shown in Equation 2.1. $\vec{d}$ denotes the vector which represents the *bag of words* of the documents and $N_{t,d}$ denotes the total amount of occurrences of term t in document d.

$$tf_{t,d} = log_{10} \frac{N_{t,d}}{||\vec{d}||} = log_{10} \frac{N_{t,d}}{\sqrt{\sum_{i=0}^{|\vec{d}|} \vec{d}_i^2}} \tag{2.1}$$

The only drawback using only the term frequency as a ranking factor is that it treats every term equally. For instance, the query for *"Reservoir dogs"* indicates the search intent for Tarantino's movie from 1992. The less often occurring term *reservoir* gives the query a different meaning and points out that the searcher's intent does not consider documents about animals. Therefore, the measure of the *inverse document frequency ($idf_t = \frac{N}{N_t}$)* is used to correct this deviation. N denotes the amount of documents in the collection and N_t the amount of documents containing the term t. Is does not matter how often the term occurs in a document, only its sole occurrence matters.

The inverse document frequency gives insight about the information content of one term. For instance the term *the*, which occurs often in English language, might occur in every single docu-

---

[1] Quote by Lt. Aldo Rain from the movie Inglorious Basterds (2009) by Quentin Tarantino

[2] Quote by Theodore Roosevelt

ment and, thus, it has an idf value close to 1. According to Google books Ngram Viewer $4 - 6\%$ of all words published in books from 1800 till 2000 are the word *the*. Whereas the term *reservoir* occurs less often in the document corpus, giving it a higher idf value than 1. According to Salton [68], the inverse document frequency is calculated as shown in Equation 2.2.

$$idf_t = log_{10}\frac{N}{N_t} + 1 \tag{2.2}$$

The information content of one term is denoted by its $tfidf$ (read *term frequency - inverse document frequency*) and is defined as shown in Equation 2.3. Each element in the document vector $\overrightarrow{d}$ stores the TF-IDF value of the respective term instead of the term's occurrence in the document. It considers the amount of occurrences of one term in a document, and also the relative information content of that term.

$$tfidf_{t,d} = tf_{t,d} * idf_t \tag{2.3}$$

Another problem is ranking the search results according to the information need encoded by the query. One very common solution is to calculate the *cosine similarity* in the *vector space model (VSM)* between the query and the documents containing at least one of the terms in the query [84]. Therefore, the query and each matching document will be encoded as K-dimensional $tfidf$ term vectors. K denotes the total amount of terms in the document collection. The values in the document vectors are either 0, if the term t_i $(0 \leq i < K)$ does not occur in the document, or $TF - IDF_{t_i,d}$ if the term does occur in the document. The same procedure is applied to the query vector. As one document contains only a small subset of the total terms in the corpus and the query an even smaller subset, the resulting document and query vectors will contain many 0 values.

As query and documents are represented as vectors, the goal is to compute the *"similarity"* between the query vector $\overrightarrow{q}$ and each of the document vectors $\overrightarrow{d}_i$. The most promising approach is to compute the angle between those vectors [84]. The smaller the angle, the more similar the vectors are. Figure 2.3 shows an example of a simple vector space. Consider the query *"Quentin Tarantino"* being represented by $\overrightarrow{q}$. Three documents in the corpus contain both query terms but their $tfidf$ values differ. In document $\overrightarrow{d}_1$ the term *Quentin* occurs frequently but the term *Tarantino* almost not at all. Document $\overrightarrow{d}_2$ instead contains both terms equally often. In the search result list those documents are ranked by the cosine similarity to $\overrightarrow{q}$, which is defined as shown in Equation 2.4. The theoretical value range is $[-1, 1]$, but in this domain it will be $[0, 1]$, as there are no negative *ifidf* values in neither query nor document vectors.

$$cos(\overrightarrow{q}, \overrightarrow{d}) = \frac{\overrightarrow{q} \bullet \overrightarrow{d}}{|\overrightarrow{q}| * |\overrightarrow{d}|} \tag{2.4}$$

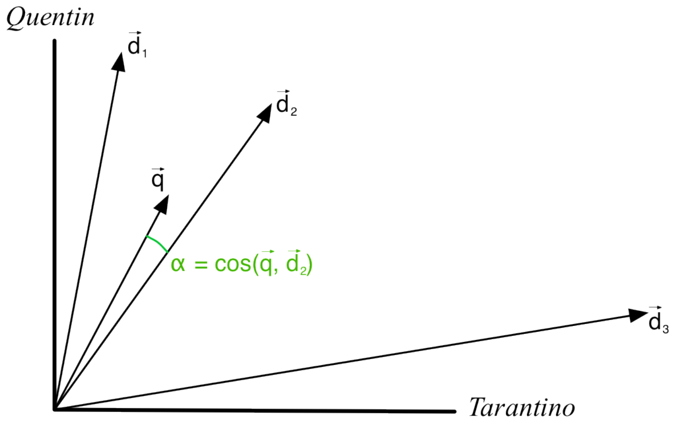


Figure 2.3: The smaller the angle α, the higher the cosine similarity of q and d_i.

The cosine similarity has to be computed each time a search is conducted for each *(query, document)* pair. Thus, to reduce computational overhead, it is beneficial if the vectors are in unit length, so the score computation become easier: $cos(\vec{q}, \vec{d}) = \vec{q} \bullet \vec{d}$.

As mentioned above, the document and query vectors will be sparse in most of the cases. Yet the computation and storage complexity are only $O(k)$, where k denotes the count of terms in the query. Due to the fact that the partial dot product of two vectors will be 0 at element i if either i-th element is 0. Thus each term occurring in the query **and** in the document vector is considered, and each term only occurring in the document, but not in the query, is discarded. Algorithm 3 demonstrates how to compute the ranking score of a document d according to query q in $O(k)$. In this example, d and q are dictionaries mapping terms to their corresponding $tfidf$ value.

```
input  : $d$, $q$
output: ranking score $s$
$score \leftarrow 0$;
forall the $term$ in $q$ do
    if $d$ contains $term$ then
        $score \leftarrow score + (term.tfidf * d[term].tfidf)$;
    end
end
```

Algorithm 3: Calculating the ranking score of a document d wrt. its query q

The $tfidf$ values of the query terms are relative to the query. In this case, the query itself is considered a document and the cosine similarity calculates the similarity between those two documents.

2.4 Comparing Ranking Algorithms

A brief look at the web search engine market reveals many big names like *Google* [36], *Bing* [14] or *Yahoo!* [43], *Duck Duck Go* [20] or *Mahalo* [44] which focus on niche markets. Search engines can differ in categories like proper ranking of document relevance, privacy, speed of result retrieval, etc. This section discusses one of the state-of-the-art evaluation method in search engine result rankings: *(Normalized) Discounted Cumulative Gain.*

Cumulative Gain of Documents

Järvelin and Kekäläinen introduced and refined a method in [46] to compare two different search engine rankings by using the cumulative gain of each document in the result list. This method requires the input of human relevance judgments for each ranked document in the results. Those relevance judgments are either binary: *document relevant* vs. *document is not relevant*; or include a broader range of values. The authors used a range from 0 to 3 in their examples to express partly fulfilled information satisfaction.

Järvelin and Kekäläinen stated the following two observations [46]:

1. *highly relevant documents are more valuable than marginally relevant documents, and*
2. *the greater the ranked position of a relevant document (of any relevance level) the less valuable it is for the user, because the less likely it is that the user will examine the document.*

The authors define a recursive equation to calculate the cumulative gain of a document at a given rank *i* (Equation 2.5). G denotes an array of human relevance judgments, e.g., $G_{rankA} = \langle 3, 2, 0, 2, 1, 1, 3 \rangle$ (for the latter examples in this section, this setting will be referred to as *rankA*). With the equation the cumulative gain of a document at rank 7 can be calculated by $CG[7] = 3 + 2 + 0 + 2 + 1 + 1 + 3 = 12$.

$$CG[i] = \begin{cases} G[1], \text{if } i = 1 \\ CG[i-1] + G[i], \text{otherwise} \end{cases} \tag{2.5}$$

To fulfill point 2 of Järvelin's and Kekäläinen's observations, the Cumulative Gain equation is not expressive enough, as a switch of the documents d_1 and d_3 still yields a *CG* of 12 (assuming the result list contains only 7 documents). They refined their method and introduced the *Discounted Cumulative Gain (DCG)*, which also takes the rank of the currently evaluated document into consideration (Equation 2.6).

$$DCG[i] = \begin{cases} G[1], \text{if } i = 1 \\ DCG[i-1] + \frac{G[i]}{\log_2 i}, \text{otherwise} \end{cases} \tag{2.6}$$

DCG now penalizes highly relevant documents which are also highly ranked, e.g., the documents *DCG* at rank 1024 is reduced by a factor of 10. Taking the same example from above into consideration, the discounted cumulative gain for document d_7 is $DCG[7] = 3+2+0+1+0.431+0.387+1.069 = 7.887$. If another ranking with different parameter configuration exchanges d_1 with d_3, the new DCG' results in $DCG'[7] = 0+2+1.893+1+0.431+0.387+1.069 = 6.78$. The prior ranking is clearly better, as $DCG[7] > DCG'[7]$.

DCG is a simple but powerful way to compare two ranking parameter configurations, which return the same amount of results for a given query. When comparing two different rankings of different search engines, *DCG* comes with some limitations, assuming both search engines return an unequal amount of results. An easy solution for this problem is to normalize the *DCG* values. For the normalization procedure, the notion of *Ideal Discounted Cumulative Gain (IDCG)* is needed, which simply computes DCG value of the human relevance judgements of the G array sorted in monotonically decreasing order, e.g. $G_{rankA_i} = \langle 3, 3, 2, 2, 1, 1, 0 \rangle$, $IDCG[7] = 3+3+1.262+1+0.431+0.387+0 = 9.08$. The definition of the *normalized DCG (NDCG)* is shown in Equation 2.7.

$$NDCG[i] = \frac{DCG[i]}{IDCG[i]} \tag{2.7}$$

In the example, the first parameter setting results into an $NDCG[7]$ of $7.887/9.08 = 0.869$, and the second parameter setting into an $NDCG'[7]$ of $6.78/9.08 = 0.747$, which again confirms the superiority of the first configuration. The advantage is that another ranking algorithm, containing more results with different relevance judgments, can now be compared to both configurations of *rankA*.

Another web search engine vendor returns for the same query as of *rankA* results, which have the following human relevance judgements: $G_{rankB} = \langle 4, 0, 2, 0, 3, 1, 2, 0 \rangle$, $DCG[8] = 4+0+1.262+0+1.292+0.387+0.356+0 = 7.297$, $IDCG[8] = 4+3+1.262+1+0.431+0+0+0 = 9.693$, $NDCG[8] = 7.297/9.693 = 0.753$. Table 2.2 summarizes the results from *rankA* and *rankB*, and shows that *rankA* with configuration c_1 is the superior ranking algorithm.

Algorithm	Configuration	$DCG[n]$	$IDCG[n]$	$NDCG[n]$
rankA	c_1	7.887	9.08	0.869
	c_2	6.78	9.08	0.747
rankB	c_1	7.297	9.693	0.753

Table 2.2: Comparison of different search engine rankings using *NDCG*.

The significant disadvantage when using the discounted cumulative gain as a search result ranking evaluation method is that all documents need to be judged by a human. This process takes a long time, and requires the user to scan through the entire document to determine its relevance. Agichtein et al. propose [1] a novel way to use implicit feedback from the web searcher to obtain a relevance value (see Section 3.3).

2.5 Term Distance

The simple inverted index structure allows to apply trivial boolean queries like *"Stuntman Mike's black car"* to retrieve documents which contain all 4 terms. The order of the terms in the query can differ from the term order in the document, for instance, the document *"While Mike was hitting the stuntman in the next scene, he was listening to the Black Eyed Peas."* does contain all terms from the query, but might not be relevant to the user's information need.

A simple approach is to create a biword index which stores two consecutive terms as the key for the inverted index, as proposed by Manning et al. [57]. The document is broken down into two-word vocabulary terms, e.g. *"While Mike"*, *"Mike was"*, *"was hitting"*, *"hitting the"*, etc. Also the query from above is split into *"Stuntman Mike's"*, *"Mike's black"* and *"black car"*. In the query process all documents will be retrieved which contain all biwords. While this approach returns better results than the one word inverted index, it is still possible to get false positives. Needless to say the space consumption of the inverted index doubles, as the one-word and two-word versions need to be saved.

A more advanced technique is the *phrase index* [57], which extends the postings lists of the original inverted index by term locations, as shown in Algorithm 2. The structure of one inverted index entry is $term : docID(location_1, location_2, ...location_n)$. Figure 2.4 shows an excerpt of an example index. The term *black* occurs on position 5, 16, 64, etc., the term *car* occurs only on position 65. All terms occur in document 42 though. Through their positions in the same document, it can be determined whether the terms are adjacent or not. In this example, the query *"Stuntman Mike's black car"* finds a document in which the terms occur in the same order as in its query, i.e. *"Stuntman"* at position 62; *"Mike's"*, *"black"* and *"car"* at positions 63, 64 and 65 respectively.

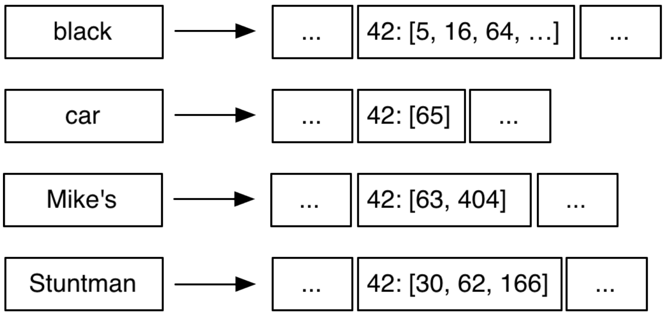


Figure 2.4: The inverted index with term locations in the postings lists.

To retrieve results from the phrase index, the intersection Algorithm 1 has to be modified in a way that the query $"term_1\ term_2"$ checks whether $term_1$ and $term_2$ occur consecutively in a document. When considering stop words, some term locations have to be skipped. For instance, in case the word *"black"* is classified as stop word, it will not be stored in the inverted index. When querying *"Stuntman Mike's black car"*, the gap between the location of *"Mike's"* and *"car"* has to be considered.

Algorithm 4 shows how to retrieve results from a phrase index. The parameter k indicates the size of a tolerable gap between two terms. After finding the common document of two query terms, the algorithm iterates through all postings lists in line 7. Line 9 checks whether the location gap between two terms is within k. The result of the algorithm is an array of $\langle docID, loc_{term_1}, loc_{term_2} \rangle$ tuples, which contains the document ID in which $term_1$ and $term_2$ have a gap $\leq k$.

The algorithm uses the fact that the document IDs and their term locations are in ascending order and, therefore, terminates in $O((n + m) * l)$. The bigger the document collection, the more time it takes to find positional intersections with respect to k. Additionally, more space will be needed. Williams at el. [83] use an approach, which exploits the biword index for frequently searched queries and the phrase index to handle queries which cannot be satisfied by the biword index. Using this approach resulted into faster information retrieval and less space consumption.

```
input : $p_1$: postings list pointer of term_1,
input : $p_2$: postings list pointer of term_2,
input : $k$: tolerable gap between two terms
output: Array $answer$ of tuples $\langle docID, term_1.pos, term_2.pos \rangle$
$answer \leftarrow []$;
while $p_1 \neq NULL \land p_2 \neq NULL$ do
    if $p_1.docID = p_2.docID$ then
        $l \leftarrow []$;
        $pp_1 \leftarrow p_1.positions$;
        $pp_2 \leftarrow p_2.positions$;
        while $pp_1 \neq NULL$ do
            while $pp_2 \neq NULL$ do
                if $|pp_1.value - pp_2.value| \leq k$ then
                    append $pp_2.value$ to $l$;
                else if $pp_2.value > pp_1.value$ then
                    break;
                end
                $pp_2 \leftarrow$ next($pp_2$);
            end
            while $l \neq [] \land |l[0] - pp_1.value| > k$ do
                delete $l[0]$;
            end
            forall the $ps$ in $l$ do
                append $\langle p_1.docID, pp_1.value, ps \rangle$ to $answer$;
            end
            $pp_1 \leftarrow$ next($pp_1$);
        end
        $p_1 \leftarrow$ next($p_1$);
        $p_2 \leftarrow$ next($p_2$);
    else
        if $p_1.docID < p_2.docID$ then
            $p_1 \leftarrow$ next($p_1$);
        else
            $p_2 \leftarrow$ next($p_2$);
        end
    end
end
return $answer$
```

Algorithm 4: Positional intersection algorithm, adapted from [57]

CHAPTER 3

Related Work

This section outlines methods and approaches to improve web search result ranking by exploiting user behavior and user profiles rather than only traditional methods (e.g., document-query similarity). While many researchers use variations of probabilistic models to improve document ranking ([11], [72]), this section focuses on papers which exploit the variants of the vector space model as proposed in [84].

3.1 Post-Click Behavior as Relevance Factor

Morita and Shinoda [60] conducted an experiment where participants were given news articles to read and were asked about the interestingness of each article. The researchers focused on the correlation between the personal interestingness value and the amount of time it took the subjects to read through the articles. In their study, they have found out that the majority spends more time on interesting articles than on uninteresting ones.

Kelly and Belkin [50] tried to reproduce the outcome of Morita and Shinoda in a more complex environment, but could not find any correlation between the time spent on one document, the so-called *dwell time*, and their personal relevance ratings for that document. In [51], the researchers observed again no relationship between the display time and the users' display rating, longer, they noted that the dwell time varies for different users and for different tasks.

Taking the limitations of the dwell time as a relevance factor into account, Guo and Agichtein proposed in [29] the metric of the *post-click behavior (PCB)* for better estimating a document's relevance. They define post-click behavior, such as cursor movement and scrolling, as measurements of actions the user does after clicking on a result of a *search engine result page (SERP)*. Guo and Agichtein observed two patterns of visiting a web page: *reading* and *scanning*. More specifically, for reading they detected a clustering of cursor positions over a horizontal line indicating a read action from left to right. In the case of scanning, the cursor positions were aligned vertically, signalizing a scroll action.

During their experiments they observed the following common patterns across all examinations:

- *Periods of horizontal reading indicate relevance*: User were more likely to slow the mouse movements down and move the cursor horizontally to read when the document is relevant.
- *Focused attention indicates relevance*: The cursor positions were clustered in a small area of relevance, whereas on less interesting areas the positions were uniformly distributed.
- *Left-prevalence*: Searchers tended to keep the mouse cursor on the left half of the screen, as most of the content of the Web is left aligned and to help reading or to prepare for a click on a link.
- *"Scanning"followed by "reading"indicates relevance*: Scanning actions through scrolling were interrupted by finding and reading an interesting area of the document.
- *Quick scrolling ("Skipping") indicates non-relevance*: The user might have become impatient and accelerated "scanning"to an even faster pace.

Guo and Agichtein measured the following post-click behavior features to estimate the personal relevance of a web site to the user: dwell time, result rank provided by a search engine, cursor movements (such as speed and range), vertical scrolling or interactions in the *Areas of Interest (AOI)*. They defined the Area of Interest as the region in a document where the main content lies. In particular they defined the AOI as the document's region with X-coordinates from 100 to 400 pixels and Y-coordinates larger than 100 pixels.

For the experiment they assigned the participants a set of tasks, which were taken from [22]. Each task comprised of a question to solve by utlizing several search queries on major search engines, e.g. Google, Yahoo!, Bing. The subject was asked to give a personal rating from 1 (did not meet information need) to 5 (completely satisfied information need) for each visited web page. During these visits Guo and Agichtein measured the PCB factors mentioned above. They used the machine learning algorithms *Ridge Regression (RR)* and *Bagging with Regression Trees (BRT)* [6] to estimate the importance of each PCB factor and combinations of those regressing to the personal relevance.

The researchers have found out that these patterns of behavioral signals correlated with the participant's explicit judgment of document relevance. Especially they found out that the distance and range of cursor travels, as well as movement speed across its vertically component, were among the most predictive signals of document relevance.

This thesis uses a similar approach to measure the importance of the input factors, such as cursor movement, henceforth, named ranking factors. The focus lies primarily on calculating scores for each search term. Under consideration of other factors, which are explained in later chapters, the summed score of all search terms result in the documents ranking score. Additionally, this approach also exploits the documents markup structure and the count of each term. Furthermore, neural networks are used to compute the weights for each ranking factor.

3.2 Personalizing Web Search

Matthijs and Radlinski argue in [58] that current information retrieval systems are not adaptive enough to provide a proper user personalization. They mention the example of an user search for *"Ajax"* and retrieving results from three different domains, such as *a web development concept*, *the football club Ajax Amsterdam* and *the Ajax cleaning products*. Without further knowledge about the user's background, all results are treated equally, for instance, giving a web developer results about the latest football scores.

Personalization of web search results has received a lot of attention. Sriram et al. [74] use a short-term personalization based on the current user session. While this approach improves the quality of the result ranking, the session data is often too sparse to personalize ideally, and also does not personalize before the second query. Speretta and Gauch [73] and Qiu and Cho [65] use the searcher's click-through data collected over a long period of time. Teevan et al. [78] use previously visited web pages, and other information such as documents on the user's hard drive or e-mails to build a search personalization profile. In [79] they discuss that profile based personalization may lack effectiveness on unambiguous queries, such as *"London weather forecast"*, and therefore, no personalization should be applied in those cases.

The user profile mentioned before can be represented in numerous ways, such as by a vector of weighted terms (e.g. [16] and [78]), a set of concepts (e.g. [55]), predefined ontologies (e.g. [23], [64], [71]) or hierarchies based on the *Open Directory Project (ODP)* [33] and corresponding keywords (e.g. [12], [54]). The remainder of this section describes the vector of weighted terms approach used by Matthijs and Radlinski [58].

For their personalization strategy they are capturing the data via a Firefox extension called *AlterEgo*. During the test, participants are surfing the web and the addon sends the user ID, the duration spend on one web page, its URL and the length of the content to a server. The server then fetches the content by the transmitted URL. If the length of the retrieved content differs by 50 from the sent length, the web page is considered as being personalized for the user, e.g., by being logged in, and is discarded.

From all fetched web sites, Matthijs and Radlinski extract six features to build the user profile: *full text unigrams* (HTML stripped content), *title unigrams* (terms inside the <title> tag), *metadata description and metadata keywords unigrams*, *extracted terms* and *noun phrases*. Algorithm from [81] is used to extract relevant terms by a number of linguistic features. The researchers were experimenting with the following three weighting schemes to find the best parameter optimization for the user profile: *term-frequency* (tf) weighting, *term-frequency inverse-term-frequency* ($tfidf$) weighting and *personalized BM25 weighting* as proposed by Teevan et al. in [79].

Matthijs' and Radlinski's personalization is used to rerank search results, which are returned by a search engine provider. In their experiments, they reranked the first 50 results provided from Google. This has the advantage that the reranking is only applied to a small subset of already generally interesting results. The researches assigned each search result different scores by applying the *Matching*, *Unique Matching*, *Language Model* and *PClick* scoring methods. Additionally, they gave already visited URLs additional weights.

The researchers applied two evaluation methods for personalized search strategies. First, they started with the *Relevance Judgments*, proposed by Teevan et al. [78], where a small group collectively judge the relevance of the top *k* documents for a set of queries. Given these judgments, they calculated the *(Normalized) Discounted Cumulative Gain* [46] or (N)DCG which reflects the quality of the applied ranking for that user. The advantage of Teevan's approach is that once judgments are made, they can be used to test many different parameter configuration without the constant input from the user. The drawbacks are that those judgments take long time to be completed and do not represent the user's normal search behavior. Matthijs and Radlinski used 3 choices for the judgments: *Not Relevant*, *Relevant* and *Very Relevant*.

After finding the best parameter configuration Matthijs and Radlinski performed an *Interleaved Evaluation* [47], [66] as an online evaluation with real users. This approach takes the first k results of a query from Google and one of the three personalization strategies, which was randomly picked. The selected strategy applied the reranking of the Google results. The Team-Draft interleaving algorithm [66] picked one result from the Google ranking (*TeamA*) and one result from the reranked list (*TeamB*) alternatingly. The interleaved result list was injected by the Firefox extension into the Google result page, making the reranking process invisible to the user. By clicking on a search result, the user gives one vote to either TeamA or TeamB. At the end of the evaluation period, the team with the most votes won, indicating the superior result ranking. The researchers performed the evaluation over two months, for 41 users who installed the Firefox extension. With their best personalization strategy, they could improve 23.9% search queries while only 8.2% got worse, for the remaining 67.9%, no improvement was found.

After the experiments, all participants were shown their keyword based user profiles. The majority was stunned how well those keywords described their profiles, and that they would use the same set of keywords to describe themselves.

Matthijs and Radlinski used term weighting to calculate a rerank of an already ranked result set from a search engine provider. This thesis uses a similar keyword centered approach to build a ranking of a set of documents, especially the markup vector and its term-frequency count (see Section 4.1) reflects the same concept as Matthijs' and Radlinski's tf weighting.

3.3 Learning from User Interactions

Matthijs and Radlinski [58] needed for their web search personalization strategy human relevance judgments, which measure the quality of search results. Similarly, the *Discounted Cumulative Gain (DCG)* equation introduced by Järvelin and Kekäläinen [46] requires human result evaluations. Although this traditional technique is used throughout several information retrieval research papers, human ratings are generally hard and expensive to obtain. Another disadvantage is that users are forced to give a judgment about search results, which invites them to examine each result carefully. This controlled laboratory environment can bias the user's natural search behavior and skews the measurement.

Morita and Shinoda [60], as well as Konstan et al. [52] have shown that the reading time of a document as an implicit rating is a strong predictor of user interest. Oard and Kim took this step further and examined [63] whether implicit feedback could replace explicit human judgments in recommender systems. Goecks and Shavlik conjectured in [25] that there is a correlation between a high degree of web site interactivity and user interest. Although their results were promising, the sample size was too small and the results were not tested against explicit human judgments. Claypool et al. [13] instead found a correlation between dwell time on a page combined with scroll actions and user interest, while individual scrolling with mouse-clicking actions did not correlate.

Agichtein et al. [1] have introduced a novel approach to circumvent human relevance judgments by exploiting the *implicit* feedback through the user's interaction with the search results. In this way they not only diffuse the problem of the controlled lab environment, but they can also aggregate the behavior of large numbers of users. That aggregation helps them to filter out the noise of so called *irrationally or maliciously behaving users*. Agichtein et al. present techniques to automatically predict user preferences for search results [1].

They conducted a case study where they analyzed the click distribution of search results of 120.000 conducted searches collected over a three week period. They found that the user click behavior is strongly biased towards the top result on the result list. The relative click count drops under 60% for clicks on the second ranked result, as compared to the first. To visualize this biased behavior, the researchers varied the position of the most relevant document within the top 10 results. Figure 3.1 illustrates the relative click frequency on documents with a varying position of top relevant document (PTR). For example, at result position 2, the most relevant document was clicked most (second bar), although clicks on other results are still distinct.

The general idea behind [1] is to create a robust *User Behavior Model*, which consist for Agichtein et al. of two components: a *relevance component*, denoting query specific behavior influenced by the apparent result relevance, and a *background component* where users click indiscriminately. They postulated that the observed value o of a feature f for a query q and a result r can be expressed as the the sum of those two components, as shown in Equation 3.1. C denotes the background component of feature f aggregated across all queries, and rel is the rel-

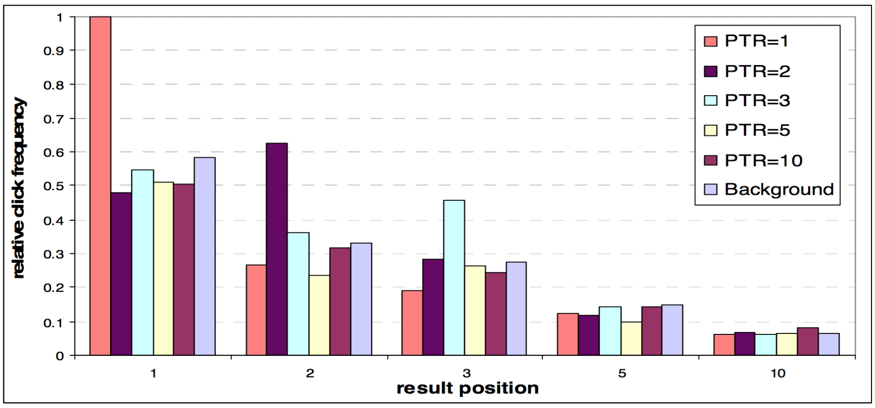


Figure 3.1: Relative click frequency for queries with varying PTR (Position of Top Relevant document). Taken from [1].

evance component influenced by the result r. A simple subtraction of the calculated background noise from the observed value o yields the desired relevance of the result.

$$o(q, r, f) = C(f) + rel(q, r, f) \tag{3.1}$$

Agichtein et al. defined a wide range of over 25 traceable features in [1] and grouped them in the following three categories:

- *Query text features*: user decide the results to examine by looking at the result title, its URL and summary. The features in this category describe the relation between the query and the snippet text such as the fraction of words which query and result summary have in common.
- *Click-through features*: contain features which describe clicks on search results such as click frequency or a click on the result above or below the currently examined search result.
- *Browsing features*: contain features which describe the general browsing behavior in relation to currently examined result such as page dwell time of dwell time on same domain.

The obtained relevance value for the set of tracked features is made more robust by averaging the values from all search attempts. Furthermore, the relevance values are fed into the tailored neural network implementation called RankNet [8], which is capable to learn how to rank a given set of items.

Agichtein et al.'s user behavior model is performing substantially more accurate in results precision than current state-of-the-art search result rankings which do not consider user behavior. That experiment was conducted 2006 and the situation may have changed since. They claim their approach is more generic, and that it can automatically adapt to new environments, such as predicting user preferences in intranet searches rather than only in web searches.

This thesis also uses similar browsing features as proposed by Agichtein et al. to model the user behavior in combination with a term related behavior focus, such as clicking or hovering over terms. Terms with high user interaction particularly influence the result ranking of documents containing those terms. While the approach in this thesis heavily relies on human explicit ratings, it is conceivable to apply implicit feedback to continuously improve the search result ranking without interfering with the user's normal browsing behavior.

CHAPTER 4

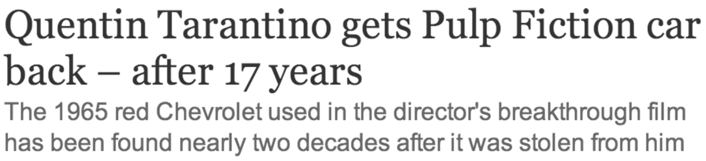
Design

The previous chapter has introduced some basic concepts of information retrieval which will be applied in this chapter. The inverted index structure introduced in Section 2.2 will be extended iteratively throughout each section. This chapter is structured as follows: Section 4.1 explains the measurement of *structural and behavioral (SAB)* ranking factors. Those are used to give the search more personal relevance than plain $tfidf$ does. Section 4.2 introduces the final ranking algorithm which considers the *term frequency*, the structural markup and personal behavioral factors as means of results ranking.

4.1 Measurement of Structural and Behavioral Ranking Factors

When building a search engine for already seen content, the way users perceive content has to be dealt with. Section 2.3 discussed how to retrieve documents which matches the user's search query by using the *cosine similarity* of the query and document vectors. As the results of documents the user gets from one query can still be very large, a ranking system is needed, which prioritizes those results with a higher personal relevance for the user. For instance, an ad which pops up while surfing the web and which is immediately closed by the user, might be less relevant than a Wikipedia page about *World War II* that she has been reading for several minutes. When designing the concepts for this search engine, it has to be considered that content is not equal content.

A simple way to prioritize content is to make use of its domain. Considering a web browser which displays an HTML site, markup such as <*h1*> or <*p*> tags can be utilized to distinguish between heading and paragraph. As shown in Figure 4.1 [61], is bolder and bigger than the rest of the content. Due to bigger and bolder font, the heading is easier to spot and, therefore, gets a higher rank than smaller sized paragraphs.


Quentin Tarantino gets Pulp Fiction car back – after 17 years

The 1965 red Chevrolet used in the director's breakthrough film has been found nearly two decades after it was stolen from him

Figure 4.1: Excerpt from an article of *The Guardian* demonstrating the visibility of marked up text.

A more sophisticated variation of the inverted index constructed by Algorithm 2 can be achieved by storing the term's occurrences in headings, and its occurrences in the remainder text instead of storing the plain location in the document. An instance of the new inverted index is shown in Figure 4.2. The term *Buscemi* occurs in document 3 (1 times in a heading and 100 times in the body) and document 4 (5 and 50 respectively). Hence a query for *Buscemi* would yield the documents *"Pulp Fiction"* and *"Reservoir Dogs"*. In this simple use case, the ranking of documents could be neglected, but it is indispensable when having thousands of search results.

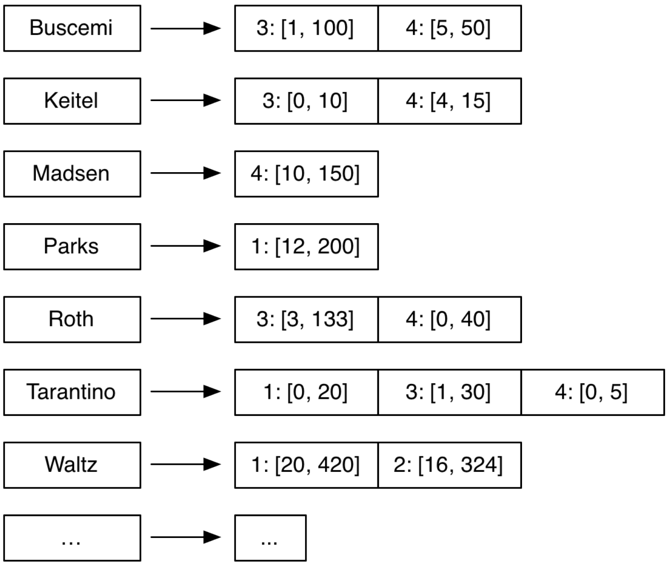


Figure 4.2: Inverted index which stores, compared to Figure 2.2a the occurrences in the markup (represented as the list in the square brackets).

Langville and Meyer [53] use the expression of a *content score* to denote a ranking mean for results. They multiply the occurrences of a term in different semantic sections with a weight to get a ranking score. Consider, for instance, the weights $w_{heading} = 20$ and $w_{paragraph} = 1$, which results into a content score of 120 ($1 * w_{heading} + 100 * w_{paragraph}$) for document 3 and 150 ($5 * w_{heading} + 50 * w_{paragraph}$) for document 4. The results are ranked in descending order by their content score.

This simple use case shows how easily the entries of the inverted index can be extended to consider ranking factors. In this section, the following structural and behavioral ranking factors, about the way how the user perceives content, are introduced:

- markup structure (*structural*) - $markup$,
- content visibility (*behavioral*) - $content_{visibility}$,
- content hovering (*behavioral*) - $content_{hovering}$,
- content clicking (*behavioral*) - $content_{clicking}$

Structural ranking factors cannot be influenced by the user and are part of the representation of the content, e.g. markup. *Behavioral* ranking factors, on the other hand, take the user's interactions as input, such as clicking or hovering over content.

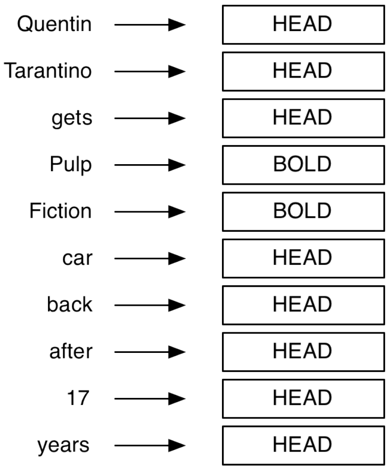
For the measurement of some of these factors it is assumed the user uses a mouse as input device. Other input sources such as the touch screen from a smartphone or an ebook reader as possible, although not all factors can be measured in this case, e.g., content hovering.

Markup Structure

Text can be encapsulated in markups, such as HTML tags. Markups are used when textual content needs to be annotated with additional meta information, for instance <*h1*> for a heading. In the case of HTML, markups yield different formatting, so an <*h1*> tag displays its contents bigger and bolder than just a normal paragraph tag <*p*>. Because of its shape and size, a heading is also more likely to be seen and read by the user than any other regular text. Therefore, the markup is a passive indicator of the importance of content.

As the inverted index stores its content based on the term, a markup containing document needs to be split up into terms, which are encapsulated by markups. For instance this simple marked up document *"**[HEAD_START]** Quentin Tarantino gets **[BOLD_START]**Pulp Fiction**[BOLD_END]** car back - after 17 years. **[HEAD_END]**"* needs to be split into terms with a correspond markup label as displayed in Figure 4.3a.

This very simple example yields that markups can also be nested. Each term's closest encapsulation markup will be considered and assigned. To get an implicit importance value for marked up sections, each markup tag gets the default value of 1. Special tags like headings or bold text get a value larger than 1. The higher the value the more important the text. Therefore a *tag-to-point* dictionary is be introduced, see Figure 4.3b. The values are selected arbitrarily for now. In Section 6.1, several configurations are examined and evaluated.



(a) Markups

Tag name	Points
HEAD_1	5
HEAD_2	4
HEAD_3	3
BOLD	3
LINK	2
PARAGRAPH	1
(other)	1

(b) Tag-to-points dictionary.

Figure 4.3: Sample term to markup assignment.

As mentioned above, only the closest encapsulating tag will be assigned to the corresponding term. This is just a heuristic though and might lead to unwanted results in some cases. For example the document *"**[HEAD_START]** Quentin Tarantino gets **[BOLD_START]**Pulp Fiction**[BOLD_END]** car back - after 17 years. **[HEAD_END]**"* will assign a lower valuation to the words *Pulp* and *Fiction*. Alternatively, the value of all encapsulating parents of one term could be accumulated to calculate the term's markup structure value.

After splitting the document into tokens, and after preparsing those, the terms are stored in the label vector $\overrightarrow{labels} = (term_1, term_2, term_3, ..., term_n)$. The index of each term corresponds to the word wise position in the document. Note that one instance of a term can occur several times in the label vector, e.g., for the document *"Well, well, well. What do we have here?"* the term *well* occurs at index 1, 2 and 3 of its label vector. The markup structure vector $\overrightarrow{markup} \in \mathbb{R}^n$ contains of markup points for each term, as shown in Figure 4.3b. The term labeled with $\overrightarrow{label_i}$ has the markup points of $\overrightarrow{markup_i}$. Considering the example from Figure 4.3a those vectors are $\overrightarrow{label} = (17, after, back, car, fiction, gets, pulp, quentin, tarantino, years)$ and $\overrightarrow{markup} = (5, 5, 5, 5, 3, 5, 3, 5, 5, 5)$. Each term instance occurring in the document has corresponding markup points.

Content Visibility

When evaluating the markup structure $\overrightarrow{markup_t}$, it is assumed that the whole content of one document (and, therefore, every single term) is visible at once. This might the case be sometimes but in general, the user only sees a fraction of the whole document on screen. When the user loads, for instance, a web site, she always gets the upper part of the page presented. After processing all important content, the user scrolls down and new content appears. It is also possible that the user does not even reach the end of the document, because she followed a link. Thus it is incorrect to treat all parts of the document equally when storing its terms in the index. Only those parts are going to be considered which were visible. Additionally, not all content sections have been displayed equally long.

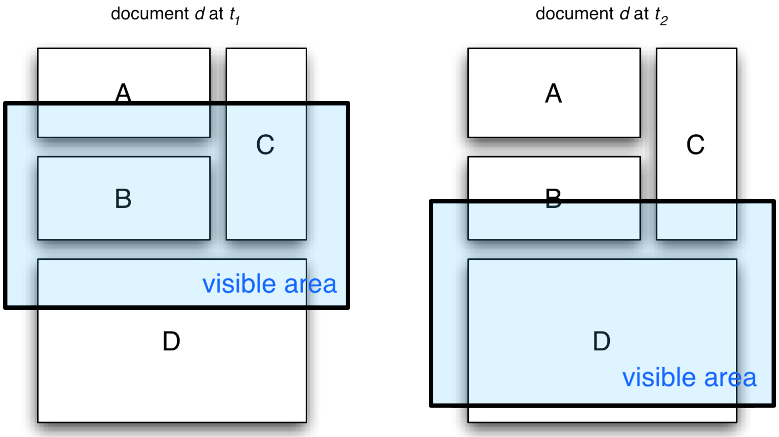


Figure 4.4: Different parts of the document are displayed when the user scrolls.

Figure 4.4 shows an example which sections of document d were visible at the time t_1 and t_2. The document consists of the 4 different-sized and different-positioned rectangular sections A, B, C and D. Those sections can be filled with any kind of information, e.g., text, images, ad banners etc. As only textual content can be evaluated, all sections will contain text for demonstration purposes. At time t_1 only B is fully visible, whereas the other sections are cropped. At time t_2, no section is completely visible, and section A is not visible at all. To measure the *content visibility*, the visibility changes over time for each section have to be considered. For instance, B was fully visible for 10 seconds, then the user scrolled down so B's upper content is cropped, and only 50% of its area is visible for another 5 seconds.

Another problem is how to measure the overall visibility of one particular content section in the document. This solution groups the fraction of a section's visible area in one vector $\vec{v}_{section}$ and their display duration in seconds in another vector $\vec{t}_{section}$. B was entirely visible (1.0) for 10 seconds and partly visible (approximately 0.5) for 5 seconds. $\vec{v}_B = \begin{pmatrix} 1.0 \\ 0.5 \end{pmatrix}$ denotes the visibility values for B and $\vec{t}_B = \begin{pmatrix} 10 \\ 5 \end{pmatrix}$ their display duration in seconds. Equation 4.1 shows how the average content visibility of one section over time is defined.

$$V = \frac{\vec{v}^T * \vec{t}}{\sum_i \vec{t_i}} \tag{4.1}$$

In this example V_B would have a value of 83.33%, which denotes the weighted visibility of

B over the entire time the document was opened. One drawback using this method is when the user scrolls down quickly through the content. This results in high-dimensional $\vec{v}$ and $\vec{t}$ vectors with very little display duration times. A simple solution for this caveat is to define a threshold value for display durations. Only if that threshold value is surpassed the section's values are appended to the vectors.

As the inverted index only stores terms, the content sections mentioned above need to be broken down into words. Although the visibility value V represents the visible fraction of one section, it does not make any sense to say 30% of the text *"What are we on a playground here? Am I the only professional?"*[1] was visible to the user. It is then hard to tell which 30% she has seen, whether it was the first or last four words of the documents. Maybe it refers to the cut off serifs of the lower part of the sentences, which can be hard to read. This value is interpreted as a likelihood whether the user could have read the text in the corresponding section or not. Alternatively, this section-wise grouping can be discarded and it can be directly applied to terms. The visibility values in $\vec{v}$ are then either 0 or 1, depending on, whether the term was displayed on the screen or not. The section-wise grouping was chosen due to the lack of API support from the host environment. If the host environment provides a function like *getVisibleText()*, then the more accurate term visibility should be chosen over section visibility.

Before showing how to translate from content section visibility to term visibility, the structure of content sections has to be considered. The example in Figure 4.4 has shown a very simple case of the logical structure of one document. Real life use cases can be more complex though, as displayed in Figure 4.5a (Reformatted excerpt from [67]). This excerpt of a document is taken from an HTML site. Its content section structure is displayed in Figure 4.5b. The section names are colored in gray. Unlike the example from Figure 4.4, this example has nested content sections.

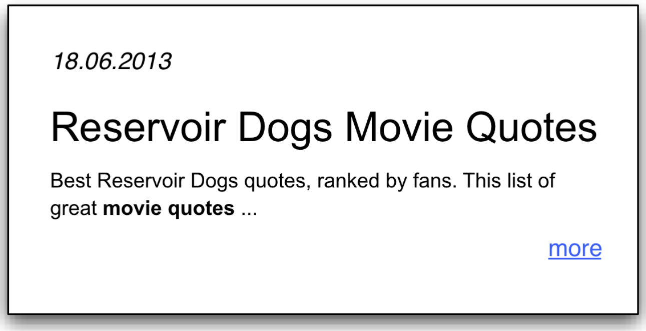
For the sake of convenience, it is assumed that each document is structured as a tree, where content sections represent nodes and terms represent leaves (see Figure 4.6). For example, *d* is the root node and has only *A* as its first level child. *A* contains the term *"18.06.2013"* and has *B* and *C* as its direct children. Note that the term *"Reservoir"* occurs both in *B* and *C*. This indicates that a term wise generalization for each document cannot be applied. Each instance of a term has to be treated independently.

To get the visibility values over time, Equation 4.1 has to be applied on all content sections of the document tree. Terms get the visibility value of their direct parent, e.g. *"Reservoir"*, *"Dogs"*, *"Movie"* and *"Quotes"* get the value V_B. It is assumed that a child content section consumes less or equal space than its parent section. So a direct value assignment is more beneficial than both visibility values V_A and V_d for the terms in section *B*.

The outcome after applying the concepts of this section is the vector $\overrightarrow{content}_{visibility}$ which contains a visibility value for each term in the document. The values in the vector correspond to

---

[1] Quote by Mr. Pink from the movie Reservoir Dogs (1992) by Quentin Tarantino



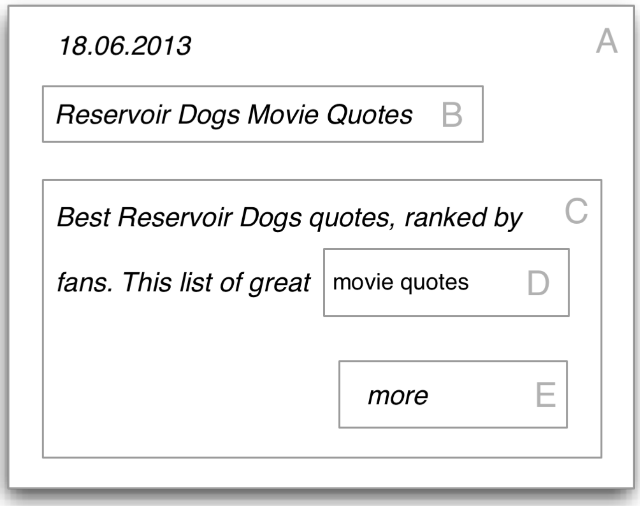
18.06.2013

Reservoir Dogs Movie Quotes

Best Reservoir Dogs quotes, ranked by fans. This list of great **movie quotes** ...

more

(a) A formatted document through HTML markups


18.06.2013
A
Reservoir Dogs Movie Quotes
B
Best Reservoir Dogs quotes, ranked by
C
fans. This list of great
movie quotes
D
more
E

(b) The content section structure

Figure 4.5: Example of an HTML document with content section structure.

the term names in $\overrightarrow{labels}$.

This approach assumes that content, once loaded, does not change while it is viewed. Many modern HTML5 applications contain animations, which pop new text into the screen without the user's assistance. Those dynamic changes are not handled in this thesis.

Content Hovering

The previous sections introduced objective ways to determine which information the user could possibly have consumed by looking at parts of the content. This section uses rather subjective means to find out which fraction of the content the user is actively perceiving. Guo et al. demonstrate in [28] that there exists a correlation between the user's mouse cursor and her gaze

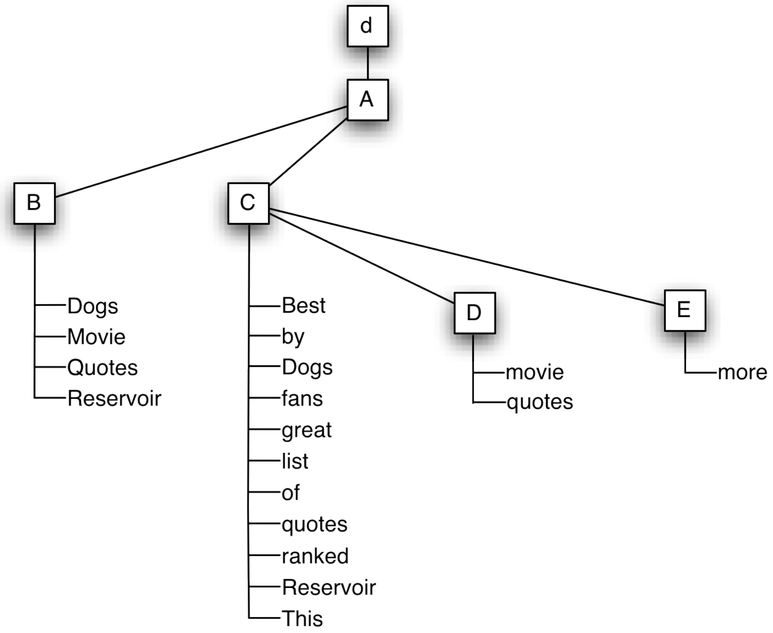


Figure 4.6: The true structure of the content sections of Figure 4.6

position. This approach takes Guo's observation one step further, and assumes that the user's gaze position is close to her mouse cursor, so content which gets hovered by the mouse is more likely to be read than other content. Therefore, hovered information is more relevant when searching for it.

Nevertheless, the user can also scroll through content. The cursor hovers over a wide range of different content sections, but it is highly unlikely the user has actively perceived information while scrolling [29]. Δt defines a time span in which the cursor should not move outside a content section in order to count its underlaying content as being perceived.

Algorithm 5 demonstrates how to weight different content sections based on their hovering interactions. The core concept is that the mouse has to be actively moving. In case the mouse has the same position for Δt milliseconds, it does not count as content hovering, as the user might be inactive and not in front of the screen. For each Δt the cursor moves inside the same content section, the hover counter for this content section is incremented. The higher a section's hover counter, the more relevant its content is to the user.

Once started, the algorithm runs in an infinite loop and constantly modifies the $hover$ array while the user is reading through a document. When the user closes the document, the values from $hover$ have to be stored in the inverted index structure. Similar to the previous section each term inherits the hover counter value of its first level parent.

```
$hover$ ←getFlatContentSections();
$oldPosition$ ← $cursor$.getPosition();
$oldSection$ ← $cursor$.getSection($oldPosition$);
while true do
    $newPosition$ ← $cursor$.getPosition();
    $newSection$ ← getSection($newPosition$);
    if $oldSection$ = $newSection$ then
        if $oldPosition$ = $newPosition$ then
            $hover[newSection]$ ← $hover[newSection]$ + 1;
        end
    end
    $oldPosition$ ← $newPosition$;
    $oldSection$ ← $newSection$;
    sleep($\Delta t$);
end
```

Algorithm 5: Calculating content relevance by mouse hovering

For better comparison, each hover counter value of the $hover$ array is divided by the total value count. The outcome of this section is a vector $\overrightarrow{content}_{hover}$, where the i-th element ($\in [0, 1]$) represents the relative hovering relevance for term at $\overrightarrow{label}_i$.

Content Clicking

Another behavioral ranking factor is the amount of clicks the user has performed in one content section. Click interactions can cause a switch of documents, i.e., the click on a hyperlink. Assuming the user is not randomly clicking her way through documents, this factor can be the most significant indicator which content the user has read.

Similar to the previous sections, it is only evaluated whether the user clicks on a content section and not the term itself. So the click values of each term have to be extracted from their first level parent content sections. The vector $\overrightarrow{content}_{click}$ represents the relative (element values $\in [0, 1]$) click counter. $\overrightarrow{content}_{click_i}$ represents for the click value of term $\overrightarrow{label}_i$.

4.2 Ranking Algorithm

The previous chapters and sections have introduced the basic structure needed to retrieve a ranked list of documents based on the query term distribution and the behavioral profile of the user's documents in the document corpus. The ranking measurement factors $\overrightarrow{markup}$, $\overrightarrow{content}_{visibility}$, $\overrightarrow{content}_{hovering}$ and $\overrightarrow{content}_{clicking}$ have been introduced in the previous section. Each term of each visited document has its corresponding ranking measurement factors. When querying for *"Death Proof"* each document which includes either *Death* or *Proof* (or both) has to be considered. For each document and each term in the query, a $\overrightarrow{factor}_{doc,term}$ vector is created, as in Equation 4.2. The markup values have been extracted from $\overrightarrow{markup}$ and grouped together with the content factors. This allows simpler matrix multiplications, which is needed for the computation of the ranking score.

$$\overrightarrow{factor}_{doc,term} = \begin{bmatrix} markup_{head} \\ markup_{bold} \\ markup_{other} \\ content_{visibility} \\ content_{hovering} \\ content_{clicking} \end{bmatrix} \tag{4.2}$$

The previous sections explained how to build the corresponding ranking measurement factor vector $\overrightarrow{vec} \in \mathbb{R}^n$, where n denotes the amount of terms in the visited document. $\overrightarrow{vec}$ denotes a generalized term for $\overrightarrow{markup}$, $\overrightarrow{content}_{visibility}$, $\overrightarrow{content}_{hovering}$ and $\overrightarrow{content}_{clicking}$. To build the vector $\overrightarrow{factor}$, each value across all ranking factor vectors for one $(term, document)$ pair needs to be collected.

Each term in the query has its own $\overrightarrow{factor}_{doc,term}$ vector for each document it occurs in. These vectors are grouped by the document their term occurs in. This collection is the $F_{query,doc}$ matrix, as shown in Equation 4.3, where k denotes the amount of query terms.

$$F_{query,doc} = \begin{bmatrix} factor^T_{doc,term_1} \\ factor^T_{doc,term_2} \\ \dots \\ factor^T_{doc,term_k} \end{bmatrix} \tag{4.3}$$

Each of the factors for each term has a corresponding weight which is grouped together in the $\overrightarrow{weights}$ vector as shown in Equation 4.4.

$$\overrightarrow{weights} = \begin{bmatrix} w_{markup_{head}} \\ w_{markup_{bold}} \\ w_{markup_{other}} \\ w_{content_{visibility}} \\ w_{content_{hovering}} \\ w_{content_{clicking}} \end{bmatrix} \tag{4.4}$$

The final definition for computing a document's ranking score with regards to the query is shown in Equation 4.5. It contains of the cosine similarities of the $tfidf$ values of the query and document vector, and the linear combination of the factors matrix $F_{query,doc}$, as well as their corresponding weights. The dimension of $F_{query,doc}$ is $\mathbb{R}^{k \times 7}$, and the resulting vector by the multiplication of $F_{query,doc} * \overrightarrow{weights}$ is in $\mathbb{R}^{k \times 1}$. The variable $p \in [0, 1]$ controls the amount of influence of the ranking factors mentioned in this chapter over the traditional cosine similarity. Chapter 6 demonstrates several p value configurations, and how they affect the overall ranking of documents.

$$score_{\vec{d},\vec{q}} = (1 - p) * cos(\vec{q}, \vec{d}) + p * \sum_{i=0}^{k} (F_{query,doc} * weights)_i \tag{4.5}$$

CHAPTER 5

Implementation

5.1 Introduction

The previous chapters have introduced concepts and algorithms which utilize the frequency of terms ($tfidf$) and the user's behavioral profile in order to re-retrieve content she has seen in the past. This chapter outlines the implementation of such a information retrieval system as a *Google Chrome* extension. This decision is justified by a worldwide usage of 42% [76] of Google's browser. Compared to other browsers such as *Microsoft Internet Explorer*, *Mozilla Firefox* or *Apple Safari*, Google Chrome has an easy extension deployment process, an expressive API documentation and a big community [37].

Chrome extensions are written in *JavaScript*, and can use a wide range of *HTML5* APIs defined by the *W3C* ([82], [19]). The programming language used in this chapter is *CoffeeScript* [3], which can be directly compiled into JavaScript before deploying the extension. CoffeeScript was selected over JavaScript due to its high readability, and the interoperability with third party JavaScript libraries, such as *jQuery* [49], *require.js* [2], *Lo-Dash* [48].

This chapter is structured as follows. Section 5.2 introduces the development and deployment setup, and also the communication of processes in an Google Chrome extension environment. Section 5.3 shows the data model and the necessary calls needed to the HTML5 *IndexedDB*. Section 5.4, 5.5 and 5.6 outline how the storing, querying and ranking of content is implemented.

5.2 Environment

Structure of the Extension

The basic Chrome extension consists of a `manifest.json` configuration file and a background page, e.g., `main.html`. The configuration file defines the name and version of the extension, its permissions, and the path to the background page, which is the entry point of code execution. Listing 5.1 depicts an excerpt of the configuration file used for the search engine.

```
{
    "name": "My Search Engine",
    "version": "0.1.0",
    "description": "Re-finding information you have already seen.",

    [...]

    "background": {
        "page": "main.html"
    },
    "content_scripts": [{
        "matches": ["*://*/*"],
        "js": [
            "lib/content_hander.js"
        ]
    }],
    "permissions": [
        "tabs", "unlimitedStorage"
    ]
}
```

Listing 5.1: Excerpt of the `manifest.json` configuration file.

The listing above shows the base structure: its background page is located in the root directory of the extension, it injects the content script `lib/content_handler.js` each time a new document finished loading, it acquires permissions for the *Tabs* API [35], and unlimited storage space to store the captured web pages.

The content script gets injected into the web site's scope (also called *content page*) after the DOM is fully loaded, and starts communicating with the background page. The latter is an HTML page, which only defines JavaScript files to include and does not provide any content. Google Chrome provides two ways for inner-extension communication: *simple one-time requests* and *long-lived connections* [40]. Due to the high amount of different content pages, long-lived connections allocate too much memory when establishing a connection with the background page. In this domain, only simple one-time requests are used.

Figure 5.1 shows the interaction between the content page and the extension. The code running in `content_handler.js` cannot directly invoke functions provided from `engine.js` or `db.js`, as they run in a different scope. For calling a function in the extension scope, the call request is wrapped into a data transfer object and sent to `main.js`, as shown in Listing 5.2.

```
// content_handler.js from content site
var msg = {type: "query", q: "Quentin Tarantino"};
chrome.extension.sendMessage(msg, function(response) {
```

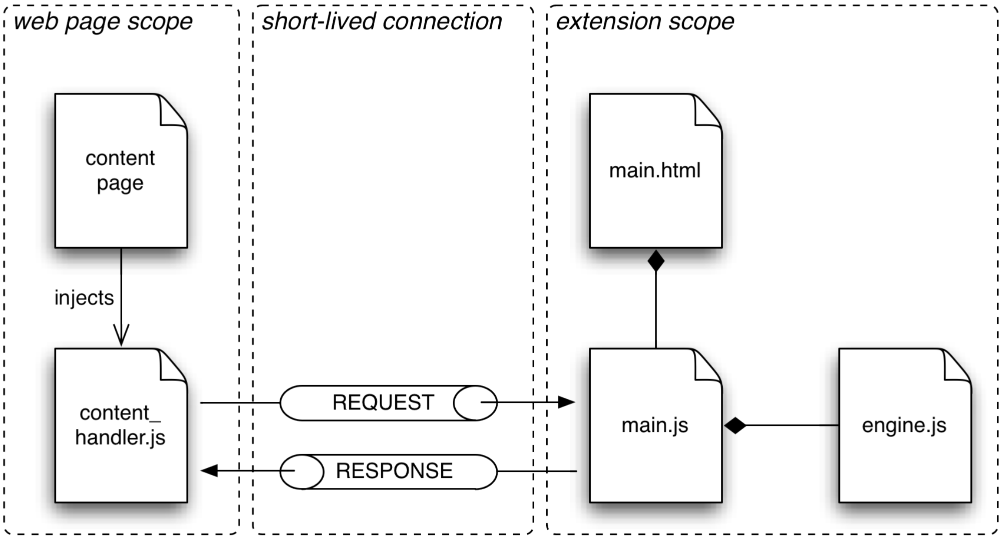


Figure 5.1: Extension structure

```
        console.log(response);
    });

    // main.js from background page
    ===============
    chrome.extension.onMessage(function(msg, sender, respondFunc) {
        engine.query(msg.q, function(results) {
            respondFunc(results);
        });
    });
```

Listing 5.2: Communication between content and background page.

The third parameter of the `onMessage` handler defines a response function (`respondFunc`) as callback, which gets invoked as soon the engine conducted the query and retrieved the results. As insertion or retrieval tasks take time to process, this asynchronous callback model prevents the user's browser from freezing during the execution.

Development and Deployment

At the beginning of this chapter CoffeeScript was introduced as a preprocessor for JavaScript. CoffeeScript was designed to improve the readability of the code by avoiding curly braces and using a Python-like indentation scheme. It also provides a set of language quirks, which allows to shorten frequently used constructs, e.g., using existence operator `?` to check whether a function or variable is defined and not `null`.

Besides CoffeeScript, the *Compass* framework [21] is used to compile *SASS* [30] code into CSS stylesheets. To reduce the compilation, copying and deploying tasks *Grunt - The JavaScript*

Task Runner [26] is utilized.

Grunt can be configured by the file `Gruntfile.js`. It can load a large set of community developed plugins (1051 plugins as of July 2013) [27]. For the implementation of the information retrieval system, the plugins `clean`, `coffee`, `compass`, `copy` and `watch` have been used. Each plugin has its own configuration represented by a JavaScript object passed to the function `grunt.initConfig`. During the development of the extension, the most useful plugin was `watch`, which monitored the local file system and triggered the `coffee` task whenever a `*.coffee` file was saved. Through Grunt, it was possible to avoid the console most of the times when developing and only switch from the IDE to the web browser. Only when an error was detected, the console gave insight in which file to look at. A sample `Gruntfile.js` can be found under `http://gruntjs.com/sample-gruntfile`.

5.3 Data storage

The previous chapters introduced the concept of documents and terms. One document d contains of multiple terms t. In the web browser domain, one document represents the textual content of a web site. Naturally, all HTML markups are considered for the ranking, but are stripped out in the content, so the user can only search for text she actually saw on the web site. The extended inverted index is stored and queried by using the HTML5 IndexedDB API. Figure 5.2 represents the structure of the database. It contains of *terms*, *documents* and *screenshots*. The latter has not been mentioned yet, and is used to show the user a miniature preview of the web site in the result list, before actually clicking on it. One term can occur in multiple documents and one document has many terms by nature. One document can be visited several times, therefore, its content can differ from visit to visit, and so can its screenshot.

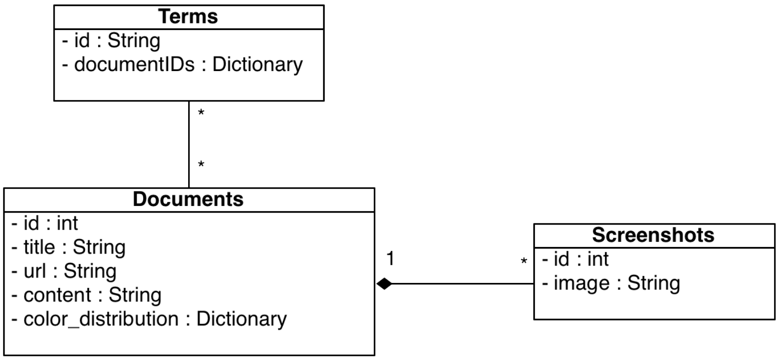


Figure 5.2: Simplified data model.

The IndexedDB Structure

As mentioned in Chapter 4, the timestamp of a document is used as its identifier. Thus, a web page visited at timestamps ts_1 and ts_2 is treated as two different documents. Listing 5.3 shows the structure of the terms table in the IndexedDB. `<xxx>` denote placeholders for concrete values, `<key>:  <value>` represent a key-to-value mapping and `[`v_1`, `v_2`, ..., `v_n`]` is an array of values.

```
...
<term_i>:
    d:
        ...
        <timestamp_j>:
            l: [10, 20, 30, ...]
            m: [1, 2, 3, 2, 20]
            f: [0.3, 0.5, 0.5]
        <timestamp_(j+1)>:
            ...
        ...

<term_(i+1)>:
    d:
        ...
...
```

Listing 5.3: Structure of the terms table.

Each term in the *terms* table has a collection of timestamps (`d`). These timestamps refer to the *documents* table, as shown in Listing 5.4. Each timestamp entry consists of the keys `l`, `m`, `f` and `d`. `l` is an array of all occurrences of `term_i` in the corresponding document. `m` denotes in which markup the term was wrapped (Table 5.1). In comparison to Section 4.1 the occurrences of the terms in the URL and inside the `<title>` tag have been considered as well. f represents an array of the normalized behavioral ranking factors $content_{visibility}$, $content_{hovering}$ and $content_{clicking}$ for term `term_i` in document `timestamp_j`.

Index	Term occurrences in:
`m[0]`	URL
`m[1]`	`<title>` tag
`m[2]`	`<h1>`, `<h2>`, `<h3>`, `<h4>` tags
`m[3]`	`<b>`, `<strong>` tags
`m[4]`	other

Table 5.1: Term occurrences in the markup array.

Listing 5.4 show the structure of the *documents* table in the IndexedDB. `u` denotes the URL of the web site, `d` the duration in seconds spent there, `c` the HTML stripped out content of the page and `t` its title. `cd` represent the color histogram of the web page's screenshot. As the screenshot is in full RGB range, and assigning a pixel count to over $(256^{256})^{256}$ different colors is infeasible, each pixel will be compared to all colors in Table 5.2. The color codes are taken from Google's Image Search color palate [38]. The counter for the color which has the highest similarity to the currently processed pixel gets increased. To compute the similarity between two color codes the CIEDE-2000 color difference algorithm [70] is used. Therefore, the colors have to be translated from the RGB to the LAB color space [32]. The color distribution is used in section 5.6 to sort the search results by color.

```
...
<timestamp_j>:
    u: <http://www.example.com/>
    d: 120
    c: "Lorem ipsum dolor ..."
    t: "Example page"
    cd:
        color_1: 0.3
        color_2: 0.1
        ...
        color_12: 0.02
<timestamp_(j+1)>:
    ...
...
```

Listing 5.4: Structure of the documents table.

Color	RGB
color$_1$	`#C70000`
color$_2$	`#FC7F00`
color$_3$	`#FFFF07`
color$_4$	`#1FC40D`
color$_5$	`#19B5BA`
color$_6$	`#0000FF`
color$_7$	`#641C96`
color$_8$	`#FC83B0`
color$_9$	`#FFFFFF`
color$_{10}$	`#878787`
color$_{11}$	`#000000`
color$_{12}$	`#784211`

Table 5.2: Colors used to determine color histogram of web page screenshots.

The *screenshots* table contains of a simple `<timestamp_j> : <base64 image>` mapping, where the screenshot content is converted into a base64 ASCII string. To reduce storage space, each screenshot is shrinked and cropped to 250x140 Pixel and saved as a JPEG image. The average screenshot size per document is 30kB. The screenshot was separated from the document object, because querying for content yields that all matching documents (and their screenshots) will be loaded into memory. As the user will only see a fraction of all results, the screenshots are stored in a separate table to improve the performance.

Wrapping IndexedDB

The IndexedDB structure provides low level functions to insert, update, retrieve and delete key-value pairs. Each action has to be wrapped in a transaction to avoid conflicts of concurrent access. Subsequent actions on the same table are wrapped in the same transaction to increase their performance. The file `db.coffee` is an IndexedDB wrapper, which abstracts the low level API calls and uses simple caching to reduce latency. Listing 5.5 shows the abstraction on how to retrieve an object from the IndexedDB.

```
class DB
    ...
    get_object: (name, id, doneCallback, failCallback) =>
        trans = @db_object.transaction [name], "readwrite"
        table = trans.objectStore name
        req = table.get id

        req.onsuccess = -> doneCallback?(req.result)
        req.onerror = failCallback
    ...
```

Listing 5.5: Retrieving one object from an arbitrary table in the IndexedDB.

The function `get_object` takes the table name, the id of the search object, a success and failure callback. `@db_object` denotes an opened IndexedDB connection. Calling the `transaction` function of this object creates a new read/write transaction for all table names in the array. `table.get` retrieves the object with `id` as identifier from `table`. This operation initiates asynchronous handling and calls `req.onsuccess` or `req.onerror` in the success or error case respectively. In case `doneCallback` is a function and defined, it gets called with the result as argument.

Besides `get_object`, the `DB` class provides the function shown in Table 5.3. `get_array`, `get_all` and `insert_array` use a single transaction to batch process a collection of requests. For instance, the latter function is used to insert an array of terms in the database. As IndexedDB does not overwrite or update an existing object with the same identifier, but throws an exception, the function has to check before whether an `id` is present. To reduce the amount of `get` requests, all identifiers of all objects stored in the database are kept in memory, in the instance variable `keys`. `keys` is a JS object and the presence of a key can be checked in $O(1)$.

Name	Arguments
`get_array`	`name`: String, `ids`: Array, `doneCB`: Func, `failCB`: Func
`get_all`	`name`: String, `limit`: Int, `doneCB`: Func, `failCB`: Func
`insert_object`	`name`: String, `object`: JS object, `doneCB`: Func, `failCB`: Func
`insert_array`	`name`: String, `objects`: Array, `doneCB`: Func, `failCB`: Func

Table 5.3: Functions provided by `db.coffee`.

5.4 Storing Content

This section describes how the content of one visited web site is preprocessed and stored in the database. Every visit consists of the following basic steps:

1. The user opens a new web site by entering a new URL in the address bar or clicking a link of an already opened page;
2. she interacts with the opened web site, i.e., scrolling, hovering and clicking content; and
3. finally closes the opened page.

New web Site visited

The content preprocessing procedure is triggered as soon as the user has *actively* spend more than 3 seconds on a newly visited web site. In some cases, the browser window or tab can be opened in the background, e.g., when *CMD/CTRL-clicking* a link, and it is hidden from the user's eyesight. From the moment the DOM tree of the page was fully loaded, it takes 3 seconds until the `content_handler` triggers the `chrome.tabs.captureVisibleTab(...)` [35] function to capture a screenshot of the web site. The Base64 encoded image string is sent to the background page as shown in Listing 5.2.

Capturing User Interaction

Section 4.1 outlined how the ranking factors $markup$, $content_{visibility}$, $content_{hovering}$ and $content_{clicking}$ are measured and represented. This subsection gives a brief overview how these values are captured using the *jQuery* library. jQuery provides functions to modify the DOM structure, and events to get modified when the DOM get changed by user's actions. As each of the mentioned ranking factors are measured for each term, the DOM tree has to be traversed first in order to identify all occurring term. The file `content_handler.coffee` contains the function `_init_nodes`, which assigns an identifier to each text-wrapping HTML tag (Listing 5.6).

```
class ContentHandler
    ...
    visibility_vector: {}
    hover_vector: {}
    click_vector: {}
    ...
    _init_nodes: ->
        counter = 1
        _rec = (node) =>
            return if @_is_dead_end(node)

            # mark node, if it contains textual content
            if node.textContent.trim().length > 0
```

```
                # assigning node a sequential identifier which is
                # evaluated in latter code
                $(node).data("mm-id", counter)

                # initialize measurement factors vector
                @visibility_vector[counter] = 0
                @hover_vector[counter] = 0
                @click_vector[counter] = 0

                counter++

            _rec(child) for child in node.childNodes

        # start traversing at <body> tag
        _rec document.body
    ...
```

Listing 5.6: Traversing and identifying all nodes in the DOM tree of the current document.

`_init_nodes` has identified all potentially interesting nodes by assigning them a sequential identifier in the HTML attribute `data-mm-id`. The following DOM nodes are not considered in the traversal as they do not wrap visible textual content: `script`, `br`, `hr`, `noscript`, `input`, `img` and `link`. Every time the user interacts with one DOM element, by hovering or clicking it, or even if the element is visible on the screen, the corresponding `*_vector` JS objects are modified, as in Listing 5.7. If any DOM element (marked by `*`) is clicked, the event handler gets called. In case the element is of interest (the `data-mm-id` is set), the `click_vector` object is incremented at the respective identifier.

```
class ContentHandler
    ...
    constructor: ->
        ...
        $(document).on "click", "*", (ev) =>
            id = $(ev.target).data("mm-id")
            @click_vector[id]++ if id?
        ...
```

Listing 5.7: Capturing all clicks inside the current document.

Capturing hover events is done as in Algorithm 5. Measuring content visibility is done as described in Section 4.1. Before the user closes the currently opened web site, by navigating to new URL or closing the browser tab, the content handler sends the normalized vectors and the corresponding terms to the background page.

Sending Content and Captured Factors to Background Page

After the user decides to leave the current web page by closing it, the content and captured factors are sent to the background page, where they are stored in the IndexedDB. The values in the variables `visibility_vector`, `hover_vector` and `click_vector` are normalized by dividing each value by the sum of all values. Furthermore, each term’s occurrence in URL, web site’s title, headings, bold markups and normal text is counted and stored as shown in Table 5.1. Listing 5.8 shows how to build the data transfer object which gets stored in the database.

```
class ContentHandler
    ...
    constructor: ->
        ...
        $(window).on "unload", (ev) =>
            dto =
                d: @duration
                type: "store"
                url: document.location.href
                title: document.title
                terms: @summarize_terms()

            # sending DTO to background page
            chrome.extension.sendMessage dto

        ...



    summarize_terms: =>
        terms = {}

        for id, value of @visibility_vector
            factors =
                v: value
                c: @click_vector[id]
                h: @hover_vector[id]

            # getting node if corresponding ’mm-data-id’
            node = @nodes_dict[id]
            for child in node.childNodes
                # discard non textual child nodes
                continue if child.nodeName != "#text"

                terms = @split_text_to_terms(child.wholeText.trim())
                for term in terms
                    terms[term] ?=
                        f: factors
                        l: @term_location[term]
                        m: [0, 0, 0, 0, 0]

                    # counting occurrences in URL/title
                    terms[term].m[0] =
                        @count_in_string(document.location.href, term)
                    terms[term].m[1] = @count_in_string(document.title, term)

                    # counting occurrences other markups
                    if ["H1","H2","H3","H4"].indexOf(node.nodeName) >= 0
```

```
                        terms[term].m[2]++
                    else if ["B", "STRONG"].indexOf(node.nodeName) >= 0
                        terms[term].m[3]++
                    else
                        terms[term].m[4]++

        return terms
```

Listing 5.8: Building the DTO which gets sent to the background page for storing.

The variable `nodes_dict` is a dictionary, which maps `mm-data-ids` to DOM nodes. The function `split_text_to_terms()` splits the textual content of one HTML tag into several tokens. Each token is preprocessed by applying the Porter Stemmer algorithm ([80], [59]) to reduce them to their stem, after which they are referred to as terms. The dictionary `term_locations` maps one term to an array of word wise integer locations in the document's content. `count_in_ string()` returns the amount of occurrences of the second argument in the first argument.

On the side of the background page, the data transfer object is split and a new entry is created in the *documents* table with the web page's URL, its title and its plain HTML-stripped content. For each term the *terms* table is either updated or a new entry is inserted. The storage process runs asynchronously and does not interfere with the user's surfing experience.

5.5 Querying

At this point, the user has already documents stored in the database to query for. The query process consists of the following steps:

1. The user types a string in the query text field,
2. the query string is split into tokens, and each token is preprocessed into a term,
3. each document in which at one term occurs is added to the unranked result list.

This implementation only searches for terms independently, and does not offer a way to group multiple terms together to a so called free text query, [57]. Listing 5.9 shows how to get an unranked list of documents containing at least one search term. Additionally, for each document in the list, the $tfidf$ value is calculated, which will be evaluated for the document score in Section 5.6.

```
    class Engine
        ...
        query: (q) =>
            @get_term_objects_from_query(q).then (term_objects) =>
                documents = {}
                for term_object in term_objects
                    for doc_id, values of term_object.d
                        tf = Math.log(values.l.length) + 1
                        idf = Math.log(@DB.documents_count /
                            Object.keys(term_object.d).length) + 1

                        term_occurrence =
                            term: term_object.id
                            tfidf: tf * idf
                            values: values

                        documents[doc_id] ?= []
                        documents[doc_id].push(term_occurrence)

                # compute document score
                ...
        ...
```

Listing 5.9: Retrieving all documents in which at least one query term occurs.

The function `get_term_objects_from_query` splits the query string into alphanumerical tokens. Each token is preprocessed by the Porter Stemmer algorithm, and the corresponding term object from the *terms* table is returned. The term object contains of the dictionary `d`, which maps document identifiers to their behavioral ranking factors. The $tfidf$ value is calculated for each *(document, term)* pair and added to the *documents* variable. In case one query term occurs only in one document, its *term frequency* logarithm is 0. Therefore, the value is corrected by adding 1.

5.6 Ranking

In this final step of, the documents which match the user's query have to be ranked according the their document score, as defined in Equation 4.5. The document score is evaluated by considering the cosine similarity of the query's and document's $tfidf$ values and their weighted behavioral ranking factors.

Computing the Cosine Similarity

The cosine similarity is calculated by the normalized query and document vector of $tfidf$ values, as defined in Equation 2.4. The $tfidf$ values for the query terms are calculated analogous to those of the document, as shown in Listing 5.9, although the tf part only contains the term frequencies of the query and not those of the document. Listing 5.10 shows how the cosine similarity is calculated.

```
class Engine
    ...
    cosine_similarity: (term_objects, document) =>
        # calculates the Euclidean distance.
        _get_length(objects) = ->
```

```
            sum = 0
            for id, tfidf of objects
                sum += Math.pow(tfidf, 2)
            return Math.sqrt(sum)

        len_q = _get_length(term_objects)
        len_d = _get_length(document)

        similarity = 0
        for term, term_tfidf of term_objects
            term = term_object.id
            if document[term]?
                similarity += term_tfidf * document[term]

        # normalizing cosine similarity
        return similarity / (len_q * len_d)
    ...
```

Listing 5.10: Computing the cosine similarity between a query and document.

Computing the Behavioral Factors Value

The second integral component of the document score is the evaluation of the behavioral factors. Listing 5.11 shows how to obtain the final score under consideration of the influence parameter `p`.

```
class Engine
    ...
    query: (q) =>
        ...
        # compute document score
        for doc_id, terms of documents
            cos_sim = @cosine_similarity(term_objects, terms)

            sum_behavior = 0
            for term_object in term_objects
                v = term_object.d[doc_id]
                if v?
                    sum_behavior +=
                        v.m[0] * @w.markup_url +
                        v.m[1] * @w.markup_title +
                        v.m[2] * @w.markup_head +
                        v.m[3] * @w.markup_bold +
                        v.m[4] * @w.markup_other +
                        v.f.v * @w.content_visibility +
                        v.f.c * @w.content_clicking +
                        v.f.h * @w.content_hovering

            documents[doc_id].score = (1-@p) * cos_sim + @p * sum_behavior
        ...
```

Listing 5.11: Computing the final document score for the result ranking.

The variable `v` contains the ranking factors $markup$ (`m`), $content_{visibility}$ (`f.v`), $content_{clicking}$ (`f.c`) and $content_{hovering}$ (`f.h`). Each of those factors is multiplied by their correspond weights, defined in `w`. The weights are set as shown in Table 5.4 as are chosen arbitrarily. For a proper implementation those weights need to be adapted with machine learning techniques for instance, which are not scope of the thesis and referred to as future work in Chapter 7. Also a proper value for the influence coefficient p has to be determined similarly. In this implementation the value is set to 0.7.

Type	Value
`w.markup_url`	0.2
`w.markup_title`	0.15
`w.markup_head`	0.15
`w.markup_bold`	0.05
`w.markup_other`	0.025
`w.content_visibility`	0.175
`w.content_clicking`	0.15
`w.content_hovering`	0.1

Table 5.4: Weight distribution of ranking factors.

Ranking the documents

In the next step, the documents have to be ordered by their score. This implementation ranks documents, which contain all k terms of the query higher than documents which contain only $k - 1$ terms, regardless their document score. Listing 5.12 shows how to consider this situation and how to "subrank"documents of the same term occurrence.

```
class Engine
    ...
    query: (q) =>
        ...
        documentsArray.sort (docA, docB) ->
            if docA.term_occurrences.length >
                                docB.term_occurrences.length
                return -1
            else if docA.term_occurrences.length <
                                docB.term_occurrences.length
                return 1
            else
                return docA.score - docB.score
    ...
```

Listing 5.12: Ranking the documents by their term occurrence and document score.

Before applying the JavaScript/CoffeeScript internal sort algorithm, the `documents` dictionary has to be transformed into an array, called `documentsArray`. The ECMAScript standard does not define which sort algorithm to use in the JavaScript implementation. Thus the algorithms may differ from browser to browser [45].

Retrieving Inner-Document Matches

At this step, the information retrieval process returned a sorted set of documents, which are ordered by the similarity of the query to each document and the measured behavioral factors. The result list does not display yet where the query matches the content in the document. To solve this problem, the positional intersection algorithm is used to compute the location in the document where $term_i$ and $term_{i+1}$ are less than k words apart. As $term_i$ and $term_{i+1}$ can occur several times in the document at different positions and with different gap lengths, the inner-document matches are ranked by their gap size in ascending order. To illustrate this problem, the inner-document ranking of content matches is shown in two examples on the document d: *"Well, if you like burgers give them a try sometime. Me, I can't usually get them myself because my girlfriend is a vegetarian which pretty much makes me a vegetarian. But I do love the taste of a good burger. Mmm."*[1]. For each example, the maximum gap size k is set to 10 and the size of the neighborhood n is set to 5. n defines the amount of neighbor terms of $term_i$ to display as query matching content. E.g. $n = 3$ displays $term_{i-n}$ ... $term_i$ ... $term_{i+n}$.

- **Example 1**: Single term query *q*: *"burger"*. *burger* occurs twice in the document, which results into 2 query matching content excerpts: *"if you like **burgers** give them a"* and *"of a good **burger**. Mmm."*. As only a single term occurs in the query, each match is equally important. Thus, the gap is set to *k*, which does not affect the ranking in this case.
- **Example 2**: Two term query *q*: *"girlfriend vegetarian"*. *girlfriend* occurs once in the document and *vegetarian* twice. The matches are: *"myself because my **girfriend** is a **vegetarian** which pretty much"* and *"makes me a **vegetarian**. But I do"*. The gap of the first match is 2 and in the second match it is set to *k* (as in example 1). The first match is ranked before the second match.

The examples above demonstrate fairly simple scenarios to obtain the gap between two terms, which can degenerate into many `if-then-else` cases when different documents and queries are chosen. For example, matches of one document can overlap when using the query *"pretty girlfriend"*. This results into *"myself because my **girlfriend** is a vegetarian"* and *"a vegetarian which **pretty** much makes me"*. Both matches show partly overlapping content which should be merged to a single match: *"myself because my **girlfriend** is a vegetarian which **pretty** much makes me"*.

Asynchronous Loading of Screenshots

The search results are transfered by *simple one-time requests* from the background page to the content page, which displays the results. The latter is an HTML file in which the results get in-

---

[1] Quote by Jules Winfield from the movie Pulp Fiction (1994) by Quentin Tarantino

jected into the `<body>` tag the farther the user scrolls down. While all results are transfered to the content page in one chuck the documents' screenshots are loaded asynchronously. Assuming the query for *"Kill Bill"* returns 2.000 results, where each document has an average screenshot size of 30kb, only the image material would be approximately 60MB large. Although all data is transfered from one thread to another on one device, it is still noticeably slow and cause the user's screen to freeze until the process is finished. The reason for this is that each screenshot has to be loaded from the database separately.

According to a study from *Optify Inc.* [42], which analyzes leaked search logs from AOL, the click-through rate of search results decreases from 42.30% of page ranked first to 2.97% of page ranked 10th and 0.30% of page ranked 20th. Instead of loading all screenshots at once, it is more efficient to load them when the corresponding result is displayed in the result list. The screenshots are loaded in chunks of 10. Loading a new chunk is triggered as soon the last result of the current chunk becomes visible on the user's screen. Listing 5.13 illustrates how to load chunks of screenshots.

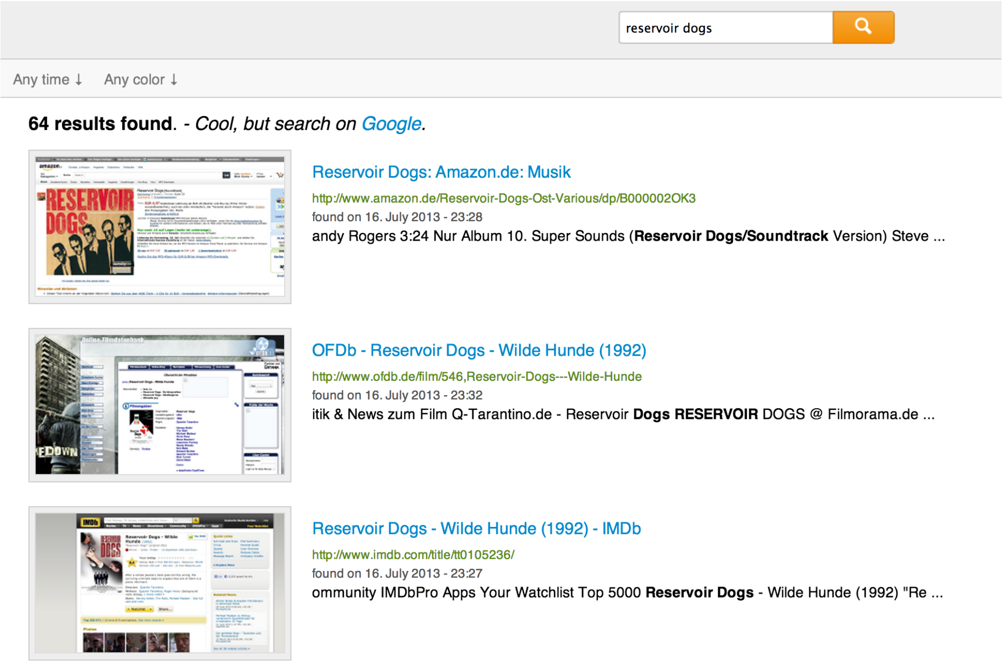
```
load_screenshots = (results, doneCallback, failCallback) ->
    dfds = []

    for result in results
        do (result) ->
            dfd = When.defer()
            dfds.push dfd.promise

            engine.get_screenshot(result.page_id).then (image) ->
                result.screenshot = image
                dfd.resolve()
            , failCallback

    When.all(dfds).then ->
        doneCallback(results)
```

Listing 5.13: Loading chunks of screenshots using Promise/A+ objects.

The function `load_screenshots` is called inside the `main.coffee` file and utilizes the *when.js* library [10] to get notified when all asynchronous tasks are finished. `engine.get_screenshot` initiates an asynchronous process and calls the callback defined in the `then` function. `When.all(dfds)` calls its callback as soon all screenshots are obtained for each entry in `results`.

Filtering Results by Colors

The purpose of storing screenshots and the color distribution of the screenshots is, primarily, so that users do not have to click on a result link in order to view the page, and, secondly, to sort the results by the distribution of one color in the screenshot. The user is given a set of 12 colors from which she can re-rank the results, as in Table 5.2. Each document has a certain color distribution stored in the `cd` field of one entry in the *documents* table (Listing 5.4). The color distribution is calculated as described in Section 5.3. The results only need to be sorted in descending order by the distribution of the color which the user has selected. Figure 5.3 shows the results of the query *"reservoir dogs"* considering the document score, and Figure 5.4 displays the results sorted by the distribution of the color *red*.



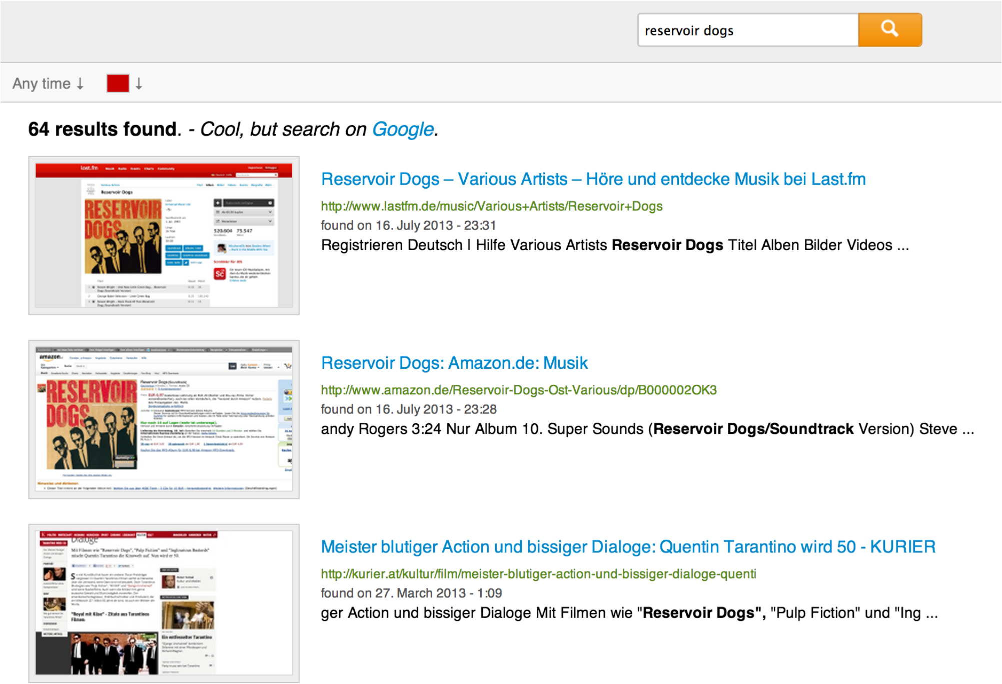
64 results found. - *Cool, but search on Google.*

Reservoir Dogs: Amazon.de: Musik
http://www.amazon.de/Reservoir-Dogs-Ost-Various/dp/B000002OK3
found on 16. July 2013 - 23:28
andy Rogers 3:24 Nur Album 10. Super Sounds (**Reservoir Dogs/Soundtrack** Version) Steve ...

OFDb - Reservoir Dogs - Wilde Hunde (1992)
http://www.ofdb.de/film/546,Reservoir-Dogs---Wilde-Hunde
found on 16. July 2013 - 23:32
itik & News zum Film Q-Tarantino.de - Reservoir **Dogs RESERVOIR** DOGS @ Filmorama.de ...

Reservoir Dogs - Wilde Hunde (1992) - IMDb
http://www.imdb.com/title/tt0105236/
found on 16. July 2013 - 23:27
ommunity IMDbPro Apps Your Watchlist Top 5000 **Reservoir Dogs** - Wilde Hunde (1992) "Re ...

Figure 5.3: Result list for query *Reservoir Dogs*.



64 results found. - *Cool, but search on Google*.

Reservoir Dogs – Various Artists – Höre und entdecke Musik bei Last.fm
http://www.lastfm.de/music/Various+Artists/Reservoir+Dogs
found on 16. July 2013 - 23:31
Registrieren Deutsch I Hilfe Various Artists **Reservoir Dogs** Titel Alben Bilder Videos ...

Reservoir Dogs: Amazon.de: Musik
http://www.amazon.de/Reservoir-Dogs-Ost-Various/dp/B000002OK3
found on 16. July 2013 - 23:28
andy Rogers 3:24 Nur Album 10. Super Sounds (**Reservoir Dogs/Soundtrack** Version) Steve ...

Meister blutiger Action und bissiger Dialoge: Quentin Tarantino wird 50 - KURIER
http://kurier.at/kultur/film/meister-blutiger-action-und-bissiger-dialoge-quenti
found on 27. March 2013 - 1:09
ger Action und bissiger Dialoge Mit Filmen wie "**Reservoir Dogs",** "Pulp Fiction" und "Ing ...

Figure 5.4: Result list for query *Reservoir Dogs*, ordered by *red* color distribution.

CHAPTER 6

Evaluation

6.1 Evaluation of Different Parameter Configurations

An experiment has been conducted to measure the structural and behavioral ranking factors presented in Section 4.1 on a real life use case, with the focus of extracting information about the significance of each factor result ranking. The participants had to install a browser extension, which tracked those factors mentioned before while surfing the internet. The browser extension was provided for Google Chrome (until version 29.0.1547.76) and Mozilla Firefox (until version 23.0.1). Each candidate had to answer ten questions, labeled as tasks, with the help of Google. The candidates were asked to rate each web page they have visited on a scale from 1 to 5, depending on how much it had helped to answer the current question.

The questions were taken and adapted from an experiment conducted at the University of Massachusetts [22]. Those questions were carefully selected to be not easily answerable by an obvious query on a web search engine. Additionally, the researches wanted to raise the user's frustration while using a web search engine to answer the questions. The intention of this experiment was not to frustrate the participants, but rather to force them examining a higher amount of web sites before find the correct answer. The following questions were used for the experiments:

1. *What is the average temperature in Vienna / Madrid / Frankfurt / Rome for summer / winter?*

2. *Name three bridges that collapsed in Germany / the US since 2001 / 2007.*

3. *In what year did Austria / Germany experience its worst drought? What was the average precipitation in the country that year?*

4. *How many pixels must be dead on a MacBook before Apple will replace the laptop? Assume the laptop is still under warranty.*

5. *Is the band Snow Patrol / Green Day / Goo Goo Dolls coming to Austria / Germany within the next year? If not, when and where will they be playing closest?*
6. *What as the best selling TV (brand model) of 2012?*
7. *Find the opening hours of MediaMarkt / Saturn nearest to Mannheim, Germany / Graz, Austria.*
8. *How much did the ATX / DAX increase / decrease at the end of its last trading day?*
9. *Find three coffee shops with WI-FI in Vienna 1st district, Austria / Mannheim, Germany.*
10. *Name four places to get a car inspection.*

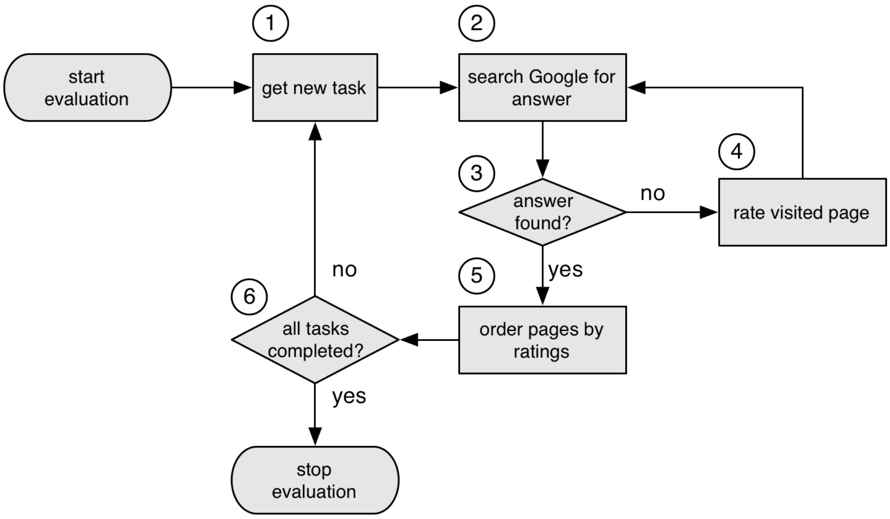
The questions contain the same structure but different instances, e.g., in question 3 one instance asks about the worst drought in Austria and the other about Germany. The participants had to answer only one instance of a question, but the choice of the instance was randomized to limit the possibility to exchange answers with other participants. As mentioned above, the questioned were localized to Central Europe. Depending on the browser setting, the questions were presented in either English or German language, as some candidates did not feel confident enough to search only English written web pages.

The rating classification of visited web pages was also taken from [22] and include the following values:

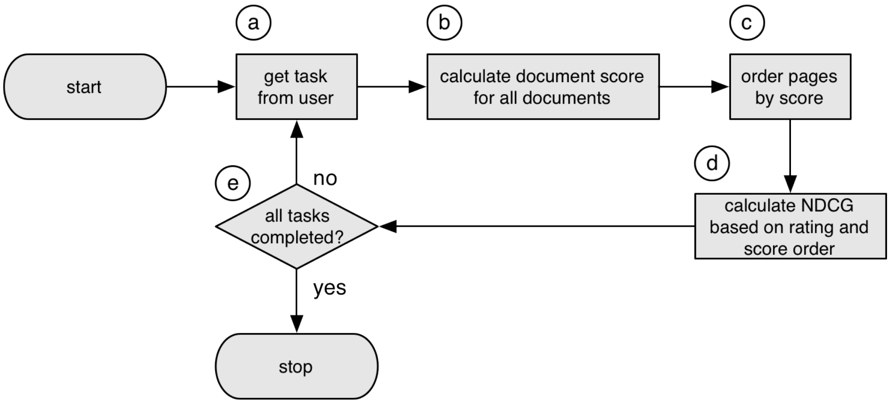
- **1 star** - Did not satisfy the information need at all.
- **2 stars** - Barely satisfied any of the information need.
- **3 stars** - Only partially satisfied the information need.
- **4 stars** - Satisfied most of the information need.
- **5 stars** - Completely satisfied the information need.

After the participant finished one task, she was prompted with a dialog to enter the answer for the current question. Each subject was told that there is no right or wrong answer and the correct answer solely depends on the personal interpretation of the question. This surprisingly lead to very divergent answers, but did not matter for the measurement of the ranking factors. Figure 6.1a depicts the question answering process for the participants.

At step ① the participant receives a new task from the browser extension, which contains a question she should answer. For each question, Google has to be queried to find the document with the perfect answer, step ②. While surfing web pages, the browser extension measures the ranking factors presented in Chapter 4. In case no answer was found, step ④, the user is prompted to rate the currently visited web page and has to continue at step◦ until the question can be answered. At step ⑤, all documents for the current task are ordered according to their



(a) User rating process


ⓐ
ⓑ
ⓒ
start
get task from user
calculate document score for all documents
order pages by score
ⓓ
calculate NDCG based on rating and score order
ⓔ
all tasks completed?
no
yes
stop

(b) Document score calculation process

Figure 6.1: Design of the question answering experiment.

ratings in descending order. Unless the user has completed all tasks, she continues the process at step ①.

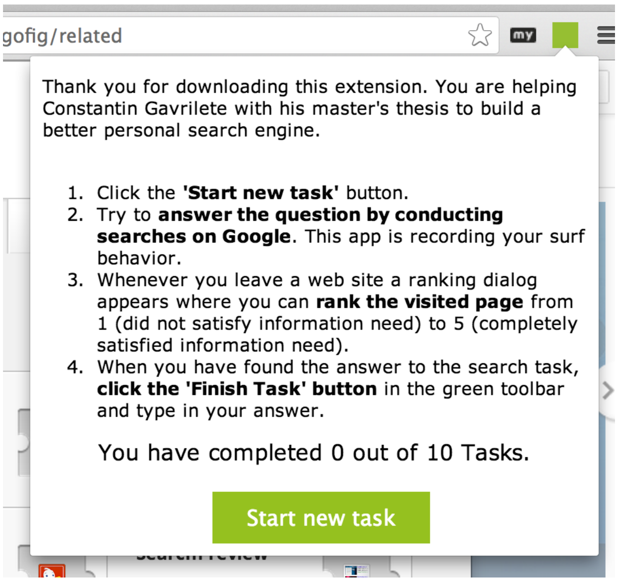
After each participant has completed each task, the document score calculation process, as shown in Figure 6.1b, is executed to evaluate the significance of each ranking factor. At step ⓐ, the documents of each submitted task are examined. The document score is calculated based on the currently evaluated ranking factor, step ⓑ. The documents for one task, for one user, are ordered by their score in descending order, at step ⓒ. Based on the document order of step ① and the order of step ⓒ, the NDCG value can be obtained, at step ⓓ. The closer the NDCG value approximates 1 the more similar both document ranking are.

Implementation

This Section uses the core implementation presented in Chapter 5, such as the ranking factor measurement and the ranking algorithm, but uses a different interface on top to retrieve the manual web site ratings. Furthermore, for the sake of evaluation, the results have to be collected centrally, which is not provided in the former implementation. Similarly to Chapter 5 this evaluation browser extension was implemented using CoffeeScript and the Grunt.js build tool. Additionally the code base was written to allow an easy deployment to Google Chrome and also Mozilla Firefox, as the implementation for those browsers are very similar and they both cover in total over 60% of the worldwide browser market [75].

The core of the implementation is the `content_handler.coffee` file, which gets injected into the currently accessed web page. As in Chapter 5, this files takes care of capturing ranking factors based on term level. For each term the click count, mouse hover duration, fraction of visibility and duration of visibility is measured.

When the browser extension gets installed it generates an identifier for the user, which is simply the timestamp provided by the JavaScript function `new Date().getTime()`. No other information about the identity of the user is captured. Additionally, the question instances were chosen at install time. A popup window showed the necessary steps to complete this survey, as shown in Figure 6.2



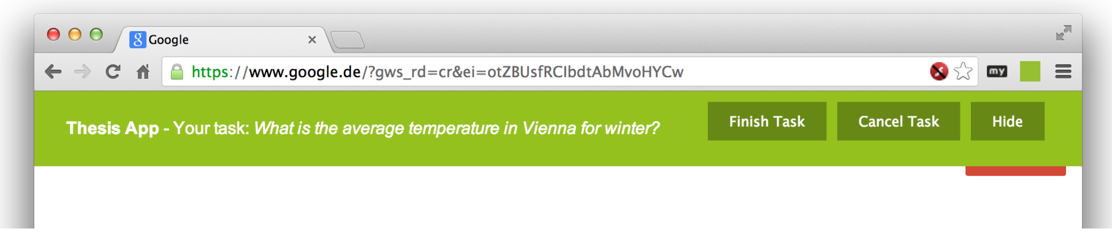
Thank you for downloading this extension. You are helping Constantin Gavrilete with his master's thesis to build a better personal search engine.

1. Click the **'Start new task'** button.
2. Try to **answer the question by conducting searches on Google**. This app is recording your surf behavior.
3. Whenever you leave a web site a ranking dialog appears where you can **rank the visited page** from 1 (did not satisfy information need) to 5 (completely satisfied information need).
4. When you have found the answer to the search task, **click the 'Finish Task' button** in the green toolbar and type in your answer.

You have completed 0 out of 10 Tasks.

Figure 6.2: Popup which gives the user instructions how to proceed with the experiment.

A click on the button *"Start new task"* displayed the next question the user had to answer. While surfing the current question was permanently displayed in the upper part of the screen, as shown in Figure 6.3.

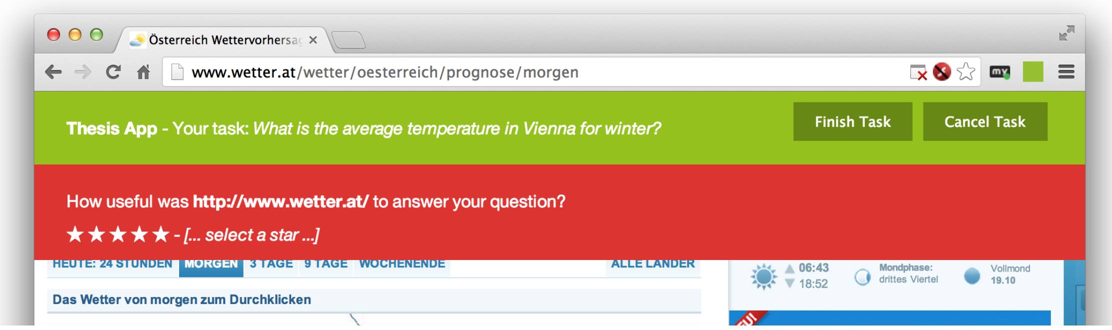

https://www.google.de/?gws_rd=cr&ei=otZBUsfRCIbdtAbMvoHYCw

Thesis App - Your task: *What is the average temperature in Vienna for winter?*

Figure 6.3: Question displayed to the user in upper part to screen.

The user had the possibility to either finish or cancel the task, or hide the green question bar to not get too distracted. A click on *"Finish Task"* prompted a dialog where the user had the answer to question. No specific format was required, as the answer is not relevant for the evaluation. This input prompt was merely meant to be a hoop for the user to deal with a task properly. In case the user would have been interrupted to complete the task, the button *"Cancel Task"* deleted all tracked information about answering the current query, so the user would not submit skewed measurements.

After leaving one website, the user was prompted to rate the previously visited page according to the extend the information need was satisfied, as shown in Figure 6.4. When clicking *"Finish Task"* an implicit rating of 5 stars was save, since it was assumed the last page helped to give the final answer.


www.wetter.at/wetter/oesterreich/prognose/morgen

Thesis App - Your task: *What is the average temperature in Vienna for winter?*

How useful was **http://www.wetter.at/** to answer your question?

★★★★★ - *[... select a star ...]*

Figure 6.4: The 1-5 star rating for the previously visited website is shown in the red area.

After finishing one task, the captured factors for the visited web sites of that current task were sent to a sever. The format of the sent data is shown Figure 6.5. `task.resultText` denotes the answer the user has entered into the input prompt and `task.taskQuery` the instance of the question for that task, `page.duration` is the duration in milliseconds or dwell time the user has spent on the current page, and `term.factor_*` and `term.markup_*` are the measured ranking factors as introduced in Chapter 4.

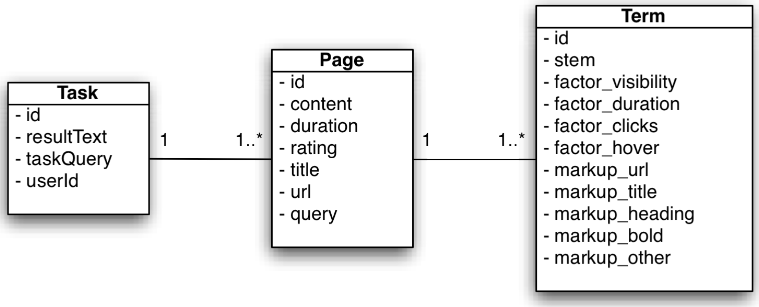


Figure 6.5: Class diagram of the data sent from the extension to the server when finishing a task.

Having the measured ranking factors and an explicit human page rating for a specific task, the weight factors, which were arbitrarily chosen in Table 5.4, can now be changed in a way to improve the *Normalized Discounted Cumulative Gain (NDCG)* to approximate to the ideal DCG value, as demonstrated in Section 2.4. Each visited and rated document is stored in the inverted index for later evaluation. To compute the NDCG value, each query from `pages.query` is rerun against the document collection to get the DCG value and divided by the IDCG value which is computed out of the human rated documents.

Result Discussion

A total amount of 181 tasks have been submitted by 28 users, who have been asked to participate in that experiment, where 16 participants could complete 8 or more tasks. For this evaluation it did not matter how many tasks one user had completed, but rather how many tasks had been submitted in total. Table 6.1 shows a statistic about the captured data. Table 6.2 shows how many users have finished each task, and how many pages have been visited on average to answer each question.

The remainder of this section shows different ranking factor weight configurations and their calculated NDCG value. As summarized in Section 2.4, the closer the NDCG value is to 1, the more likely the ranking method presented in Equation 4.5 ranks their result documents according to the human relevance ratings, e.g., documents rated with 5 stars should appear first in the result list followed by 4 stars rated documents etc. Table 6.3 gives an overview of the calculated NDCG

Total amount of tasks submitted	181
Total amount of pages submitted	531
Total amount of terms submitted	290.000
Average duration spend on one website	34s

Table 6.1: Base statistics about captured data.

Question	# of users who solved question	AVG # of visited pages for question
What is the average temperature in Vienna / Madrid / Frankfurt / Rome for summer / winter?	28	1.75
Name three bridges that collapsed in Germany / the US since 2001 / 2007.	20	2.45
In what year did Austria / Germany experience its worst rought? What was the average precipitation in the country that year?	15	5.27
How many pixels must be dead on a MacBook before Apple will replace the laptop? Assume the laptop is still under warranty.	19	2.26
Is the band Snow Patrol / Green Day / Goo Goo Dolls coming to Austria / Germanywithin the next year? If not, when and wherewill they be playing closest?	14	6.29
What as the best selling TV (brand & model) of 2012?	16	3.88
Find the opening hours of MediaMarkt / Saturn nearest to Mannheim, Germany / Graz, Austria.	18	2.17
How much did the ATX / DAX increase / decrease at the end of its last trading day?	17	2.29
Find three coffee shops with WI-FI in Vienna 1st district, Austria / Mannheim, Germany.	18	2.61
Name four places to get a car inspection.	16	2

Table 6.2: Statistics about each task / questions.

values for each user while only evaluating one ranking factor. Unlike for the document score calculated by Equation 4.5 in this calculation only the behavioral and structural ranking factors are evaluated and not the $tfidf$ component of the equation.

As shown in Table 6.3, every NDCG value for each ranking factor, is almost close to 1.0, which means that each factor is individually distinctive enough to rank the documents of one task in the same order, as provided by the human page ratings, at step ④ of Figure 6.1a. This proves the hypothesis, that the degree of personal relevance correlates with the user's interactivity with the document's content.

User	m_url	m_title	m_head	m_bold	m_other	f_cl	f_dur	f_hov	f_vis
user01	0.95	0.97	0.93	0.96	0.96	0.93	0.96	0.97	0.95
user02	0.97	0.97	0.97	0.97	1.00	0.97	1.00	0.97	1.00
user03	0.97	0.95	0.93	0.97	0.95	0.89	0.94	1.00	1.00
user04	0.97	0.96	0.95	0.98	0.96	0.94	0.93	0.96	0.96
user05	0.93	0.94	0.92	0.94	0.93	0.96	0.96	0.95	0.94
user06	0.94	0.88	0.85	0.85	0.85	0.81	0.90	0.94	0.91
user07	0.96	0.96	0.96	0.99	0.99	0.95	0.96	0.99	0.98
user08	0.94	0.85	0.83	0.84	0.82	0.87	0.79	0.88	0.88
user09	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.97
user10	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user11	0.96	0.96	0.95	0.95	1.00	0.95	0.96	0.96	1.00
user12	0.95	0.97	0.96	0.94	0.94	0.96	0.94	0.98	0.94
user13	0.95	0.95	0.93	0.93	0.97	0.94	0.92	0.92	0.93
user14	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user15	0.95	0.98	0.94	0.96	0.98	1.00	0.96	1.00	0.98
user16	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user17	0.99	0.96	0.97	0.95	0.98	0.95	0.98	0.97	0.98
user18	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
user19	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user20	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
user21	0.93	0.95	0.94	0.91	0.94	0.91	0.93	0.94	0.97
user22	0.78	1.00	0.78	1.00	0.78	0.78	0.78	1.00	0.78
user23	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user24	0.90	0.90	0.93	0.89	0.91	0.92	0.92	0.92	0.89
user25	0.94	0.99	0.97	0.96	0.97	0.94	0.95	0.98	0.95
user26	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user27	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
user28	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
AVG	**0.96**	**0.97**	**0.95**	**0.96**	**0.96**	**0.95**	**0.95**	**0.97**	**0.96**

Table 6.3: NDCG values separately for each ranking factor.

6.2 Benchmark of HTML5 Storage Structures

This section concludes the taken design decisions by measuring the performance of the most common use cases when using different HTML5 storage structures. *IndexedDB* [62] has been selected to be the superior large scale database since the W3C stopped working on the specifications for *WebSQL* [31] in November 2010. Despite the fact that the latter database never made it as a *W3C Candidate Recommendation*, it is the only HTML5 database which can be used on desktop and mobile Safari [19]. Table 6.4 shows the current implementation status of those databases for the most recent versions of Mozilla Firefox, Apple Safari and Google Chrome.

While Chrome supports both IndexedDB and WebSQL, the other browsers only provide one database implementation. Microsoft Internet Explorer is excluded intensionally since it does not support JavaScript extension development.

	IndexedDB	WebSQL
Firefox 23.0.1	X	
Safari 6.0.4 (8536.29.13)		X
Chrome 29.0.1547.65	X	X

Table 6.4: Browser support of HTML5 databases.

The benchmarks in this section compare the performance development of insert, update and search tasks executed on IndexedDB and WebSQL for each of the above mentioned browsers. The database layer described in Chapter 5 supports only IndexedDB, but can be easily replaced by another layer supporting WebSQL. Alternatively, an additional layer can be placed between the *Data Access Object* layer and the database implementation, which exposes the same API as IndexedDB does, but internally calls WebSQL functions. This abstraction procedure for not supported HTML5 features is called *polyfilling*. Facebook provided an IndexedDB polyfiller which uses WebSQL as the database implementation to support Apple Safari [34].

The only performance measurement is the execution time of one use case, which can be measured by the JavaScript functions `console.time()` and `console.timeEnd()` as shown in Listing 6.1.

```
console.time("expensive function");
expensiveFunc();
console.timeEnd("expensive function");

// result: 'expensive function: 121ms.'
```

Listing 6.1: JavaScript CPU execution profiling.

Benchmark Use Cases

There are several resources online which benchmark IndexedDB against WebSQL, e.g., [9], but almost all of them perform the tests on a very low level basis. This benchmark although groups several low level API calls to one use case and measures the execution duration of many subsequent use cases. The following use cases are benchmarked in the next sub section:

- **Insert**; the insertion occurs when the user leaves one website. All measured factors are transformed in their correspondent term and document values as shown in Listing 5.3 and 5.4 respectively. Typically, one document consists of many terms and one screenshot, so this use case consists of a variable count of low level insert operations for the term table, one insert for the document table and one for the screenshot table.

- **Update**; updates occur only in the term tables, e.g., when an already stored website is visited again. Assuming that the term is already present in the database, its document list (property d, as in Listing 5.3) has to include the newly visited web page.
- **Search**; the user only searches for terms, but expects a list of documents in return. The search task, therefore, first retrieves all matching terms from the database and from their d property all matching documents.

Results

The benchmark has been executed on an Apple MacBook Pro Retina with 2,7 GHz Intel Core i7 and 16 GB 1600 MHz DDR3 RAM for the following browsers: Mozilla Firefox 23.0.1, Apple Safari 6.0.4 (8536.29.13) and Google Chrome 29.0.1547.65. The results are summarized in Table 6.5 and plotted in Figure 6.6. As mentioned before, Safari does not support IndexedDB, as well as Firefox does not support WebSQL. The average web site has between 1000 and 3000 terms, so a best and worst case for 100 and 10000 terms respectively has been added.

Storage	**Action**	**# of terms**	**Firefox 23.0.1**	**Safari 6.0.4 (8536.29.13)**	**Chrome 29.0.1547.65**
IndexedDB	insert	100	2 ms	N/A	5 ms
		1.000	10 ms	N/A	34 ms
		10.000	93 ms	N/A	290 ms
	update	100	3 ms	N/A	9 ms
		1.000	29 ms	N/A	101 ms
		10.000	271 ms	N/A	926 ms
	search	10	11073 ms	N/A	45 ms
		50	11721 ms	N/A	4 ms
		100	11528 ms	N/A	5 ms
WebSQL	insert	100	N/A	42 ms	41 ms
		1.000	N/A	315 ms	201 ms
		10.000	N/A	2.828 ms	2.247 ms
	update	100	N/A	166 ms	175 ms
		1.000	N/A	1.758 ms	1.783 ms
		10.000	N/A	15.317 ms	16.959 ms
	search	10	N/A	1.253 ms	1.787 ms
		50	N/A	5.633 ms	9.455 ms
		100	N/A	11.084 ms	18.294 ms

Table 6.5: CPU execution time of insert, update and search tasks on HTML5 databases.

When comparing the IndexedDB results, Firefox is performing slightly better than Chrome for *insert* and *update* operations, but is slow for search operations. Write tasks are allowed to have a longer execution time, as they occur only once, when the user leaves a web page, and are processed in the background. Slow search operations, although, hinder a fluent search experience,

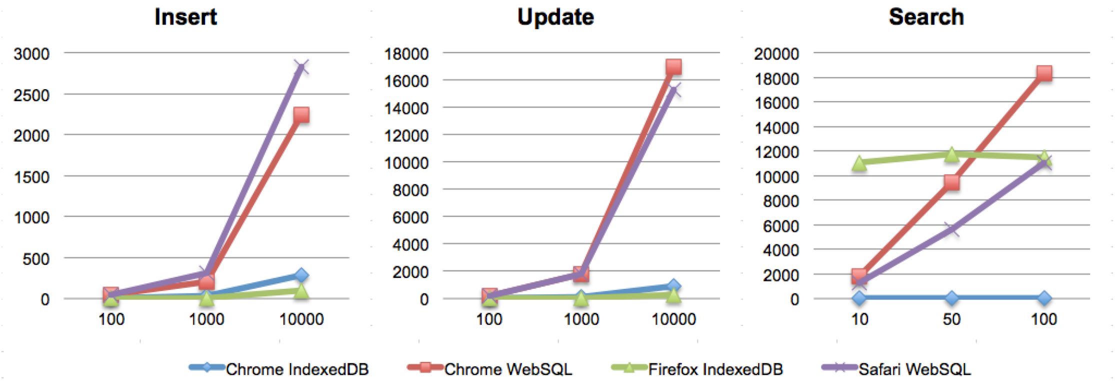


Figure 6.6: Plot of insert, update and search operations on different browser / storage combinations.

as it cannot be expected from the user to wait up to 11 seconds until the search engine returns results. This behavior can be avoided through proper caching of term - document mappings. Interestingly, after several runs of the search task, Chrome performs worse when searching for 10 terms, but is almost 10 times as fast when searching for 50 or 100 terms. Caching inside the V8 JavaScript engine could be a possible explanation.

For WebSQL Safari and Chrome are performing almost identically for insert and update operations, but Chrome is slightly slower in searching for documents.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

This thesis has summarized state-of-the-art concepts to design and build an information retrieval system. Furthermore it has analyzed related work in the area of *web search personalization* and *result reranking*, and introduced a method to retrieve the relevancy of a visited web site by analyzing the terms on that page. More specifically, how the user has been interacting with the web page's content by *clicking*, *hovering* or simply *seeing* information on her screen. Additionally, the markup structure of terms is considered when calculating the relevance value of one document.

A ranking method has been introduced, which, firstly, uses the traditional *term frequency inverse document frequency* in the *vector space model* to calculate the similarity between the query and the result documents, and secondly, considers the measured relevancy as an additional component to give a score to a document.

An implementation has been discussed how to build an information retrieval system for already seen content as a browser extension for Google Chrome. While Chrome's performance on insert, update and search operations on *IndexedDB* is the fastest, other browser-HTML5-storage combinations have proven to be too slow to assist the user in searches for content.

An online experiment has been conducted to get insight, how the measured structural and behavioral ranking factors influence the document ranking. The results have proven the hypothesis, that the degree of content interaction correlates with the personal relevance of one document.

7.2 Future Work

This thesis laid the ground work for a concept of a personal search engine, with the focus of re-retrieving already perceived content. The following list summarizes open research tasks, which could improve the accuracy and performance of the service:

- *Carrying out large scale evaluations.* For the experiment in this thesis, the surf behavior of 28 participants has been evaluated. A bigger user base needs to be tested, to find out which of the proposed ranking factors influences the ranking of documents the most.
- *Using implicit feedback instead of human judgments.* As the latter information is hard and expensive to get, implicit feedback can be obtained to determine the rating of one search result without interfering with the user's search behavior, so longer empirical tests can be executed.
- *Automatic learning of ranking factor weights.* As not every user has the same web surfing behavior, some ranking factors are more important to *userA* than to *userB*. In combination with implicit feedback and machine learning algorithms such as RankNet [8], factor weights can be learned automatically and individually for every user.
- *Estimating a proper value for p-coefficient.* The p-coefficient in the document score equation denotes the influence of traditional $tfidf$ weighting over behavioral and structural relevance factors. While binary values are not beneficial, a proper setting of p has to be examined and tested.
- *Implementing a desktop solution.* The configuration Google Chrome with IndexedDB as an HTML5 storage structure works well in practice, but others do not. This is the case because of the lack of implementation of IndexedDB across all browsers. A simple solution is to implement the information retrieval system as a desktop software, which hooks into networking API of the operating system. All needed information can be intercepted and stored in a faster key-value database such as MongoDB [41] or Redis [69].
- *Considering dynamic document changes.* The content of documents can either be static or it can change dynamically, such as web sites using *Asynchronous JavaScript and XML (AJAX)*. The concepts in this thesis highlight methods do not highlight volatile content and hence leave that use case open for future work.

Bibliography

[1] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. Learning User Interaction Models for Predicting Web Search Result Preferences. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 3–10, New York, NY, USA, 2006. ACM.

[2] James Burke Andy Chung. RequireJS. Website. `http://requirejs.org/`; visited in July 2013.

[3] Jeremy Ashkenas. CoffeeScript. Website. `http://coffeescript.org/`; visited in July 2013.

[4] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[5] Roger E. Bohn and James E. Short. How much information? 2009. Report on American Consumers. Technical report, Global Information Industry Center. University of California, San Diego, 2009.

[6] Leo Breiman. Bagging Predictors. *Mach. Learn.*, 24(2):123–140, August 1996.

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[8] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to Rank Using Gradient Descent. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[9] Lucas Calje. Test WebSQL in Chrome and IndexedDB in Firefox and Chrome. Website. `https://github.com/facebook/IndexedDB-polyfill`; visited in September 2013.

[10] Brian Cavalier. A solid, fast Promise/A+ and when() implementation. Website. `https://github.com/cujojs/when`; visited in July 2013.

[11] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic Information Retrieval Approach for Ranking of Database Query Results. *ACM Trans. Database Syst.*, 31(3):1134–1168, September 2006.

[12] Paul Alexandru Chirita, Wolfgang Nejdl, Raluca Paiu, and Christian Kohlschütter. Using ODP Metadata to Personalize Search. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 178–185, New York, NY, USA, 2005. ACM.

[13] Mark Claypool, David Brown, Phong Le, and Makoto Waseda. Inferring User Interest. *IEEE Internet Computing*, 5(6):32–39, November 2001.

[14] Microsoft Corporation. Bing. Website. `http://www.bing.com/`; visited in September 2013.

[15] Christopher Manning Dan Jurafsky. Natural Language Processing. Website. Available online at `https://www.coursera.org/course/nlp`; visited in September 2013.

[16] Mariam Daoud, Lynda Tamine-Lechani, and Mohand Boughanem. Learning User Interests for a Session-Based Personalized Search. In *Proceedings of the second international symposium on Information interaction in context*, IIiX '08, pages 57–64, New York, NY, USA, 2008. ACM.

[17] The Internet Movie Script Database. Django Unchained Script at IMSDb. Website. `http://www.imsdb.com/scripts/Django-Unchained.html`; visited in September 2013.

[18] Maurice de Kunder. WorldWideWebSize.com | The size of the World Wide Web (The Internet). Website. `http://www.worldwidewebsize.com/`; visited in September 2013.

[19] Alexis Deveria. Can I use... Support tables for HTML5, CSS3, etc. Website. `http://caniuse.com/#cats=HTML5`; visited in July 2013.

[20] DuckDuckGo. Search DuckDuckGo. Website. `https://duckduckgo.com/`; visited in September 2013.

[21] Christopher M. Eppstein. Compass Home | Compass Documentation. Website. `http://compass-style.org/`; visited in July 2013.

[22] Henry A. Feild, James Allan, and Rosie Jones. Predicting Searcher Frustration. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 34–41, New York, NY, USA, 2010. ACM.

[23] Susan Gauch, Jason Chaffee, and Alaxander Pretschner. Ontology-Based Personalized Search and Browsing. *Web Intelli. and Agent Sys.*, 1(3-4):219–234, December 2003.

[24] Go-Gulf.com. How people spend their time online. Website. `http://www.go-gulf.com/blog/online-time/`; visited in July 2013.

[25] Jeremy Goecks and Jude Shavlik. Learning Users' Interests by Unobtrusively Observing Their Normal Behavior. In *Proceedings of the 5th international conference on Intelligent user interfaces*, IUI '00, pages 129–132, New York, NY, USA, 2000. ACM.

[26] GruntJS.com. Grunt: The JavaScript Task Runner. Website. `http://gruntjs.com/`; visited in July 2013.

[27] GruntJS.com. Plugins - Grunt: The JavaScript Task Runner. Website. `http://gruntjs.com/plugins`; visited in July 2013.

[28] Qi Guo and Eugene Agichtein. Towards Predicting Web Searcher Gaze Position from Mouse Movements. In *CHI '10 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '10, pages 3601–3606, New York, NY, USA, 2010. ACM.

[29] Qi Guo and Eugene Agichtein. Beyond Dwell Time: Estimating Document Relevance from Cursor Movements and Other Post-Click Searcher Behavior. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 569–578, New York, NY, USA, 2012. ACM.

[30] Nathan Weizenbaum Hampton Catlin and Chris Eppstein. Sass - Syntactically Awesome Stylesheets. Website. `http://sass-lang.com/`; visited in July 2013.

[31] Ian Hickson. Web SQL Database. W3C Working Group Note, November 2010.

[32] Gernot Hoffman. CIELab Color Space. PDF. `http://docs-hoffmann.de/cielab03022003.pdf`; visited in July 2013.

[33] AOL Inc. ODP - Open Directory Project. Website. `http://www.dmoz.org/`; visited in September 2013.

[34] Facebook Inc. IndexedDB polyfill (via Web SQL Database). Website. `https://github.com/facebook/IndexedDB-polyfill`; visited in September 2013.

[35] Google Inc. chrome.tabs - Google Chrome. Website. `https://developer.chrome.com/extensions/tabs.html`; visited in July 2013.

[36] Google Inc. Google. Website. `https://www.google.com/`; visited in September 2013.

[37] Google Inc. Google Chrome extension API. Website. `https://developer.chrome.com/extensions/api_index.html`; visited in July 2013.

[38] Google Inc. Google Images. Website. `http://www.google.com/imghp`; visited in September 2013.

[39] Google Inc. Google Ngram Viewer. Website. `http://books.google.com/ngrams`; visited in September 2013.

[40] Google Inc. Message Passing - Google Chrome. Website. `http://developer.chrome.com/extensions/messaging.html`; visited in September 2013.

[41] MongoDB Inc. MongoDB. Website. `http://www.mongodb.org/`; visited in September 2013.

[42] Optify Inc. The Changing Face of SERPs - Organic Click Through Rate. PDF. `http://www.optify.net/wp-content/uploads/2011/04/Changing-Face-oof-SERPS-Organic-CTR.pdf`; visited in July 2013.

[43] Yahoo! Inc. Yahoo. Website. `http://www.yahoo.com/`; visited in September 2013.

[44] Mahalo.com Incorporated. Mahalo.com. Website. `http://www.mahalo.com/`; visited in September 2013.

[45] Ecma International. ECMAScript Language Specification - ECMA-262 Edition 5.1. Website. `http://www.ecma-international.org/ecma-262/5.1/#sec-15.4.4.11`; visited in July 2013.

[46] Kalervo Järvelin and Jaana Kekäläinen. IR Evaluation Methods for Retrieving Highly Relevant Documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR ’00, pages 41–48, New York, NY, USA, 2000. ACM.

[47] Thorsten Joachims. Optimizing Search Engines Using Clickthrough Data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD ’02, pages 133–142, New York, NY, USA, 2002. ACM.

[48] Kit Cambridge John-David Dalton, Blaine Bublitz and Mathias Bynens. Lo-Dash. Website. `http://lodash.com/`; visited in July 2013.

[49] The jQuery Foundation. jQuery. Website. `http://jquery.com/`; visited in July 2013.

[50] Diane Kelly and Nicholas J. Belkin. Reading Time, Scrolling and Interaction: Exploring Implicit Sources of User Preferences for Relevance Feedback. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR ’01, pages 408–409, New York, NY, USA, 2001. ACM.

[51] Diane Kelly and Nicholas J. Belkin. Display Time as Implicit Feedback: Understanding Task Effects. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR ’04, pages 377–384, New York, NY, USA, 2004. ACM.

[52] Joseph A. Konstan, Bradley N. Miller, David Maltz, Jonathan L. Herlocker, Lee R. Gordon, and John Riedl. GroupLens: Applying Collaborative Filtering to Usenet News. *Commun. ACM*, 40(3):77–87, March 1997.

[53] Amy N. Langville and Carl D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.

[54] Fang Liu, Clement Yu, and Weiyi Meng. Personalized Web Search by Mapping User Queries to Categories. In *Proceedings of the eleventh international conference on Information and knowledge management*, CIKM '02, pages 558–565, New York, NY, USA, 2002. ACM.

[55] Fang Liu, Clement Yu, and Weiyi Meng. Personalized Web Search For Improving Retrieval Effectiveness. *IEEE Trans. on Knowl. and Data Eng.*, 16(1):28–40, January 2004.

[56] Adobe Systems Software Ireland Ltd. PDF-Reader, PDF-Viewer | Adobe Reader XI. Website. `http://www.adobe.com/de/products/reader.html`; visited in September 2013.

[57] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[58] Nicolaas Matthijs and Filip Radlinski. Personalizing Web Search Using Long Term Browsing History. In *Proceedings of the fourth ACM international conference on Web search and data mining*, WSDM '11, pages 25–34, New York, NY, USA, 2011. ACM.

[59] Chris McKenzie. Porter-Stemmer/PorterStemmer1980.js at master - kristopolous/Porter-Stemmer - GitHub. Website. `https://github.com/kristopolous/Porter-Stemmer/blob/master/PorterStemmer1980.js`; visited in July 2013.

[60] Masahiro Morita and Yoichi Shinoda. Information Filtering Based on User Behavior Analysis and Best Match Text Retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, pages 272–281, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[61] Guardian News and Media Limited. Quentin Tarantino gets Pulp Fiction car back - after 17 years. Website. `http://www.theguardian.com/film/2013/apr/29/quentin-tarantino-pulp-fiction-car-17-years`; visited in September 2013.

[62] Eliot Graff Andrei Popescu Jeremy Orlow Joshua Bell Nikunj Mehta, Jonas Sicking. Indexed Database API. W3C Candidate Recommendation, July 2013.

[63] Douglas Oard and Jinmook Kim. Implicit Feedback for Recommender Systems. In *in Proceedings of the AAAI Workshop on Recommender Systems*, pages 81–83, 1998.

[64] Alexander Pretschner and Susan Gauch. Ontology Based Personalized Search. In *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '99, pages 391–, Washington, DC, USA, 1999. IEEE Computer Society.

[65] Feng Qiu and Junghoo Cho. Automatic Identification of User Interest for Personalized Search. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 727–736, New York, NY, USA, 2006. ACM.

[66] Filip Radlinski, Madhu Kurup, and Thorsten Joachims. How Does Clickthrough Data Reflect Retrieval Quality? In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 43–52, New York, NY, USA, 2008. ACM.

[67] Ranker. Reservoir Dogs Movie Quotes | Best Movie Quotes from Reservoir Dogs. Website. `http://www.ranker.com/list/reservoir-dogs-movie-quotes/movie-and-tv-quotes`; visited in September 2013.

[68] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[69] Salvatore Sanfilippo and Pieter Noordhuis. Redis. Website. `http://redis.io/`; visited in September 2013.

[70] Gaurav Sharma, Wencheng Wu, and Edul N. Dalal. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. *Color Research & Application*, 30(1):21–30, February.

[71] Ahu Sieg, Bamshad Mobasher, and Robin Burke. Web Search Personalization with Ontological User Profiles. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 525–534, New York, NY, USA, 2007. ACM.

[72] David Sontag, Kevyn Collins-Thompson, Paul N. Bennett, Ryen W. White, Susan Dumais, and Bodo Billerbeck. Probabilistic Models for Personalizing Web Search. In *Proceedings of the fifth ACM international conference on Web search and data mining*, WSDM '12, pages 433–442, New York, NY, USA, 2012. ACM.

[73] Micro Speretta and Susan Gauch. Personalized Search Based on User Search Histories. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '05, pages 622–628, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Smitha Sriram, Xuehua Shen, and Chengxiang Zhai. A session-Based Search Engine. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 492–493, New York, NY, USA, 2004. ACM.

[75] StatCounter. Top 5 Browsers from Sept 2012 to Aug 2013 | StatCounter Global Stats. Website. `http://gs.statcounter.com/#browser-ww-monthly-201209-201308`; visited in September 2013.

[76] StatCounter. Worldwide browser distribution from June 2012 to June 2013. Website. `http://gs.statcounter.com/#browser-ww-monthly-201206-201306`; visited in July 2013.

[77] Jaime Teevan, Eytan Adar, Rosie Jones, and Michael A. S. Potts. Information re-retrieval: repeat queries in Yahoo's logs. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 151–158, New York, NY, USA, 2007. ACM.

[78] Jaime Teevan, Susan T. Dumais, and Eric Horvitz. Personalizing Search via Automated Analysis of Interests and Activities. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 449–456, New York, NY, USA, 2005. ACM.

[79] Jaime Teevan, Susan T. Dumais, and Daniel J. Liebling. To Personalize or Not to Personalize: Modeling Queries with Variation in User Intent. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 163–170, New York, NY, USA, 2008. ACM.

[80] C.J. van Rijsbergen, S.E. Robertson, and M.F. Porter. New Models in Probabilistic Information Retrieval. 1980.

[81] Thuy Vu, AiTi Aw, and Min Zhang. Term Extraction Through Unithood and Termhood Unification. In *IJCNLP*, pages 631–636. The Association for Computer Linguistics, 2008.

[82] W3C. HTML 5.1 Editor's Draft. Website. `http://www.w3.org/html/wg/drafts/html/master/`; visited in July 2013.

[83] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast Phrase Querying With Combined Indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, October 2004.

[84] S. K. M. Wong, Wojciech Ziarko, and Patrick C. N. Wong. Generalized Vector Spaces Model in Information Retrieval. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '85, pages 18–25, New York, NY, USA, 1985. ACM.

[85] Budi Yuwono and Dik Lun Lee. Search and Ranking Algorithms for Locating Resources on the World Wide Web. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 164–171, Washington, DC, USA, 1996. IEEE Computer Society.