Michael Holzer BSc BSc

Improving query suggestions for rare queries on faceted documents.

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Roman Kern

Know-Center
Institut für Wissenstechnologien

Graz, April 2015

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

Abstract

The goal of this thesis is to improve query suggestions for rare queries on faceted documents. While there has been extensive work on query suggestions for single facet documents there is only little known about how to provide query suggestions in the context of faceted documents. The constraint to provide suggestions also for uncommon or even previously unseen queries (so-called rare queries) increases the difficulty of the problem as the commonly used technique of mining query logs can not be easily applied.

In this thesis it was further assumed that the user of the information retrieval system always searches for one specific document - leading to uniformly distributed queries. Under these constraints it was tried to exploit the structure of the faceted documents to provide helpful query suggestions. In addition to theoretical exploration of such improvements a custom datastructure was developed to efficiently provide interactive query suggestions.

Evaluation of the developed query suggestion algorithms was done on multiple document collections by comparing them to a baseline algorithm that reduces faceted documents to single facet documents. Results are promising as the final version of the new query suggestion algorithm consistently outperformed the baseline.

Motivation for and potential application of this work can be found in call centers for customer support. For call center employees it is crucial to quickly locate relevant customer information - information that is available in structured form (and can thus easily be transformed into faceted documents).

Contents

List of Figures

1 Introduction

In this introductory chapter the reasons for this work are described. After the motivational part an outline over the rest of the thesis is provided.

1.1 Motivation

The original motivation for this work was provided by call centers for customer support. The employees of the call centers have access to a software system to provide customer support. When a customer calls for support a common task is to retrieve some information about the customer, either the customer account or some more detailed data, e.g. an order. Naturally this step should be performed as quickly as possible so that the employee can move on to handle the customer's actual problem.

If the customer can provide a unique identifier (such as an account number) the information retrieval problem is quite trivial. But in the absence of such information the search becomes more challenging and time consuming: the search might be performed using data like customer name, address or order information (e.g. product names). It is here where the system should be interactive and responsive to enable the employee to quickly zero in on the desired customer document.

All customer related data resides in a relational database, therefore its structure is well-known and can (and should) be exploited in the search process. However it is undesirable to present a cluttered user interface to the call center employee where she has to enter data into specific fields. Rather a single textfield to input the query is the preferred user-friendly option, as all the major web search engines demonstrate.

1 Introduction

For this single textfield we want to provide useful query suggestions. Query suggestions can help a user in several ways. Firstly it may speed up the process of entering the query, a crucial property in the commercial setting. Secondly, when the user is unsure about the spelling of a word, a query suggestion may already present the complete word (either derived from the first characters of the word entered by the user or from the other query terms even before the user starts typing the next word), drastically reducing the time needed to find the right spelling using trial and error. Additionally a query suggestion showing the term(s) the user was about to enter anyway can improve the user experience by reassuring the user that she is on the right track (see *8 Design Patterns for Autocomplete Suggestions* 2014 for a user study supporting this claim).

Query suggestions should satisfy certain properties to be useful for the user. Despite the lack of any a priori knowledge about the mapping of query terms to document facets the system should be able to capture the intent of the user's query and provide query suggestions that match the user query's intended albeit unstated term to document facet mapping. Additionally it is crucial that every suggested query should have a non-empty result set: a query that returns zero result documents is certainly not useful. Yet another aspect is the order of the terms in a query. If there is an inherent or commonly accepted order for the facets of the suggested query's terms (e.g. the order of address facets such as street, street number, postal code and city is usually given by convention - although it may vary depending on the current locale) then this order should be respected in the suggested query.

Previous research in the area of query suggestions focuses mainly on extracting information from query logs. We will argue that in our setting these techniques are not applicable because of the distribution of the queries - contrary to the common power law distribution of queries this work is based on the assumption that the queries are uniformly distributed - a consequence of having a unique target document for each information need. Therefore mining query logs for query suggestions is not feasible.

This thesis will discuss approaches that rely heavily on the indexed documents to provide query suggestions - not only for common queries but also for infrequent or even new queries.

1.2 Research Question

As described in the previous chapter the goal of this thesis is to provide helpful query suggestions on faceted documents. Additionally we make the assumption that each information need is satisfied by a single target document. So, put concisely the research question is:

Improving query suggestions on faceted documents for information needs satisfied by a single target document.

1.3 Outline

In chapter 2 we will elaborate both on the problem itself and on its setting. It is here where we will argue that the query logs can not be exploited as in systems where the queries are distributed according to a power law.

In chapter 3 we will discuss related work in the literature. We will give a short general overview about existing techniques to support the user of an information retrieval system. Existing work on query suggestions will be discussed in more detail. Since we argue that the queries in our target system are mainly one-off queries we will also look at how query suggestions can be provided for rare or long-tail queries. Most of the existing work on query suggestions focuses on single-facet documents, so another section will be about strategies to handle multi-facet (or faceted) documents, e.g. in scoring.

In chapter 4 the techniques used to provide query suggestions in the target system will be presented.

Chapter 5 will discuss interesting parts of the implementation of the presented query suggestion approaches.

In chapter 6 we will present our evaluation strategy and the evaluation results of our approach in comparison with the baseline.

Discussion and interpretation of the evaluation results will be done in chapter 7.

In chapter 8 we will provide a concluding discussion about the work done. We will summarize our improvements and discuss remaining open issues. Also we will present some ideas for further research based on the work done in this thesis.

2 Problem and Setting

This chapter will present the problem this thesis is trying to solve as well as a use case where the solution will be applied.

We will start in section 2.1 with some general reasoning why information retrieval (IR) is an essential challenge of the current time. Section 2.2 looks at IR systems from a user's perspective - in particular it discusses some challenges the user may have to overcome to fulfill her information need.

In section 2.3 we will formalize the problem setting. This includes various aspects of the IR context, such as the structure of both single documents and the whole document collection or the expected nature of information needs.

A use case is presented in section 2.4. It is here where the reasons for the various requirements from section 2.3 will become more transparent.

2.1 Finding the Needle in the Haystack

In an age where the size of data is often measured in terabytes or even higher magnitudes users are often faced with an abundance of information when all they want is just an answer to a single question - their *information need*. The most common setting is the web (in 2013 estimated to a size of 4 zettabytes - see Blog, 2014), but others include digital encyclopedias (e.g. Wikipedia) or large e-commerce sites (e.g. Amazon); the latter can be seen as a subcategory of enterprise systems.

Typically a user only searches for a very small part of the available information - the proverbial needle in the haystack. Unfortunately the haystack

tends to be a rather large one. The need for efficient and effective information retrieval systems is thus rather obvious - the user's search should lead to the desired answer as fast as possible.

2.2 Problems a User of an IR System Faces

While the task of building a decent IR system is indisputably a challenging one, using it might be hard as well. In this section we will touch a few problems a user may face. The typical consequence of each of these problems is that the user's query fails to retrieve the documents needed to satisfy her information need.

2.2.1 Spelling

The user might misspell a word, or maybe even worse, a word might have multiple possible spellings. A particularly severe case is the last name of the Russian mathematician Pafnuty Chebyshev caused by the transliteration from the Cyrillic script to the Latin alphabet: the English and German Wikipedia entries alone each list four used spellings in English and German, respectively. Misspelling a word (or using a spelling that deviates from the one used in the indexed documents) in a query might fail to produce the desired results.

2.2.2 Word Forms

Misspelling a word is only one possible cause for an unsuccessful search against an index that actually contains documents that would satisfy the user's information need. The index may contain slightly modified forms of the searched word that adhere to the language's linguistic rules (i.e. are not misspellings) and fail to match the word entered in the query.

Languages have various grammatical rules that cause a word to appear in different forms. Examples from the English language are plurals of nouns (e.g. query - queries) or inflections of verbs (e.g. search, searches, searched,

searching). The user has no way of knowing which form of a word was actually used at index time - she can only guess.

2.2.3 The Vocabulary Problem

The Vocabulary Problem was first described in Furnas et al., 1987. It refers to the problem that different people use different words when naming the same thing. An example from the cited paper shows how a group of typists used different words to describe the same text editing operations. For the user of an IR system this means her search may fail even though the desired document is present in the index: the user might use a different word in her query than the author of the indexed document did even though user and author meant the same thing.

2.2.4 Synonymy

Related to the vocabulary problem and leading to similar problems is synonymy: different words may have the same meaning. As an example the words “monitor”, “screen” and “display” can all be used to refer to a computer’s display. The user of an IR system will obviously have a hard time guessing the right term that was used in the indexed documents.

2.2.5 Homonymy

Homonymy describes the concept of words that have multiple meanings. For example the words “Java” and “Python” gained new additional meanings with the invention of the respective programming languages. Homonymous words are a problem in IR because documents unrelated to the user’s information need can match the user’s query thus decreasing the quality of the results. A user interested in the programming language “Python” won’t be interested in result documents related to snakes.

2.2.6 Incomplete Knowledge to Specify the Query

A problem of a different kind appears when the user has incomplete knowledge about the searched domain. As an example imagine somebody wants to find a movie she saw some time ago - without remembering any "hard" facts such as the title or the names of actors. It is very likely that the searcher will experience difficulties finding this movie using a web search engine.

2.3 Problem Setting

In this section the problem setting of this thesis is described. We do this by specifying a set of characteristics and properties of the environment that are needed to make our approach applicable. This way the ideas from this work can be easily transferred to various systems that satisfy the described characteristics and properties.

In the first subsections the structure of both the indexed documents and the user-issued queries will be discussed. Then we describe characteristics of typical information needs and associated search processes. Finally it is discussed how query suggestions can support the user in the described setting.

2.3.1 Faceted Documents

In this subsection it is defined what is meant by a *Faceted Document*, the type of document we will consider in our setting.

A *document* is the unit of data that an information retrieval system processes. At index time the document is provided to the system to make it searchable, and at query time documents are returned as search results.

A document can simply be a single string of text, but it can also have a set of metadata associated with it. These metadata (e.g. author, category) are commonly called *facet*s. The values of facets are typically single words or short, descriptive phrases.

A *faceted document* is a document that consists solely of a set of facets. In particular this means that a faceted document has no *body* - no large section of text is associated with a document.

Faceted search is usually associated with metadata (as for example in metadata-based image search like in Yee et al., 2003) - faceting is provided not on the primary content of the documents but on their metadata. In our setting a document's metadata constitute the whole document.

2.3.2 Queries

This subsection describes the type of input entered by the user.

A user-friendly IR interface is usually as simple as possible and demands as little domain-specific knowledge from the user as possible. This is most popularly reflected by the common interface solution for web search: a single textfield. In this work we aim to provide a similar approach for search on faceted documents.

The input entered by the user (the *query*) should consist of keywords - the user should not be bothered with matching keywords to facets. This would either require knowledge of the internal representation of the facets, namely the names of the facets used in the index, or some cluttered interface that provides this information. Both options are undesireable from a user's perspective since they complicate (and thus slow down) the search process.

2.3.3 Information Need to Document Relation

An important property of an information need I is the set of documents R_I in the index that match I. Typically it is not known a priori how many documents R_I contains. It might also be the case that the user doesn't need to see all documents in R_I to satisfy her information need - for example there could be redundant information across documents in R_I.

From a user's perspective it is desirable that documents that satisfy the information need are at the top of the results page, for example among

the top ten results. To formulate this idea we define S to be the user satisfaction, and $P(S)$ the probability that the user is satisfied. Of course user satisfaction depends on the result documents R, so we are actually interested in $P(S|R)$.

For a ranked retrieval system the order of the result documents matters. The user looks at the highest ranked result documents first, so it makes sense to include this notion in the model. For this we define R_k to be the top k result documents for $k \in \mathbb{N}$. So $P(S|R_k)$ is the probability that the user's information need is satisfied by the top k result documents.

In this thesis we will look at a special case, where it is known that for any information need I the set R_I of matching documents has a constant size, i.e. $|R_I| = k$ for some $k \in \mathbb{N}$. In particular we will dicuss the case where $k = 1$, i.e. each information need is satisfied by exactly one document. This property has some implications on the user's search process as will be discussed next.

2.3.4 Search Process

Given the premise that each information need is satisfied by exactly one document we can deduce some characteristics of the search process. Using the search categories introduced in Jansen, Booth, and Spink, 2008 we can place our users' searches in either the navigational or transactional category. It depends on the actual system what the user typically does once the desired document is located. Since the user's goal is to zero in on the unique target document the search is certainly not informational.

The main goal of an information system is to satisfy its users' information needs - and it should do this as quickly as possible. In our case this means to retrieve the target document with minimal effort from the user.

Two important factors of the retrieval speed are the amount of text the user has to enter and the time she has to wait for feedback from the IR system. Minimizing the latter demands the immediate display of search results for the partially entered query as the user types. Minimizing the amount of typing for the user requires a bit more discussion.

The user's goal is to enter enough information to uniquely identify the target document while at the same time avoiding it to enter redundant information. Query terms that can be deleted from the query such that the new query still uniquely identifies the target document are redundant and thus not worth entering in the first place. Without detailed knowledge of the document collection the user can not know whether a query term she is about to enter is redundant or not. Here query suggestions can help the user in refining the query in the right way.

2.3.5 Query Suggestions

Query suggestions should guide the user in fulfilling her information need. In our setting a query suggestion is useful if it helps to find the unique target document. Thus a query suggestion should reduce the number of matching documents while keeping the target document in the result set - precision should be increased towards a value of 1, recall should be constantly at 1.

Query suggestions can essentially operate on two levels: on facet level and on phrase level. On the facet level a query suggestion can add term(s) from a facet that is not part of the current query. On the phrase level a query suggestion can extend a phrase from a single facet that is already part of the current query. Phrase extension might again happen on two levels: completing a partially entered word or adding one or more words to continue a phrase.

We can put this more formally, starting with the most general form of query suggestion: Given a query Q a query suggestion is another (related) query Q' that is computed from Q by some means. Adding or completing a partial term can be formulated as follows: Let $Q = q_1 \dots q_n$ be the original query, where q_n is a partially entered term (potentially the empty string), then a query suggestion Q' for Q that completes the term q_n has the form $Q' = (Q \setminus \{q_n\}) \cup \{t\})$, where q_n is a prefix of t.

Note that the last query suggestion Q' is written in set notation. Sets are by definition unordered - the terms of Q' can be reordered with respect to Q to match the user's expectations about the structure of the query terms. For example the newly completed (or added) term q_n might have some close

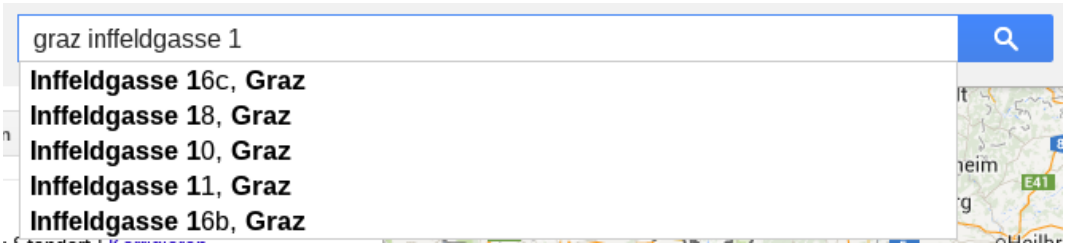


Figure 2.1: Query suggestion with term reordering example from *Google Maps* 2014

relationship with a previously entered query term q_i, so it might make sense to group those terms together in Q'. See figure 2.3.5 for an example of term reordering in a query suggestion in the setting of finding an address.

So far we only discussed cases where query suggestions add terms, but it may also be desirable to remove terms from a query. A user might have gone wrong in her quest to find the target document - for example she might have entered a term that is not present in any document together with the other query terms. This causes the result set to be empty, an undesirable situation. Suggested queries can delete terms from the query such that the result set is not empty any more. In formal terms $Q' = Q \setminus q_i$ is a query suggestion obtained by term deletion.

2.4 Customer Support - a Use Case

In this section we describe a use case of this work: Call centers that provide customer support where employees need to locate customer related data.

When a customer calls for support the employee's first task is typically to locate customer related data in the system. This task should be done as fast as possible because only then the customer's actual problem can be handled. To find the relevant customer data the customer has to provide some information. The employee has to work with this information to perform a search.

The information provided by the customer can be of varying quality for the search process: if a unique identifier such as an account number is available

then the search is over quickly - looking up such an identifier is an easy task. However, if the customer can only provide data like her/his name and address then the task gets more challenging and time consuming. Names of people, streets and places often also present spelling challenges - for the searcher this means she might be entering a wrong term leading to unsatisfying search results.

One way of mitigating such problems is to shorten the feedback cycle for the user by providing query suggestions. This way for example potential spelling mistakes can be avoided already at query time because the correct spelling is present in a query suggestion.

2.4.1 System Characteristics

Documents

The documents in the target system represent information about customers. This includes information about the customer's personal data (such as name and address) as well as customer related data such as the customer's orders. This information resides in a relational database and is thus available in structured form. From this representation we can build faceted documents (e.g. a document containing a customer's personal data) that can be fed into an index.

A Single Target Document for Each Information Need

The information need of a customer support employee is usually satisfied by exactly one document. Either she wants to access the customer's whole account or some detail of it, for example a particular order. Thus the goal of the search is to zero in on this single target document.

Query Input

Entering the query should be as easy as possible - the easiest way is to just enter the data provided by the customer into a single textfield. It is

not desireable to match the customer's data to specific fields of a complex search form, or to provide facet information for the terms entered in a single textfield (e.g. "name:Holzer street:Lerchengasse"). Both options simply cost time and significantly complicate the employees task.

Query distribution

Queries in our target system are not distributed according to a power law - as it is common for example in the setting of web search. Rather queries are distributed uniformly over the indexed data - the probability to be in need for customer support is approximately the same for every customer. Along with the requirement that query suggestions should be provided if a customer is calling for support for the first time this implies that a query suggestion approach based solely on query logs simply can't work. Thus the complete collection of indexed documents needs to be considered when computing query suggestions.

The Data

The size of the customer bases managed by the use case system range from a couple of thousand to potentially millions, so scalability is a primary concern.

The facets of the index documents include but are not limited to customer contact data such as first and last name, various address fields, telephone number and email address. In addition the index contains information like the customer's ordered products and payment details.

2.5 Summary

Now the problem this thesis is trying to solve has been formally introduced. After a discussion of a general set of challenges that a user of an IR system faces our problem setting was described. This includes a set of characteristics and properties of the IR environment that will be taken as given in this

thesis from now on. Also a use case originating from a real world system was presented, showing that this work is not only purely of acadamic but also of practical value.

3 Existing Work - State of the Art

In this chapter we take a look on related work. We start with an overview of techniques that assist a user during a search session. A separate section is dedicated to the technique that is the topic of this work: query suggestion. Other sections discuss existing work on specific aspects of our problem: dealing with rare (or long-tail) queries and searching on faceted documents.

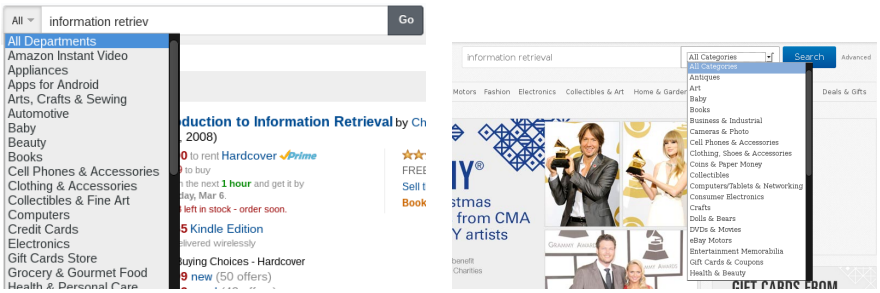
3.1 Guiding the User to Fulfill Her Information Need

There are many ways how an IR system can help the user to fulfill her information need that are presented in the literature and/or in use in existing systems. In this section some of them are presented.

3.1.1 Filtering

Many IR systems provide the user with some means to narrow down the part of the document collection that should be searched by a query, i.e. some kind of filter is applied (see Baeza-Yates, Ribeiro-Neto, et al., 2011).

An example for this are categories on e-commerce sites such as Amazon or Ebay (see figure 3.1), provided for selection in a dropdown field. A user that performs an exploratory search on chess books may enter 'chess book' in the query field, but this results in matches not only in the 'Books' department but also in others like 'Software' and 'Music' - certainly not what the user



(a) Category filter on Amazon (b) Category filter on Ebay

Figure 3.1: Examples of category filters.

expects. If she instead only enters the query 'chess' and selects the 'Books' department only books will appear in the results.

A slightly different approach is used at sites like Yelp [1] or Zvents [2] that offer search interfaces for locations (e.g. restaurants) and events, respectively. They both have an additional textfield for the geographic location, and Zvents has another one for time.

These approaches are not necessarily a departure from the popular choice of using a single textfield for query input. In the case of e-commerce sites the default option for the category is usually simply 'All', and Zvents let's the user choose a default location that is preselected in subsequent searches.

Providing additional input fields clutters the user interface, therefore default interfaces usually provide only limited filtering capabilities. Also further refinement of the filter - if possible at all - results in longer request-response cycles. Another shortcoming of this filtering technique is that it is not possible to combine multiple filters. Thus there is often the need for a more flexible and interactive approach such as faceting, discussed in the next subsection.

[1] www.yelp.com
[2] www.zvents.com

3.1.2 Faceting

Faceting provides the means to interactively refine the results of a query along multiple dimensions of metadata. This technique has been researched by Hearst et al. (see Hearst et al., 2002, English et al., 2002, Yee et al., 2003 and Hearst, 2008).

Originally developed for metadata-based image search, query refinement using faceting is now common practice for various kinds of document collections. Faceting not only supports combining multiple filters (for example on an e-commerce site the user might choose a price range and a customer rating threshold) but also filtering on hierarchical metadata (for example a location might be subsequently refined by the sequence *Europe* $\rightarrow$ *Austria* $\rightarrow$ *Styria* $\rightarrow$ *Graz*). On each iteration the user is presented with a list of facets where she can select among provided options (for example typical price ranges); for hierarchical metadata this results in a nice way of refining large sets (for example the location facet might start out at the continent level and end up at the city level). At each iteration the user interface provides information about the currently selected faceting filters, along with the possibility to revert or change previous choices. Typically each displayed filter option also provides information on how many documents would still be matched if the filter would be selected.

Faceted search interfaces are a well-received and established solution for exploratory searches. However it is not a desirable solution for our use case system because the request-response cycle is slowed down when the user searches for specific terms among both the available facets and the suggested filter options on each iteration. This is caused by the mismatch between exploratory search (targeted by the faceted search paradigma) and the need to hunt down a single target document.

3.1.3 Spelling Correction

Misspelling a word in a query is an obvious problem. Without a correction of the spelling error the IR system can't properly satisfy the user's information need since misspelled words do not match against the indexed documents

as expected by the user. Approaches for spelling correction are described for example in Manning, Raghavan, and Schütze, 2008.

Spelling suggestions can be presented both at query time and on the search results page. At the time of writing Google for example currently implements both variants. During query time the interactive query suggestion feature includes suggestions for possible spelling corrections. On the results page Google suggests a corrected query if it suspects a spelling error in the query. For common misspellings Google even performs the search on the corrected query, suggesting the original supposedly misspelled query for users that insist on their spelling.

Spelling correction relies on a similarity measure for queries such that the *nearest* correct query can be suggested. Correct queries are commonly identified using the indexed data or query logs. Spelling correction can either be done on single terms or context-sensitively on the complete query.

3.1.4 Stemming

Stemming describes the technique of mapping various forms of a word to one common *stem*. Note that this algorithmically computed stem doesn't necessarily coincide with the grammatical word stem. An example for a stemming algorithm is the Porter stemmer, described in Porter, 1980. Performing stemming at index time as well as on the query terms mitigates the problem of *Word Forms*: after stemming the terms "searched" and "searches" are both reduced (or *stemmed*) to "search" and the exact word form used in the indexed documents and the queries doesn't matter anymore. A potential downside of unreflected use of this technique is that it is no longer possible to search for specific forms of a word, e.g. a search for "searched" will always also yield results for "searches".

3.1.5 Query Expansion

Query expansion adds - possibly transparent to the user - additional terms to a query. The reasoning behind query expansion is that adding relevant terms to the query should increase the effectiveness of the search.

Query expansion can be both opaque and transparent to the user. Query expansions can be presented to the user as suggestions both interactively at query time or on the results page as alternative searches. If query expansion is done transparently to the user it is called automatic query expansion.

A survey about automatic query expansion is provided by Carpineto and Romano, 2012. They provide a taxonomy to classify different approaches with respect to various aspects of query expansion, including the data source and feature extraction method used to compute query expansions.

It is noteworthy that early work used the indexed documents as the data source, while later the focus shifted to query logs. Others use external data (e.g. Wikipedia) or try to exploit the context of the search session.

He and Ounis, 2007 use both the indexed documents as well as an external data source for query expansion. They also exploit that the indexed documents consist of multiple facets (title, anchor texts and body) to adjust the weights of the terms in the expanded query.

Techniques for query expansion can be computationally expensive and are therefore sometimes applied offline for popular queries. Broder et al., 2009 suggested a way to overcome these problems for rare queries by reusing computations done for related popular queries.

3.1.6 Autocompletion

As the user types a query it can be tried to guess the rest of the intended query of the currently entered term. More formally, given a partial query $Q_p = q_1 q_2 \cdots q_k$ where the q_i are the query terms and q_k is a potentially partially entered term, the system tries to compute the user's intended query $Q = Q_p Q_a$ where Q_a is the part that the user has not typed yet. Some systems try to only compute the next query term, that is if q_k is partially entered to complete it to a full term, and if q_k is completely entered to provide the next query term q_{k+1}.

Note that the autocompletion problem is a subset of the problem of providing query suggestions: each autocompletion is also a query suggestion.

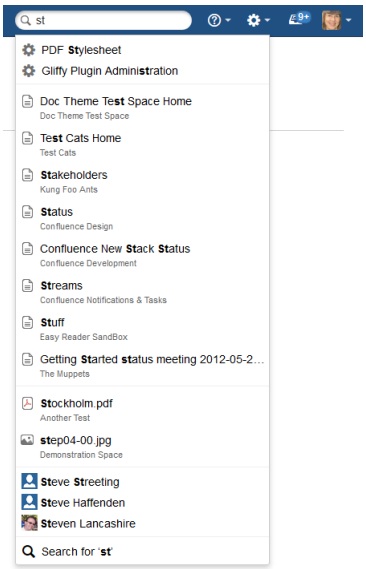


Figure 3.2: Autocompletion in Atlassian's Confluence.

Autocompletion is nowadays provided by all major web search engines and it is also common in enterprise systems, for example in the Confluence Wiki by Atlassian (see figure 3.2), often as part of a more general query suggestion feature.

Various kinds of autocompletion have been studied in the literature, we will briefly describe a few of them that have some similarity to what we want to provide.

In Bast and Weber, 2006a autocompletion (in the context of a corpus of single-facet documents) is provided on the last partially entered query term. Their system only suggests autocompletions that lead to non-empty result sets when appended to the already entered query. Instead of using an inverted index a new datastructure was invented for the computation of autocompletions. This datastructure (called *HYB*) is based on the idea to precompute inverted lists for unions of words.

The same authors extended their approach in Bast and Weber, 2006b to also provide limited autocompletion on facets. Given a query with a partially entered last query term q_t they compute *categories* that contain matches for

the already entered query and match q_t with their name. A category is one particular facet, so this autocompletion on facets can by no means be called complete.

Autocompletion on structured data is explored in Nandi and Jagadish, 2007. In their system the query is entered in a single textfield, having the general form $Q = key_1 : value_1\ key_2 : value_2 \cdots key_n : value_n$. This format is not strict, it is possible to omit either the key or the value part. Autocompletion is provided on both the key and value parts, always ensuring non-empty result sets. The authors employed a combination of datastructures (a Lucene-based inverted index, trie-based datastructures and schema-backed functions for structural query validation) to compute query suggestions along with display information (e.g. result set size or type of key).

The concept of autocompletion is taken to a new level by Hawking and Griffiths, 2013 using what they call extended query autocompletion. However, their approach does not only append terms to the current query, rather it falls into the more general notion of query suggestions. Thus discussion of this paper is deferred to 3.2.3.

3.2 Assisting the User with Query Suggestions

One particular way to help the user formulating her query is to suggest new, alternative or more specific queries to the one the user has typed. Approaches for query suggestions differ mainly in the way they are computed and how they are presented to the user. Query suggestions are often computed from query logs, alternatives directly use the indexed data. Computed query suggestions can be presented to the user interactively as she enters the query or along with the results for further exploration.

3.2.1 Deriving Query Suggestions

Query suggestions need to be computed by some means, and different methods have been researched in the literature. The main input value to the computation is always the query. Depending on the algorithm additionally

used inputs can be the indexed data, logs of previously issued queries and even external data. In this subsection we will take a brief look at the various algorithms presented in the literature.

Computing query suggestions from query logs

A popular approach to compute query suggestions that works well for common queries is mining query logs. Particular techniques vary in sophistication and the way the query logs are processed. Probably the simplest use of query logs is to compute the frequency of each query. This has been used in Church and Thiesson, 2005 to provide query suggestions for wildcard queries. More sophisticated is the use of clickthrough data, for example in Beeferman and Berger, 2000 or Cao et al., 2008. Providing query suggestions for long-tail queries is more difficult due to the data sparsity, but Broccolo et al., 2012 overcame this problem by mining the query logs not on the query but on the query term level. Thus they were able to provide query suggestions even in the case of a new query that was not yet present in the query logs.

Computing query suggestions from the indexed data

In settings where query logs are not available for query suggestion computation other approaches need to be devised. Bhatia, Majumdar, and Mitra, 2011 computed query suggestions directly from the indexed documents. Phrases from the document collection are extracted and indexed, on query time the indexed phrases are matched against the query and appropriate phrases presented as query suggestions.

Further approaches to computing query suggestions

A third class of algorithms computing query suggestions makes use of external data, i.e. data not directly linked to the searched document collection (such as the index or associated query logs). A linguistic approach is to use e.g. thesauri to find relevant synonym terms, or generally related terms. If

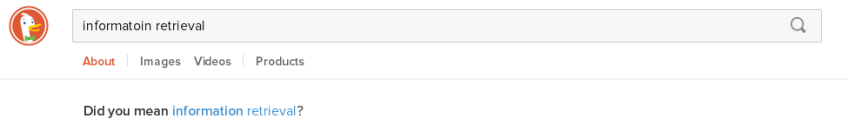


Figure 3.3: Example for non-interactive query suggestion, taken from duckduckgo.com

we consider query expansion as some sort of query suggestion (from an existing query a new query is computed to provide better search results), see Hsu, Tsai, and Chen, 2006 for an example.

3.2.2 Ways of Presenting Query Suggestions to the User

Query suggestions are usually opaque to the user, she can decide to use a suggestion or to neglect it. To enable the user to make this decision, query suggestions must be presented to her. There are mainly two places where query suggestions can be shown to the user: either along with the query results or already next to the query when it is entered. Choosing the former results in non-interactive, choosing the latter in interactive query suggestions.

Non-interactive Query Suggestions

Non-interactive query suggestions are only computed for queries that are submitted by a user. Therefore this approach is computationally not very expensive, but the feedback cycle is quite long when compared to interactive query suggestions. A well-known usecase of non-interactive suggestions is a common feature of popular web search engines. If a query issued by a user has only a small number of hits the engine tries to suggest an alternative query. The small number of results might be for example caused by a common misspelling that can be corrected by the search engine to suggest an improved query. See figure 3.3 for an example from the web search engine DuckDuckGo [3].

[3] http://www.duckduckgo.com

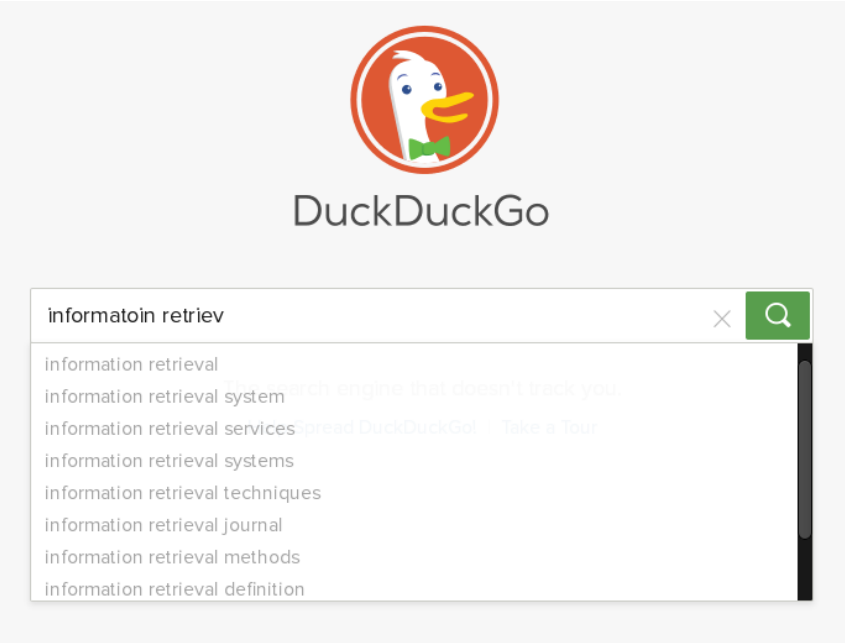


Figure 3.4: Example for interactive query suggestions, taken from duckduckgo.com

Interactive Query Suggestions

Interactive query suggestions are computed as the user enters her query. This method is computationally more expensive - the suggestions need to be computed with every keystroke of the user. Autocompletion of partially entered terms can be seen as a special case of interactive query suggestions. See figure 3.4 for an example from the web seach engine DuckDuckGo. Note how the suggestions provide term completion, spelling correction and extension of the current query by a new term all at once.

3.2.3 Query Suggestions for Faceted Documents

Note that all discussed work so far operates on documents without facets. A work that has a similar intent to ours is Bast and Weber, 2006b. This paper describes an IR system where autocompletion is provided for the last query term that the user is currently entering. The indexed document collection

is the English Wikipedia, along with category information (a Wikipedia document can be assigned to any number of categories). Autocompletions are shown for the term (occurrences in the document text) and for matching category names.

Another very interesting system is described in the recent paper Hawking and Griffiths, 2013. The authors extended the autocompletion technique to provide suggestions based on *triggers* (from the entered query). Suggestions are associated with *actions* that can differ according to the type of the suggestion. The explored actions also contain an approach for interactive faceting by suggesting filter options on facets. Note that this is significantly different from what we want to do: our intention is to actually hide the complexity of facets from the user, while still exploiting their structure in providing query suggestions.

3.3 Infrequent Queries - Making the Problem Harder

The typical way to implement query suggestions is to mine the query logs for information. The query logs are analyzed to identify similar queries that could lead to better results. For rare queries this approach has an obvious shortcoming: there are only very few or even no relevant queries in the query log. This lack of data might cause bad suggestions, or the system might fail to produce any suggestions at all.

One solution is to compute query suggestions directly from the indexed data. This approach is harder than mining query logs because suggested queries have to be computed from relevant documents. This involves deducing relevant terms from a set of documents - it is arguably easier to find similar user-entered queries where users have done the work of identifying relevant terms for an information need.

3.4 Where to Search - Fulltext Queries on Faceted Documents

Searching for query terms in a single-facet document is equivalent to searching for the query terms in the single facet of the document. Having documents with multiple facets raises some questions on how the search for query terms in a document should be performed.

One straight-forward approach would be to refer this decision to the user. The user could supply for each query term the document facet where it should occur. However this has several drawbacks, one of them being that the user needs to know the names of the facets. This might be acceptable for some expert systems, but in general users expect to be able to use a system without prior training.

A common technique is to reduce the multi-facet search to single-facet search by accumulating the content of all facets in a special catch-all facet that is then used for searching. Weighting facets can be simulated by simply adding the content of important facets multiple times to the catch-all facet. A drawback of this approach is that potential relationships between the contents of different facets are not used.

Another approach is to simply search for all query terms in all document facets and assigning weights to the facets. This way e.g. a title facet could be weighted higher than a body facet such that an occurrence of a query term in the title facet is assigned a higher score than an occurrence in the body facet.

3.5 Summary

This chapter presented existing work on various aspects of our problem on query suggestions for faceted documents, showing this work's place in the IR literature. In particular we showed how this work operates on a problem space where multiple challenges overlap: first of all the general setting of providing useful query suggestions. Then the aspect of faceted documents is added, as well as the requirement to provide suggestions

for any query, however rare it is. Faceted documents imply that standard approaches for query suggestion need at least be adapted to solve the problem, and providing suggestions also for rare queries rules out the applicability of a whole range of standard techniques, such as mining query logs. The next chapter contains the discussion of an approach to tackle all these challenges.

4 Approach

In this chapter it is described how query suggestions for queries on faceted documents are computed.

The goal of this thesis is to provide helpful query suggestions on faceted documents. We will compute query suggestions without using query logs to be able to provide suggestions not only for common frequent queries but also for infrequent and even previously unseen queries. Additionally the query suggestions should guide the user to find her unique target document. The only way to satisfy these requirements is to use the whole corpus of indexed documents for the computation of query suggestions.

In the first section 4.1 the problem is discussed in terms of its computational complexity. As it will be shown the problem to provide the best possible query suggestions is NP-complete.

The section 4.2 gives a high-level view of the steps involved in query suggestion computation and the requirements on the suggested queries.

The following sections go into the details of the various steps, including the mapping of query terms to facets (4.3), term ordering (4.4) and computing terms to add to the original query (4.5 and 4.6).

4.1 Problem Complexity

The goal of this thesis is to enable the user to find the target document as fast as possible, i.e. with as few keystrokes as possible. Mathematically this can be formulated as an optimization problem. In this section we take a look at the mathematical complexity of this optimization problem.

Being able to compute the best possible query for a given target document would also be useful in evaluating the quality of a query that uniquely identifies the target document. However, as it will turn out, it is generally not feasible to compute the optimal query.

4.1.1 Linear Integer Program Formulation

The optimization problem can be formulated as a linear integer program. In this subsection we will develop such a formulation. We start with some definitions that we will use subsequently.

Definition 1. Let D be a set of documents, $d \in D$. For a word w we write $w \in d$ if the word w occurs in d and $w \notin d$ otherwise. The set of all words in D is W, and the set of words in d is $W(d) = \{w \in W | w \in d\}$. The function δ is used to express whether a document d contains an occurrence of a word w or not:

$$\delta(d, w) = \begin{cases} 1 & \text{if } w \in d \\ 0 & \text{otherwise} \end{cases}$$

To study the complexity of the optimization problem we use a simplified version. Instead of counting the user's keystrokes we count the number of words that the query must contain to match only the target document d.

Theorem 1 (Integer program formulation)**.** Let $d \in D$ be the target document. Let $W(d) = \{w_1, w_2, \ldots, w_s\}$ be the set of words that occur in d. Then the following equations formulate the optimization problem of finding the smallest set of words that uniquely identify d.

$$\min \sum_{i=1}^{s} \lambda_i, \text{ s.t.} \tag{4.1}$$

$$\lambda_i \in \{0,1\} \tag{4.2}$$

$$\sum_{i=1}^{s} \lambda_i \delta(d, w_i) \geq 1 \tag{4.3}$$

$$\prod_{\substack{i=1 \\ \lambda_i=1}}^{s} \delta(\tilde{d}, w_i) = 0 \quad \forall \tilde{d} \in D \setminus \{d\} \tag{4.4}$$

Proof. The binary variables λ_i represent the presence of the word $w_i \in W(d)$ in the query. Thus the inequality 4.3 holds if and only if the query represented by $\lambda = (\lambda_1, \ldots, \lambda_s)$ contains a word that occurs in d. This in turn is equivalent to d matching the query.

Let $\tilde{d} \in D \setminus \{d\}$. The equation 4.4 holds for $\tilde{d}$ if and only if there is at least one index i such that $\lambda_i = 1$ and $\delta(\tilde{d}, w_i) = 0$. But this in turn is equivalent to $\tilde{d}$ not satisfying the query represented by λ.

So the equations 4.3 and 4.4 ensure that the query represented by λ is matched uniquely by d.

Since the sum of the λ_i is minimized, the number of words in the query is minimzed as well. Therefore the integer program solves the problem of minimizing the query to uniquely match d. □

Note that the above integer program formulation contains nonlinear constraints. However it is possible to avoid the nonlinear constraints and obtain a linear integer program:

Theorem 2. Let $d \in D$ be the target document. Let $W(d) = \{w_1, w_2, \ldots, w_s\}$ be the set of words that occur in d. Then the following equations formulate the optimization problem of finding the smallest set of words that uniquely identify d.

$$\min \sum_{i=1}^{s} \lambda_i, \text{ s.t.} \tag{4.5}$$

$$\lambda_i \in \{0, 1\} \tag{4.6}$$

$$\sum_{i=1}^{s} \lambda_i \delta(d, w_i) \geq 1 \tag{4.7}$$

$$\sum_{i=1}^{s} (\delta(\tilde{d}, w_i) - 1)\lambda_i < 0 \quad \forall \tilde{d} \in D \setminus \{d\} \tag{4.8}$$

Proof. Note that the only equation that changed in comparison to the previous integer program formulation is the inequality 4.8 that replaced 4.4. So it suffices to show that those two inequalities enforce equivalent constraints.

We start with a simple transformation of 4.8:

$$\sum_{i=1}^{s} (\delta(\tilde{d}, w_i) - 1)\lambda_i < 0 \Leftrightarrow \tag{4.9}$$

$$\sum_{i=1}^{s} \delta(\tilde{d}, w_i)\lambda_i < \sum_{i=1}^{s} \lambda_i \tag{4.10}$$

We only need to look at the indexes i such that $\lambda_i \neq 0$ (otherwise the corresponding terms are contributing 0 to both sides of the inequality). Then the inequality only holds if and only if there is an index i such that $\delta(\tilde{d}, w_i) = 0$. But this is equivalent to the product in 4.4 being 0. □

Note that the above linear integer program is actually a so-called 0-1 integer linear programming problem. It is known that 0-1 integer linear programming is an NP-complete problem. This raises the question whether our query minimization problem is actually NP-complete as well. This is discussed in the next subsection.

4.1.2 NP-completeness

Definition 2 (Unique target query)**.** Let D be a set of documents and Q a query. We call Q a *unique target query* if

$$|\{d \in D | d \text{ satisfies } Q\}| = 1.$$

If $d \in D$ is the only document satisfying the query Q, then we call Q a *unique target query for* d.

In the previous section we discussed the optimization problem - now we discuss the associated decision problem.

Definition 3 (Unique target query decision problem)**.** Let D be a set of documents, $d \in D$ the target document. The *unique target query decision problem (UTQ)* is formulated as follows:

Is there a unique target query Q for d such that Q contains k words?

Lemma 1. The unique target query decision problem is in NP.

Proof. We have to show that it is possible to verify a given solution in polynomial time. Let a query Q be a solution. It is possible to check for every document whether it satisfies Q or not. Furthermore this can obviously be done in polynomial time, so $UTQ \in NP$. □

In the reduction step of the NP-hardness proof of UTQ we will use the *Set Cover Problem (SCP)*, which is known to be NP-complete (for a proof, see Karp, 1972):

Definition 4 (Set Cover Problem)**.** Given a *universe* $U = \{1, 2, \ldots, m\}$ and a set S of n subsets of U, the *set cover problem* is formulated as follows:

Is there a subset of S of size k such that their union covers the universe?

Or equivalently:

$$\exists T \subseteq S : \bigcup_{t \in T} t = U \quad \text{and} \quad |T| = k$$

As a first step we reduce the set cover problem to its dual problem, which we call the *Empty Set Intersection Problem*:

Definition 5 (Empty Set Intersection Problem)**.** Given a *universe* $U = \{1, 2, \ldots, m\}$ and a set S of n subsets of U, the *Empty Set Intersection Problem* is formulated as follows:

Is there a subset of S of size k such that their intersection is empty?

Or equivalently:

$$\exists T \subseteq S : \bigcap_{t \in T} t = \emptyset \quad \text{and} \quad |T| = k$$

Lemma 2. The empty set intersection problem is NP-hard.

Proof. Let $U = \{1, 2, \ldots, m\}$ be the universe, and $S = \{S_1, S_2, \ldots, S_n\}$ be the set cover subsets of U. Let $\tilde{S}_i = U \setminus S_i$ and $\tilde{S} = \{\tilde{S}_1, \tilde{S}_2, \ldots, \tilde{S}_n\}$ be the empty set intersection subsets of U. Let $I \subseteq \{1, 2, \ldots, n\}$ with $|I| = k$. If $\{\tilde{S}_i | i \in I\}$ is a solution to the empty set intersection problem then $\{S_i | i \in I\}$ is also a solution of the set cover problem:

$$\begin{aligned}
&\bigcap_{i \in I} \tilde{S}_i = \emptyset \Rightarrow \\
&U = (\bigcap_{i \in I} \tilde{S}_i)^c = \bigcup_{i \in I} \tilde{S}_i^c = \bigcup_{i \in I} S_i \Rightarrow \\
&\bigcup_{i \in I} S_i = U
\end{aligned}$$

It can easily be seen that this reduction can be done in polynomial time. □

Now everything is prepared to prove the NP-hardness of UTQ:

Lemma 3. UTQ is NP-hard

Proof. We reduce the empty set intersection problem to UTQ.

Let $U = \{1, 2, \ldots, m\}$ be the universe, and $S = \{S_1, S_2, \ldots, S_n\}$ be the subsets of the empty set intersection problem. The universe U corresponds to the documents $D \setminus \{d\}$, where d is the target document of the UTQ problem. Each S_i corresponds to a word w_i and its elements correspond to all the documents that contain the word w_i. More precisely:

- $W = \{w_1, w_2 \ldots w_n\}$.
- $d_i = \{w_j | i \in S_j\}$, $i \in \{1, 2, \ldots, m\}$
- $d = W$
- $D = \{d_i | i \in U\} \cup \{d\}$.

Now let $Q = \bigcup_{i \in I} w_i$ be a unique target query for d with k words ($|I| = k$). We claim that

$$\bigcap_{i \in I} S_i = \emptyset.$$

S_i represents all the documents that contain the word w_i. If the intersection of the S_i above would contain an element j this would mean that a document $d_j \neq d$ would contain all elements of the query Q. But Q is a unique target query for d, a contradiction. Thus the claim is proven.

The reduction can obviously be done in polynomial time. □

Combining the lemmas 1 and 3 we finally get the following theorem:

Theorem 3. UTQ is NP-complete.

4.2 Steps of Query Suggestion Computation

This section outlines the steps necessary to compute query suggestions.

The generic algorithm for computing query suggestions consists of the following three steps:

1. Selection of candidate documents.
2. Compute terms for query suggestions.

3. Add the suggested terms to the original query and if deemed appropriate reorder the query terms.

The following subsections elaborate on the individual steps.

4.2.1 Selection of Candidate Documents

In this step relevant documents for the current query are identified. Only documents that match the current query (or at least a part of it) are selected for further computations since documents not related to the current query can't help the user to satisfy her information need. Section 4.5 covers this step.

4.2.2 Compute Terms for Query Suggestions

Once the documents that match the user's current query are identified the algorithm can start to compute which terms should be added to the current query. This is the hard part of the problem: How do we select those terms? A new term should satisfy certain properties - these can be encoded into a quality function that ranks terms for suggestion. Section 4.6 describes how this can be done.

4.2.3 Rearrange the Original Query

Simply appending a new term to a query might irritate the user because the updated query might be in conflict with the user's expecations about the structure of the data represented by the query terms.

Suppose the user searches for a customer using name and address data. If the current query is "Holzer Lerchengasse Graz" and the system returns suggestions 27 and 48 for street numbers, then "Holzer Lerchengasse Graz 27" does not capture the structure of the data well - "Holzer Lerchengasse 27 Graz" would be a much more natural suggestion, respecting the close relationship of street name and street number.

4.3 Probabilistic Mapping of Query Terms to Facets

This section describes how the indexed data can be used to probabilistically assign each query term to a facet of the indexed documents. A use case of such a mapping is the reordering of the terms of a suggested query which will be discussed further down.

First the query needs to be split into terms. For a query Q consisting of n terms we define $q_1, q_2, \ldots, q_n$ to be the individual terms. After the individual terms have been identified we can proceed to the next step: probabilistically compute their facets.

Let q be a query term. Now the target is to find the facet f_q from the set of all facets F that most probably contains q. Thus we seek

$$f_q = \arg\max_{f \in F} P(f|t)$$

Using the definition of conditional probability

$$P(a|b) = \frac{P(a \cap b)}{P(b)}$$

we get

$$P(f|t) = \frac{P(f)P(t|f)}{P(t)}.$$

$P(t|f)$ can be estimated by simple term occurrence counting:

$$P(t|f) = \frac{\text{\# occurrences of t in f in all documents}}{\text{\# of terms in f in all documents}}$$

$P(f)$ and $P(t)$ can also be estimated by simple term occurrence counting. Thus we have all ingredients to estimate $P(f|t)$. Finally iterating over all facets we can estimate f_q.

Given f_{q_i} for all $i \in \{1, 2, \ldots, n\}$ a facet-aware query Q' can be formulated:

$$Q' = f_{q_1}{:}q_1 \; f_{q_2}{:}q_2 \; \cdots \; f_{q_n}{:}q_n$$

The mapping of terms to facets computed this way might not always be clear-cut - there might be ties (or almost ties) between several facets for a single term. To respect this one could also continue computations with several mapped facets for a single term - weighted by the estimated probabilities.

4.4 Term Reordering According to Predefined Facet Order

Given a term t that is suggested for addition to a query Q it is desirable to insert it at a position where the user expects it according to her knowledge about the structure of the data. Somehow the system has to be taught this expected structure. The simplest way of doing this is to statically predefine an order on the facets.

Let $F = \{f_1, f_2, \ldots, f_k\}$ be the set of facets of the indexed documents (Note that this approach assumes F to be a statically known set of facets that doesn't change). One way of defining an order would be to define a total order $<_F$ on all facets of F. However, this approach has a serious drawback. Suppose the user enters terms in a different order - should the query be reordered after the input of each term? This seems like an annoyance to the user. A more sensible approach is needed.

Defining a less strict ordering relation on the facets lowers the probability that a user-entered query will be reordered. Our approach is to only define orderings on groups of facets that have an order that is well-understood by the users of the system. Suppose for example there are facets for the street and street number parts of an address. The common order for these facets is the street followed by the street number. Using a partial order by defining total orders only on small disjunct subsets of F triggers fewer reorderings on the user-entered part of a query while keeping the expected order for certain facets.

Suppose the ordered facet groups are $G = \{G_1, G_2, \ldots, G_l\} \cup \{G_{unordered}\}$, where $G_i = (f_{G_i,1}, \ldots, f_{G_i,k_{G_i}})$ and $G_{unordered}$ is the group of all facets that are not part of any ordered facet group. Note that the facet groups need to be pairwise disjunctive (otherwise the partial order on F potentially leads

to conflicting reorderings). Now every facet group has an associated total internal ordering $\leq_{G_i}$ (for $G_{unordered}$ we define all facets to be equal with respect to the ordering).

Using these ordered facet groups G we can define what we mean by an ordered query Q:

Definition 6 (Ordered Query)**.** Let Q be a query. Q can be split into query term groups $Q = Q_1 Q_2 \dots Q_k$ using a term to facet mapping function f as follows:

- Let $Q = q_1 q_2 \dots q_n$ be the query terms of Q.
- Starting from left to the right query terms coming after each other that are mapped by f to facets in the same facet group G_j are grouped together in a query term group Q_i.

Thus we can define a function G that maps a query term group Q_i to the facet group G_j that contains the facets to which the query terms from Q_i are mapped to: $G(Q_i) := G_j$.

Q is an *ordered query* if and only if it satisfies the following properties:

- If two different query term groups are mapped to the same facet group then this facet group is $G_{unordered}$. More formally:

$$i \neq j \wedge G(Q_i) = G(Q_j) \Rightarrow G(Q_i) = G(Q_j) = G_{unordered}$$

- Inside a query term group $Q_i = t_1 t_2 \dots t_l$, $l \geq 1$, the query terms are ordered according to its facet group $G(Q_i) = G_j$:

$$u < v \Rightarrow f(t_u) \leq_{G_j} f(t_v)$$

  Note that this also well-defined for query term groups Q_i with $G(Q_i) = G_{unordered}$.

4.4.1 Inserting a New Query Term Into an Ordered Query

Let t be a suggested term and $f_t = f(t)$ its facet and G_t the ordered facet group that contains f_t. Where should t be inserted in the original query Q to get the query suggestion Q_{sugg}?

If f_t is not part of any ordered facet group (that is, $G_t = G_{unordered}$), then t is simply appended to Q, so $Q_{sugg} = Qt$. Otherwise there are again two cases to distinguish.

Let $Q = Q_1 Q_2 \cdots Q_k$ be the query term groups of Q. Suppose there is no query term group Q_i of Q with $G(Q_i) = G_t$. Then t is again appended to Q, resulting in $Q_{sugg} = Qt$.

If there is a query term group Q_i with $G(Q_i) = G_t$ then t is inserted into this group. Let $Q_i = q_1 q_2 \cdots q_l$. t needs to be inserted into Q_i to get an updated query term group Q_i'. If $f_t <_{G_t} f(q_1)$ then $Q_i' = tQ_i$. Otherwise it is possible to define an index $\hat{u} = \max\{u | f(q_u) \leq_{G_t} f_t\}$ and insert t in Q_i after $q_{\hat{u}}$. So $Q_i' = q_1 \cdots q_{\hat{u}} t q_{\hat{u}+1} \cdots q_l$. Note that $q_{\hat{u}+1} \ldots q_l$ might be empty. After t is inserted into Q_i the query suggestion is given as $Q_{sugg} = Q_1 \cdots Q_{i-1} Q_i' Q_{i+1} \cdots Q_k$.

4.4.2 Reordering a Complete Query

Suppose a user entered several query terms without accepting a query suggestion. Then the query terms are not necessarily ordered according to the predefined orderings for the facet groups. Thus in general the terms in a query suggestion must be reordered completely - it is not sufficient to ensure that the newly suggested term is inserted according to the orderings.

However, the general case can be reduced to the one discussed above: inserting one term into a query that respects the orderings of the facet groups. Let $q_1 q_2 \cdots q_m$ be an unordered query Q. To sort the query terms q_i we define another query Q' and initialize it to $\emptyset$, the empty query. While the query Q is not empty, we take the first term from Q and insert it into Q', respecting the orderings from the ordered facet groups, and delete it from Q. Note that Q' respects the orderings from the ordered facet groups in every iteration, and thus at the end Q' is completely ordered.

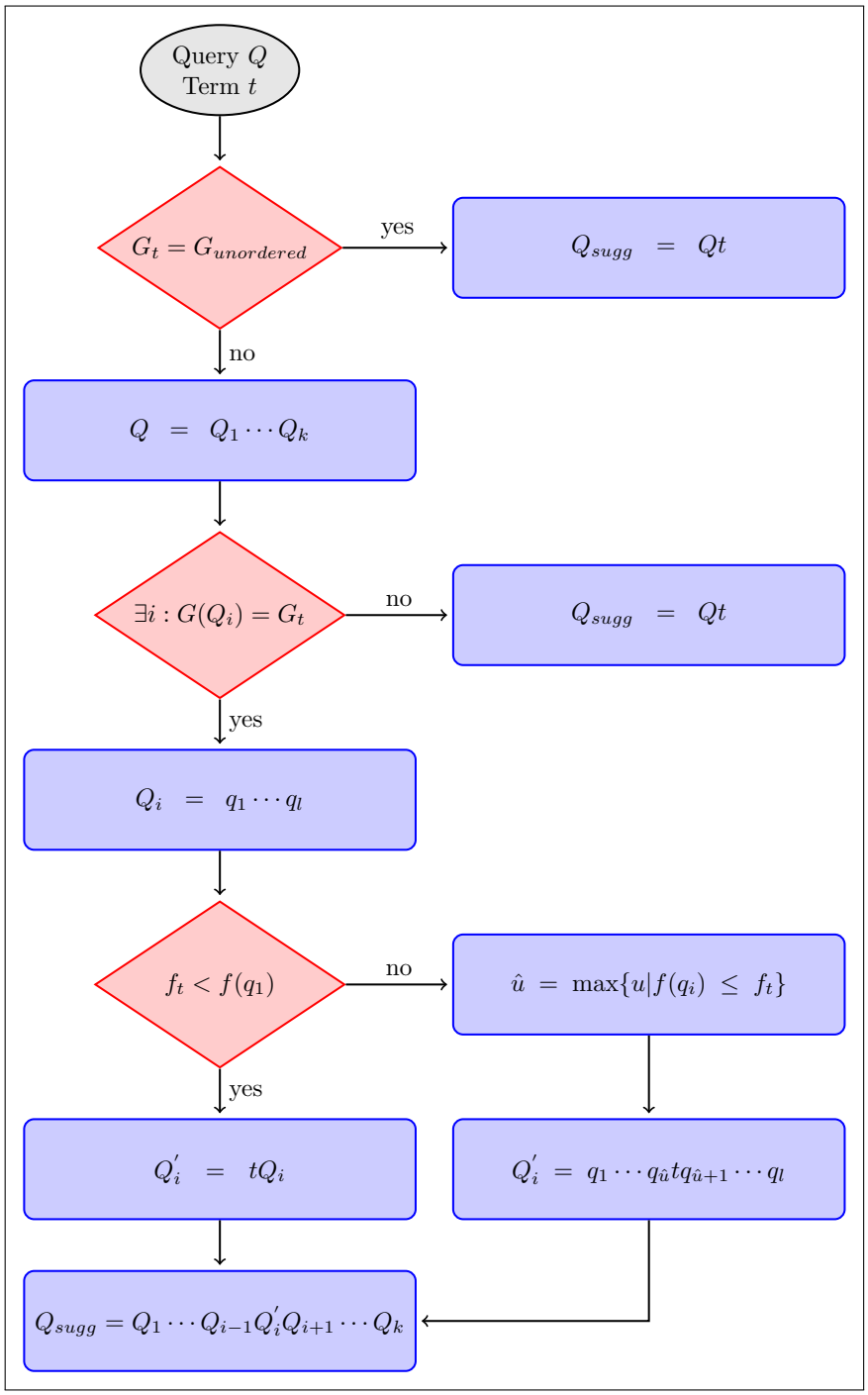


Figure 4.1: Outline of the procedure to insert a new term t into an ordered query Q to obtain a query suggestion Q_{sugg}.

4.5 Selecting a Set of Documents for the Computation of Term Suggestions

In this section it is described how, given a current query, a set of documents is chosen as input for the algorithm that computes terms for query suggestions. Essentially there are two states of the current query that need to be considered:

- The last query term is partially entered.
- The last query term is fully entered (followed by a space).

The subsections below discuss both cases. In the following let Q be the current query, consisting of the terms $q_1 q_2 \dots q_k$.

4.5.1 Fully Entered Last Query Term

Let q_k be a fully entered query term. It is noteworthy that for all query terms (except the last one in the final query the user issues) it can be easily determined when the term is fully entered: it is followed by a space character - the user is about to start entering the next query term.

Now it is straight forward to determine a set of documents that should be mined for terms for query suggestion: all documents that still match the query Q should be considered. However, the user-entered query Q is facet-agnostic, so it is necessary to construct a facet-aware query $\hat{Q}$ from Q. Without guessing a mapping from query terms to facets there is only one sensible approach: search for every query term in every facet.

As an example, suppose we have the following current query:

$$Q = \text{Donald Knuth Programming}$$

and in our document collection we have the facets *author*, *title* and *publisher*. Then (assuming query terms are combined by logical *and*) the following facet-aware query is constructed:

$$\hat{Q} =$$
$$(\text{author:donald } OR \text{ title:donald } OR \text{ publisher:donald}) AND$$
$$(\text{author:knuth } OR \text{ title:knuth } OR \text{ publisher:knuth}) AND$$
$$(\text{author:programming } OR \text{ title:programming } OR \text{ publisher:programming})$$

With the facet-aware query $\hat{Q}$ it is ensured that all matching documents contain each word from Q in some facet. So if a term t is selected from any of the matching documents, then the query $Q' = q_1 q_2 \dots q_k t$ (or rather its corresponding facet-aware query $\hat{Q}'$) is guaranteed to have a non-empty result set.

4.5.2 Partially Entered Last Query Term

Having solved the case where the last query term is fully entered, let's move on to the case where the last query term q_k is partially entered. Obviously it is not possible to use the same approach as above, since the facet-aware query $\hat{Q}$ won't have the result set expected: it is necessary to do prefix search for the partially entered term q_k. If $\hat{q_k}$ is the term the user is currently entering, then q_k is a prefix of $\hat{q_k}$, but a document containing q_k doesn't necessarily contain the full term $\hat{q_k}$.

There are basically two solutions to the problem, with different impacts on the running time of the algorithm. The first solution adjusts the facet-aware query $\hat{Q}$, while the second solution defers the prefix string problem to the selection of terms for query suggestions.

Adapting the facet-aware query $\hat{Q}$ for the prefix q_k of $\hat{q_k}$ can simply be done by replacing q_k with q_k*, representing prefix search in Lucene. So, to return to our example from above, suppose the last term *programming* was only partially entered, say: *pro*. The corresponding facet-aware query now looks like this:

$$\hat{Q} =$$
$$(\text{author:donald } OR \text{ title:donald } OR \text{ publisher:donald}) AND$$
$$(\text{author:knuth } OR \text{ title:knuth } OR \text{ publisher:knuth}) AND$$
$$(\text{author:pro* } OR \text{ title:pro* } OR \text{ publisher:pro*})$$

While this is a perfectly valid solution (all the points made in the section above still hold), performing a prefix search is a lot more costly than a plain string search. Since a fast running time is a crucial property of our algorithm, we chose a different approach.

Deferring the prefix problem to later in the algorithm when terms are selected for query suggestions, the partial term q_k is simply omitted from the facet-aware query. Continuing our example, the facet-aware query is now the following:

$$\hat{Q} =$$
$$(\text{author:donald } OR \text{ title:donald } OR \text{ publisher:donald}) AND$$
$$(\text{author:knuth } OR \text{ title:knuth } OR \text{ publisher:knuth})$$

Later, when the terms for query suggestions are actually selected, the algorithm only accepts terms that have q_k as a prefix (note that this step is also necessary in the first solution, albeit on a smaller set of terms). So we trade faster search for iterating on a larger set of terms later on.

Having selected a set of documents we can now move on to the next part of the algorithm - computing terms for query suggestions from those documents.

4.6 Computing Terms for Query Suggestions

At the heart of our problem is the selection of terms for query suggestions - a query suggestion needs to add a useful term to the current query. After the

previous section showed how a set of relevant documents can be obtained from the current query, this section deals with the problem of selecting useful terms from these documents for query suggestions.

We start by defining desirable properties of terms for query suggestions in 4.6.1, then we describe how these properties can be encoded into an objective (or *quality*) function in 4.6.2. In 4.6.3 a variant of the objective function that tries to further exploit the relationships between facets is presented. Finally an approach to smoothen the objective function is presented in 4.6.4.

4.6.1 Desirable Properties of Term Suggestions

Before presenting an objective function, let's consider what properties a term should fulfill to be a useful term for query suggestions:

- It should be present in a large fraction of the candidate documents, and preferably in the same facet. If a term is present in many candidate documents it is likely that the user is searching for one of these documents.
- It should further minimize the result set (remember: the user is searching for a single document). Or put in a different way: It should not suggest redundant terms (a redundant term is a term that is present in all candidate documents).
- It should not lead to an empty result set (otherwise it would not be a helpful query suggestion!).

4.6.2 Quality Function

To quantify the above requirements for a suggestion term in a compact way we introduce an objective function that respects all these requirements.

We start by introducing the building blocks of the final objective function. Let D be the set of candidate documents, $n = |D|$, let F be the set of facets of documents in D and let t be an arbitrary term.

If $f \in F$ and $d \in D$ then $f(d)$ is the facet f of d. We write $t \in f(d)$ if t occurs in the facet f of d.

$rdc_D(t)$ describes the number of documents in D where t occurs in some facet ("relative document count"):

$$rdc_D(t) = |\{d \in D | \exists f \in F : t \in f(d)\}|$$

$rfc_D(t)$ counts the number of facets of documents in D that contain occurrences of t ("relative facet count"):

$$rfc_D(t) = |\{f \in F | \exists d \in D : t \in f(d)\}|$$

We define $r_D(t)$ as a redundancy factor of t with respect to D:

$$r_D(t) = \begin{cases} \frac{1}{n+1} & \text{if } rdc_D(t) = n \\ 1 & \text{otherwise} \end{cases} \tag{4.11}$$

Now we can define our objective (or *quality*) function $q_D(t)$:

$$q_D(t) = \frac{rdc_D(t)}{rfc_D(t)} r_D(t) \tag{4.12}$$

Some properties of $q_D(t)$:

- $q_D(t) \geq 0$
- $q_D(t) > 0 \Leftrightarrow \exists d \in D : \exists f \in F : t \in f(d)$
- Let t_1, t_2 be two terms such that $rfc_D(t_1) = rfc_D(t_2)$ and $r_D(t_1) = r_D(t_2)$. Then $q_D(t_1) \leq q_D(t_2) \Leftrightarrow rdc_D(t_1) \leq rdc_D(t_2)$.
- Let t_1, t_2 be two terms such that $rdc_D(t_1) = rdc_D(t_2)$ and $r_D(t_1) = r_D(t_2)$. Then $q_D(t_1) \leq q_D(t_2) \Leftrightarrow rfc_D(t_1) \geq rfc_D(t_2)$.
- Let $|D| > 1$ and let t_1, t_2 be two terms such that $rdc_D(t_1) = 1$, $rdc_D(t_2) = |D|$ and $rfc_D(t_1) = rfc_D(t_2)$. Then $q_D(t_1) > q_D(t_2)$. (Note that this is ensured by the $(n+1)$ term in the denominator of the redundancy factor, using n would fail to ensure this property.)

Given the above properties we can conclude that $q_D(t)$ satisfies all our requirements of an objective function. Computing k terms for query suggestions can now be done by computing $q_D(t)$ for all terms t and select the terms that yield the top k values for $q_D(t)$.

4.6.3 Facet Aware Quality Function

The quality function q_D as described above is independent of the other terms in the query. However, if some information is known about the (expected) structure of the query this can be exploited as well.

To be able to do this there are essentially two building blocks needed: firstly there needs to be some way to map a query term to a specific facet and secondly the expected structure of the query (with respect to the order of the facets the query terms are mapped to) needs to be known. Mapping query terms to facets is discussed in section (4.3). Dealing with an expected structure of queries is discussed in section (4.4).

If a suggested term would fit into the expected query structure the quality function can boost this term. Suppose for example a term from a facet A is supposed to be followed by a term from a facet B and the last entered query term has been mapped to facet A. Then a suggested term t that is mapped to facet B should be boosted. As a more concrete example suppose the term "Michael" was the last query term. "Michael" can be mapped to a first name, and after a first name a last name is expected. The suggested term that is mapped to the lastname facet should be boosted with respect to terms that are mapped to other facets.

Let's put this ideas into a more formal notation. Let q_D be the quality function from above, let f_D be a function that maps terms to facets with respect to a set of documents D, let t_q be the last term of the current query and let b be the boost factor for suggested terms that are mapped to the expected facet. First we define a function δ_f that represents the expected structure of the query. For two facets f_1 and f_2 it decides whether a term of facet f_2 is expected to follow a term of facet f_1 in a query:

$$\delta_f(f_1, f_2) = \begin{cases} 1 & \text{if a term of facet } f_2 \text{ is expected to follow a term of facet } f_1 \\ 0 & \text{otherwise} \end{cases}$$

Now we can define a function $b_D(t)$ that computes the facet boost factor for a suggested term:

$$b_D(t) = \begin{cases} b & \text{if } \delta_f(f_D(t_q), f_D(t)) = 1 \\ 1 & \text{otherwise} \end{cases}$$

Finally we can define our facet aware quality function q_D^f:

$$q_D^f(t) = b_D(t) q_D(t) \tag{4.13}$$

Note that q_D^f can not only consider relationships between different facets but also boost suggested terms that are mapped to the same facet as the last query term - it all depends on information encoded in the function δ_f. Boosting terms of the same facet can make sense for facets that contain multiple words (e.g. last names or names of cities are not always single words, they can also be composed of multiple words).

4.6.4 Dampening the Redundancy Factor

The redundancy factor, defined in equation (4.11) introduces a rather crude point of discontinuity when a term occurs in all documents of D. To mitigate this one can try to smoothen this discontinuity by using a dampening function. Using two different dampening functions we introduce two more variations of our quality function (note that these alternative redundancy factor functions can be used both in the facet-agnostic and the facet-aware quality functions defined above):

$$r_D^{sqrt}(t) = \begin{cases} \frac{1}{\sqrt{n+1}} & \text{if } rdc_D(t) = n \\ 1 & \text{otherwise} \end{cases} \tag{4.14}$$

$$r_D^{log}(t) = \begin{cases} \frac{1}{\log(n)+1} & \text{if } rdc_D(t) = n \\ 1 & \text{otherwise} \end{cases} \tag{4.15}$$

4.7 Summary

This chapter started with bad news: the optimization problem lying at the heart of our IR problem was shown to be based on an NP-complete problem. However, the NP-completeness mainly concerns the evaluation of a query for a specific document, computing query suggestions for an unknown document works more or less the other way around. Thus we continued by developing a heuristic approach for computing query suggestions. The whole process of computing query suggestions in our setting was formalized, separated into multiple steps. The main step is the computation of terms to be added to the current query. The actual selection of the terms is delegated to a quality function, assessing the relevance or usefulness of a term for the current query. Several quality functions were introduced, though one forms the basis of the others. This quality function not only considers the total amount of documents that contain a term, but also takes into account that the user's goal is to find a single document.

Having laid the theoretical foundations in this chapter, the next one provides some insights into the details of the implementation of the developed ideas. In particular, it focuses on one aspect that was neglected in this chapter: performance in the sense of response time to the user - a superior query suggestion algorithm is worthless if the user can pause for a cup of coffee while waiting for suggestions to appear.

5 Implementation

This chapter presents the most important aspects of the implementation of the ideas described in the previous chapters. After the first section reveals the technologies used in the implementation the following sections present the most interesting parts of the implementation.

After settling on the theoretical aspects of a query suggester the main practical challenge is to achieve acceptable performance in terms of response time. Of all the parts necessary for the computation of query suggestions as described in the previous chapter the step selecting terms for query suggestions is practically the most demanding one. The main reason for this is that the algorithm to identify terms for query suggestions operates on the complete index - potentially a very large set of data. There were other non-trivial challenges that needed solving in the implementation, but to both avoid bloating of this section and to try to focus on ideas that could prove to be useful to others we limit the discussion of implementation details to this topic.

In section 5.2 we briefly present a first naive implementation that directly uses the index provided by Lucene. This implementation was used in the beginning to test the practicality of ideas for query suggestion.

In terms of computation time the approach of directly using the index provided by Lucene proved to be unacceptable. A more specialized representation of the document collection was needed. In section 5.3 a rather simple array-based solution is presented that provides sensible response times for computation of terms for query suggestions. Both constructing the data structure for a given document collection and using the data structure to identify terms for query suggestions are described.

5.1 Used Technology

For the implementation the programming language Java was chosen. Common information retrieval functionality was built upon the Apache Lucene library [1]. Lucene is a library providing functionality for various aspects of full text search. At its core is an implementation of an inverted index, surrounded by a vast array of supporting and supplementing features like tokenization, analysis, query parsing, scoring or hit highlighting.

Lucene was used for building and querying inverted indices for document collections. This includes Lucene's capabilities for tokenization, analysis, queryparsing and scoring. Fast querying of an inverted index is also crucial for query suggestion since the set of documents matching the current query needs to be computed at every keystroke the user performs.

5.2 Using Lucene to Compute Terms for Query Suggestions

This section briefly describes the attempt to directly use a Lucene index for the computation of terms for query suggestions.

Being used for full text search on the document collection the Lucene index is already available, so the straight forward solution is to use it for the computation of terms for query suggestions as well. To compute all the expressions needed in the quality function described in equation 4.12 we need to count the occurrences of all the terms in all facets of all documents that match the current query. While it is possible, such an iteration is not idiomatic for an inverted index - an inverted index is good to map from terms to documents that contain them, not for iteration over terms in a document. In addition, to actually count term occurrences it is also necessary to perform string comparisons, another time consuming task. Consequently, it turns out that this approach can not be used to provide real time query suggestions.

[1] http://lucene.apache.org

5.3 Using a Custom Datastructure to Compute Terms for Query Suggestion

With the first approach (using the Lucene index directly) failing, another solution was needed. This time a custom datastructure was designed to provide better performance.

5.3.1 Requirements

The custom datastructure needs to provide fast implementations for the following operations on a set of documents D:

- For each term t count the number of documents in D that contain t.
- For each term t count the number of facets in D that contain t.

Since the set of documents is arbitrary (it depends on the user's current query), the datastructure has to contain information about the complete document collection.

5.3.2 Datastructure

One issue of the Lucene based implementation was the usage of string comparisons. To mitigate this issue we assign a unique integer to each term that occurs in the document collection, introducing a mapping from strings to integers. Thus each document is represented by an array of integers. To be more precise it is also needed to account for the facets of the documents. This can be done by representing each facet by an array of integers, and the documents in turn by an array of facets - so each document is represented by an array of arrays of integers. Note that this induces a mapping of facets to array indices. However, our algorithm only needs the number of facets that contain occurrences of a term, not the facets themselves.

See figure 5.1 for an example of transforming two documents into their integer arrays representation. The associated term to integer mapping is shown in figure 5.2.

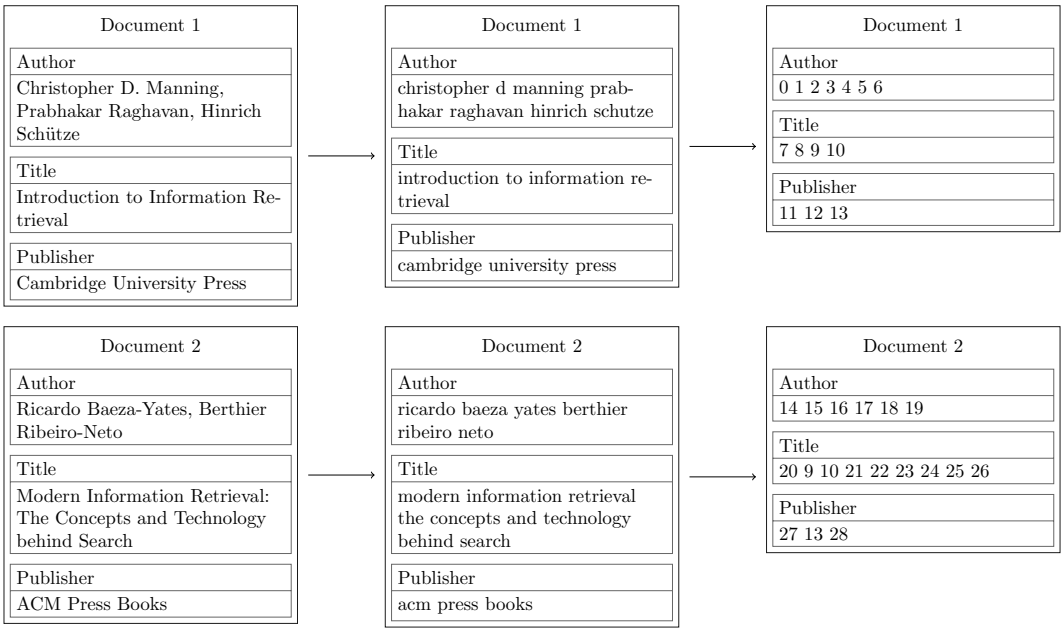


Figure 5.1: Transforming documents to arrays of integers.

0	christopher	8	to	16	yates	24	technology
1	d	9	information	17	berthier	25	behind
2	manning	10	retrieval	18	ribeiro	26	search
3	prabhakar	11	cambridge	19	neto	27	acm
4	raghavan	12	university	20	modern	28	books
5	hinrich	13	press	21	the		
6	schutze	14	ricardo	22	concepts		
7	introduction	15	baeza	23	and		

Figure 5.2: Mapping of terms to integers.

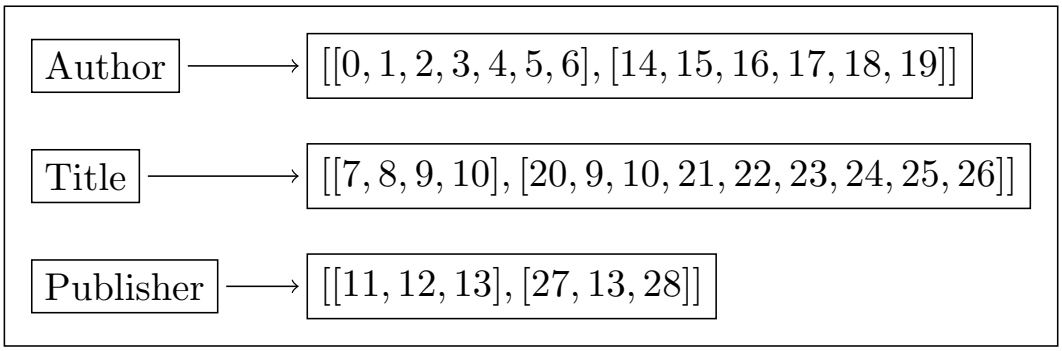


Figure 5.3: Representing the facets of a document collection by arrays of integers

So far we represent each document as an array of arrays of integers. For reasons that will be explained in the subsection 5.3.4 we split the documents into their facets and represent each facet of the *document collection* as an array of documents, in turn represented by arrays of integers. Figure 5.3 shows an example of this representation.

5.3.3 Construction

Since each document of the document collection is represented by an array of integers what is needed is esentially a mapping of terms to integer IDs. To account for term analysis (i.e. tokenization and normalization) it is the analyzed terms that are mapped to IDs. However, to provide sensible query suggestion terms it is needed to keep the unnormalized form of terms for presentation to the user. Thus the construction algorithm needs to keep track of the following:

- The mapping of analyzed terms to integer IDs.
- The mapping of integer IDs back to the unnormalized terms for presentation to the user.
- The next available integer ID.

The construction algorithm simply iterates over all documents, and for each document over all its facets. Term to integer ID mapping is global (over the whole document collection), so for each encountered term the

algorithm checks whether its analyzed form has already been assigned an ID and reuses it if this is the case, otherwise a new ID is assigned. For each facet of a document this results in an array of integer IDs. For each facet of the document collection this results in an array of arrays of integer IDs, one for each document. In the implementation this is represented by a $Map < String, int[][] >$, each entry representing a facet of the document collection. In addition to this map the second part of the output of the construction of our datastructure is an array of $TermInfo$ objects that contain the normalized term as well as its unnormalized form. The indices of this array correspond to the term IDs from the mapping of terms to integer IDs. So to find the term given its ID is a single array access (see figure 5.2 for an example of such an array).

The construction algorithm interates over all terms of all documents in the collection. Let n be this total number terms. For each term the most expensive operations are the access and potential update of the term to integer ID mapping (all other operations can easily be done in constant time). Using a HashMap these map operations have a best case running time of $\mathcal{O}(1)$, but it can be worse. However, it is quite easy to achieve $\mathcal{O}(\log n)$ worst case running time using a different datastructure, resulting in an overall construction time of $\mathcal{O}(n \log n)$.

Note that the construction algorithm avoids the usage of $Integer$ objects (as opposed to the primitive int values) - an intricacy of the Java programming language that wraps primitive integer values into objects to make integers usable in generic collections (or rather in generics in general). Previous versions of this algorithm used integer objects to provide the mapping back from integer IDs to the term information (using a $Map < Integer, TermInfo >$) which turned out to be a serious performance issue: the JVM had a lot of work to do garbage collecting short-lived Integer objects.

5.3.4 Usage

Now that we have the documents represented as arrays of integer IDs (partitioned by facets) we need to put this datastructure into use. When we want to compute terms for query suggestion we need essentially two inputs (apart from our datastructure): a set of documents (given by ID)

and a quality function that quantifies the relevance of a term for query suggestion. The quality function gets as input the following information about a candidate term: the normalized and unnormalized form of the term, and the number of documents and facets that contain occurrences of the term (more accurately: occurrences of terms that are normalized to the same token).

Thus we need to count occurrences of terms in facets and in documents over a set of document IDs. In the following parts we will discuss how our algorithm solves both problems.

The running time of the various steps of our algorithm will mainly depend on the total number of terms in the selected document set. We will use k to refer to this number.

Preparation for Counting the Number of Facets that Contain a Term

A straight-forward implementation would be to iterate over all documents and keep track of the terms encountered in each facet. However, both the running time and the space usage are less than optimal - we can do much better.

Firstly, note that we already partitioned the integer term ID arrays for the documents by facets. So, given a document ID and a facet we can quickly access its terms. As the first step the algorithm accumulates for each facet all the term IDs for all documents, resulting in a single integer array for each facet. It is here (and in the following steps) where the partition by facets has significant benefits over partitioning by document. This step is a simple concatenation of arrays, having running time $\mathcal{O}(k)$.

To be able to iterate over all facets simultaneously we sort these integer arrays (a simple upper bound for the running time is $\mathcal{O}(k \log k)$). With these sorted arrays it is quite straightforward to iterate over all terms over all facets. For each term ID we iterate over all the facets and count the facets that contain the term. And since the number of facets can typically be bounded by a constant the simultaneous iteration turns out to be efficient as well (essentially the time complexity is $O(k)$) - the overall running time is dominated by the sorting step.

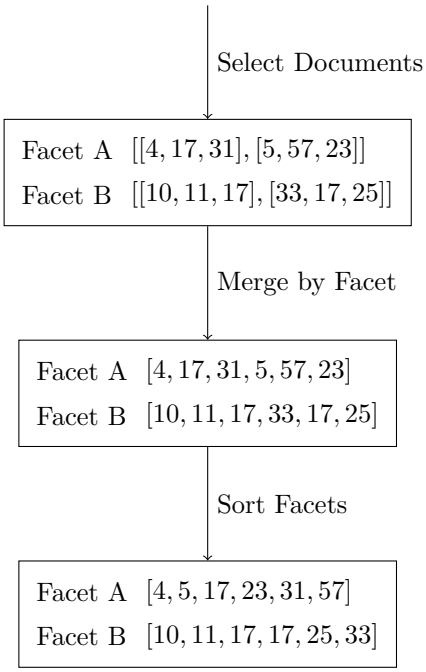


Figure 5.4: Preparation for counting the number of facets that contain a term.

See figure 5.4 for an illustration of the preparation step for counting the number of facets that contain a term.

Preparation for Counting the Number of Documents that Contain a Term

The algorithm for counting the number of facets that contain an occurrence of a term can easily be extended to also compute the total number of occurrences of a term in a set of documents - when counting the number of facets that contain a specific term we also have the information how often the term actually occurs in a facet, so a simple counter is enough. However, we don't want the total number of occurrences of a term in a set of documents, we need the number of documents that contain an occurrence of a term.

Our datastructure is actually less suited for this than for counting the

number of facets containing a term, but it can still be done quite efficiently. At first we need to collect all term occurrences for each document. This step can be done in linear time with respect to the total number of term occurrences in the set of documents (simply copying and merging the arrays from the facet partitions is enough).

Now each document is represented by an array of integer IDs. These arrays can contain duplicates - a term can occur multiple times in a document. However, we are not interested in the number of times a term occurs in a document, only whether it occurs in a document or not. The easiest and quite efficient way get rid of these duplicate entries is to first sort the arrays and then copy the unique entries to a new array (now duplicates can be easily identified since they occur next to each other in the sorted array).

At this point we have for each document the set of terms it contains. The arrays representing these term sets are now concatenated into a single array. Sorting this array finally enables efficient counting of the number of documents that contain a term - just as it was done for the facet counting part above.

See figure 5.5 for an illustration of the preparation step for counting the number of documents that contain a term.

While the approach described above works, a different and arguably more efficient alternative approach trades more memory usage for a better running time. The idea behind this second approach is essentially to do the aggregation of terms of a document into a pseudo catch-all facet when setting up the datastructure. This way the elimination of duplicate terms only needs to occur once at construction time and not at each invocation - this part can now be done analogously to the other ("real") facets: simply concatenate the arrays for the selected documents by facet.

The running times (both construction and usage) are not affected by this change - it's still $\mathcal{O}(n \log n)$ for construction and $\mathcal{O}(k \log k)$ for usage.

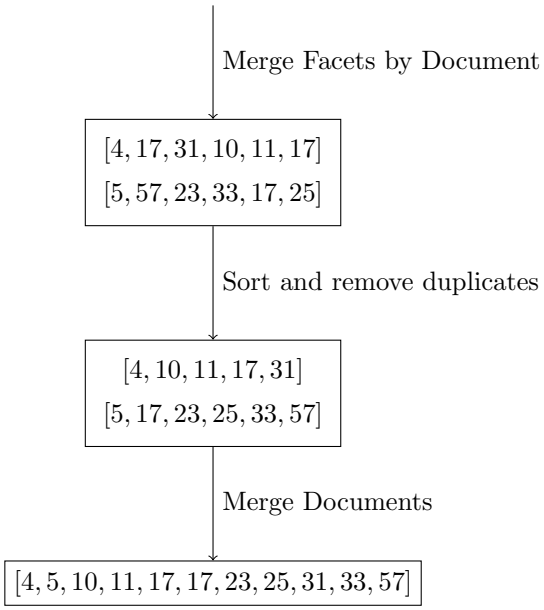


Figure 5.5: Preparation for counting the number of documents that contain a term.

Counting the Number of Documents and Facets that Contain a Term

Now we are finally ready to actually count the number of documents and facets that contain a term. In fact, the counting of the number of facets and the number of documents that contain a term can now be done simultaneously - providing all the information we need about a term at the same time.

Since both the arrays containing the term IDs for the facets as well as the array containing the term IDs for all documents are already sorted we can iterate simultaneously over all those arrays, counting the number of occurrences of each term ID while iterating, both the number of facets and the number of documents. For each iterated array we need to keep track of the currently visited element, and globally we need to keep track of the minimum term ID that is currently visited by any array. For this minimum term ID we actually count the occurrences - all arrays containing an occurrence of this ID have it as their currently visited ID, thus at this point counting is very easy. Note that for the facet arrays we ignore multiple

occurrences of a term ID (we are only interested whether the term ID occurs in a facet of not), whereas for the document array we count them (duplicates inside a document were already removed in the preparation step). Once the occurrences of the minimal term ID have been counted the currently visited element is advanced to the next element in all arrays that contained the minimal term ID. Finally the next globally minimal term ID is computed and the iteration continues. Since the algorithm iterates over all arrays only once and for each term ID only constant time operations are executed the running time of this step is bounded by $\mathcal{O}(k)$.

See figure 5.6 for an illustration of this process.

Ranking and Selecting Terms

Once all the information needed about a term in a set of documents is collected this information is handed to a quality function. This function contains the crucial logic in deciding which terms should be suggested - for a given term it computes a score, a single number. Thus we can rank all terms according to a quality function. Keeping a sorted list of all terms is rather inefficient. It suffices to keep the k top scoring terms, where k is the number of suggestions finally presented to the user.

This far the algorithm operated only on the integer IDs, but we need to present the original term to the user. The term can be easily recovered from our datastructure - it contains a mapping from integer IDs to the associated terms.

One detail we ommitted so far is that certain terms are not eligible for use in query suggestions. Terms that occur already in the user's query should certainly not be presented again as suggestions, so those terms need to be filtered out by the suggestion algorithm. This can easily be done by comparing a candidate term for suggestion with the terms already present in the query as part of the ranking step.

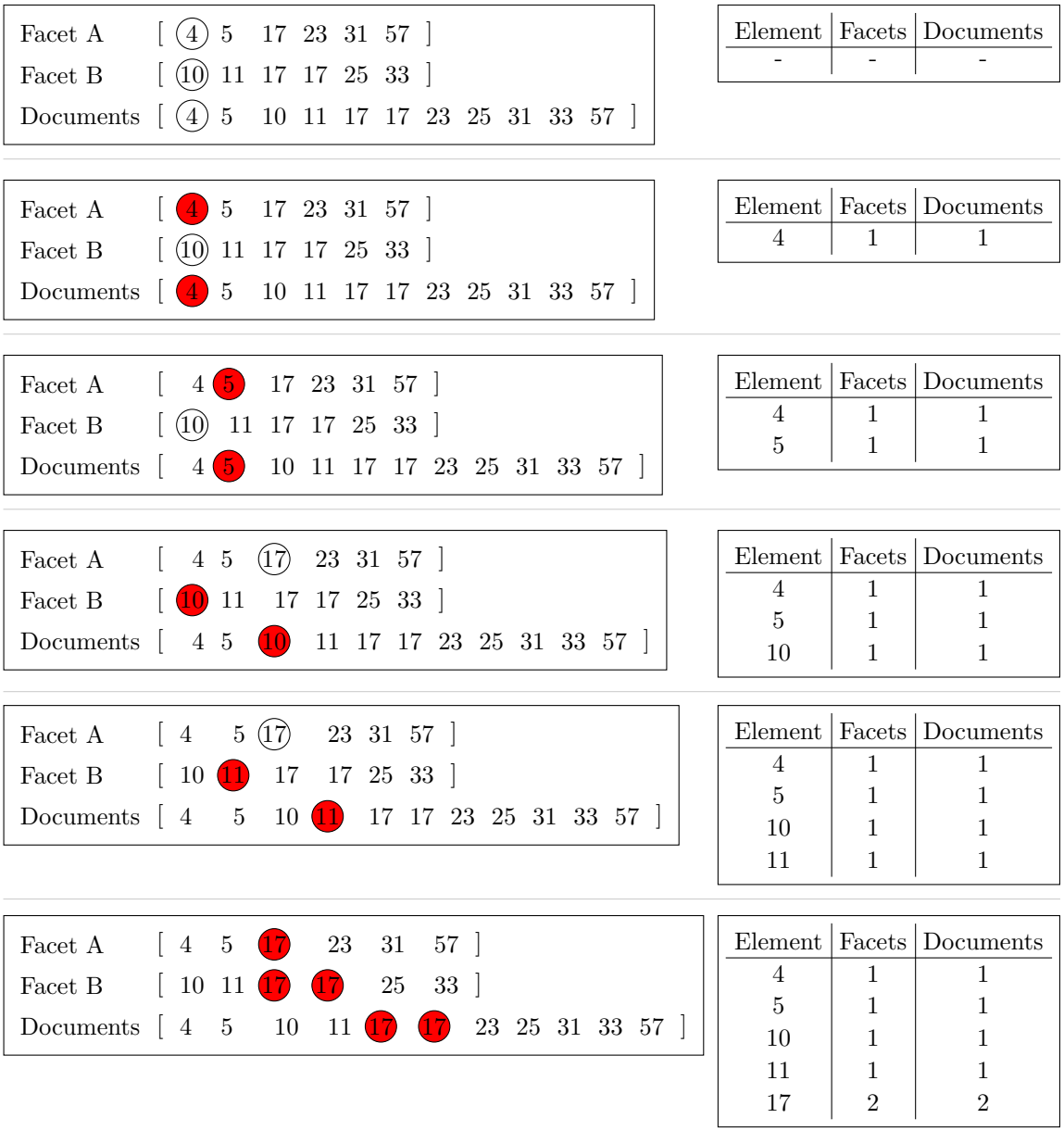


Figure 5.6: Simultaneous counting of number of facets and documents that contain a term. Circles indicate the currently visited element in an array, the globally minimal term ID among the currently visited elements is highlighted with red. The tables at the right show the counted elements. The *Facets* column shows the number of facets containing an element, and the *Documents* column shows the number of documents containing an element.

Computing Term Suggestions for Partially Entered Last Query Terms

So far we only discussed the case of suggesting complete terms, i.e. the user is about to start entering a new query term. But what about the case when the user has partially entered the last query term (i.e. she is currently typing a word)?

As it turns out we only need to adjust the last part of the computation of terms for query suggestions. We start the computation as if the partial last term hasn't been entered, i.e. we operate on the set of documents that match the user's query without the partial last query term. In the final ranking step we simply filter out all terms that do not start with the already entered prefix - this way our algorithm only suggests terms that have the partially entered last query term as a prefix.

5.3.5 Limitations

The current implementation of the datastructure described above has some shortcomings. Those proved to be not relevant for research purposes, but may be more problematic in a real-world setting.

RAM-only

Currently the whole datastructure is loaded into the working memory. For very large document collections this can become a problem. The solution would be to persist the datastructure into the file system and only load/keep the parts needed to/in RAM - similarly to Lucene's file based inverted index.

Non-Incremental

It is so far not possible to apply changes in the document collection to the datastructure. Once the datastructure is constructed for a document collection it remains static. A workaround for evolving document collections would be to periodically rebuilt the query suggestion datastructure (taking

$\mathcal{O}(n \log n)$ time). However, a more sensible approach would be to implement an update feature into the datastructure itself. As interesting as this topic is, for this thesis it was out of scope.

5.4 Summary

Computing query suggestions as outlined in chapter 4 is a computationally expensive problem. Unfortunately it does not seem to be possible to directly use an inverted index (as provided by Lucene) as the basis for the computations. Another approach was needed and this chapter introduced a custom datastructure that enabled efficient calculation of terms for query suggestions. This datastructure is able to provide the needed data for selection of terms for query suggestions for document collections with millions of documents with a response time in the magnitude of milleseconds.

6 Evaluation

So far the discussion was solely on theoretical and implementational aspects of query suggestions. In this chapter the developed algorithms are put into practice by evaluating them against a baseline query suggestion algorithm. The evaluation transforms the claims about the capabilities of our algorithms into numbers. By performing the evaluation on document collections of different characteristics we get some insights into the strenghts and weaknesses of the algorithms.

The methodology of our evaluation is presented in the first section. This includes the description of all involved query suggestion algorithms: the used baseline as well as our algorithms (varied by choosing different parameters). Along with the used algorithms this section also presents how the performance of query suggestions is put into numbers - a crucial point of the evaluation.

The document collections used for evaluation are described in section 6.2. Those collections vary in various characteristics, e.g. in size (number of documents) and term distributions.

The results of the evaluation runs are presented in section 6.3. Note that the interpretation and discussion of these results is deferred to the next chapter that is dedicated entirely to this.

6.1 Methodology

To evaluate the performance of an IR system there are two main choices available: either do a user study or do automated tests without human interaction. A big advantage of the latter is that once the setup is done it can

be performed repeatedly at low cost (in terms of e.g. time or space for and coordination of people). So if one decides to do an evaluation on a different document collection it is much cheaper to redo an evaluation based on automated tests than to repeat a user study with dozens of participants.

Because of both the reasons outlined above as well as others we decided to do our evaluation based on automated tests. For the setup of those tests there are various aspects that need to be resolved. How do you model your users? How do you quantify the results or benefits of query suggestions? The following subsections describe how our evaluation setup addressed these issues.

6.1.1 Quantifying the Performance of Query Suggestions

To gain any understanding about the performance of a query suggestion algorithm there needs to be some means to quantify the usefulness of query suggestions to the user. The main goal is to enable the user to satisfy her information need more quickly. Setting factors aside that are beyond the influence of an IR system (such as the time the user spends thinking), the limiting factors are the user's interaction with the system. In our case this means typing into a textfield, as well as selecting query suggestions.

Those two interactions can be quite simply quantified by counting keystrokes: typing a letter is one keystroke, thus counting 1. Counting the acceptance of a query suggestion is slightly more involved: one possibility to accept a query suggestion is by selecting it with a mouse click, but that doesn't match well with counting key strokes. Another way is by using the arrow keys to navigate to the suggestions. Pressing the *down* key once selects the top-ranked query suggestion. Each time the *down* key is pressed again, the selection moves down to the next query suggestion. Finally accepting a query suggestion is done by pressing the *enter* key. So to accept the query suggestion ranked at place k, the user needs to press the *down* key k times and hit *enter* once, resulting in $k + 1$ key strokes.

So now we have a model to quantify the costs of entering a query: the number of letters typed plus the number of key strokes needed for accepting query suggestions. Next we need to find a way to generate representative

queries. As it turns out it is simpler to look at this problem from the other side, the other side of a query being its target: a document. Since our presumption is that in our setting each information need is satisfied by a unique result document we know that an entered query has a single document as the target. So to model an information need we start from a document d and generate a query from d.

This gets us to the next question: How, given a document, do we generate a query? First, note that a query Q that consists of all the terms of a document d appended together can only be matched by d (otherwise our assumptions are violated). The only thing to decide is the order how the terms are appended: randomly, facet after facet, or iterate over all facets, taking a single term each round? We decided to choose the facet after facet approach: define an order on the facets, concatenate the terms of each facet and then concatenate these term blocks by the defined order on the facets. Note that ideally one would compute the shortest possible query that has only d in its result set. However, this is unfeasible for large document collections, as this optimization problem (or more correctly: a simplified version of it) was shown to be NP-complete in 4.1.2.

Now we have all ingredients that are needed to do the evaluation of a query suggestion algorithm on a query for a target document d. The evaluation tries to mimic the behaviour of a "perfect" user: it "types" the query, accepting provided suggestions if they cost less in terms of key strokes than completely writing the current term. This continues as long as the query is ambiguous (i.e. the size of the result set contains more than one document). As soon as only the target document d is the only document still matching the current query, the "typing" is stopped and the costs of entering the query are summed up.

This evaluation process is illustrated in figure 6.1. The generated query is "USA Alfred Hitchcock". "USA" is typed, no suggestions are used, so the costs are equal to the number of letters in the word: 3. The next term is "Alfred". Here, after typing "Alf" (adding 3 to the costs), the term is present in the list of suggestions on the second position, so the costs for accepting the suggestion are 3: 2 for hitting the *down* key twice to select the suggestion and 1 to accept the selected suggestion by hitting the *enter* key. Note that simply typing the whole term "Alfred" would have resulted in the same

Action	Content	Costs
Type	USA	3
Type	Alf	3
Select Suggestion	Alfonso Alfred AlFayyum Alfred-Pischof-Gasse Aelfgifu	3
Type	H	1
Select Suggestion	Hitchcock Hardware Hurts Heart Hart	2
	Total costs:	12

Figure 6.1: Example evaluation of query.

total costs. For the last term in the example, the right term appears in the suggestions on the top position after entering just the first letter, giving a total cost of 3 for the term "Hitchcock", a clear win over just typing the term (costing 9). So the summed up costs for the whole query are 12 (typing the whole query would have resulted in a total cost of 18).

6.1.2 Baseline

As the baseline we use an approach that adapts a standard query suggestion algorithm that operates on single-facet documents for faceted documents.

More specifically we mimic the strategy to collect data from all facets into a single facet. When selecting terms for query suggestions this algorithm selects the terms that have the most occurrences in this *catch-all* facet in the selected documents.

To fit this baseline algorithm into the query suggestion framework developed for our new query suggestion algorithm the easiest way is to design a quality function with the same parameters as our quality functions 4.12 and 4.13. This means that for a term t the function gets two numbers as its input: $rfc_D(t)$ (the number of facets that contain an occurrence of t) and $rdc_D(t)$ (the number of documents that contain an occurrence of t). For the baseline quality function we simply return $rdc_D(t)$:

$$q^b_D(t) = rdc_D(t) \tag{6.1}$$

In addition to this "real" baseline we also evaluate the suggestion algorithms against the worst case: no query suggestions at all are provided to the user. Put differently, this mimics a user who doesn't use query suggestions at all, simply typing her query without any assistance. This "bottom baseline" is included as a sort of sanity check for our query suggestions: Do they help at all? Because if typing only is not worse than utilizing query suggestions then our query suggestion algorithms have clearly gone wrong.

6.1.3 Quality Functions

Both quality functions from chapter 4, the facet-agnostic 4.12 and the facet-aware 4.13 are evaluated. For the facet-aware quality function the order of the facets is provided, matching the order of the facets used in the evaluation query construction outlined above. Additionally two variations of the facet-aware quality function are evaluated as well: they employ the dampened redundancy factor functions defined in 4.14 and 4.15.

6.1.4 Common Parameters

Some parameters of the query suggestion algorithm don't depend on the used quality function. Perhaps the most obvious one is the number of query suggestions that are presented to the user. In the evaluation runs this number was set to 5. Another common parameter is the maximal number of documents that are used for the computation of terms for query suggestion. If this number is set too high the response time of the algorithm suffers, resulting in an inacceptible user experience. If it is set too low then the query suggestions might not be as useful as they could be. For the evaluation this number was set to 5000. This means, if the current query is still matched by more than 5000 documents in the document collection, then only the 5000 top ranked documents are further processed by the query suggestion algorithm.

For all facet aware quality functions one can choose two more parameters: whether to boost terms from the same facet and the boost factor for terms that match the given facet order. In the evaluation runs a constant boost factor of 2 was used, and terms from the same facet as the previous term were always boosted.

6.2 Test Data

For an evaluation one obviously needs data - in this case a document collection. This section describes how the document collections used in the evaluation runs were generated. Basically two kinds of document collections were generated: one containing people (names and addresses) owning books (title, author, year of publication) and one with several facets containing words drawn from a set of Zipf distributed words (same set of words, different distributions for each facet).

6.2.1 People and Books

This document collection contains documents representing people and books "owned" by people. Each person is randomly assigned between 0

Facet	Description
firstName	First name of the person
lastName	Last name of the person
personType	Type of the person (by Wikipedia), may be empty
cityName	Person's home city
cityCountry	Person's home country
cityPopulation	Population of person's home city
streetName	Street the person lives in
houseNumber	House number of person's home
bookTitle	Title of the owned book
bookAuthor	Author of the owned book
bookYear	Publication year of the owned book
type	"person" or "ownedBook"

Table 6.1: Document structure for *People and Books* index

and 10 books. See table 6.1 for a description of the documents' structure.

The data are taken from multiple sources as described in the following paragraphs.

Person Name and Type

These data were parsed from the English Wikipedia (*Wikipedia* 2014). Name and person type were taken from the "Infobox" part of articles about people. Names on wikipedia are not split up into first and last name, rather all parts of the name are concatenated to the full name. To extract first and last name we employed a rather crude heuristic to take the first word (indicated by a following whitespace character) as the first name and concatenate the remaining words to represent the last name. Some full names also contain a comma - then the part preceding the comma is typically the lastname and the part after the comma constitutes the first name(s). In the context of our evaluation this simple heuristic is enough, since the correctness of first/last names is less important than having realistically distributed first/last names.

The header of Wikipedia Infoboxes contains information about a person's occupation - this was used as the person type.

Person data for index documents were generated by first randomizing the order of items and then iterate over them. Given the natural distribution of names on Wikipedia this leads to a realistic distribution of names in the generated document collection.

Cities

Population data about cities was taken from *Mongabay* 2014. A list of the world's cities with a population over 200.000 was generated, including the city's country. For each person chosen from the Wikipedia-backed names list a city was randomly picked from the list of cities. The probability for each city was weigthed by its population, again resulting in a realistic distribution in the document collection.

Street and House Number

Street names were obtained from *Statistik Austria* 2014 in the form of lists of streets from the Austrian cities Graz and Vienna. To account for different lengths of streets, each street was assigned a number of houses. The number of houses was generated using a gamma distribution with parameters $k = 3$ and $\theta = 2.0$, scaled by a factor of 10. For each person a random street was chosen, weighted by the number of houses on the streets. After a street was chosen, a house number within the range of 1 and the number of houses on the chosen street was selected uniformly.

Books

A list of almost 70.000 books was generated using data available at *Internet Book List* 2014. For each book the list contains its title, author and publication year. Each generated person was assigned a random number of k books (uniformly between 0 and 10). For each of the assigned books a document was generated containing both the person's and the book's data (in addition

to the document representing the person alone), giving a total of $k + 1$ documents for each person.

To differentiate between documents for owned books and for people, a facet representing the document type was introduced (with values being either "person" or "ownedBook").

6.2.2 Multiple Facets with Zipf Distributed Words

To complement the first (people and books) document collection with something different we generated another set of documents. This time we started simply from a list of English words (taken from *Mieliestronk's list of English words* 2014). For each of the facets (10, simply named by the letters from a to j) the following steps were executed:

- Shuffle the complete list of words.
- Pick words according to a Zipf distribution.

So each facet has a word distribution that is typical for natural language, but due to the shuffling step the probabilities of the single words are different among the facets.

The parameters for the Zipf distribution were chosen as N the number of words in the word list, and $s = 1.1$.

See table 6.2 for an example document from this collection.

6.3 Results

The following subsections present the evaluation results on various document collections.

To reference the evaluated quality functions concisely each of them was assigned a short descriptive name. Table 6.3 lists these names along with the respective quality function.

Facet	Content
a	remembers resourcefulness malpractice remembers zooms beats refinancing quiff tarriest expatriated
b	gateposts fugue revives goodhumoured shrewdness cloisters sensory blatantly carcinogenesis isles
c	lustier blunting distraction inconceivable undercoat disputed escarpment disputed fitly lustier
d	reciprocating undaunted mediates flippancy satiny capitalisation ascertainable manuals reside stadiums
e	amoral permitting conjurers creationism methylated vexing mildest clod tilts neatness
f	subjectivity assailants enfranchised obsesses assailants traumatic sines capacitors prosthesis innumerate
g	papas protease sloughed darkrooms constituency constituency purities lanky knuckleduster constituency
h	lees optimality swappers justly forecourts hastily fugitive theoreticians veering cox
i	frumps contravene norms coinages is gobbets integrationist ethological toddle typewritten
j	gated dopey obtained inverter flirts gated championing regal dripping inverter

Table 6.2: Example document for our Zipf distributed words collection.

Name	Quality function
writer	No suggester (typing only)
mostCommon	$q^b_D(t) = rdc_D(t)$ (the baseline)
experimental	$q_D(t) = \frac{rdc_D(t)}{rfc_D(t)} r_D(t)$ (see 4.12)
facetAware	$q^f_D(t) = b_D(t) q_D(t)$ (see 4.13)
logDampened	$q^{sqrt}_D = b_D(t) \frac{rdc_D(t)}{rfc_D(t)} r^{sqrt}_D(t)$ (see 4.14)
sqrtDampened	$q^{log}_D = b_D(t) \frac{rdc_D(t)}{rfc_D(t)} r^{log}_D(t)$ (see 4.15)

Table 6.3: Quality function names

Three result tables were generated for each of the evaluated document collections. These three tables are crosstables comparing the evaluated quality functions in terms of "Number of wins", "Amount gained" and "Maximal gain". To explain what the entries in these crosstables mean we look at the entry for a pair of quality functions A and B. Let k be the value for A versus B in the "Number of wins" crosstable. This means that for k evaluated documents the costs (in terms of keystrokes) for the quality function A were lower than the costs for the quality function B. In the "Amount gained" crosstable a value of k is the total sum of keystrokes saved by using the quality function A over B (summing only the costs of evaluated documents where A lead to lower costs than B). Finally a value of k in the "Maximal gain" crosstable means that the highest number of keystrokes saved by using A over B on any evaluated document was k.

Additionally for each evaluated document collection a comparison of the total costs (the sum of the costs for all evaluated documents) was computed.

6.3.1 Small Document Collections

People and Books

The document collection has 50.000 documents, 1.000 randomly picked documents were evaluated. The chosen facet order for evaluation queries was the following: *bookTitle, bookAuthor, firstName, lastName, personType, cityName, cityCountry, cityPopulation, streetName, houseNumber*. Facets not in this list were randomly appended to the ordered query. The results are shown in the tables 6.4 (Number of wins crosstable), 6.5 (Amount gained crosstable) and 6.6 (Maximum amount gained crosstable), a comparison of the total costs by quality function is shown in 6.7.

Zipf Distributed Words

The document collection has 50.000 documents, 1.000 randomly picked documents were evaluated. The chosen facet order for evaluation queries was alphabetically from a to j. The results are shown in the tables 6.8

Number of wins	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	815	x	48	95	38	38
experimental	816	31	x	82	16	16
facetAware	839	410	423	x	17	17
logDampened	843	515	516	272	x	0
sqrtDampened	843	515	516	272	0	x

Table 6.4: Number of wins crosstable (People with books small)

Amount gained	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	11427	x	60	119	49	49
experimental	11402	35	x	98	20	20
facetAware	11946	638	642	x	19	19
logDampened	12259	881	877	332	x	0
sqrtDampened	12259	881	877	332	0	x

Table 6.5: Amount gained crosstable (People with books small)

Max gain	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	64	x	4	4	4	4
experimental	64	2	x	4	4	4
facetAware	66	5	5	x	2	2
logDampened	66	5	5	3	x	0
sqrtDampened	66	5	5	3	0	x

Table 6.6: Maximal gain crosstable (People with books small)

writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
34884	23457	23482	22938	22625	22625

Table 6.7: Total costs (People with books small)

Number of wins	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	969	x	218	129	125	125
experimental	967	105	x	27	21	21
facetAware	977	520	577	x	19	19
logDampened	977	512	567	14	x	0
sqrtDampened	977	512	567	14	0	x

Table 6.8: Number wins crosstable (Zipf distributed words small)

Amount gained	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	7995	x	354	184	179	178
experimental	7767	126	x	31	22	22
facetAware	8781	970	1045	x	22	22
logDampened	8776	959	1031	17	x	0
sqrtDampened	8776	959	1031	17	0	x

Table 6.9: Amount gained crosstable (Zipf distributed words small)

(Number of wins crosstable), 6.9 (Amount gained crosstable) and 6.10 (Maximum amount gained crosstable), a comparison of the total costs by quality function is shown in 6.11.

6.3.2 Large Document Collections

People and Books

The document collection has 1.000.000 documents, 20.000 randomly picked documents were evaluated. The chosen facet order for evaluation queries was the following: *bookTitle*, *bookAuthor*, *firstName*, *lastName*, *personType*, *cityName*, *cityCountry*, *cityPopulation*, *streetName*, *houseNumber*. Facets not

Max gain	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	31	x	4	3	3	3
experimental	30	4	x	2	2	2
facetAware	34	7	7	x	2	2
logDampened	34	7	7	2	x	0
sqrtDampened	34	7	7	2	0	x

Table 6.10: Maximal gain crosstable (Zipf distributed words small)

writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
27369	19374	19602	18588	18593	18593

Table 6.11: Total costs (Zipf distributed words small)

Number of wins	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	19867	x	1999	3660	1057	1057
experimental	19861	1219	x	3081	431	431
facetAware	19855	9563	10057	x	370	370
logDampened	19865	13040	13558	8676	x	0
sqrtDampened	19865	13040	13558	8676	0	x

Table 6.12: Number wins crosstable (People with books large)

in this list were randomly appended to the ordered query. The results are shown in the tables 6.12 (Number of wins crosstable), 6.13 (Amount gained crosstable) and 6.14 (Maximum amount gained crosstable), a comparison of the total costs by quality function is shown in 6.15.

Zipf distributed Words

The document collection has 500.000 documents, 18.800 randomly picked documents were evaluated. The chosen facet order for evaluation queries was alphabetically from *a* to *j*. The results are shown in the tables 6.16 (Number of wins crosstable), 6.17 (Amount gained crosstable) and 6.18 (Maximum amount gained crosstable), a comparison of the total costs by quality function is shown in 6.19.

Amount gained	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	307550	x	2621	4175	1189	1189
experimental	306404	1475	x	3444	468	468
facetAware	317375	14003	14418	x	420	420
logDampened	328018	21661	22086	11064	x	0
sqrtDampened	328018	21661	22086	11064	0	x

Table 6.13: Amount gained crosstable (People with books large)

Max gain	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	71	x	4	4	4	4
experimental	70	6	x	4	3	3
facetAware	72	6	6	x	3	3
logDampened	72	8	8	6	x	0
sqrtDampened	72	8	8	6	0	x

Table 6.14: Maximal gain crosstable (People with books large)

writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
909934	602454	603600	592626	581982	416255

Table 6.15: Total costs (People with books large)

Number of wins	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	18060	x	5204	2583	2473	2473
experimental	18087	2308	x	725	534	534
facetAware	18190	10282	11408	x	370	370
logDampened	18199	10253	11358	496	x	0
sqrtDampened	18199	10253	11358	496	0	x

Table 6.16: Number wins crosstable (Zipf distributed words large)

Amount gained	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	201380	x	8631	3966	3784	3784
experimental	195635	2886	x	909	648	648
facetAware	217193	19779	22467	x	486	486
logDampened	217265	19669	22278	558	x	0
sqrtDampened	217265	19669	22278	558	0	x

Table 6.17: Amount gained crosstable (Zipf distributed words large)

Max gain	writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
writer	x	0	0	0	0	0
mostCommon	44	x	8	7	7	7
experimental	44	5	x	4	4	4
facetAware	46	9	9	x	5	5
logDampened	46	9	9	3	x	0
sqrtDampened	46	9	9	3	0	x

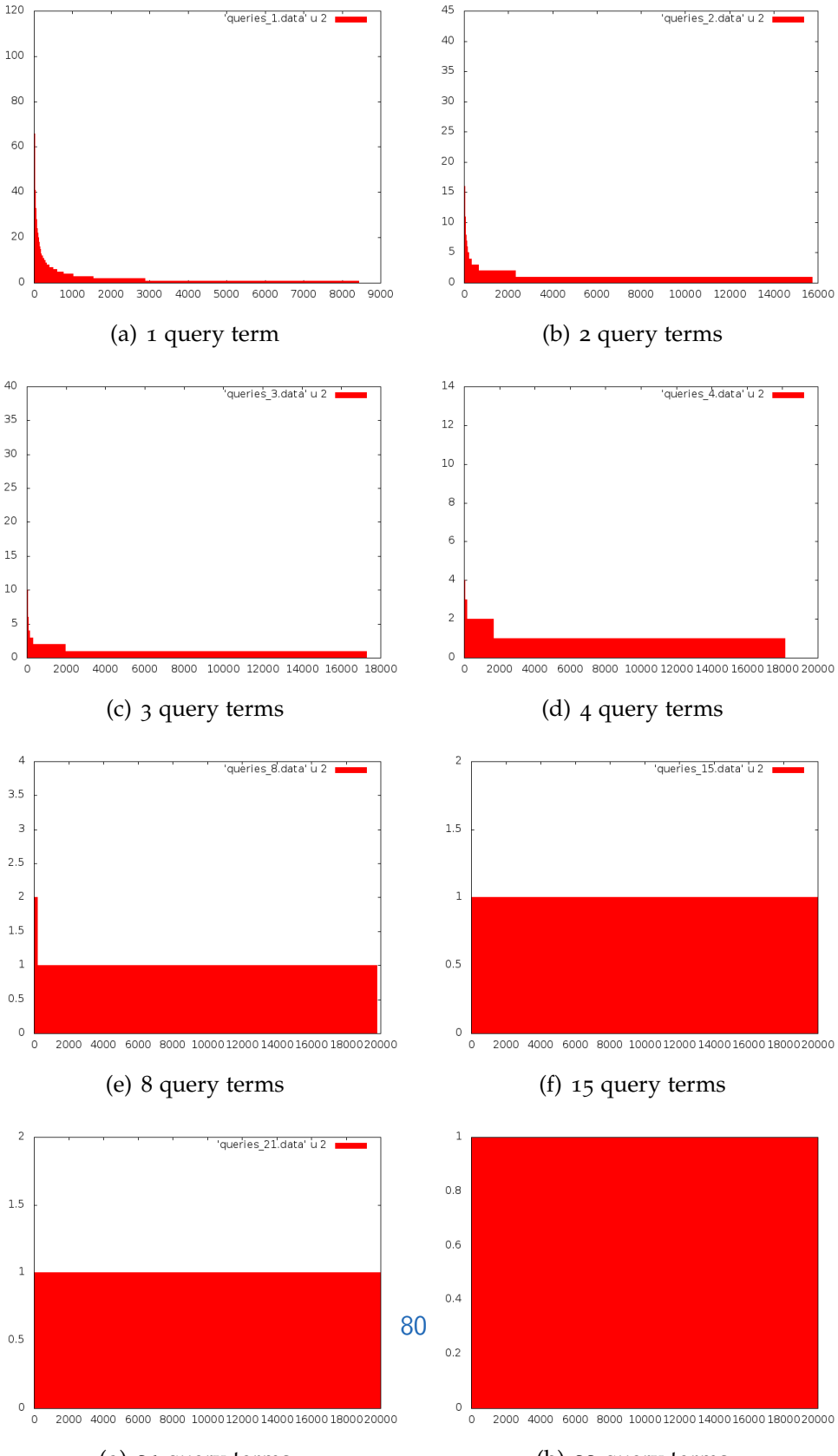
Table 6.18: Maximal gain crosstable (Zipf distributed words large)

writer	mostCommon	experimental	facetAware	logDampened	sqrtDampened
633520	432140	437885	416327	416255	416255

Table 6.19: Total costs (Zipf distributed words large)

6.3.3 Query distributions

The diagrams in figure 6.2 show how the distribution of queries evolves from a single query term to the complete query. The queries are taken from the evaluation with the large "People and Books" index. Note that the bars on the left hand side of the diagrams are sometimes too small to be visible: there always is a bar on the left hand side that extends to the top of the scale. The series of figures nicely illustrates how the distribution transforms from the typical exponential distribution to a uniform distribution (caused by the unique target document presumption).



(a) 1 query term

(b) 2 query terms

(c) 3 query terms

(d) 4 query terms

(e) 8 query terms

(f) 15 query terms

(g) 21 query terms

(h) 22 query terms

Figure 6.2: Evolution of query distribution.

7 Discussion

The last chapter presented evaluation results in plain numbers. In this chapter the meaning of these numbers is discussed.

To make this chapter more readable we will use abbreviations for the evaluated quality functions:

- *writer*: W
- *mostCommon*: MC
- *experimental*: EXP
- *facetAware*: FA
- *logDampened*: LD
- *sqrtDampened*: SD

7.1 Numeric Comparison to the Baseline

7.1.1 Base Quality Function

Starting with the negative: looking at the tables in the previous chapter showing the evaluation results one rather disappointing fact stands out. EXP consistently performs a little worse than the MC, our baseline. In the direct comparison MC has some more wins than EXP: 48 to 31 and 218 to 105 out of 1000 on the small document collections, 1999 to 1219 (out of 20000) and 5204 to 2308 (out of 18800) on the large document collections. So this is a first drawback, however the gap is not too big. Let's reexamine EXP again:

$$q_D(t) = \frac{rdc_D(t)}{rfc_D(t)} r_D(t)$$

Remember that the redundancy factor $r_D(t)$ was designed to explicitly punish redundant terms that occur in all documents matching the current query. So if the user is typing a redundant word then EXP is simply inferior to the basic MC. The goal of EXP is to find non-redundant terms, especially those that reduce the size of the set of matching documents to 1, i.e. it works best on the last term of the query. This has shown not to be sufficient to outperform MC in the evaluation - it needed additional modifications.

7.1.2 Facet Aware Quality Function

Designing the quality function to prefer terms that shrink the set of matching documents proved to be insufficient. Providing more information to the quality function by making it aware of the expected order of the facets (leading from EXP to FA) improved the performance. On all evaluated document collections FA won over MC on more than 40% (on the "Zipf distributed words" collection even more than 50%) of the evaluated documents. It is also noteworthy that the number of wins of MC over FA is almost double the number of wins of MC over EXP on the "people with books" collections - on the Zipf distributed words collections that factor is inversed. This is one of two parts of the evaluation where the different distribution of words in the facets of the two kinds of collections showed quite drastic effects. Comparing FA and EXP directly to each other the situation is almost the same as for FA and MC, only a little clearer in favor of FA.

The gap in performance between the two facet agnostic quality functions (MC and EXP) is drastically smaller than the gaps in performance between both MC and EXP in comparison to the facet aware quality function (FA). This shows that adding the facet awareness has a drastically higher effect on the performance of the quality function than operating solely on the document and facet counts of terms.

7.1.3 Dampened Facet Aware Quality Functions

We evaluated two dampened quality functions: one uses the logarithm (LD), the other one the square root (SD) as its dampening function. As it turns

out both functions have exactly the same effect on the performance of the quality function: the evaluation results of SD and LD are identical.

The effects of applying those dampening functions differs drastically on the two kinds of document collections. On the "people with books" collections it leads to another improvement, on the "Zipf distributed words" collections LD and SD are very slightly worse than FA - both in the direct comparison and in comparison to MC.

On the small "people with books" collection the improvement against the non facet aware quality functions (including the baseline) is around 100 more wins than FA (in total wins on more than 50% of the evaluated documents) and only 38 and 16 losses against MC and EXP, respectively. In the direct comparison to FA the dampened quality functions win on more than 25% of the evaluated documents while losing on less than 2%.

On the large "people with books" collection the dampened quality functions win on more than two thirds of the evaluated documents against the MC and EXP, while losing on about 5% against MC and 2% against EXP. Against FA they win on more than 40% of the evaluated documents while losing on less than 2%. This indicates that the positive effect of applying a dampening function increases with the size of the document collection.

As mentioned above the situation is different on the "Zipf distributed words" collections. There is amost no difference between the evaluation results of the dampened functions and FA. The margins are so small that they are rather meaningless. Therefore we conclude that the effects of dampening depend very much on the characteristics of the document collection. It might change nothing at all or improve performance further (in comparison to FA).

7.2 Observations

Given the discussion of the evaluation results in the previous section one can observe a number of effects of the various modifications to the quality functions. This section attempts to provide reasons for those effects.

7.2.1 Difficulties

Narrow Margins

So far we only discussed one third of the tables from the evaluation chapter. These were the tables that showed the binary outcome of the comparisons of the quality functions: the number of wins and losses. There is a reason for the neglegence of the other tables. Those tables show the amount of keystrokes that were gained by using one quality function over another. For each evaluation there is one table where the gains are summed up and one where the maximal gains are shown. Already the maximal gains are rather small numbers (the largest being 8, in the small "people with books" collection the largest number is just 5), but when the table with the summed gains is combined with the one showing the wins it turns out that the majority of wins is by the smallest possible margin: by 1 keystroke.

So is all the effort practically for nothing? We'd like to argue against that. Even though on a large part of the documents there are simply no differences between various quality functions (on some evaluated documents the computed suggestions provided no improvement over simply writing the query terms, i.e. even W did not lose against any other quality function) there are also documents where the right quality function can make a larger difference. Most of the time the gains will be small, but it is rather obvious that (picking the right quality function) the gains are consistently there. And those gains should certainly be provided to the users. Additionally, as the chapter about existing work showed, research on this topic is just at its beginning and it is likely that future works will further increase the gains.

Hard to Win

In the subsection above the narrow (and sometimes nonexistent) margins were discussed. Reasons for these narrow margins are that the documents often provide only very little chances for a quality function to win keystrokes over another quality function. We briefly discuss two sources for this difficulty: short words and short queries.

First we look at short words. Suppose the query contains the phrase "to have and have not". For a two letter word like "to" term suggestions are simply worthless: it takes at least to keystrokes to select a suggestion - it takes the same amount of keystrokes to type a two letter word. For a three letter word the suggestion must be the top suggestion before the user starts typing - otherwise the cost of typing the whole word beats using suggestions again. To put this more concisely into numbers: given a term of length n, the maximal possible gain with query suggestions is $n - 2$. Since the majority of words used in texts tend to be short (for a reference see for example Miller, Newman, and Friedman, 1958) this issue shows its effects.

The special kind of documents (faceted documents) we studied had quite little content. If the document collection is not very large and the distribution of the words is rather flat then it takes only a few terms in a query to make it unambiguous. And short queries mean that there are only very few opportunities for the quality functions to outperform each other.

7.2.2 The Bad Parts

The quality function designed to zero in on the unique target document quickly did not perform as well as hoped. It was not a meaningful improvement over the baseline. On the contrary, on the evaluated documents it was slightly worse.

Another modification showed mixed results. Smoothing the quality function with a dampening function worked quite well on one kind of document collections, but failed to improve over other quality functions on the second kind of document collections.

7.2.3 The Good Parts

The largest gains were achieved by adding facet awareness to the quality function by boosting terms that matched the expected facet order. The gains achieved by this were consistent over all evaluations. In addition this not only achieved gains, but also reduced the losses to a negligable amount.

Losing one or two keystrokes on only a few percent of the documents in a collection won't even be perceivable to users.

Another step in the right direction was the introduction of dampening functions into the quality function. While leading to improvements on one kind of evaluation document collection the decline in performance on the second kind of evaluation document collection was negligable. In short this means possible gains at a very small risk of losses.

7.3 Summary

This chapter discussed the evaluation results from the previous chapter. It turns out that it is quite hard to save the user more than a few keystrokes when she enters a query: the margins between the various quality functions are very narrow. This can be (at least partially) explained by short terms and short queries: both provide only very few opportunities to save keystrokes.

On the plus side two measures showed great potential: making the quality function aware of the expected order of facets in the query and apply a dampening function to smoothen the quality function. The positive effect of adding facet awareness indicates that the key to better query suggestions is to understand the structure of the document collection as well as how the users' perceive the relationships between the facets. Finally the dampening functions' positive contribution shows that too abrupt leaps in the quality function hurt its performance - intuitively a smoother function is more predictable and leads to more consistent results.

8 Conclusion

Searching for a specific document in a large collection of faceted documents is a challenging task. Supporting the user with query suggestions along the way enables her to complete this task as fast as possible. In this chapter we review what the work of this thesis accomplished in improving query suggestions for rare queries on faceted documents and provide a short outlook on future work.

8.1 Review

To rank new terms for query suggestions a number of quality functions were introduced. These quality functions were designed to exploit knowledge about the nature of both the information need and the document collection. They punished redundant terms that did not decrease the number of documents matching the current query and used information about the facets of the documents.

Evaluation of the quality functions on two kinds of faceted document collections showed substantial gains against a baseline quality function (the baseline quality function simply counts the number of documents that contain occurrences of a term). While the initial attempt to punish redundant terms by implementing a redundancy factor did not lead to the hoped improvements, a large gain in performance was achieved by making the quality function aware of the expected order of facets in the query. Dampening the redundancy factor showed additional potential.

To maximize the usefulness of query suggestions to the user they need to be presented interactively. A custom datastructure was developed to compute query suggestions within the time constraints of an interactive system.

8.2 Outlook

While this thesis made progress in several directions, there are still many aspects open for exploration. In the next paragraphs we will touch a few of them.

The developed quality functions were evaluated in one rather specific way. More extensive evaluation can improve the understanding of these quality functions. So far it was evaluated how query suggestions computed using these quality functions can reduce the number of keystrokes for a predetermined query. Another evaluation may investigate how well the quality functions perform in suggesting new terms or whether the performance varies on the position of the suggested term in the query (early terms versus final terms). Also very interesting would be a user study to also capture the subjective effects of the query suggestions on users.

For the evaluation the order of the facets in the queries was selected arbitrarily. While in some domains this might work well, in others this order might not be known beforehand. In those cases the facet order should be for example learned from query logs.

The developed datastructure for the computation of query suggestions currently does not support efficient updates - it basically has to be reconstructed from scratch when updates to the document collection need to be incorporated in the query suggestions. This might be acceptable for some systems, but others may need a (near) real time view of the document collection. Therefore the datastructure should be adapted to enable efficient updates.

8.3 Summary

Providing query suggestions on faceted documents for rare queries is a problem that has various challenging aspects to it. This thesis showed progress on some of them (e.g. providing suggestions for any query, exploiting the structure of faceted documents or computing suggestions in realtime). Others are still open (e.g. support updates in the datastructure used for

computation of query suggestions) or need more extensive investigation (e.g. user study to get feedback about the subjective effects of query suggestions). The scope of this thesis needed limitation, thus open topics are deferred to future work.

Appendix

Bibliography

8 Design Patterns for Autocomplete Suggestions (2014). URL: http://baymard.com/blog/autocomplete-design (visited on 11/08/2014) (cit. on p. 2).

Baeza-Yates, Ricardo, Berthier Ribeiro-Neto, et al. (2011). *Modern Information Retrieval*. second edition. ACM press New York (cit. on p. 16).

Bast, Holger and Ingmar Weber (2006a). "Type less, find more: fast autocompletion search with a succinct index." In: *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, pp. 364–371 (cit. on p. 21).

Bast, Holger and Ingmar Weber (2006b). "When you're lost for words: Faceted search with autocompletion." In: *SIGIR*. Vol. 6, pp. 31–35 (cit. on pp. 21, 25).

Beeferman, Doug and Adam Berger (2000). "Agglomerative Clustering of a Search Engine Query Log." In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '00. Boston, Massachusetts, USA: ACM, pp. 407–416. ISBN: 1-58113-233-6. DOI: 10.1145/347090.347176. URL: http://doi.acm.org/10.1145/347090.347176 (cit. on p. 23).

Bhatia, Sumit, Debapriyo Majumdar, and Prasenjit Mitra (2011). "Query suggestions in the absence of query logs." In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM, pp. 795–804 (cit. on p. 23).

Blog, VSAT Global Series (2014). *In 2013 the amount of data generated worldwide will reach four zettabytes*. URL: http://vsatglobalseriesblog.wordpress.com/2013/06/21/in-2013-the-amount-of-data-generated-worldwide-will-reach-four-zettabytes/ (visited on 01/17/2014) (cit. on p. 5).

Broccolo, Daniele et al. (2012). "Generating suggestions for queries in the long tail with an inverted index." In: *Information Processing & Management* 48.2, pp. 326–339 (cit. on p. 23).

Broder, Andrei et al. (2009). "Online expansion of rare queries for sponsored search." In: *Proceedings of the 18th international conference on World wide web*. ACM, pp. 511–520 (cit. on p. 20).

Cao, Huanhuan et al. (2008). "Context-aware query suggestion by mining click-through and session data." In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 875–883 (cit. on p. 23).

Carpineto, Claudio and Giovanni Romano (2012). "A survey of automatic query expansion in information retrieval." In: *ACM Computing Surveys (CSUR)* 44.1, p. 1 (cit. on p. 20).

Church, Kenneth and Bo Thiesson (2005). "The wild thing!" In: *Proceedings of the ACL 2005 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, pp. 93–96 (cit. on p. 23).

English, Jennifer et al. (2002). *Flexible search and navigation using faceted metadata*. Tech. rep. Technical report, University of Berkeley, School of Information Management and Systems, 2003. Submitted for publication (cit. on p. 18).

Furnas, George W. et al. (1987). "The vocabulary problem in human-system communication." In: *Communications of the ACM* 30.11, pp. 964–971 (cit. on p. 7).

Google Maps (2014). URL: https://www.google.at/maps (visited on 11/08/2014) (cit. on p. 12).

Hawking, David and Kathy Griffiths (2013). "An enterprise search paradigm based on extended query auto-completion: do we still need search and navigation?" In: *Proceedings of the 18th Australasian Document Computing Symposium*. ACM, pp. 18–25 (cit. on pp. 22, 26).

He, Ben and Iadh Ounis (2007). "Combining fields for query expansion and adaptive query expansion." In: *Information processing & management* 43.5, pp. 1294–1307 (cit. on p. 20).

Hearst, Marti (2008). "Uis for faceted navigation: Recent advances and remaining open problems." In: *HCIR 2008: Proceedings of the Second Workshop on Human-Computer Interaction and Information Retrieval*. Citeseer, pp. 13–17 (cit. on p. 18).

Hearst, Marti et al. (2002). "Finding the flow in web site search." In: *Communications of the ACM* 45.9, pp. 42–49 (cit. on p. 18).

Hsu, Ming-Hung, Ming-Feng Tsai, and Hsin-Hsi Chen (2006). "Query expansion with conceptnet and wordnet: An intrinsic comparison." In: *Information Retrieval Technology*. Springer, pp. 1–13 (cit. on p. 24).

Internet Book List (2014). URL: http://www.iblist.com/ (visited on 11/07/2014) (cit. on p. 71).

Jansen, Bernard J, Danielle L Booth, and Amanda Spink (2008). "Determining the informational, navigational, and transactional intent of Web queries." In: *Information Processing & Management* 44.3, pp. 1251–1266 (cit. on p. 10).

Karp, Richard M (1972). *Reducibility among combinatorial problems*. Springer (cit. on p. 33).

Manning, Christopher D, Prabhakar Raghavan, and Hinrich Schütze (2008). *Introduction to information retrieval*. Vol. 1. Cambridge university press Cambridge (cit. on p. 19).

Mieliestronk's list of English words (2014). URL: http://mieliestronk.com/wordlist.html (visited on 11/07/2014) (cit. on p. 72).

Miller, George A, Edwin B Newman, and Elizabeth A Friedman (1958). "Length-frequency statistics for written English." In: *Information and control* 1.4, pp. 370–389 (cit. on p. 85).

Mongabay (2014). URL: http://www.mongabay.com/igapo/2005_world_city_populations (visited on 11/07/2014) (cit. on p. 71).

Nandi, Arnab and HV Jagadish (2007). "Assisted querying using instant-response interfaces." In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, pp. 1156–1158 (cit. on p. 22).

Porter, Martin F (1980). "An algorithm for suffix stripping." In: *Program: electronic library and information systems* 14.3, pp. 130–137 (cit. on p. 19).

Statistik Austria (2014). URL: http://www.statistik.at (visited on 11/07/2014) (cit. on p. 71).

Wikipedia (2014). URL: http://en.wikipedia.org (visited on 11/07/2014) (cit. on p. 70).

Yee, Ka-Ping et al. (2003). "Faceted metadata for image search and browsing." In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, pp. 401–408 (cit. on pp. 9, 18).