Bachelor Thesis at the Faculty of Engineering of the Albert-Ludwigs-University of Freiburg im Breisgau

Fast Approximate Title Matching

Mirko Brodesser

August 30, 2010

Albert-Ludwigs-University of Freiburg im Breisgau
Faculty of Engineering
Department for Computer Science

Supervisors
Prof. Dr. Hannah Bast
Marjan Celikik

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Bachelorarbeit mit dem Titel

Fast Approximate Title Matching

selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet wurden und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen sind, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Bachelorarbeit nicht, auch nicht auszugsweise, bereits anderweitig verwendet wurde.

_____ _____

Ort, Datum Unterschrift

Contents

Abstract

Given a set of titles and a query, we want to find the title with the largest similarity to the query. The queries may contain spelling mistakes and words which are not part of the correct entry. Furthermore, no word separators in the query are expected and words of the correct title may be missing. Titles are usually short and consist of multiple words and numbers. We analyse several existing similarity measures like Edit distance and Jaccard similarity. Besides the quality, efficiency is the other requirement of our algorithm and therefore we analyse and compare the best existing algorithms for efficient fuzzy match [CGGM03] and efficient duplicate detection [XWLY08]. We propose a new similarity measure as well as algorithm for its efficient computation in order to improve the existing algorithms in terms of quality and efficiency. We show on two different collections that our algorithm is in most of the cases more efficient than the related work, yet achieves better quality.

1 Introduction

We consider the following problem: Given a clean set of titles, which means the words do not contain spelling mistakes and are correctly separated by space characters, we want to find the title which has the largest similarity to the query.

The query in general is erroneous. It may contain typographical errors and the words may not be separated at all. Furthermore there may be missing words, and it may contain additional words which are not part of a correct title (trash).

The usual way to define the similarity between two strings is a similarity function. A similarity function determines how similar these objects are and returns (if normalized) a value between 0 and 1. A high value indicates a high degree of similarity. The choice of a specific similarity function is crucial for the quality. Another challenge is how to compute the distance efficiently against a large dataset of records.

The goal of this thesis is to develop an algorithm which returns the title with the highest similarity to a query in the shortest possible time. We propose a new similarity measure and an algorithm to solve this problem efficiently. We compare our algorithm to one of the best existing approaches, namely the near duplicate detection algorithm from [XWLY08], both in terms of quality and efficiency.

The thesis is structured as follows. In the following section we will give the problem definition. In Section 2 we compare different existing similarity measures. Furthermore we propose two ideas of a new similarity function and then define and discuss our own similarity function. Section 3 is about the related approaches and we analyse two existing algorithms proposed by Chaudhuri et al. [CGGM03] and Xiao et al. [XWLY08]. Our own algorithm which uses our similarity measure is introduced in Section 4. In Section 5 we show the experimental results on two different datasets. In Section 6 we discuss possible improvements; conclusions are drawn in Section 7.

1.1 Problem Definition

We are given a large set of clean records (titles). Clean means that each record contains only the title with correctly separated words. Furthermore the records do not contain any spelling mistakes. In a pre-processing step all non-alphanumerical characters from both, the query and the records are removed.

The query is expected to contain superfluous words and spelling mistakes. Words may be concatenated and words of the correct title may be missing. Furthermore, the order of the non-superfluous words in the query is expected to be correct. Hence,

the query is considered to be a string of characters. The goal is to find the title, given an erroneous version of it, the query.
For example, a query including a typographical error and additional words is “Napoeon Dynamite (2004) DVDRip KVCD by Brady” and should match the record “Napoleon Dynamite (2004)”.

2 Similarity Measures

2.1 Desired Properties

To clarify the desired properties of a similarity function better, we present some example queries which were part of our experiments:

Id	Query	Correct Record
Q1	Der letzte Lude	Andi Ommsen ist der letzte Lude
Q2	inge-casablanca.xvid	Casablanca
Q3	ideocracy	Idiocracy
Q4	vcf-district9-a	District 9
Q5	haco-almostfamous-xvid-1	Almost Famous

The queries *Q1* to *Q5* contain additional words as well as spelling mistakes and missing words. These are typical queries for our problem definition. Based on our experiments we formulate the properties which we expect our similarity measure to have:

1. The absence of valid words in the query should be tolerated, but punished.

2. Additional superfluous words ("trash") in the query should be only slightly punished.

3. Spelling mistakes should be allowed, but punished (i.e. correct match is preferred over fuzzy match).

4. Differences in the order of the matching words from the query and the correct record should be punished.

5. Concatenations in the query should not be punished.

2.2 Overview

There exist several widely used similarity measures in the context of record matching. In the following we will give the definitions of the most relevant measures and then discuss the advantages and disadvantages of each of them with respect to our problem definition.

2.3 Comparing Existing Similarity Measures

Most of the existing algorithms use the same tokenization for the query and the record, we will therefore analyse the existing similarity measures from this point of view. We will take words as well as n-grams into account. The n-grams of a word w are all substrings of w with length n. In general, n-gram tokens are tolerant concerning spelling mistakes, because only parts of the words have to match.
In the following, each of the variables X and Y denotes a set of tokens.

2.3.1 Overlap Similarity

The Overlap similarity $O(X, Y)$ is defined as follows:

$$O(X, Y) = \frac{|X \cap Y|}{min(|X|, |Y|)}$$

The definition of the Overlap similarity implies that the ordering of the tokens is ignored. This goes against property 4 from the desired properties. In the special case when the query is short another problem exists. A short query implies that the query consists only of few tokens. This implies that longer records are more likely to have a high similarity. According to the desired properties we do not prefer longer records in general. In other words, the big problem is that the absence of tokens is not punished. Combined with superfluous words, it is very likely that wrong assignments are made. The following example demonstrates this problem. Assume the query Q is "intowildpart1", then the candidates could be:

Id	Record	Matching tokens
R1	into the wild 2007 2007	{"int", "nto", "wil", "ild"}
R2	into the wild blue 1999 tv 1999	{"int", "nto", "wil", "ild"}
R3	due south 1997 call of the wild part1 2 12 1999	{"wil", "ild", "par", "art", "rt1" }

With the 3-grams as tokens, *R3* would have the highest Overlap similarity, since $min(|Q|, |R1|) = min(|Q|, |R2|) = min(|Q|, |R3|)$ and $|X \cap R3| > |X \cap R1| = |X \cap R2|$. This means $O(Q, R3) > O(Q, R1)$, which leads to a wrong assignment, since *R1* is the correct title.

2.3.2 Jaccard Similarity

The Jaccard similarity $J(X, Y)$ is defined as follows:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

For this similarity measure we assume that each record contains each token only once. By that the definition implies the difference in length is taken into account. In order to have 100% similarity, the query and the record need to have equal length. For our problem definition this is convenient on the one hand because the difference in the length has to be part of the similarity function. Otherwise very long records would have an advantage over short ones to obtain a high similarity value by chance. On the other hand, the queries are expected to contain tokens which are not part of the correct title. In this case, the Jaccard similarity of the query and the correct record can be quite low. However this is not a problem since we expect the additional tokens not to match very well with each of the records. Therefore, the similarity will be quite low for all of the records. For example assuming the query Q is "almost famous trash trash2", candidates could be:

Id	Record
R1	almost famous
R2	trash
R3	the art of trash

Even though $J(Q, R1) = \frac{1}{2}$ is not high, this is not a problem because $J(Q, R2) = \frac{1}{4}$ and $J(Q, R3) = \frac{1}{7}$ are lower.
Another implication of the definition is that the order of the tokens is ignored. In our case, this is improper because we expect the queries to have no ordering errors.

The Jaccard similarity with 3-grams as tokens is a frequently used approach to allow for spelling mistakes. Few spelling mistakes lead to the loss of only few 3-grams. The influence on the similarity is therefore small. Compared to the Jaccard similarity using words as tokens, this seems to be the more robust approach. For example assume the query $Q1$ is "almosz famozs", then the 3-gram Jaccard similarity $J(Q1, R1) = \frac{5}{11}$ is still high enough compared to $J(Q1, R2) = J(Q1, R3) = 0$. The following example illustrates another shortcoming of the Jaccard similarity with 3-grams as tokens. Assume the query $Q2$ is "alzost famzus" then the similarity $J(Q2, R1) < J(Q1, R2)$, even though both queries have exactly one spelling mistake per word. Thus, the position of the spelling mistake has an influence on the similarity, which is not desirable. Furthermore we give experimental evidence in Section 5 to support our claims and we show that Jaccard similarity is indeed worse than our own similarity measure, presented in Section 2.4.

2.3.3 Edit Distance

The Edit distance $ED(x, y)$ is defined between two strings x and y. It is the *minimum number of character edit operations required to transform x into y.* Character edit operations are insertions, deletions and replacements. For example, as illustrated in Figure 2.1, the Edit distance between "famous" and "fame" is three.
One approach to use the Edit distance between a record and a query is the following. First, for both the record and the query all non-alphanumerical characters are

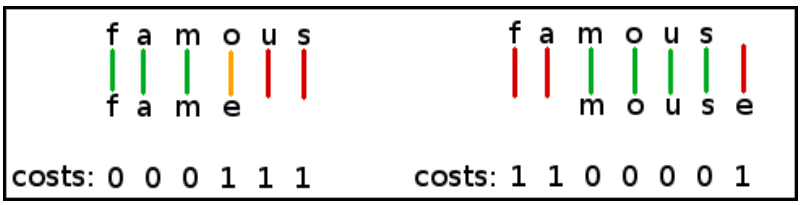

Figure 2.1: Two examples for the Edit distance.

removed. The second step is to calculate the Edit distance. Afterwards, the record with the smallest Edit distance to the query is chosen as the matching record. However, the following example shows one of the problems of this approach concerning our problem definition:

Query (Q):	a	b	c	d	e	f	g	h
Record1 (R1):	i	j	c	d	e	f	k	l
Record2 (R2):	c	d	e	f				

We observe that $ED(Q, R1) = ED(Q, R2) = 4$ but due to our problem definition we do not consider spelling mistakes in short words because they could lead to considerably different words. Furthermore we expect queries to include superfluous words. In this case, "ab" and "gh" would be superfluous words. Taking these facts into account makes clear that the similarity between Q and $R2$ should be higher than between Q and $R1$. Furthermore, the Edit distance between the correct record and the query can become very large, as the following example shows:

Query (Q):	no country foroldmen dvdripbymaster2000
Record1 (R3):	no country for old men
Record2 (R4):	my country

Taking the record with the smallest Edit distance leads to the correct result, since $ED(Q, R3) = 18$ is smaller than $ED(Q, R4) = 29$. However, to calculate all records with a large Edit distance of, for example, 20 is very inefficient. A different application of the Edit distance is to find those words, which approximately match. One way to do this is to calculate the costs to get from the i-th word in the query to the i-th word in the record. This approach is based on the assumption that on the one hand there are no additional words in the query and on the other hand no words in the query are missing. The assumption does not hold true for our problem definition and therefore this approach cannot be applied in our case.
An alternative approach is to allow for each word of the query to match with the closest word in the record. This approach entails with two conflicts concerning our desired properties. First of all, it ignores the order of the words, and secondly, concatenations are not considered. The latter, for example, leads to a problem for the query $Q =$ "thefastandthefurious" and the following records:

Id	Record
R50	the fast and the furious
R51	max and the furious fly

The set of closest words from Q and *R50* would only consist of "furious", equal to the set of closest words from Q and *R51*. Hence, the similarity between Q and $R50$ would be the same like the similarity between Q and $R51$.

2.4 Our Similarity Measure

In the following subsection we discuss different possible approaches to fulfil as many of the desired properties as possible. Afterwards we propose a similarity measure which takes into account most of these properties.

2.4.1 Two Different Approaches

We came up with two new approaches, both with partly different goals which appeared to be hard to combine in one measure. We will illustrate both approaches and their advantages.

The first approach is to sum up all overlapping matches. Gaps in the query as well as in the record are punished. Further, the order of the matches has to be exactly the same in the query and the record. Otherwise we define the similarity as zero. The advantage of this approach is, that superfluous words in the query are not punished and that words missing in the query are also not punished. The following example demonstrates this. Let the query Q be "der letzte lude"; then its candidate set may be the following:

Id	Record
R1	andi ommsen ist der letzte lude
R2	der letzte akt

This implies that the similarity between Q and $R1$ is the length of all matching words, which is $length(der) + length(letzte) + length(lude) = 3 + 6 + 4 = 13$. The similarity between Q and $R2$ is $length(der) + length(letzte) = 9$. Thus, $R1$ is the most similar record to the query Q, which is the correct assignment. The similarity measure also has its drawbacks, as the following example with the query Q "into the wild part 1" shows. Let the candidate set be the following:

Id	Record
R10	due south 1997 call of the wild part 1 2 12 1999
R11	into the wild 2007 2007

The similarity between Q and $R10$ is $length(the) + length(wild) + length(part1) = 3+4+5 = 12$. Compared to the latter, the similarity between Q and $R11$ is smaller. Hence, the assignment based on the largest similarity is wrong.

In the following we introduce another approach with a different advantage. This particular similarity measure is a weighted Jaccard similarity. Tokens are words and each word has a weight. If a word appears correctly in the query, its weight is its

length. If a word matches approximately with the query, its weight is its length minus a punishment value. The punishment depends on the Edit distance of the approximate match to the correct word. The similarity of the query and record is calculated depending on the sum of the weights and the length of the query and the record. With this approach, the similarity between Q and $R11$ is larger than the similarity between Q and $R12$. In the following we will only consider the latter similarity function. In the next subsection, we give a clean definition. Afterwards we comment on how the desired properties are fulfilled.

2.4.2 Weighted Jaccard Similarity

For the following definition assume there are no spelling mistakes in the query. Later on we show how this problem is solved. Further, we assume to know the non overlapping substrings of the query which match with words from the record. For example, for the query Q "almostfamoustrash" we know that "almost" and "famous" match with the words from the record "almost famous". For a set of strings S, we define $W(S)$ as the sum of the weights of all strings in S. We define the weight of a string as its length. With R denoting the set of words of a record, Q denotes the set of substrings from the query which are equal to one of the elements of R. With this notation we can define the weighted Jaccard similarity $WJ(Q, R)$:

$$WJ(Q, R) := \frac{W(Q \cap R)}{W(Q \cup R)}$$

The problem of spelling mistakes is solved in the following manner. When the matching substrings are retrieved, there are also approximately matching substrings considered. For example the query "almoszfamouz" leads to the valid words "almost" and "famous". These approximate matches get a lower weight depending on the Edit distance of the closest substring from the query. The higher the distance, the lower the weight.

We discuss the weighted Jaccard similarity with respect to the desired properties.

1. Missing words are not a problem, as long as there are not too many. The following example demonstrates this. Assume the query Q is "bullets broadway" and its candidate set is the following:

Id	Record
R40	bullets over broadway 1994 1994
R41	broadway 1929 1929
R42	mr broadway 1933 1933

   Then the similarity between Q and $R40$ is:

$$WJ(Q, R40) = \frac{W(\text{“bullets”, “broadway”})}{W(\text{“bullets”, “over”, “broadway”, “1994”, “1994”})}$$

$$= \frac{7+8}{7+4+8+4+4} = \frac{15}{27}$$

In the same way we compute $WJ(Q, R41) = \frac{8}{23}$ and $WJ(Q, R42) = \frac{8}{25}$. $WJ(Q, R40)$ is the largest similarity, which means the missing of "over" has not been a problem for this query.

2. The punishing of additional words in the query is low. This has already been illustrated by the example for the Jaccard similarity. Therefore, this requirement is also fulfilled for the weighted Jaccard similarity.

3. Spelling mistakes are taken care of by allowing approximate matches of the substrings.

4. Since the ordering of the tokens is not taken into account, this requirement is not fulfilled.

5. Since the (approximate) substrings are calculated to get the matching words for a record, concatenated words do not influence the result.

3 Related work

3.1 Record Matching

Several methods for approximate string matching over sets of texts have been proposed (e.g., [XWLY08, CGGM03, LLL08, AGK06, GIJ+01, Nav01, XL08, Ukk83, NBY98]). The general approach is to pre-process the set of records, to build an n-gram index. For each word of a record all possible substrings of length n are indexed. The index holds lists for each substring. Each list consists of those record ids, which include the substring. The candidate set for a given query is generated from those records, which contain n-grams from the query. For efficiency reasons, some existing algorithms (e.g., [XWLY08, CGGM03]) avoid looking at all of these lists to reduce the candidate set. From the candidate set, the record with the largest similarity to the query is calculated. As similarity measures, Edit distance and Jaccard similarity with n-grams are commonly used. In the following subsections we will discuss two of the most relevant algorithms concerning this problem.

3.2 Robust and Efficient Fuzzy Match for Online Data Cleaning

The authors of [CGGM03] address a very similar problem as discussed in this paper. They expect the records to be clean and the queries to be erroneous. The types of errors in the query which are expected on the one hand are spelling mistakes, on the other hand wrong words like “Corporation” instead of “Company”. By assigning weights to the words the algorithm ensures that errors in the less important words do not lead to a wrong result. An example is the query “Boeing Company”, which is, in terms of pure Edit distance, more similar to “Bon Company” than “Boeing Corporation”, even though “Boeing” is the more important word in this case. Because of the weights, which depend on the document frequency, popular words like “Corporation” and “Company” would get lower weights and therefore the query “Boeing Company” would be assigned to the correct record “Boeing Corporation”. However, in our case we do not expect this type of error where a semantically similar word is given in the query instead of the correct word. Therefore we expect not to profit from this. Furthermore, their algorithm is able to deal with different columns, which in our case does not contribute to an improved result because we are dealing with one-column records. Furthermore the authors assume that a non-matching token from the query is an erroneous version of a token in the record. In our case this

holds true only sometimes because we expect our queries to include trash. Trash means words that are not part of the correct title. Another problem with respect to our requirements are concatenations we expect our queries to have. For example, we expect queries like "almostfamous" instead of "Almost Famous". Their algorithm does not take this problem into account, as it is inherently word-based. To achieve its efficiency, the algorithm described in [CGGM03] uses a probabilistic approach to create a candidate-set.

In the following we briefly describe the respective algorithm. The query as well as the records are tokenized to sets of words. Each token gets assigned its inverse document frequency as its weight. The proposed fuzzy match similarity function is defined as $1 - min(tc(u, v)/W(u), 1)$, where $tc(u, v)$ are the costs to transform the query u to the record v and $W(u)$ is the sum of the weights for all tokens from u. The costs of transformation are calculated with the minimum transformation cost function as described in [SW81]. To achieve a high efficiency, the use of an approximate fuzzy match function is proposed to generate the candidate set: Each token from the query is allowed to match its closest token in the record and differences in the ordering are ignored. Disregarding these two characteristics only leads to an overestimation of the similarity and therefore no candidate is lost here. Furthermore, a probabilistically chosen set of n-grams is used to represent each token. This implies that for each query only records with a certain number of shared n-grams have to be identified. The candidate set is created from the shared n-grams, using a pre-processed database which contains for each n-gram, its position, the frequency, its column and the set of corresponding records. Using the database, a candidate set is created with the approximate fuzzy match function. Afterwards, the fuzzy match similarity is used to filter those records from the candidate set which have a similarity greater or equal to the given threshold.

3.3 Efficient Similarity Joins for Near Duplicate Detection

Xiao et al. in their paper [XWLY08] focus mainly on the efficiency aspect. Their proposed algorithm "ppjoin" uses the Jaccard similarity as a similarity measure. It can be extended to the other well-known similarities, i.e., *Overlap similarity*, *Edit distance* and *Cosine similarity*. With respect to our problem definition, Jaccard similarity is expected to be the most promising one. However, it has some shortcomings, which were discussed in Section 2.3.2. The intention behind the authors algorithm was to find the *near duplicates* in a given record-database and therefore it has a similarity threshold. Based on this threshold, different filtering techniques are used to enable finding the duplicates efficiently. The authors make use of positional filtering as well as size-filtering to be able to skip as many of the pairs of the candidates as possible. The effect of these filters depends strongly on the similarity threshold. In our case, the similarity threshold has to be *very* low, to be able to

cope with the expected large amount of "trash" in the queries. In the following we briefly describe the proposed algorithm. In the pre-processing phase the records are tokenized. Tokens can be either words or n-grams. For each record, the tokens are ordered by their increasing document frequency and prefixes of the records are indexed. A prefix of a record are the first p tokens, where p depends on the similarity threshold. The higher the threshold, the smaller the value of p. Thus, less tokens are indexed. To allow positional filtering, the position of each token is indexed as well. The authors propose size filtering, which is used to ignore candidates where the size difference between the records implies that the given similarity threshold cannot be reached. The following example illustrates size filtering. We consider the records u and v and a Jaccard similarity threshold of 0.5. Assume $A \dots D$ are tokens.

$$\begin{aligned} u &= [\mathbf{A}] \\ v &= [\mathbf{A}, \mathbf{B}, C, D] \end{aligned}$$

Taking the difference in length of u and v into account, it becomes obvious that the Jaccard similarity between u and v has to be smaller than 0.5. Hence, this candidate pair can be omitted.

Furthermore, positional filtering is used to ignore candidates where the known overlap and the positions of common tokens imply that the threshold cannot be reached. The following example which was substantially described in [XWLY08], demonstrates positional filtering. Consider the records x and y with the tokens $A \dots F$ and a similarity threshold of 0.8.

$$\begin{aligned} x &= [\mathbf{A}, \mathbf{B}, C, D, E] \\ y &= [\mathbf{B}, \mathbf{C}, D, E, F] \end{aligned}$$

Assume the bold printed tokens are indexed and the similarity measure applied is the Jaccard similarity. Then a similarity threshold of 0.8 implies that x and y have to share at least 5 tokens to meet the similarity threshold. Looking at the positions of the common token B allows to calculate the maximum number of common tokens, as the current overlap plus the minimum number of unseen tokens. Since the tokens are checked from left to right, x has 3 and y has 4 unseen tokens. Hence, the maximum number of common tokens is $1 + min(3, 4) = 4$, which is smaller than 5. This leads to the conclusion that the similarity between x and y is smaller than 0.8. Accordingly, this candidate pair can be omitted.

Additionally, a generalization of the positional filtering to the suffixes of the records is proposed to reduce the candidate set. Notice that the positional filtering as described before cannot be applied to the suffixes, because the suffixes are not indexed and therefore the position of a token in the suffix is not directly available. Notice that due to our problem definition, we modified the algorithm to find the nearest records for a given query.

Taking everything into account, indicates that "ppjoin" can not achieve the desired efficiency and the possible similarities do not exactly match our requirements. We show this in our experiments section, i.e., Section 5.

4 Our Algorithm

In this section we will describe in detail our solution to the problem defined in Section 1.1. The challenge is, again, to develop an efficient algorithm for our weighted Jaccard similarity function, as described in Section 2.4.2. The idea is to consider all valid approximate substrings from the query. Valid approximate substrings are substrings which approximately match with words from the vocabulary. Due to extracting the valid approximate substrings, concatenations are taken care of, because the words are retrieved during the extraction. Notice that all non-alphanumerical characters from the query are removed. We demonstrate this with the following example for the query "thefastandfirious", where the word "the" is missing and one typographical error exists. Its valid approximate substrings could look like this:

Original query:	t	h	e	f	a	s	t	a	n	d	f	i	r	i	o	u	s
Approx. substring:											f	u	r	i	o	u	s
Approx. substring:	t	h	e	f	t												
Approx. substring:						s	t	a	n	d							
Approx. substring:				b	a	s	t	a									
Approx. substring:			f	a	u	s	t										
Approx. substring:				f	a	s	t										
Approx. substring:	t	h	e														
Approx. substring:								a	n	d							
Approx. substring:					a												

Table 4.1: Valid approximate substrings for "thefastandfirious".

In a pre-processing phase, all records are tokenized and their words are indexed. The inverted index maps each word to a inverted list of record ids that contain the word. From the inverted lists of the valid approximate substrings, the candidate set is generated by matching all approximate substrings with the records from the inverted lists. To accomplish this, the inverted lists are merged to a candidate set C and for each candidate the information is stored, with which of the approximate substrings it matches. The algorithm tries to match the valid approximate substrings by decreasing length. The reason is that always the longest substrings of the query should be allowed to match the record. For example, assuming the same query as above, "thefastandfirious, possible candidates could look like this:

Id	Record
R90	the fast and the furious
R91	the theft of the mona lisa

In this example, $R91$ would match with “theft“. The substring ”the” would not be allowed to match the same record, because it overlaps with “theft” in the query. The record $R90$ would first match “furious”, since it is the longest approximate substring, then “fast”, etc.

Matching the substrings by decreasing length implies our algorithm is greedy. In most cases, this leads to the optimal solution, but it does not have to, as the following example shows. Suppose the query is “abcdef” and the following candidate record:

Id	Record
R92	abc abcde def

Our algorithm allows “abcde” to match, because it is the longest matching substring, but letting “abc” and “def” match would result in a higher similarity. However, since this case is expected to be extremely rare, we choose the greedy solution, because it is simple and fast to compute. Afterwards the record with the largest similarity is calculated from the candidate set.

One observation is that the inverted lists of very frequent tokens like “the” are very long. The words corresponding to these long lists are called stopwords. The following example demonstrates the previous observation. Consider the query “the fast and the furious”. Then the sizes of the inverted lists would look as follows:

Word	Size of inverted list
“the”	140941
“and”	76900
“fast”	1375
“furious”	85

The inverted lists of “the” and “and” would lead to a large candidate set, accordingly to a high running time for the calculation of the most similar record.

In the following we discuss two different approaches how to efficiently generate a small candidate set from the valid approximate substrings of the query.

4.1 Approach 1

A first approach to avoid long inverted lists could be to concatenate stopwords in the records with their neighbours. For the following record “will” is considered as a stopword:

Id	Record	Record with concatenated stopwords
R90	there will be blood	there willbe blood

Thus, only the strings “there”, “willbe” and “blood” would be indexed. The effect is that the inverted index will contain more lists, but in general they will be shorter. On the one hand the short inverted lists lead to a small candidate set which allows to efficiently calculate the record with the largest similarity. On the other hand, it affects the true similarity if a stopword is missing in the query. For example, the query “there be blood” would find as matches with the record *R90* only “there” and “blood”, because “willbe” is not an approximate substring of the query. Due to this expected loss of quality we will not consider this approach anymore and propose a different approach.

4.2 Approach 2

Our second approach is as follows: For each record, only those words are indexed, which are not stopwords. If a record only consists of stopwords, we will index them. For a query, we merge the inverted lists of all approximate substrings. The merged lists constitute the candidate set. Since the stopwords will be indexed only in rare cases, we can expect to have a small candidate set. Afterwards, the algorithm tries to match each of the stopwords with each of the candidates. We consider the above example, illustrated in Table 4.1. We assume that “and”, “the” and “a” are marked as stopwords. The inverted lists of *all* approximate substrings (“furious”, “theft”, “stand”, “basta”, “faust”, “fast”, “the”, “and” and “a”) are now merged and constitute the candidate set. The inverted lists of the stopwords also have to be considered, since some few records may contain only stopwords, which in this case were indexed. However, the inverted lists of the stopwords are expected to be very short, since the vast majority of records contains stopwords as well as non-stopwords. Notice that after merging the inverted lists, it is already known which of the non-stopwords match with the candidate records. The candidate set could look like the following:

Candidate Id	**Words of the candidate**
$C1$	*the* **fast** *and the* **furious**
$C2$	*i am* **furious**
$C3$	*go* **fast**

The bold words denote the words which are known to match the candidate record. The words in italic letters are those for which it is unknown, in the current state, whether they match with the query. After this step, for each candidate record the information is available which of the indexed words match with it. The next step is to check for all approximate substrings which are stopwords and all candidate records whether they match. For our example, the result is the following:

Candidate Id	**Words of the candidate**
$C1$	**the fast and** the **furious**
$C2$	i am **furious**
$C3$	go **fast**

Finally all information is available to calculate the record with the largest similarity from the candidate set. This is simply done by iterating over all candidates.

4.3 Our Full Algorithm

We now describe our full algorithm "atMatch" (Approximate Title Match) which uses the above described approach. The pre-processing phase is different from many algorithms of this type. Therefore, we explain it briefly.

Algorithm 1 Preprocess(R, S)

Input: A set of records R, and a set of stopwords S.

```
1: for all record r ∈ R do
2:    T = empty set of strings;
3:    T ← tokenize(r);
4:    N = empty set of strings;
5:    N ← T\S;
6:    if |N| > 0 then
7:       addToIndex(N);
8:    else
9:       addToIndex(T);
10:   end if
11: end for
```

Algorithm 1 is designed to pre-process the record data. At first the tokens, which are all words and numbers, are extracted. If the tokens contain at least one non-stopword, then the non-stopwords are indexed. Otherwise, if the record contains only stopwords, as is rarely the case, all of them are indexed.

Algorithm 2 atMatch(Q)

Input: The normalized querystring Q.
Output: The record with the largest similarity to the query.

```
1: V = empty set of pairs (word, weight);
2: V ← getAllValidApproximateSubstrings(Q);
3: S = empty set of stopwords;
4: I = empty set of indexed words;
5: seperateIndexedWordsAndStopWords(V, I, S);
6: C ← merged candidate-set constituted by I;
7: maxSimilarity ← 0;
8: maxRecord ← NULL;
9: for all c ∈ C do
10:    for all s ∈ S with decreasing length do
11:       if length(s) ≥ minLength(s)) then
12:          fillGap(c, s);
13:       else
14:          break;
15:       end if
16:    end for
17:    if similarity(Q, c) > maxSimilarity then
18:       maxSimilarity = similarity(Q, c);
19:       maxRecord = c;
20:    end if
21: end for
22: return maxRecord;
```

Algorithm 2 describes our "atMatch" algorithm. In lines 1-5, all valid approximate substrings V from the query are retrieved and separated into two sets S and I. I is the subset of V containing all indexed words of V. S contains all stopwords from V. Notice that $S \cup I = V$ but that S and I do not have to be a partition of V, since stopwords can also be indexed. The candidate set is constituted in line 6, by merging the inverted lists. Each candidate contains the information which words are already known to match the associated record.

In lines 9-16 the stopwords of the query are matched with all candidate records. The "fillGap(candidate, stopword)" procedure checks if the given stopword is allowed to match the candidate. Notice that it has to allow a small overlap in the query, since approximate substrings can be longer than the original substring.

Lines 11-14 of Algorithm 2 describe how the matching of the stopwords is improved. Since on the one hand many stopwords like "a" are short, but on the other hand many records do not contain these short words, the algorithm checks if the stopwords length is greater or equal to the length of the smallest word of the candidate. By this, it is avoided to attempt to fill gaps which do not exist in the record.

The following algorithm describes how the valid approximate substrings are retrieved from the querystring.

Algorithm 3 getAllValidApproximateSubstrings(Q)

Input: The normalized querystring Q. The minimum length m of a valid substring s which leads to ignoring all substrings of s.

Output: All valid approximate substrings of Q.

```
1: V = empty set of pairs (word, weight);
2: S ← getAllSubstrings(Q);
3: for all String s ∈ S do
4:   if s is valid word and not substring of a valid substring with length ≥ m then
5:     V ← V ∪ {(s, length(s))};
6:   else
7:     A = validApproximateSubstrings(s);
8:     for all a ∈ A which are not substrings of a valid substring with length ≥
       m do
9:       V ← V ∪ {(a, punishedWeight(a, s))}
10:    end for
11:  end if
12: end for
13: return V;
```

Notice that the valid approximate substrings are efficiently retrieved with the algorithm described in [CB09]. Line 5 and 9 describe how the weights of exact and approximate substrings are calculated. For an exact match, the weight equals the substrings length. An approximate substring a gets a punished weight. Depending on its closest exact substring s, it is calculated as follows:

$$punishedWeight(a, s) = length(s) * (1 - \frac{editdistance(a, s)}{max(length(a), length(s))})$$

Thus, the lower the Edit distance between the original string and the approximate match, the lower the punishment.

Lines 4 and 8 describe that valid substrings of the normalized query of at least length m are considered as correct matches. Therefore, the substrings of such a correct match are not considered any more. Our algorithm uses this technique with a fixed minimum length m of 14. Thus, not many substrings can be omitted. We chose such a high value, because of the following problem. Consider the query "hangoverstar" with the following relevant records:

Id	Record
R100	over and out
R101	hangover
R102	hundred stories of hangovers from the last century

Choosing a minimum length of for example 8 for correct words leads to “hangovers” being recognized as a correct match. Thus, on the one hand “over” is not considered any more which in this case has no influence on the quality. On the other hand, the substring “hangover” is also not considered anymore, which results in not finding the correct record R101.

5 Experiments

In this section, we present our experimental results.

We implemented and used the following algorithms in the experiments:

- **ppjoin:** Is an efficient algorithm for near duplicate detection. It uses a threshold and positional filtering to achieve efficiency. As standard similarity measure the Jaccard similarity is used. The experiments were run with this similarity measure [XWLY08].
- **atMatch:** Is our proposed algorithm. It uses stopwords to achieve short running times. The used similarity measure is our weighted Jaccard similarity which makes use of approximately matching words to deal with spelling mistakes, as described in Section 2.4.2.

5.1 Setup

All algorithms were implemented in C++. To make fair comparisons, all algorithms use Google's `dense_hash_map` for the inverted index. Other algorithms like [LLL08] were not considered, because they use the same similarity measures and therefore no improvement of the precision can be expected.

We performed the experiments on a machine with 2 quad-core 2.8 GHz processors and 32 GB RAM running Ubuntu 9.10. All algorithms were compiled using GCC 4.4 with `-O3` flag. We used two real datasets in the experiments. They were selected to cover a wide spectrum of characteristics.

- **IMDB:** This dataset is a snapshot of the movie titles from the IMDB website. It contains almost 1.5 million records. Each record is a concatenation of the title and its release year. All records were cleaned by removing the non-alphanumerical characters.
- **DBLP:** This dataset is a snapshot of the bibliography records from the DBLP website. It contains almost 1.4 million records. Each record consists of the title of a publication. Similar to the IMDB dataset, the records were cleaned by removing all non-alphanumerical characters.

For the IMDB dataset we chose a random set of 100 existing approximate titles. For the DBLP dataset we chose randomly 100 of the clean titles in the dataset. Afterwards we added different types of errors to them:

1. Typos: For each query, we randomly changed one letter/number per 10 characters.

2. Adding words: For each query, we randomly added one word per 12 characters.

3. Removing words: For each query, we randomly removed one word per 15 characters.

4. Concatenations: For each query, we randomly added a concatenation of two words per 6 characters.

In order to see the effect of each type of error we created all possible fifteen combinations of them.

We compared our algorithm to ppjoin with the following parameter settings: For ppjoin we ran the experiments with 3-grams as tokens and different thresholds. For our algorithm we used a fixed normalized Edit distance threshold of 0.2 for the approximate substrings. For both of the datasets we created a set of input queries, were we assigned the correct record from the dataset to each query. We use the following measures for the evaluation of the experiments:

1. Average elapsed time: The elapsed time to process the set of input queries divided by their number.

2. Correct assignments: The percentage of input queries assigned to the correct record.

5.2 Results

First we present the results of the algorithms on the IMDB and then on the DBLP dataset. Afterwards the results are compared. Notice that for ppjoin, the threshold is appended to its name.

5.2.1 IMDB

We first compare the performance of ppjoin and our algorithm on the IMDB dataset.

Algorithm	Average elapsed time	Correct assignments
PPJOIN-0.05	468.69 ms	69.72 %
PPJOIN-0.1	201.88 ms	69.72 %
PPJOIN-0.2	107.34 ms	64.22 %
PPJOIN-0.3	60.89 ms	54.13 %
PPJOIN-0.4	39.29 ms	44.59 %
PPJOIN-0.5	22.23 ms	24.77 %
PPJOIN-0.6	10.53 ms	8.25 %
PPJOIN-0.7	4.38 ms	2.75 %
PPJOIN-0.8	1.97 ms	0.92 %
PPJOIN-0.9	0.45 ms	0 %
ATMATCH	46.20 ms	78.90 %

Table 5.1: Experimental results for IMDB.

As Table 5.1 shows, for ppjoin the quality depends heavily on the threshold. On the one hand, with decreasing threshold the quality increases, but on the other hand the elapsed time increases. Our algorithm outperforms ppjoin on the IMDB dataset. Independent from the threshold, the quality achieved by our algorithm is better. Compared to the best quality of ppjoin, our algorithm is four times faster at least.

5.2.2 DBLP

We now compare the performance of both algorithms on the DBLP dataset. Since the results of ppjoin with a threshold larger than 0.3 in general do not provide high quality, we omit them.

In the following four tables, *TE* denotes *typo errors*, *AE* denotes *added words errors*, *RE* denotes *removed words error* and *CE* denotes *concatenation errors*. To indicate whether or not an error type is activated, *T* and *F* are abbreviations for "True" and "False".

Algorithm	Avg. time	Corr. assignments	TE	AE	RE	CE
PPJOIN-0.1	3755.53 ms	99 %	T	F	F	F
PPJOIN-0.2	2201.04 ms	99 %	T	F	F	F
PPJOIN-0.3	1191.99 ms	99 %	T	F	F	F
ATMATCH	803.95 ms	100 %	T	F	F	F
PPJOIN-0.1	4838.65 ms	99 %	F	T	F	F
PPJOIN-0.2	2473.52 ms	99 %	F	T	F	F
PPJOIN-0.3	1517.45 ms	99 %	F	T	F	F
ATMATCH	1254.02 ms	100 %	F	T	F	F
PPJOIN-0.1	2729.21 ms	62 %	F	F	T	F
PPJOIN-0.2	1687.99 ms	62 %	F	F	T	F
PPJOIN-0.3	828.71 ms	62 %	F	F	T	F
ATMATCH	358.53 ms	62 %	F	F	T	F
PPJOIN-0.1	4156.40 ms	98 %	F	F	F	T
PPJOIN-0.2	2152.50 ms	98 %	F	F	F	T
PPJOIN-0.3	1297.55 ms	98 %	F	F	F	T
ATMATCH	778.27 ms	100 %	F	F	F	T

Table 5.2: Experimental results for DBLP with exactly one type of error. The error types are exactly the above described ones.

Table 5.2 shows the effect of exactly one type of error applied to the queries. Except for the removed words, the effects are not big and there is no significant difference for ppjoin and our algorithm in terms of quality. Depending on the threshold, our algorithm is up to eight times faster than ppjoin.

Algorithm	Avg. time	Corr. assignments	TE	AE	RE	CE
PPJOIN-0.1	3541.73 ms	99 %	T	T	F	F
PPJOIN-0.2	1758.90 ms	99 %	T	T	F	F
PPJOIN-0.3	1103.48 ms	97 %	T	T	F	F
ATMATCH	1205.17 ms	100 %	T	T	F	F
PPJOIN-0.1	1591.15 ms	44 %	T	F	T	F
PPJOIN-0.2	818.44 ms	44 %	T	F	T	F
PPJOIN-0.3	468.36 ms	39 %	T	F	T	F
ATMATCH	260.37 ms	51 %	T	F	T	F
PPJOIN-0.1	2804.73 ms	98 %	T	F	F	T
PPJOIN-0.2	1418.01 ms	97 %	T	F	F	T
PPJOIN-0.3	785.36 ms	97 %	T	F	F	T
ATMATCH	800.77 ms	100 %	T	F	F	T
PPJOIN-0.1	3030.64 ms	65 %	F	T	T	F
PPJOIN-0.2	1534.79 ms	65 %	F	T	T	F
PPJOIN-0.3	921.01 ms	65 %	F	T	T	F
ATMATCH	505.06 ms	66 %	F	T	T	F
PPJOIN-0.1	4274.09 ms	98 %	F	T	F	T
PPJOIN-0.2	2231.43 ms	98 %	F	T	F	T
PPJOIN-0.3	1285.97 ms	98 %	F	T	F	T
ATMATCH	1316.18 ms	100 %	F	T	F	T
PPJOIN-0.1	2441.97 ms	61 %	F	F	T	T
PPJOIN-0.2	1284.32 ms	61 %	F	F	T	T
PPJOIN-0.3	771.05 ms	61 %	F	F	T	T
ATMATCH	359.66 ms	62 %	F	F	T	T

Table 5.3: Experimental results for DBLP with all combinations of exactly two types of errors. The error types are exactly the above described ones.

Table 5.3 shows the impact of exactly two types of errors on the DBLP testset. Again, there is no significant difference in the quality for ppjoin and our algorithm. In most cases, our algorithm is faster by a factor of three, compared to the best quality results of ppjoin.

Algorithm	Avg. time	Corr. assignments	TE	AE	RE	CE
PPJOIN-0.1	1857.43 ms	48 %	T	T	T	F
PPJOIN-0.2	968.91 ms	48 %	T	T	T	F
PPJOIN-0.3	553.75 ms	36 %	T	T	T	F
ATMATCH	278.51 ms	52 %	T	T	T	F
PPJOIN-0.1	3249.26 ms	98 %	T	T	F	T
PPJOIN-0.2	1623.21 ms	96 %	T	T	F	T
PPJOIN-0.3	1177.87 ms	61 %	T	T	F	T
ATMATCH	1190.63 ms	100 %	T	T	F	T
PPJOIN-0.1	1539.99 ms	45 %	T	F	T	T
PPJOIN-0.2	783.13 ms	45 %	T	F	T	T
PPJOIN-0.3	439.46 ms	34 %	T	F	T	T
ATMATCH	260.24 ms	51 %	T	F	T	T
PPJOIN-0.1	2723.11 ms	64 %	F	T	T	T
PPJOIN-0.2	1405.16 ms	64 %	F	T	T	T
PPJOIN-0.3	824.11 ms	61 %	F	T	T	T
ATMATCH	505.30 ms	66 %	F	T	T	T
PPJOIN-0.1	1533.88 ms	44 %	T	T	T	T
PPJOIN-0.2	782.76 ms	44 %	T	T	T	T
PPJOIN-0.3	594.46 ms	34 %	T	T	T	T
ATMATCH	272.63 ms	52 %	T	T	T	T

Table 5.4: Experimental results for DBLP with all combinations of at least three types of errors. The error types are exactly the above described ones.

For the experiments on DBLP with at least three types of errors, our algorithm achieves a better quality in most of the cases. The main difference is the running time between ppjoin and our algorithm. To achieve a similar quality, our algorithm is always two to three times faster than ppjoin.

Taking everything into account, our algorithm performed better on both datasets in terms of quality. For both algorithms, the running times were much lower on IMDB compared to DBLP. The reason for the big difference in the running times are the lengths of the titles. The IMDB titles are in general noticeably shorter than the DBLP titles. Therefore, the IMDB queries mostly lead to a smaller candidate set, which for both algorithms results in a shorter running time.

6 Possible Improvements

We propose different ways of improving our algorithm, which due to time-constraints were not integrated to our algorithm.

6.1 Considering the Ordering

One way to improve the quality is to take into account the order of the tokens. We give two examples. The first one shows that considering the order has potential to improve the quality. The second example indicates that it is difficult to find a good similarity function which exploits the order.

Assume the query Q is "date movie xvid", the relevant candidates could be the following:

Id	Record
R121	date movie 2006
R122	movie date 2006

With our proposed similarity function (Section 2.4.2), the similarity between Q and $R121$, and between Q and $R122$ would be equal. Allowing no ordering mistakes in this case would lead to the correct assignment of $R121$. For the next example assume the query Q is "no country for men folderhacker". The relevant candidates could be:

Id	Record
R131	no country for old men
R132	no country

Notice that the query is missing the word "old" and contains the superfluous word "folderhacker". Since the superfluous word contains the substring "old", the ordering of the tokens in $R131$ and Q would not be the same. Ignoring the record because of the ordering mistake would lead to the wrong assignment of $R132$.

As the two examples indicated, there is certain potential for improvement, but the crucial point is to *correctly* punish words which are in the wrong order, since ignoring the candidates with only small ordering mistakes can lead to wrong assignments.

6.2 Ignoring Substrings

As explained in Section 4.3, ignoring correct substrings of a certain length can reduce the size of the candidate set, hence improve the efficiency. Since we only rudimen-

tary exploited this, there is still room for improvement. The problem concerning this approach are correct substrings like “hangovers”. Ignoring *all* approximate substrings of “hangovers” would lead for example to ignoring “hang”, “over”, “cover” and “hangover”. Since “hangover” is part of the correct title, it should not be ignored. One approach to solve this problem could be to ignore only substrings of half the length of the original substring. In the case of “hangovers”, only “hangover” would not be ignored. Since *approximate* substrings are retrieved from the query to deal with spelling mistakes, ignoring the *approximate* substrings of sufficiently long correct substrings seems to be another reasonable approach. For the previous example this would lead to ignoring the word “cover”, because it is an approximate substring of the correct substring “hangovers”.

6.3 Popularity

A different way to improve the quality of the results is to take into account the popularity of each record. Especially when there are matches with exactly the same weight, the additional information could be used to determine which record is more likely to be the correct one. For example, the IMDB dataset contains several movies with same title, but different years of production. If a query does not contain the year of production the current algorithm cannot decide which one is the correct title since the similarity to the query would be the same. Choosing the corresponding movie with the higher popularity is a reasonable approach to decide which title is the expected one.

6.4 Custom Stopwords

We compare the number of occurrences of a few popular words from DBLP to their frequency in the IMDB dataset.

Word	#Occurrences in DBLP	#Occurrences in IMDB
analysis	56692	50
data	81458	81
model	36262	613
system	41324	268

Table 6.1: Word frequencies for certain words in DBLP an IMDB.

As shown in Table 6.1, there are several words in the DBLP dataset which have a high frequency, and therefore they are called stopwords. They can not be in general considered as stopwords since their frequency in, for example, the DBLP dataset is very low. We conclude that the stopwords should not only be a fixed set of common words like “the”, “and”, etc., but a custom set, depending on the

dataset. One possible solution for this problem is to calculate the set of stopwords by their frequency. Marking all words as stopwords occurring in at least one percent of the records seem to be a reasonable approach. In general, this could lead to an improvement of the efficiency, since it is expected to lead to smaller candidate sets.

7 Conclusion

The goal of this thesis was to develop an algorithm which efficiently finds the correct title from a given set of titles for an erroneous query. We compared existing algorithms and similarity measures that could be reasonably applied to our problem. Based on the problem definition, we proposed a weighted Jaccard similarity to improve the quality of the results of the existing similarity measures. Additionally, we analysed different approaches on how to avoid large inverted lists to create an efficient algorithm. We proposed a new algorithm, which makes use of approximate substrings to calculate the weighted Jaccard similarity, efficiently executed by avoiding to index stopwords.

By using the IMDB and DBLP datasets we showed the quality and efficiency of our algorithm compared to the existing algorithm ppjoin. The results indicated that our algorithm achieves a better quality and is on average two to five times faster. Furthermore we discussed different potential ways of further improving our algorithm, both in terms of quality and efficiency.

Bibliography

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[CB09] Marjan Celikik and Holger Bast. Fast error-tolerant search on very large texts. In *SAC*, pages 1724–1731, 2009.

[CGGM03] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 313–324, New York, NY, USA, 2003. ACM.

[GIJ+01] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[LLL08] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 257 –266, 7-12 2008.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1): 31–88, 2001.

[NBY98] Gonzalo Navarro and Ricardo A. Baeza-Yates. A practical q -gram index for text retrieval allowing errors. *CLEI Electron. J.*, 1(2), 1998.

[SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1): 195–197, March 1981, http://view.ncbi.nlm.nih.gov/pubmed/7265238.

[Ukk83] Esko Ukkonen. On approximate string matching. In *FCT*, pages 487–495, 1983.

[XL08] Chuan Xiao, Wei Wang 0011, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1): 933–944, 2008.

[XWLY08] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW '08: Proceeding*

of the 17th international conference on World Wide Web, pages 131–140, New York, NY, USA, 2008. ACM.