Hasso-Plattner-Institut für Softwaresystemtechnik
Fachgebiet Informationssysteme

Effective and Efficient Similarity Search in Databases

Dissertation
zur Erlangung des akademischen Grades
„Doktor-Ingenieur“
(Dr.-Ing.)
in der Wissenschaftsdisziplin „Informationssysteme“

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dustin Lange, M.Sc.

Postdam, den 31.01.2013

Abstract

Given a large set of records in a database and a query record, similarity search aims to find all records sufficiently similar to the query record. To solve this problem, two main aspects need to be considered: First, to perform effective search, the set of relevant records is defined using a similarity measure. Second, an efficient access method is to be found that performs only few database accesses and comparisons using the similarity measure. This thesis solves both aspects with an emphasis on the latter.

In the first part of this thesis, a *frequency-aware similarity measure* is introduced. Compared record pairs are partitioned according to frequencies of attribute values. For each partition, a different similarity measure is created: machine learning techniques combine a set of base similarity measures into an overall similarity measure. After that, a similarity index for string attributes is proposed, the *State Set Index* (SSI), which is based on a trie (prefix tree) that is interpreted as a nondeterministic finite automaton. For processing range queries, the notion of *query plans* is introduced in this thesis to describe which similarity indexes to access and which thresholds to apply. The query result should be as complete as possible under some cost threshold. Two query planning variants are introduced: (1) *Static planning* selects a plan at compile time that is used for all queries. (2) *Query-specific planning* selects a different plan for each query. For answering top-k queries, the *Bulk Sorted Access Algorithm* (BSA) is introduced, which retrieves large chunks of records from the similarity indexes using fixed thresholds, and which focuses its efforts on records that are ranked high in more than one attribute and thus promising candidates.

The described components form a complete similarity search system. Based on prototypical implementations, this thesis shows comparative evaluation results for all proposed approaches on different real-world data sets, one of which is a large person data set from a German credit rating agency.

Zusammenfassung

Ziel von Ähnlichkeitssuche ist es, in einer Menge von Tupeln in einer Datenbank zu einem gegebenen Anfragetupel all diejenigen Tupel zu finden, die ausreichend ähnlich zum Anfragetupel sind. Um dieses Problem zu lösen, müssen zwei zentrale Aspekte betrachtet werden: Erstens, um eine effektive Suche durchzuführen, muss die Menge der relevanten Tupel mithilfe eines Ähnlichkeitsmaßes definiert werden. Zweitens muss eine effiziente Zugriffsmethode gefunden werden, die nur wenige Datenbankzugriffe und Vergleiche mithilfe des Ähnlichkeitsmaßes durchführt. Diese Arbeit beschäftigt sich mit beiden Aspekten und legt den Fokus auf Effizienz.

Im ersten Teil dieser Arbeit wird ein *häufigkeitsbasiertes Ähnlichkeitsmaß* eingeführt. Verglichene Tupelpaare werden entsprechend der Häufigkeiten ihrer Attributwerte partitioniert. Für jede Partition wird ein unterschiedliches Ähnlichkeitsmaß erstellt: Mithilfe von Verfahren des Maschinellen Lernens werden Basisähnlichkeitsmaßes zu einem Gesamtähnlichkeitsmaß verbunden. Danach wird ein Ähnlichkeitsindex für String-Attribute vorgeschlagen, der *State Set Index* (SSI), welcher auf einem Trie (Präfixbaum) basiert, der als nichtdeterministischer endlicher Automat interpretiert wird. Zur Verarbeitung von Bereichsanfragen wird in dieser Arbeit die Notation der *Anfragepläne* eingeführt, um zu beschreiben welche Ähnlichkeitsindexe angefragt und welche Schwellwerte dabei verwendet werden sollen. Das Anfrageergebnis sollte dabei so vollständig wie möglich sein und die Kosten sollten einen gegebenen Schwellwert nicht überschreiten. Es werden zwei Verfahren zur Anfrageplanung vorgeschlagen: (1) Beim *statischen Planen* wird zur Übersetzungszeit ein Plan ausgewählt, der dann für alle Anfragen verwendet wird. (2) Beim *anfragespezifischen Planen* wird für jede Anfrage ein unterschiedlicher Plan ausgewählt. Zur Beantwortung von Top-k-Anfragen stellt diese Arbeit den *Bulk Sorted Access-Algorithmus* (BSA) vor, der große Mengen von Tupeln mithilfe fixer Schwellwerte von den Ähnlichkeitsindexen abfragt und der Tupel bevorzugt, die hohe Ähnlichkeitswerte in mehr als einem Attribut haben und damit vielversprechende Kandidaten sind.

Die vorgestellten Komponenten bilden ein vollständiges Ähnlichkeitssuchsystem. Basierend auf einer prototypischen Implementierung zeigt diese Arbeit vergleichende Evaluierungsergebnisse für alle vorgestellten Ansätze auf verschiedenen Realwelt-Datensätzen; einer davon ist ein großer Personendatensatz einer deutschen Wirtschaftsauskunftei.

Acknowledgments

I am very thankful to my advisor Felix Naumann, who gave me the opportunity to work on a motivating and challenging research project. With his inspirational advice and encouragement, Felix supported me during all the phases of research.

Working at HPI was always a pleasure. With my roommate Johannes and almost-roommates Tobias and Ziawasch, I found real friends and had a great time at work. I will also remember Jana, Uwe, Arvid, Christoph, Jens B., Gjergji, Alex, Toni, Zuo, and Max as thoughtful friends and colleagues. As co-authors of one of my papers, I thank Dandy, Astrid, and Ulf for their commitment.

I thank Schufa Holding AG for supporting this work. In particular, I thank Stephan Springob, Boris Zerban, Michael Stolz, and Tim Hoffmann for their valuable input.

Throughout my entire studies (and also the life before and beyond that), my family encouraged and supported me wherever they could. I thank my parents Ilona and Rüdiger, my brother Marc, and also my family-in-law Maria, Achim, Sarah, and Sven for their support. For the extensive support of a particular person I am especially grateful: On every day, my husband Jens gives me a feeling of warmth and comfort.

Dustin Lange

Contents

1 Introduction

With ever-growing amounts of data and the ability and desire to integrate and query more and more databases, there is a need for efficient processing of this data. Traditional relational database systems are built for fast retrieval of data from a large corpus. With SQL and efficient index structures, such as the B^+-tree, retrieval of records with exact matches in their attribute values from even very large databases can be implemented with little effort. However, a query may also be inaccurate, as it may contain typing errors or missing values, and also a database record may contain incorrect or incomplete information. In this case, an index that only finds exact matches cannot be used. A traditional database system neither offers the possibility to define what is a similar record, nor does it perform a fast retrieval of those records.

The field of research that solves this problem is called similarity search: Given a set of records in a database and a query record, *similarity search* aims to find all records in the database that are sufficiently similar to the query record.

Similarity search is an important technology for many applications. In the following, we introduce several application scenarios:

– **Person Database:** Consider a person data set that typically contains information such as the name, the birth date, and the address of individuals that are somehow related to an organization. Often, queries against this data set have to be answered extremely fast, e. g., to process online orders or to support call centers. In many cases queries may contain information that differs from the information stored in the data set. For instance, there may be typing errors, outdated values, or sloppy data or query entries.

  In Table 1.1, we show an example for an excerpt from a person table with a query. Note that the query is structured exactly as the records from the database. In this example, our desired search result contains the records 1, 3, and 4, because all these records have attributes values that are similar to those from the query record. However, each record contains at least one attribute value differing from the query, e. g., record 3 contains an additional middle name "F.", and record 4 has a different birth month. Thus, an exact search over all records cannot find the correct entries. In addition, considering the case that the shown records are part of a database with millions of records, the need for a novel search application for finding relevant records fast is evident.

  Parts of this thesis were created during a research project with Schufa Holding AG, a German credit rating agency. Schufa maintains a large database with information about the creditworthiness of most adult German persons. The information is provided and queried by partner companies, such as banks, telephone companies, or online shops. For Schufa, it is important to select the correct entry from its database and return the correct

	Name	**Address**	**BirthDate**
Record 1	John Miller	48 5 Av, New York, NY 10014	05-Jun-68
Record 2	Peter Smith	27 Main St, Sacramento, CA 95811	30-Jul-47
Record 3	John F. Miller	43 5 Av, New York, NY 10041	15-Jun-67
Record 4	John Milller	42 6 Av, New York, NY 10041	05-Jul-68
Record 5	David White	123 Lake St, Chicago, IL 60601	03-May-63
Record 6	Richard Harris	20 Main St, Scramento, CA 95811	30-Jul-48
⋮	⋮	⋮	⋮
Query	John Miller	42 5 Av, New York, NY 10041	05-Jun-68

Table 1.1: Example data for a person table with example query

information about a person for each individual query despite any differing attribute values in query and database records. We refer to this use case throughout the thesis as our running example and show comparative evaluation results with the according data set with 66 million records for most presented algorithms, among other real-world data sets.

– **Product Database:** Another domain that often contains erroneous entries or queries is product data. For example, consider a database of an online shop for notebooks, where each notebook model may have configurations (RAM size, display size, etc.). A query to the shop product database should handle any abbreviations or alternative names of the model, typos of the manufacturer, or differing measures for the configuration options; the query result should display any corresponding product entries including configuration variants.

– **Image Database:** When a large image database is available, a user may want to find images that are similar given a query image. Depending on the use case, image similarity can be defined in different dimensions. First, the colors of two images can be compared using a color histogram that counts the numbers of pixels in the images with specific color values. Two images thus might have a similar impression although they show different motives. Two images can also be judged similar if they show the same motive. For example, an image of the Eiffel tower at day and another image of the illuminated tower at night do show the same motive although they will not share many similar colors or pixels. When meta-information is available, it may also be possible to compare any labels that have been assigned to the images. Images with similar labels might be related to each other and thus relevant to a user, even if they do not share any motives or colors.

To build a similarity search system, two main aspects need to be considered, namely effectiveness and efficiency. First, to perform *effective search*, the set of relevant records is to be defined. In contrast to an exact query, i. e., all records with the exact attribute values specified in the query are to be found, a similarity query is usually specified with a similarity measure, i. e., a function that compares two records (in particular the query record and a record from the database) and assigns a value for their similarity. The main types of queries are range queries (find all records with a similarity above a threshold) and top-k queries (find the k records with the highest similarity). To perform *efficient search* given a similarity measure, an access method is to be found that performs only few database accesses and comparisons using the similarity measure.

The two goals of the search system are contradictory: To find all similar records, a detailed comparison with many records is required, resulting in a large query time. On the other hand, a very fast search may miss relevant records. In this thesis, we resolve this trade-off as follows: We follow a user perspective with a limited amount of query time. Thus, the goal of the search system is to be as effective (find as many relevant records) as possible while observing a cost limit (perform only up to a limited amount of database retrievals and comparisons). We consider both range and top-k queries in this thesis.

Similarity search systems are related to database systems, search engines, and duplicate detection systems. Database systems are built for efficient storage and retrieval of data [Garcia-Molina et al., 2009]. They are optimized for answering exact point or range queries with suitable index structures, such as B^+-trees [Bayer and McCreight, 1970, Comer, 1979], but usually only offer only very limited support for similarity queries. Similarly, the goal of search engines is to retrieve and rank all documents relevant to a search query, which in most cases consists of a (usually small) set of keywords [Manning et al., 2008]. Inverted indexes allow efficient access to all documents containing a specific term [Zobel and Moffat, 2006]. Ranking the result set of documents for a keyword query is a task comparable to assigning similarity scores to the result set of database records for a similarity search query. However, search engines typically do not support similarity queries as defined in this thesis, they rather suggest alternative keywords if they previously encountered a query with similarly spelled, but more frequent keywords [Whitelaw et al., 2009]. Another related area is duplicate detection, which is the problem of finding all sets of records in the database that refer to the same real-world entity [Naumann and Herschel, 2010]. Similar to similarity search, duplicate detection aims to find records that are similar. While duplicate detection is usually a batch job, aiming to find all duplicates in the database at once, similarity search separately answers queries by finding only those records that are similar to a given query. Consequently, similarity search applications need to adhere to much stricter time constraints.

In the past decades, many researchers have addressed the field of similarity search. First, if similarity search is to be performed within vector data, there exist many appropriate similarity measures and index structures [Böhm et al., 2001]. Another popular group of approaches addresses the metric space [Chávez et al., 2001, Zezula et al., 2006], which requires the similarity measure to fulfill the metric requirements, in particular the triangular inequality, which is often a handicap for modeling similarity measures [Skopal and Bustos, 2011]. For similarity search on a set of strings, several algorithms for specific similarity measures, such as the edit-distance, have been proposed [Navarro, 2001]. We give more detailed overviews on similarity measures in Section 3.1 and on similarity indexes in Section 4.1.

This thesis is guided by the experience with real-world similarity search problems, in particular with the person data set from Schufa. On the one hand, the data set inspired us to improve existing methods with observations from the data. For example, by regarding frequencies of values, we developed a method [Lange and Naumann, 2011b] that improves learnable similarity measures [Bilenko and Mooney, 2003] (see Chapter 3). On the other hand, we observed that existing methods are not capable of performing an efficient *and* effective search as described above. Fast methods are limited to specific groups of similarity measures, such as vector-based or metric measures. Because we defined several attribute-specific similarity measures by hand and have a complex overall similarity measure, a novel retrieval method was required to allow fast query answering [Lange and Naumann, 2011a, Lange and Naumann, 2013] (see Chapter 5). Instead of building a large index for the complex overall measure, we favor a solution that creates several attribute-specific and thus lightweight similarity indexes (see Chapter 4) and then combines their results according to the overall similarity measure.

Contributions

With this thesis we make the following contributions:

- **Frequency-aware Similarity Measure:** We introduce a novel approach for similarity measurement that exploits frequencies of values. We partition data according to value frequencies and learn one composed similarity measure for each partition. This approach can be applied to any similarity measure that combines several base similarity measures or other learnable similarity measures. We compare different partitioning strategies: a greedy partitioning algorithm, equi-depth partitioning, random partitioning, as well as a novel partitioning algorithm inspired by genetic programming. Evaluation on two real-world data sets shows that frequency-partitioning gives better results than frequency-oblivious methods, and that genetic partitioning achieves best results among the compared partitioning strategies. Chapter 3 describes our approach to frequency-aware similarity measures in detail.

- **State Set Index for String Similarity Search:** We present a novel index structure State Set Index (SSI) for fast retrieval of similar strings based on the edit-distance. SSI is based on a trie that is interpreted as a nondeterministic finite automaton and includes a novel state labeling approach, where only information on the existence of states is stored. Different from previous approaches, state transitions do not need to be stored and can be calculated on-the-fly. Using this highly space-efficient labeling strategy, SSI is capable of indexing very large string sets with low memory consumption on commodity hardware. SSI allows a graceful trade-off between index size and search performance by parameter adjustment. These parameters, namely labeling alphabet size and index length, determine the trade-off between index size and query runtime. Our evaluation reveals that SSI outperforms other state-of-the-art approaches in the majority of cases in terms of index size and query response time. In particular, on a data set with more than 170 million strings and a distance threshold of 1, SSI outperforms all other methods we compared to. SSI is described in Chapter 4.

- **Query Planning for Range Query Processing:** For fast retrieval, we create several lightweight similarity indexes for the different attributes of our data. To describe which similarity indexes to use with which thresholds for query answering, we introduce the notion of query plans for similarity search. Result completeness and execution cost are used as performance metrics to evaluate query plans for accessing similarity indexes. We distinguish two variants of query planning: Static planning selects one query plan at compile time that is used for all queries. Query-specific planning selects an individual plan for each query. For both planning variants, exact and approximative algorithms for optimization of query plans based on the specified metrics are proposed. We evaluate our approach on the Schufa data set and show that our algorithms select plans achieving almost always complete results, even if only few comparisons are allowed. Our query planning algorithms are topic of Chapter 5.

- **Bulk Sorted Access for Top-k Query Processing:** For processing top-k queries, we present the Bulk Sorted Access Algorithm (BSA), which is based on the well-known Threshold Algorithm (TA) [Fagin et al., 2001]. BSA retrieves several bulks of IDs of records with similar attribute values. To save unnecessary comparisons, BSA prefers records with several high attribute similarity values. We analytically and experimentally compare BSA with TA on two large real-world data sets (10 million and 2.2 million records) and show that our method outperforms TA in most cases. All details on BSA can be found in Chapter 6.

Outline

This thesis is structured as follows. We begin with an overview of our similarity search system in Chapter 2 before describing the components of the system in detail in the following chapters. Chapter 3 introduces the similarity model used throughout the thesis. We also propose the novel similarity measure for comparing database records that exploits frequencies of values. Chapter 4 contains an introduction to similarity indexes for fast retrieval of similar values given specific similarity measures. We present an index structure for string similarity search, the State Set Index (SSI), and compare the method with previous index structures. For subsequent chapters, we assume that we have created one similarity index for each attribute, and that we have an overall similarity measure composed of attribute-specific measures. In Chapter 5, we then introduce query plans as a means of describing how to access the similarity indexes and how to combine the results. We describe static and query-specific algorithms for selecting query plans based on the criteria result completeness and execution cost. Chapter 6 adds the BSA method for answering top-k queries with similarity indexes by retrieving bulks of IDs of relevant records and combining results into a priority queue. For Chapters 3 to 6, related work is described at the end of each chapter. We conclude the thesis and give an overview on open research questions for future work in Chapter 7.

Most parts of this thesis have previously appeared in the following conference and journal publications:

- Chapter 3:

  Lange, D. and Naumann, F. (2011b). Frequency-aware similarity measures. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 243–248

- Chapter 4:

  Fenz, D., Lange, D., Rheinländer, A., Naumann, F., and Leser, U. (2012). Efficient similarity search in very large string sets. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 262–279

  The contents of this conference paper are based on the Master's thesis by Dandy Fenz [Fenz, 2011], supervised by the author of this thesis.

- Chapter 5:

  Lange, D. and Naumann, F. (2011a). Efficient similarity search: Arbitrary similarity measures, arbitrary composition. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1679–1688

  Lange, D. and Naumann, F. (2013). Cost-aware query planning for similarity search. *Information Systems (IS)*, 38(4):455–469

2
Similarity Search System

This thesis proposes several approaches to sub-problems of similarity search. These approaches can be used individually as fragments of similarity search applications. Because the components were designed to interact with each other, we describe in this chapter how they work together in the complete similarity search system that we implemented.

Architecture

We first give an overview on the architecture of our similarity search system. Figure 2.1 visualizes the involved components. In the following, we describe the components and their interactions in greater detail by explaining the processing of a query.

A *query* contains query values for some or all attributes. Each query is either a *range query* (for all records with a minimum overall similarity) or a *top-k query* (for the k most similar records). In both cases, a cost limit can be specified that restricts the number of comparisons to be executed. The following components are responsible for query answering:

– **Query planning:** The query is received by the query planning component. The task of this component is to create a plan that describes which base similarity measures (and thus corresponding similarity indexes) and thresholds are to be used for answering the query.

  In traditional database systems, query processing works as follows [Garcia-Molina et al., 2009]: The query is written in SQL and exactly describes the result. The query is parsed and transformed into a logical query plan, which is an expression tree in relational algebra describing which operators are required to execute the query. After that, the most efficient implementation of each operator and the execution order is determined, which is described in a physical query plan. In both query optimization steps, all alternative operators, arrangements of operators, and implementations are equivalent regarding the desired query result, so that only the most efficient plan is to be selected.

  In contrast to traditional query processing, our approach considers query plans as approximations of the expected query result. Because we consider the used overall similarity measure (which can be arbitrarily chosen) a black box and only combine the available base similarity measures in the query plan, we need to estimate result completeness, for which we use training data. In the sense of traditional planning methods our plan is a logical query plan, because it describes which similarity measures are to be combined and which thresholds to apply. How to perform optimized physical planning based on our logical plans is not covered in this thesis.

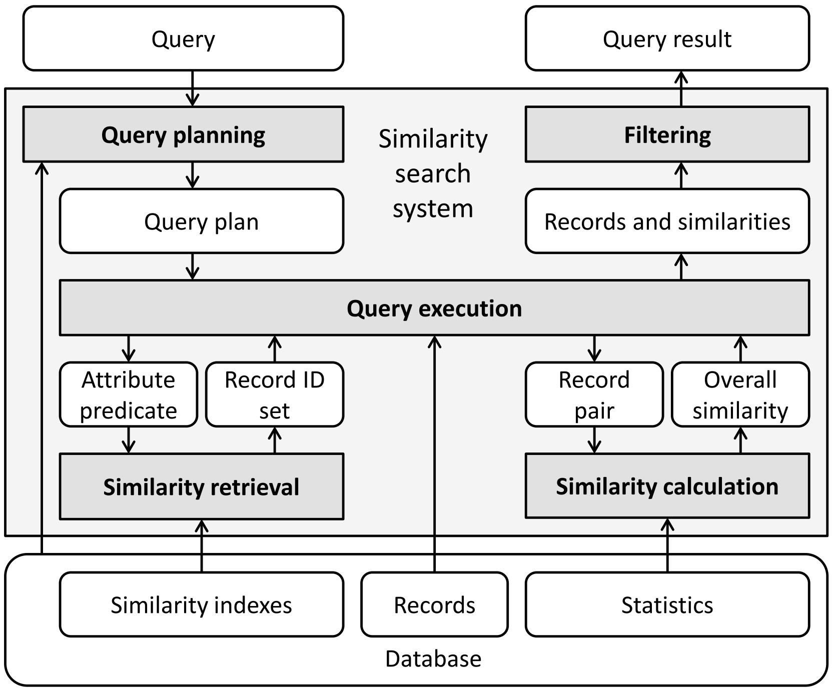


Figure 2.1: Architecture of the similarity search system

For *range query* processing (Chapter 5), we introduce two planning approaches: The first is to select a *static plan*, which is created in advance and applied for all queries. In this case, the query planner has no other function than providing the static plan. A second approach is *query-specific planning*: An individual plan is selected for each query based on available information about the query values. For performance reasons, first the template is determined, and then the thresholds are optimized. In both cases, the two main criteria to determine the best plan are result completeness, estimated by evaluating the plan performance on a set of training queries, and query cost, where the estimation is performed by retrieving and combining frequencies of query values from the database.

For processing *top-k queries* (Chapter 6), our proposed approach performs a bulk sorted access to retrieve many similarities for all attributes. This approach can be implemented as a static query plan that is a disjunction of all attributes with previously determined retrieval thresholds.

– **Query execution:** The created query plan is executed by retrieving IDs of relevant records and calculating the similarity of these records with the query record. First, all predicates of the plan are individually sent to the similarity retrieval component to determine the set of record IDs that fulfill the predicates. Then, the sets are combined according to the logical composition specified in the query plan.

  In case of *range queries*, all records with IDs in the resulting set are retrieved from the database. For each such record, the overall similarity is calculated, and the record and its similarity to the query record are sent to the filter.

  A *top-k query* is executed by first building a priority queue from the retrieved attribute similarities, where the priority is the expected overall similarity. The record IDs are then processed in decreasing priority. A record is retrieved from the database and then its overall similarity to the query record is determined; the result is sent to the filter. With the used retrieval thresholds and information about the already processed records, we can determine when the priority queue cannot contain any relevant records and query execution can be stopped.

- **Similarity retrieval:** As part of a query plan, an attribute predicate specifies an attribute and a minimum similarity threshold. We use indexes to speed up similarity retrieval at query time.

  The choice of indexes depends on the given retrieval task. For retrieving all exact matches in a database, B^+-trees or hash tables are usually used [Bayer and McCreight, 1970, Peterson, 1957]. For answering keyword queries given a document corpus, search engines use inverted indexes [Manning et al., 2008], which store pointers of terms to the documents they occur in. To allow fast scanning of inverted indexes, B^+-trees are created on the set of indexed terms.

  In our similarity search system, we use *similarity indexes* (Chapter 4) to retrieve the set of IDs of all records that fulfill an attribute predicate in a query plan, i. e., that have an attribute value with a similarity to the query value above a given threshold. A similarity index is created for a specific similarity measure and exploits its characteristics. Various similarity indexes have been proposed for specific groups of similarity measures, such as measures for the vector space or metric space. We later describe a similarity index for the edit distance on string attributes.

- **Similarity calculation:** We use a *similarity measure* (Chapter 3) to determine the overall similarity of two records, in particular of the query record and a record from the database. In our setting, we use similarity measures that are composed of attribute-specific measures.

  Our similarity measure accesses the database statistics to determine the frequencies of the attribute values of the compared records. According to the determined frequencies, the record pair is assigned a frequency partition, such as *frequent values* or *rare values.* We then use the measure that we created with machine learning techniques for the selected partition to determine the overall similarity of the compared records.

- **Filter:** The filter processes the retrieved records and overall similarity values according to the query settings.

  For a *range query*, the filter simply evaluates whether the overall similarity fulfills the specified query range and, in case of positive evaluation, immediately passes the record to the query result.

  For answering a *top-k query*, the filter returns the k records with highest overall similarity. In contrast to range query processing, query execution must be completed before the result can be determined. Because a record processed lately may be among the top-k records, any previously seen record may lose its position in the result set, so that the correct result set is known only after query execution.

Before queries are answered, some preparations are required to allow fast query processing:

- **Similarity measure:** The configuration of our similarity measure is learned using a set of training queries with correct result records (Chapter 3). Our learning method selects a set of similarity measures, each responsible for handling the similarity of a specific range of the frequencies of the name values.

- **Similarity indexes:** For each attribute, we create a similarity index to reduce similarity calculations at query time (Chapter 4).
- **Query planning:** Depending on the selected query planning method (Chapter 5), some pre-calculations are required. If *static planning* is selected, one plan is determined in advance that is used for all queries. For *query-specific planning*, a completeness tree is created that predicts the completeness of query plans using training queries at query time.

Example

Consider a small person database and a query for the attribute values FirstName = `Peter`, LastName = `Smith`, and City = `New York`. Additional query parameters are given as follows: The type is a range query with a range (minimum overall similarity) of 0.90, query-specific planning is to be used, and the cost limit is 5 (which is selected quite low for the purpose of demonstration). The query planner retrieves the frequencies of the attribute values in the database and then determines an appropriate query plan, which consists of predicates that require the similarity regarding an attribute to be at least as large as the specified threshold:

$$(\mathsf{FirstName} \geq 0.9 \wedge \mathsf{LastName} \geq 0.85) \vee (\mathsf{City} \geq 1.0)$$

The plan is executed by sending three tasks to the similarity receival component to determine the sets of IDs of records (referred to as S in the following) that fulfill the predicates from the query plan:

$$\begin{aligned} S(\mathsf{FirstName} \geq 0.9) &= \{3, 5, 9, 13\} \\ S(\mathsf{LastName} \geq 0.85) &= \{2, 5, 9, 14\} \\ S(\mathsf{City} \geq 1.0) &= \{1, 5, 15\} \end{aligned}$$

The overall set of IDs of records is then created according to the logical operators given in the query plan, where a conjunction of terms is transformed into an intersection of sets, and a disjunction is transformed into a union:

$$(S(\mathsf{FirstName} \geq 0.9) \cap S(\mathsf{LastName} \geq 0.85)) \cup S(\mathsf{City} \geq 1.0) = \{1, 5, 9, 15\}$$

All records with IDs in the determined set are retrieved from the database. The overall similarity of the query record and the retrieved records (where r_{ID} refers to the record with the given ID) are calculated:

$$\begin{aligned} sim_{Overall}(q, r_1) &= 0.91 \\ sim_{Overall}(q, r_5) &= 0.98 \\ sim_{Overall}(q, r_9) &= 0.88 \\ sim_{Overall}(q, r_{15}) &= 0.81 \end{aligned}$$

Finally, the filter is applied with the query range 0.90 and the query result is returned consisting of the records:

$$\{r_1, r_5\}$$

□

3 Similarity Measures

The problem of determining the similarity (or distance) of two records in a database is a well-known, but challenging problem. The representations of same real-world objects might differ due to typographical errors, outdated values, and sloppy data or query entries. A measure is needed to quantify the similarity of two representations; this measure is used to define the result set of a similarity search query.

In this chapter, we propose a novel comparison method for database records, which partitions the data using value frequency information and then automatically determines similarity measures for each individual partition. We show that this method indeed finds a different configuration for each partition and that we achieve an overall better precision than a corresponding frequency-oblivious measure.

This chapter is structured as follows: We describe base similarity measures and how to combine them in Section 3.1. In Section 3.2, we describe the details of our approach for including frequencies in similarity measures. We continue with a detailed evaluation in Section 3.3. Related work is discussed in Section 3.4, and the chapter is concluded in Section 3.5.

3.1 Defining Similarity

In this section we describe the similarity model used throughout the thesis. After introducing base similarity measures, we show different techniques to combine several base similarity measures into an overall measure.

3.1.1 Definitions and Examples

This thesis follows the common notion of defining individual similarity measures for different attributes and attribute types; for instance, dates are compared differently than names or addresses. These individual similarities are subsequently combined to define the global similarity of two records.

We first split the problem of measuring the similarity of two records into smaller subproblems. We define one similarity measure for each attribute.

Definition 3.1 (Base similarity measure)**.** *A* base similarity measure *for an attribute a is a function that compares the attribute values of a from two records from a universe U of possible records:*

$$sim_a : (U \times U) \to [0, 1] \subset \mathbb{R} \tag{3.1}$$

where a high similarity value means that the two compared values are very similar, a low value means they are very dissimilar.

Some similarity measures are defined as distance measures, while the transformation into a similarity measure is often trivial. For example, the distance of two numbers is given by their difference. The similarity of two numbers can be defined as the difference between the maximum distance (an application-specific number) and the actual difference between the numbers, the result is then scaled to $[0, 1]$ if necessary.

For various data structures, researchers have proposed specific similarity (or distance) measures, of which a selection is described in the following.

- **Strings:** A string is a sequence of characters. The most commonly used string similarity (or distance) measures cover insertion, deletion, and substitution operations of characters [Navarro, 2001]. The Levenshtein or edit distance of two strings is the minimal number of insertion, deletion, or substitution operations of characters to transform one string into the other [Levenshtein, 1966]. The Hamming distance is only defined for strings of equal length, and is calculated as the number of substitution operations to transform one string into the other [Hamming, 1950].

  Other measures include phonetic similarity. For instance, the Soundex encoding of a string consists of the first character of the string and three digits that represent the pronunciation of the remaining consonants in the string [Russell, 1918]. If two strings have the same Soundex encoding, they are pronounced similarly. While Soundex is optimized for the English language, other variants have been proposed for German (Kölner Phonetik [Postel, 1969]) and other languages (Double Metaphone [Philips, 2000]).

- **Graphs:** A graph consists of nodes and edges between these nodes. The similarity of two graphs is often defined by their structural similarity or by the similarity of any labels of their elements (e. g., using the string similarity measures described above). Structural similarity can be defined as the number of insertion or deletion operations of nodes and edges to transform one graph into the other, also referred to as graph edit distance [Sanfeliu and Fu, 1983].

- **Vectors:** The Minkowski distance refers to a commonly used family of distance measures for comparing two vectors consisting of several real numbers, t. The Minkowski distance of two vectors is defined as the p-th root of the sum of the differences of the vectors' components, where the differences are raised to the p-th power. With $p = 1$, the distance is called the Manhatten distance, and with $p = 2$, the Euclidean distance is obtained. Another relevant vector similarity measure is the cosine similarity, defined as the cosine of the angle between the two vectors.

- **Texts:** A text consists of a sequence of terms. To determine the similarity of texts, the texts are usually transformed into vectors representing the frequencies of terms in the texts [Manning et al., 2008]. Then, a vector similarity measure can be applied to the vectors, such as the cosine similarity.

In our person data use case, we have the functions $sim_{\mathsf{FirstName}}$, sim_{LastName}, $sim_{\mathsf{BirthDate}}$, sim_{City}, and sim_{Zip}. All base similarity measures can be chosen independently. For example, we could use Jaro-Winkler distance [Winkler, 1999] for $sim_{\mathsf{FirstName}}$, the relative distance between dates for $sim_{\mathsf{BirthDate}}$, and value difference for sim_{Zip}. The base similarity measures can also test for equality (e. g., for email addresses).

To calculate the overall similarity of two records, we integrate the base similarity measures into an overall judgement.

Definition 3.2 (Composed similarity measure)**.** *A* composed similarity measure *is an arbitrary function that is composed of the base similarity measures and compares two records from a universe U of possible records:*

$$sim_{Overall} : (U \times U) \rightarrow [0, 1] \subset \mathbb{R} \tag{3.2}$$

where a high similarity value means that the two compared records are very similar, a low value means they are very dissimilar.

For example, the weighted sum, the maximum, and the minimum of the base similarity measures are composed similarity measures. In Section 3.1.2, we describe our implementation of a more complex composed similarity measure using learning techniques.

A property of composed similarity measures that is relevant in subsequent chapters is monotonicity:

Definition 3.3 (Monotonous similarity measure)**.** *A composed similarity measure is a* monotonous similarity measure *if for any base similarity measure sim_a the following holds:*

$$\begin{aligned} &\forall r_1, r_2, s_1, s_2 \in U : sim_a(r_1, r_2) > sim_a(s_1, s_2) \\ &\wedge \forall a' \neq a : sim_{a'}(r_1, r_2) \geq sim_{a'}(s_1, s_2) \\ &\Rightarrow sim_{Overall}(r_1, r_2) \geq sim_{Overall}(s_1, s_2) \end{aligned} \tag{3.3}$$

Our query planning and top-k retrieval algorithms in Chapters 5 and 6 require monotonous similarity measures. Whether the composed similarity measure obtained by the methods proposed in this chapter is monotonous, depends on several design decisions. If logistic regression is chosen as learning method to combine the base similarity measures (Section 3.1.2) and all learned weights are positive, then the resulting measure is monotonous. For decision tree and SVM models, no such guarantee can be given. Furthermore, the partitioning method introduced in Section 3.2.3 gives a measure consisting of several learned models. The resulting measure is monotonous if the same measure (from the same partition) is used across all record comparisons, which depends on the design of the frequency function in Secion 3.2.1.

3.1.2 Learnable Similarity Measures

In the following, we describe how to obtain a composed similarity measure using machine-learning techniques. We consider the task of judging whether two records are similar to be a classification task: the classes are *isSimilar* and *isDissimilar*; the features are the results of the base similarity measures. We assume to have enough training data for supervised learning methods. In our use case, we have a large set of training queries with result records manually labeled as similar or dissimilar. In case of lacking training data, active learning methods could be chosen [Sarawagi and Bhamidipaty, 2002].

In this section, we discuss established machine learning techniques for classification. The following criteria are relevant for selecting machine learning techniques in our use case:

– **Performance:** Of course, we require the learning technique to generate convincing results. We measure success with precision and recall.

– **Interpretability:** The output of the overall similarity measure should be a value that expresses not only a preference for a classification as similar or dissimilar records, but also the classifier's certainty on this decision. This allows us to manually fine-tune a threshold for classification based on a detailed evaluation.

– **Comprehensibility:** The details of the learning technique should be comprehensible. For the users of the overall similarity system, it is often useful to not only to understand the overall model, but also to analyze the reasons for a specific similarity judgement. The ability to manually fine-tune the model is a plus.

We chose three popular classification methods for further analysis: logistic regression, decision trees, and support vector machines. The methods are described in details in the following and empirically compared in Section 3.3.

Logistic Regression

As an intuitive and comprehensible learning technique, we first suggest using logistic regression. The overall similarity is calculated as follows (where a iterates over all attributes):

$$sim_{LogReg}(r_1, r_2) = \frac{1}{1 + e^{-g(r_1, r_2)}} \tag{3.4}$$

with

$$g(r_1, r_2) = w_0 + \sum_a w_a \cdot sim_a(r_1, r_2) \tag{3.5}$$

The weights w_a are determined by minimizing the error probability of the classifier on the training data. The weights offer an intuitive interpretation of the learned model and directly show the relevance of the individual base similarity measures. These weights are optimal for our training data, but can easily be adjusted later on, if necessary.

Decision Tree

A decision tree consists of a set of nodes and leaves. Each internal node represents a split point with a condition (e. g., similarity of attribute FirstName is at least 0.8) and each leaf node represents the decision for a class (e. g., *isSimilar* or *isDissimilar*). While traversing the tree, conditions of internal nodes are evaluated, and the leaf class is returned. A popular algorithm for constructing a decision tree is C4.5 [Quinlan, 1993].

A generated decision tree is easy to understand for a human because of its visual representation. Additionally, some aspects of the tree can easily be adjusted, if necessary. Replacing a leaf by a new node or changing thresholds in decisions is quite simple.

To create a decision tree that combines our base similarity measures, we consider each base similarity measure as a separate attribute.

Support Vector Machine

Support vector machines (SVM) are a more recent development in machine learning [Vapnik, 1998]. An SVM learns to classify data points by determining a hyperplane that maximally separates the differently classified training data points. The points that are nearest to the hyperplane are called support vectors. Since such a hyperplane often cannot be found in the original data space, data is transformed into another data space by applying a kernel function.

The model of an SVM consists of a set of support vectors, often in a transformed data space. Thus, the model is difficult to understand; fine-tuning seems impossible. Although SVMs thus cannot provide comprehensive models, we still want to consider SVMs in the following due to their strong general classification performance [Kotsiantis, 2007].

Name	**Street**	**City**	**Birth date**
Arnold Schwarzenegger	27, Main St	Sacramento, CA 95811	30-Jul-47
Arnold Schwarzenegger	20, Main St	Sacramento, CA 95801	30-Jul-48
Peter Smith	48, 5 Av	New York, NY 10014	5-Jun-68
Peter Smith	4, 5 Av	New York, NY 10041	5-Jun-67

Table 3.1: Examples for differently handled records with rare and frequent names

Training and test data of an SVM consist of a set of data points, each point comprising a set of feature values. Features represent the main characteristics of the learning problem. In our case, we use each base similarity measure as a separate feature. Since we require all similarity values to be in the range between 0 and 1, no scaling is necessary.

3.2 Exploiting Frequencies

In our use case, we have data about a large set of persons. We need to define a similarity measure for comparing persons, but our findings are equally applicable to other entity types, such as products, bills, etc. We want to answer the question: Can information about data distribution improve our similarity measurement?

Specifically, we want to judge similarity differently for different frequencies of specific attributes. We use value frequencies to determine how useful a value is to identify an object. Table 3.1 shows an example: If two person records agree on a rare name, such as Arnold Schwarzenegger, then we are already quite sure that the two records represent the same person, even if other attribute values slightly disagree. We decide that the two first records shown in Table 3.1 represent the same person, even if there are small variations in the address and birth date fields. If, on the other hand, two records agree on a frequent name, say Peter Smith, then we cannot be sure that the records refer to the same person, since there are many persons with this name. On the contrary, we demand that other attribute values also agree. We would not say that the two latter records in Table 3.1 are referring to the same person, since we cannot forgive the variations in the address and birth date fields. Our intuition is thus that frequencies of attribute values have an effect on the importance of individual base similarity measures.

In this section, we propose two approaches to exploit frequencies in learning similarity measures. A requirement for both approaches is a frequency function that is introduced in Section 3.2.1. In Section 3.2.2, we describe how to adapt the machine learning techniques to include frequencies. We subsequently introduce a partitioning approach in Section 3.2.3 where different composed similarity measures are used for different frequency ranges. The introduced approaches are empirically compared in Section 3.3.

3.2.1 Frequency Function

In the following, we define a function that determines the information from the two records that is used to perform a frequency-specific judgement. Because the comparison, which is to be influenced by the frequency, operates on two records, the function also operates on these two records.

First of all, we need to select attributes for frequency evaluation (in our use case, we select the attributes FirstName and LastName). For two compared records, we then determine the value frequencies of the selected attributes. We define a frequency function f that takes as arguments two records from a universe U of possible records and determines a frequency:

$$f : U \times U \to \mathbb{N} \tag{3.6}$$

Modeling the frequency function is a domain-specific task. In the following, we describe how we modeled this function in the Schufa use case. Our goal is to partition the data according to the name frequencies. We have two name attributes in our data model (FirstName and LastName) and need to handle several data quality problems: swapping of first and last name, typos, and combining two attributes (so that one frequency value is calculated). In the following, we refer to the attribute values as follows: F_1 is the FirstName and L_1 is the LastName of the first record, and F_2 and L_2 are the respective values of FirstName and LastName of the second record. First and last name may be switched (e. g., F_1=`Arnold`, L_1=`Schwarzenegger`; F_2=`Schwarzenegger`, L_2=`Arnold`). We take this possible switch into account by calculating the attribute similarities for both attribute value combinations (switched and non-switched) and proceeding with the combination that results in a larger similarity value:

$$(\mathsf{F}_1, \mathsf{L}_1, \mathsf{F}_2, \mathsf{L}_2) := \begin{cases} (\mathsf{F}_1, \mathsf{L}_1, \mathsf{F}_2, \mathsf{L}_2) & \text{if } sim_{\mathsf{F}}(\mathsf{F}_1, \mathsf{F}_2) + sim_{\mathsf{L}}(\mathsf{L}_1, \mathsf{L}_2) \geq \\ & \quad sim_{\mathsf{F}}(\mathsf{F}_1, \mathsf{L}_2) + sim_{\mathsf{L}}(\mathsf{L}_1, \mathsf{F}_2) \\ (\mathsf{F}_1, \mathsf{L}_1, \mathsf{L}_2, \mathsf{F}_2) & \text{else} \end{cases} \tag{3.7}$$

where sim_F returns the similarity of two first names and sim_L the similarity of two last names. Moreover, typos may occur in attribute values (e. g., F_1 = `Arnold`; F_2 = `Arnnold`). We assume that at least one of the spellings is correct and that a typo leads to a less frequent name. Although this is not always true, this heuristic works in most cases. Thus, we take the larger frequency value of both spellings:

$$f_{\mathsf{FirstName}}(r_1, r_2) = \max(f_v(\mathsf{F}_1), f_v(\mathsf{F}_2)) \tag{3.8}$$

$$f_{\mathsf{LastName}}(r_1, r_2) = \max(f_v(\mathsf{L}_1), f_v(\mathsf{L}_2)) \tag{3.9}$$

where f_v returns the frequency of the respective value. Lastly, we combine the frequency values of first and last name. We argue that the less frequent name is more distinguishing and helpful (e. g., `Schwarzenegger` is more distinguishing than `Arnold`). Thus, we take the smaller frequency of the different attribute values as the result of our frequency function:

$$f(r_1, r_2) = \min(f_{\mathsf{FirstName}}(r_1, r_2), f_{\mathsf{LastName}}(r_1, r_2)) \tag{3.10}$$

Experiments with alternatives, namely using the maximum or the average instead of the minimum, showed that minimum is in fact the best accumulation function in our use case.

This description reflects the characteristics of our person data use case; for other data sets, the individual frequency function must be adjusted accordingly. For the DBLP data set (described in Section 3.3.5), our frequency function works similar to the Schufa function explained above, except that there is no check for switched first and last names, since the name parts are not split in this data set.

If a monotonous similarity measure is to be created (cf. Definition 3.3) and the partitioning approach from Section 3.2.3 is used, the frequency function should only consider the query record, so that it is guaranteed that the same partition and thus the same similarity measure is used for all record comparisons.

3.2.2 Frequency-enriched Models

A first idea to exploit frequency distributions is to alter the models that we learned with the machine learning techniques introduced in Section 3.1.1. One could manually add rules to the models, e. g., for logistic regression, we could say “if the frequency of the name value is below 10, then increase the weight of the name similarity by 10 % and appropriately decrease the weights of the other similarity functions”. Manually defining such rules is cumbersome and error-prone.

Another idea is to integrate the frequencies directly into the machine learning models. We add the frequencies of the compared entities’ attribute values as an additional feature to the discussed models. We call the resulting models *frequency-enriched models.* Some machine learning techniques can only handle normalized feature values (e. g., logistic regression and SVMs). Since all similarity values are required to lie in the range $[0, 1]$, we need to scale the frequency values accordingly. We apply the following scaling function:

$$scaled\_f(r_1, r_2) = \frac{f(r_1, r_2)}{M}, \tag{3.11}$$

where M is the maximum frequency of a value in the data set.

Adding attributes to the learned model means adding complexity and mixing information. The models become “polluted” with frequency information. The comprehensibility of the created models is lower, since each model contains mixed decisions based on similarity values and frequencies. As an example, consider a logistic regression model created with the base similarity measures and the frequency as attributes where all attributes have positive weights. In this case, to achieve a high overall similarity, it is necessary to have not only high base similarity values, but also a high frequency; a similarity of 1.0 can only be reached with the maximum frequency. On the other hand, a high frequency means that a medium overall similarity is always given, even if all base similarity values are very low. Thus, adding frequencies directly to the model gives malformed models. As our experiments in Section 3.3 show, this idea is clearly outperformed by the partitioning approach that we describe in the following section.

3.2.3 Partitioning

We propose to partition compared record pairs based on frequencies and create different models for the different partitions. These models are equal to the ones learned in Section 3.1.1, we just create several of them. The models still decide only on the basis of similarities. Since the models do not contain any additional information about the frequencies, they are still as easy to interpret and adjust as the original models.

We partition compared record pairs into n partitions using the determined frequencies. The number of partitions is an important factor for this process. A too large number of partitions results in small partitions that can cause overfitting. A too small number of partitions leads to partitions too large for discovering frequency-specific differences. To determine a good number of partitions as well as a good partitioning, we need a *partitioning strategy.* We introduce several strategies in Section 3.2.4.

In the following, we formally define partitions. The entire frequency space is divided into non-overlapping, continuous partitions by a set of endpoints:

$$\Pi = \{\pi_i \mid i \in \{0, \ldots, n\} \land \pi_0 < ... < \pi_n\} \tag{3.12}$$

with $\pi_0 = 0$ and $\pi_n = M + 1$, where M is the maximum frequency in the data set. A *partition* I_i is an interval defined by its endpoints:

$$I_i = [\pi_i, \pi_{i+1}) \tag{3.13}$$

A *partitioning* I is a set of partitions that covers the entire frequency space:

$$I = \{I_i \mid i \in \{0, \ldots, n-1\}\} \tag{3.14}$$

A record pair (r_1, r_2) falls into a partition $[\pi_i, \pi_{i+1})$ iff the frequency function value for this pair lies in the partition's range:

$$\pi_i \leq f(r_1, r_2) < \pi_{i+1} \tag{3.15}$$

For each partition, we learn a composed similarity measure using the learning techniques presented in Section 3.1.1. With the set of composed similarity measures for all partitions at hand, we can judge the similarity of unseen record pairs. For a record pair, we first determine the partition that the pair belongs to with Formula (3.15). We then apply only the composed similarity measure that has been learned for this partition. By retaining frequency lists for all frequency-relevant attributes, the frequency lookup is quite fast. Thus, our approach hardly affects the query time. This is quite important for our use case as each query needs to be answered in less than a second.

3.2.4 Partitioning Strategies

An important problem in partitioning our data is to determine the number of partitions as well as the endpoints that separate the partitions. Since the number of partitions is determined by the number of endpoints, we only need to determine a set of endpoints.

A complete search on the frequency space is too expensive. For a maximum frequency M, we could define up to $n = M - 1$ endpoints, resulting in a separate partition for each possible frequency. Overall, there are 2^{M-1} possibilities to choose endpoint sets, since each distinct frequency $f \in \{1, \ldots, M\}$ can be either contained or not contained in an endpoint set. For a reasonable maximum frequency of $M = 1,000,000$, there are obviously too many endpoint combinations to consider. Even if we consider only the number of distinct frequencies that actually occur in the data set as possible endpoints, the computation costs are too high. In our use case data set, the frequency of the most frequent last name is 616,381. The number of distinct frequencies is 4629, which results in 2^{4628} theoretically possible different partitionings.

To efficiently determine a good partitioning, we suggest the following partitioning strategies, which we empirically compare in Sections 3.3.4 and 3.3.5:

- **Random partitioning:** To create a random partitioning, we randomly pick several endpoints $\pi_i \in \{0, \ldots, M+1\}$ from the set of actually occurring frequencies of the considered attribute values. The number of endpoints in each partitioning is also randomly determined. The maximum number of partitions in one partitioning as well as the total number of generated initial partitionings are fixed. For our use case, we define a maximum of 20 partitions in one partitioning.

- **Equi-depth partitioning:** We divide the frequency space into e partitions. Each partition contains the same number of records from the original data set R. For our use case, we create partitionings for $e \in \{2, \ldots, 20\}$.

  Note that we also evaluated equi-width partitioning. Due to the skewed frequency distribution in our use case, many of the resulting partitions contain no record pairs, which are necessary for training and testing our similarity measures. Thus, we do not consider equi-width partitioning in the following.

- **Greedy partitioning:** We define a list of endpoint candidates $\{\pi_0, \ldots, \pi_n\}$ by dividing the frequency space into segments with the same number of records (similar to equi-depth partitioning, but with fixed, large e, in our case $e = 50$). We then begin learning a similarity measure for the partition defined by the first candidate endpoints $[\pi_0, \pi_1)$. Then we learn a measure for the second partition that extends the current partition by moving its upper endpoint to the next endpoint candidate: $[\pi_0, \pi_2)$. We compare both partitions (i. e., the learned measures for the partitions) using F-measure (harmonic mean of precision and recall). If the extended partition achieves better performance, the process is repeated for the next endpoint slot. If not, the smaller partition is kept and a new partition is started at its upper endpoint; another iteration starts with this new partition. This process is repeated until all endpoint candidates have been processed.

This greedy partitioning algorithm stops after encountering a worse F-measure. A worse F-measure is an indicator that another model is required for the set of records in the analyzed partition; thus, it makes sense to create a new partition for this set.

- **Genetic partitioning:** Another partitioning approach is inspired by genetic algorithms. Since we have achieved overall best results with this approach in our experiments, we elaborate on genetic partitioning in the following section.

3.2.5 Genetic Partitioning

To determine a good partitioning, but keep the number of generated and evaluated partitions low, we apply ideas inspired by genetic algorithms [Banzhaf et al., 1998, Koza, 1992]. Genetic algorithms in turn are inspired by biological evolution. From an initial population, the fittest individuals are selected. These individuals "breed" offspring that ideally combine the advantages of their parents and are thus fitter than them. Random genetic mutation and crossover produce even fitter offspring.

Indeed, genetic algorithms are a good fit to our problem. A specific partitioning contains several endpoints, some of which are a good choice, while others should be replaced or removed completely. Deciding whether an endpoint is a good choice is difficult, as we can only evaluate performance of partitions, and these are defined by *two* endpoints. In addition, we do not know the optimal number of partitions in advance. Due to their intelligent selection algorithms and random events, genetic algorithms can efficiently handle these choices (as we also empirically show in Sections 3.3.4 and 3.3.5).

The detailed steps of our *genetic partitioning algorithm* are the following:

- **Initialization:** We first create an initial population consisting of several random partitionings. These are created as described above with the random partitioning startegy.

- **Growth:** Each individual is grown. We learn one composed similarity function for each partition in the current set of partitionings using the techniques presented in Section 3.1.2.

- **Selection:** We then select some individuals from our population for creating new individuals. A fitness function determines the quality of partitionings. In our case, we select F-measure on our training data as fitness function. For each partition, we determine the maximum F-measure that can be achieved by selecting an appropriate threshold for the similarity function. We weight the partitions' F-measure values according to the compared record pairs in each partition to calculate an overall F-measure value for the entire partitioning. We then select the partitionings with highest weighted F-measure. For our use case, we select the top five partitionings. Note that we use a test set for evaluation that is different from the training set used for learning the composed similarity measure for each partition.

- **Reproduction:** We build pairs of the selected best individuals (during all iterations) and combine them to create new individuals. Two techniques are applied to each pair of partitionings:

  - *Recombination*: We first create the union of the endpoints of both partitionings. For each endpoint, we randomly decide whether to keep it in the result partition or not. Both decisions have equal chances.
  - *Mutation*: We randomly decide whether to add another new (also randomly picked) endpoint *and* whether to delete a (randomly picked) endpoint from the current endpoint set. All possibilities have equal chances.

  A partition is too small if we have too few training and test data available. To avoid too small partitions (and thus overfitting), we define a minimum partition size (we set this value to 20 record pairs in our experiments with 1000 training instances). Randomly created partitionings with too small partitions are discarded. In our use case, we create two new partitionings for each partitioning pair processed in the reproduction phase.

- **Termination:** The resulting partitionings are evaluated and added to the set of evaluated partitionings. The selection/reproduction phases are repeated until a certain number of iterations is reached or until no significant improvement can be measured. In our use case, we require a minimum F-measure improvement of 0.001 after 5 iterations. The algorithm returns the partitioning that has been evaluated best during all iterations.

Genetic algorithms are variants of hill climbing algorithms. Thus, they can run into local maxima, i. e., a good, but non-optimal solution that cannot be improved because too few variations are considered. As our algorithm applies several random evolution steps, this danger is not as high as for other algorithms. Several parameters of the algorithm influence the probability to find a good solution, in particular the number of initial partitionings and the number of iterations. We evaluate the effects of these parameters for our use case in Section 3.3.3.

We improve efficiency of the genetic partitioning algorithm by using caches for generated learners as well as evaluation results. As shown in Section 3.3.3, this approach reduces runtime by more than 70 %.

3.3 Evaluation

In this section, we provide a detailed evaluation of our approach. We describe the Schufa data set in Section 3.3.1. We then show in Section 3.3.2 results of the machine learning methods applied without partitioning the data. We continue with a detailed analysis of the genetic partitioning algorithm in Section 3.3.3. We compare different partitioning strategies and show improvements achieved by our approach in Section 3.3.4. Since our motivation stems from a specific use case, we conducted most experiments on the Schufa data set. In Section 3.3.5, we show results of experiments on a different data set, which was extracted from DBLP.

We evaluate the effectiveness of our approach with precision and recall. Precision is the fraction of true positives among all positively evaluated record pairs. High precision means that a similarity measure is good at finding *only* true positives. Recall is the fraction of true positives among all true matches. A similarity measure with high recall is good at finding *all* true positives.

To combine these metrics into an overall metric, we choose F-measure, the harmonic mean of precision and recall. As harmonic mean, F-measure has the advantage of forcing both precision and recall to be high. If only one of them has a high value, then F-measure is significantly lower than the arithmetic mean. Thus, F-measure helps us to determine the overall best similarity measure.

We performed all tests on a workstation PC. Our test machine runs Windows XP with an Intel Core2 Quad 2.5 GHz CPU and 8 GB RAM.

We used Weka [Hall et al., 2009] as implementation of the machine learning techniques. We used C4.5 [Quinlan, 1993] as decision tree algorithm, LIBSVM [Chang and Lin, 2001] for Support Vector Machines, and a logistic regression algorithm that employs ridge estimators [Le Cessie and Van Houwelingen, 1992].

3.3.1 Preparation of Schufa Data Set

We evaluate our approach on real-world data from Schufa, a large credit agency. The Schufa database contains information about the credit history of about 66 million people.

Our data set consists of two parts: a person data set and a query data set. The person data set contains about 66 million records. The most relevant fields for our search problem are first name, last name, birth date, and address data (street, city, zip). The query data set consists of 2 million queries to this database. For each query, we know the exact search values (most record fields are mandatory), and the result obtained by Schufa's current system. This result contains up to five candidate records.

The Schufa system automatically evaluates its confidence. A confidence value is assigned to each result. If only one result could be found and if its confidence is above a pre-determined high threshold, then the result is automatically accepted. Results with a confidence below a pre-determined low threshold are automatically discarded. In some cases, hand-crafted decision rules can be applied. In all other cases, Schufa is particularly careful: An expert determines whether one of the results can be accepted or not.

Thus, there are many manually evaluated queries (the least confident, most difficult cases) that we can use for evaluating our approach. We randomly selected 10,000 of these most difficult queries for evaluating our system: we built record pairs of the form *(query, correct result)* or *(query, incorrect result)*, depending on how the result has been evaluated by the expert. We compare with the results of Schufa's current system and we show whether our system would allow Schufa to save some of the (expensive) manual decisions without losing precision.

The automatically evaluated cases are not interesting for us: by comparing with Schufa's current system, we could only determine whether our system would act in the same way, without knowing whether these decisions are correct.

For our experiments in this chapter, we have defined hand-crafted similarity measures for the name, the address, and the birth date of persons as base similarity measures.

3.3.2 Evaluation of Similarity Measure Composition Techniques

In our first experiment, we evaluate the results of the different learning techniques described in Section 3.1.1; no frequency information is available yet. We apply 10-fold cross validation (CV). The results form the basis of this section as a reference point to measure improvements achieved by partitioning.

As baseline, we additionally show results for the existing system of Schufa, a manually-tuned system consisting of a set of rules. Regarding the low absolute values (an F-measure of 0.8 is of course not acceptable for a real-world system), keep in mind that these experiments are run only on the most difficult sample, i. e., on precisely those record pairs where an automated decision was not possible and thus a human decision was required. To evaluate the system, we "enforce" a decision by applying a decision threshold on the calculated similarity.

For each evaluated method, we determine maximum F-measure results, i. e., the highest F-measure value of all calculated precision-recall points for different decision thresholds. Table 3.2 shows average results of the best F-measure values of each CV run. Logistic regression achieves highest F-measure of all learning methods. All results are relatively close and improve the current system by only 2 percentage points. This improvement may seem low, but we need to remember that we chose a quite difficult test set with mainly uncertain query results. We show the impact of our further ideas in the following experiments.

	F-measure	Precision	Recall
Logistic regression	**0.832**	**0.739**	0.951
Decision tree	0.826	0.734	0.945
SVM	0.830	0.733	**0.958**
Current system	0.817	0.722	0.940

Table 3.2: Best F-measure results and according precision and recall values for learning methods and Schufa's current system

3.3.3 Genetic Partitioning Results

We now evaluate the genetic partitioning algorithm presented in Section 3.2.5. We randomly divided the 10k queries into 9k for training and 1k for testing. In Section 3.3.4, we present the results of a complete 10-fold CV run, but in this section, we cover only one of these runs to better analyze the behavior of the genetic partitioning algorithm. Note that because of this difference, the resulting absolute values of this experiment are not comparable with the other experiments.

The results of one genetic partitioning run per learning technique are shown in Figure 3.1. We illustrate the results of each genetic partitioning iteration. The green points show the highest F-measure of all evaluated partitionings in the iteration; the red points show the lowest result. We added F-measure for the composed similarity measures without partitioning as baseline.

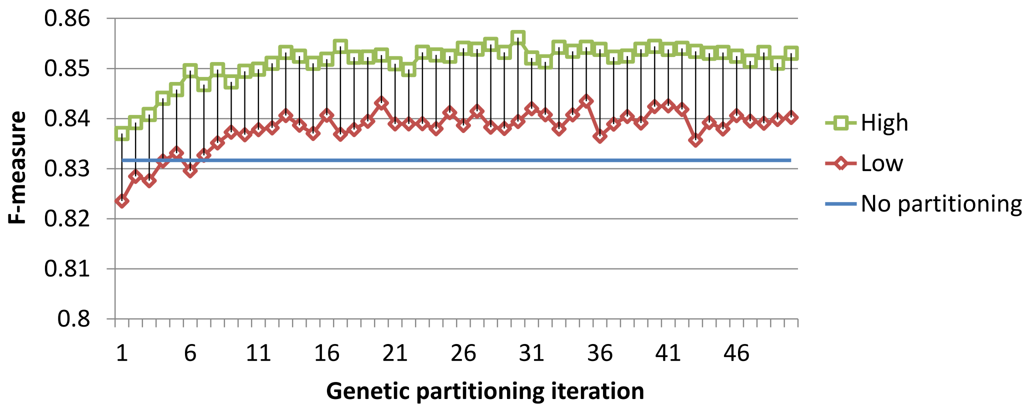
For all three evaluated learning techniques, we can see that the baseline result lies in between the results of the first partitioning round. Even without genetic partitioning, we can improve our initially learned similarity measures at least with some of the random partitionings.

For the following iterations of logistic regression and SVM (Figures 3.1a and 3.1b), we can see a relatively fast improvement of both the highest and the lowest F-measure of the partitionings created via recombination and mutation of successful partitionings. However, after a certain amount of iterations, no further improvement can be measured. The necessary number of iterations differs for the different techniques.

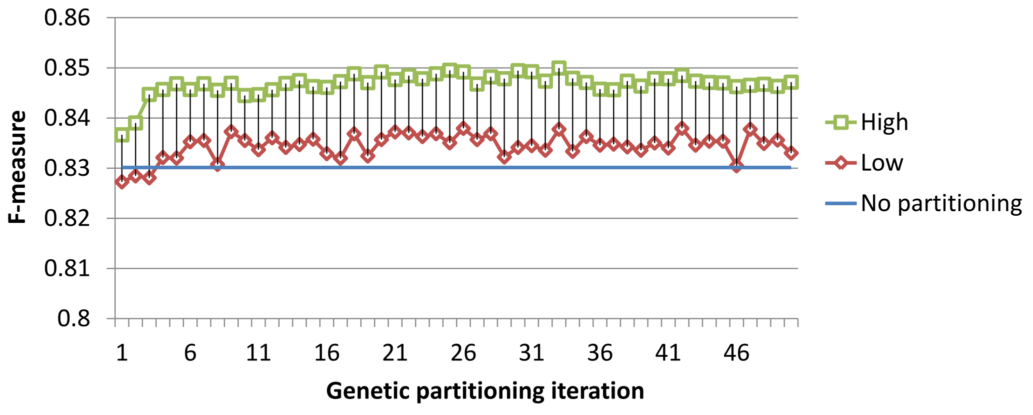
For decision trees (Figure 3.1c), we observe a much slower improvement of the best partitioning results per iteration. Additionally, there are still quite often results that are approximately equal to or worse than the results without partitioning. The gap between best and worst results grows slowly. We assume that the pruning behavior of decision trees is responsible for this growth. To avoid overfitting, decision tree algorithms prune leaves with too few elements. In some cases, the changes of one genetic partitioning iteration seem to be too small to trigger the insertion of several new nodes that are not pruned. Thus, the generated trees do not change considerably, and there is only little improvement in one iteration. We conclude that the number of necessary genetic partitioning iterations also depends on the learning method.

Relevance of Initial Partitionings

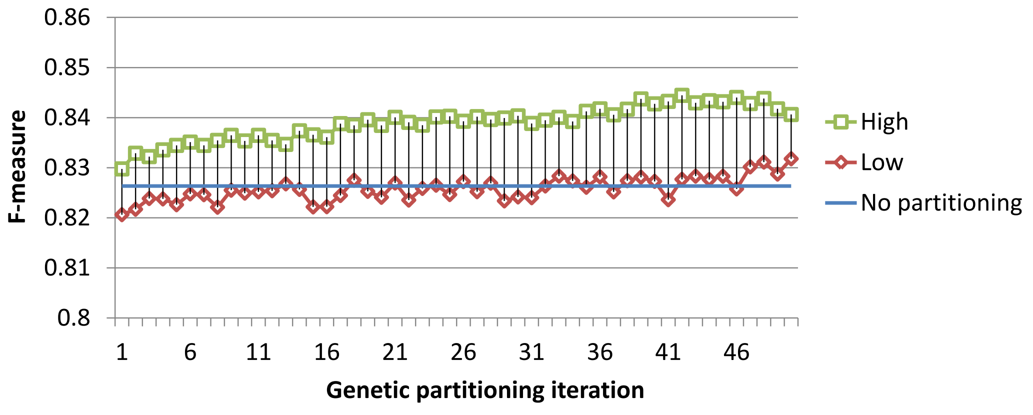
We continue our analysis with the most promising learning technique: logistic regression. We now analyze whether the generated partitionings in the later iterations are mainly recombinations of the random partitionings of the initial set or whether mutation actually helps discovering



(a) Logistic regression



(b) SVM



(c) Decision tree

Figure 3.1: Results for genetic partitioning iterations per learning technique

new and better partitionings. For all iterations of a genetic partitioning run with logistic regression (same setting with 9k records for training and 1k records for testing), we have calculated the fraction of partition endpoints generated in this iteration that were already included in the first iteration.

Figure 3.2 shows that already after few iterations, a large part of the generated endpoints is not related to the initial endpoints. Additionally, we observe that a certain amount of initial endpoints remains in the final set of endpoints. There seem to be enough random mutation steps in the genetic partitioning algorithm, so that the initial set of endpoints does not dominate the generated partitionings.

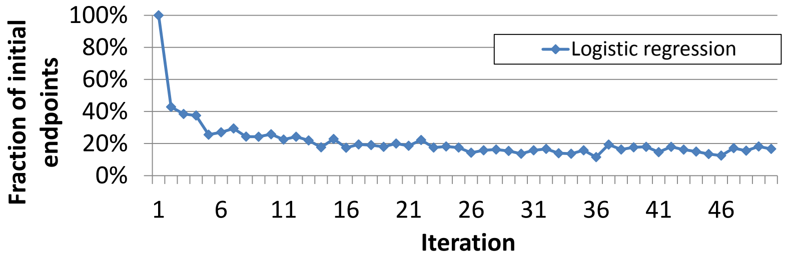


Figure 3.2: Fraction of initial endpoints

Efficiency Improvements with Caches

To efficiently compute a good partitioning, we exploit the reuse of learned similarity measures and evaluation results in genetic partitioning. For each created partitioning, the learned overall similarity measures and evaluation results (F-measure) of all unseen partitions are stored in a cache. Whenever a new partitioning contains a partition that has been created before, the similarity measure from the cache is reused. Also, the evaluation result is obtained from the cache for evaluating the partitioning.

In Figure 3.3, we show the elapsed time after 50 genetic partitioning iterations for logistic regression. The graphs show improvements achieved using a cache for the learned similarity measures (Learner cache) and for the evaluation results (Result cache). With a learner cache, the overall runtime can be significantly reduced by 72 %. With an additional cache for the evaluation results, this rate can be increased to 76 % in total.

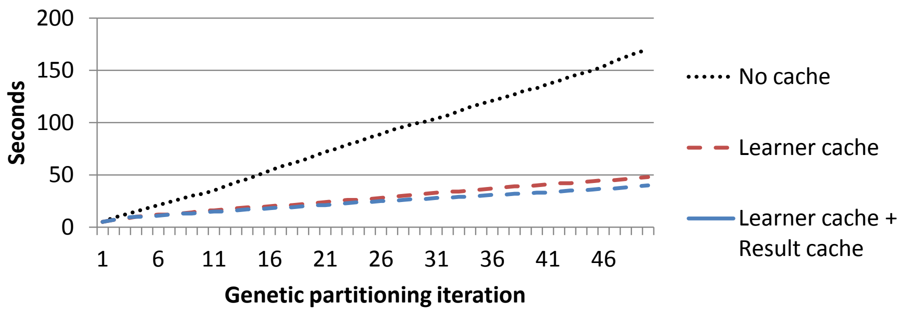


Figure 3.3: Elapsed time after each genetic partitioning iteration with cache usage (all results using logistic regression)

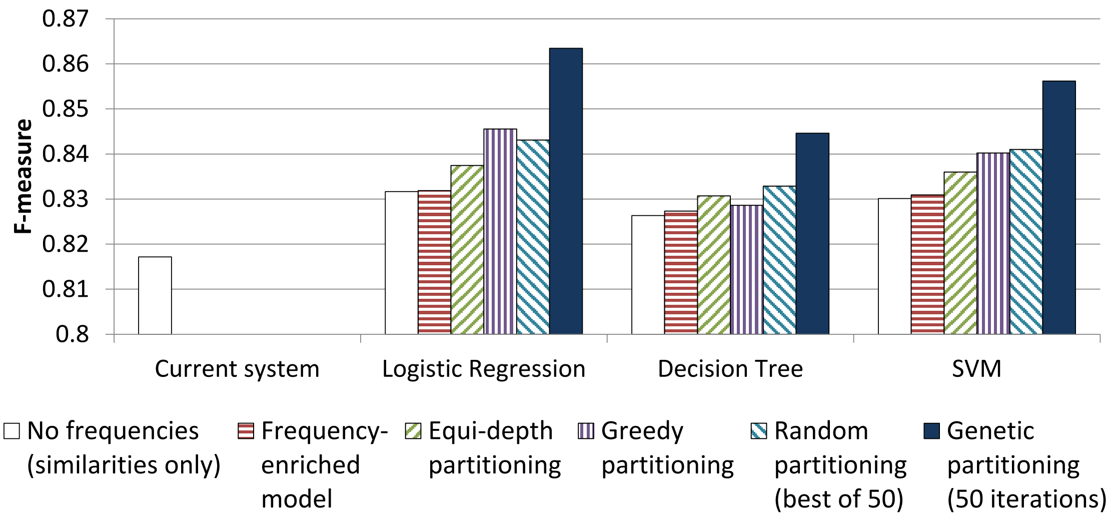


Figure 3.4: Comparison of frequency approaches and partitioning strategies for Schufa data set

3.3.4 Comparison of Partitioning Strategies: Schufa Data Set

In this experiment, we compare the results of the learned similarity measures when applying the different partitioning strategies described in Section 3.2.4. All results are determined using 10-fold CV on the 10k query data set containing difficult queries. We show the results in Figure 3.4.

For all learning methods, we observe considerable improvements with almost all partitioning strategies. The partitioning approaches also achieve better results than the frequency-enriched model, for which no significant improvement over the frequency-oblivious model can be measured. As we have pointed out in Section 3.2.2, the frequency-enriched models perform mixed decisions based on frequencies and base similarity values and are thus often malformed.

For equi-depth partitioning, greedy partitioning, and random partitioning, we can see no clear leader across all learning methods. The achieved results are specific to each method. For all learning methods, genetic partitioning achieves the overall best results.

When comparing learning methods for genetic partitioning, logistic regression achieves better results than SVM, while decision trees are relatively far behind. As discussed earlier, we believe that the pruning behavior of decision trees is responsible for their small gain with genetic partitioning.

We further show the result of Schufa's manually developed current system. In total, we can improve this system's F-measure by approx. 6 %, which is a significant and valuable improvement for this use case.

In an additional experiment, we analyzed the effects on regular cases (i. e., all queries are considered, not only the most difficult cases) by running the generated classifiers on the entire 2m query set. For this experiment, we retrained our learners (from the genetic partitioning runs) with a set of 10k randomly selected queries from the entire query set. Note that for all automatically evaluated queries, we assume that Schufa's current system is always "correct", since there is no manual evaluation for these queries. However, we compare our learners with this system, for the sake of completeness. In Table 3.3, we show the results for the differently evaluated query sets. Although all decisions that do not agree with Schufa's current system have been considered incorrect, the improvements achieved in the manually evaluated query set (with most difficult queries) still lead to overall slightly better results for all learning methods.

	Manually evaluated queries (**150k**)	All queries (**2m**)
Logistic regression	0.8634	0.9849
Decision tree	0.8446	0.9849
SVM	0.8562	0.9860
Current system	0.8172	0.9833

Table 3.3: Results on entire query set. Evaluation is based on expert decisions where available (*manually evaluated*) or automatic decisions of Schufa's current system (*automatically evaluated*).

3.3.5 Comparison of Partitioning Strategies: DBLP Data Set

To investigate whether our approach also works for other data sets, we prepared another data set from DBLP [Ley, 2009], a bibliographic database for computer sciences. The main problem in DBLP is the assignment of papers to author entities. As described by Ley, DBLP is not perfect and needs to handle joins and splits: Author entries that falsely summarize publications from several authors need to be split into distinct author entries; author entries that represent the same entity, but have different names, need to be joined [Ley, 2009]. In contrast to other work that focuses on artificially injected errors [Rastogi et al., 2011] or specific groups of names [Ferreira et al., 2010, Han et al., 2004a], we want to handle the actual DBLP problems on a larger scale. We constructed our data set from cleaned parts of DBLP, where different aliases for a person are known or ambiguous names have been resolved. In DBLP, cleaning is done manually (due to author requests) and automatically (using fine-tuned heuristics) [Ley, 2009].

We created a new data set consisting of paper reference pairs where each pair that can be assigned to one of the following categories:

1. Two papers from the same author
2. Two papers from the same author with different name aliases (e. g., with or without middle initial)
3. Two papers from different authors with the same name (i. e., the name is ambiguous)
4. Two papers from different authors with different names

For each paper pair, the matching task is to decide whether the two papers were written by the same author. The data set contains 2,500 paper pairs per category (10k in total). This does not represent the original distribution of ambiguous or alias names in DBLP (where around 99.2 % of the author names are non-ambiguous), but makes the matching task more difficult and interesting. We provide this data set on our website[1].

Results. We list our similarity measures in Table 3.4[2]. Similar to the previous experiment on the Schufa data set, we applied 10-fold cross validation to compare the partitioning strategies from Section 3.2.4. The results in Figure 3.5 are similar to the results of the Schufa data set. Without partitioning, all three machine learning techniques achieve similar F-measure results of about 0.86.

[1] `http://www.hpi-web.de/naumann/data`

[2] For the Schufa data set, we cannot disclose the used similarity measures.

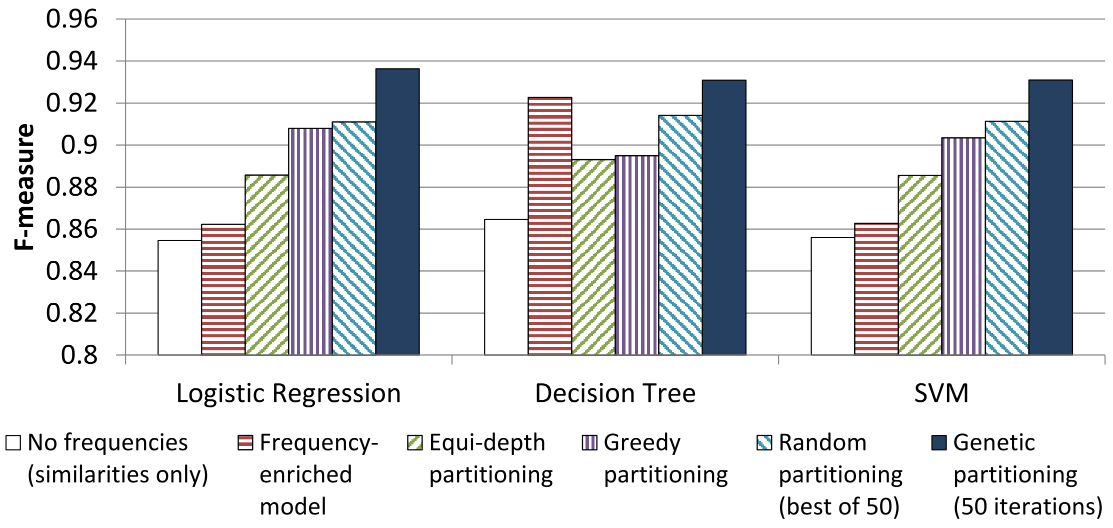


Figure 3.5: Comparison of frequency approaches and partitioning strategies for DBLP data set

Attribute	**Similarity measure**
Title	MongeElkan
Author	MongeElkan of concatenated author names
Author	JaroWinkler of concatenated author names
Author	Exact match of all author names (boolean)
Author	Max JaroWinkler-Similarity of any author combination
Author	JaroWinkler similarity of any author combination ≥ 0.9 (boolean)
Journal/ Booktitle	MongeElkan of journal or booktitle (longer value is used)

Table 3.4: Our similarity measures for DBLP data set

Incorporating only frequencies into these models (without partitioning) improves performance only by a small amount. An exception are decision trees, for which we can measure a significant improvement. For this data set, the decision tree algorithm could determine that the frequency attribute is relevant for the classification task.

From the partitioning strategies, genetic partitioning again clearly outperforms all other strategies with overall F-measure results of about 0.93. The results show that random partitioning is not enough to achieve best results. The equi-depth and greedy partitioning strategies perform worse than the other partitioning strategies, but still better than the no-partitioning approach without frequencies. For this data set, too, partitioning always improves the results.

Overall, we can measure a significant improvement by genetic partitioning of about 9 %, which is larger than the improvement on the Schufa data set. Our experiments show that our approach works on different data sets, but that the actual amount of improvement depends on the data set.

3.4 Related Work

In the last decades, many researchers have worked on duplicate detection, where similarity measures are used to define which records are duplicates. Recent surveys provide a general overview on duplicate detection [Elmagarmid et al., 2007, Winkler, 1999].

Bilenko et al. use machine learning techniques to learn similarity measures for duplicate detection [Bilenko and Mooney, 2003]. They learn base similarity measures using SVMs and apply another SVM to combine the base similarity measures. Sarawagi and Bhamidipaty use active learning for interactive duplicate detection [Sarawagi and Bhamidipaty, 2002]. They also employ machine learning techniques (decision trees, naive Bayes, and SVMs) to combine base learners. We adapt and greatly extend this approach by creating a set of learners for different partitions of the data. Shen et al. propose to create a set of different matchers for different portions of the data [Shen et al., 2007]. They focus on the task of integrating a set of data sources and create different similarity measures for comparing entities from the different sources. In contrast to their work, we partition data according to frequencies and not based on different sources of the data. Moreover, we employ a set of similar matchers, i. e., we learn one similarity function for each of the partitions – but all of them with the same machine learning technique. Another idea is to use actual attribute values for partitioning (e. g., using a Country attribute, we learn one measure for each different country). While this idea depends on the availability of attributes values that are suitable for partitioning, our approach is more generally applicable as it only exploits meta-information of the values (frequencies).

There are also approaches to duplicate detection/entity resolution that exploit knowledge about frequencies. Bhattacharya and Getoor apply collective entity resolution; by incorporating knowledge about references between entities, entity resolution can be improved [Bhattacharya and Getoor, 2007]. Their approach starts by resolving entities with least frequent names, since "two references with the name 'A. Ansari' are more likely to be the same, because 'Ansari' is an uncommon name." Torvik and Smalheiser propose a probabilistic approach to author name disambiguation [Torvik and Smalheiser, 2009]. In their model, they include the frequency of names to predict the probability that two names match. They also recognize that "if the name is very unusual (e. g., D. Gajdusek), the chances are better that any two randomly chosen articles with that name are written by the same individual than if the name is very common (e. g., J. Smith)." We also exploit this insight, but with an entirely different approach.

There exist several ideas to generate learners from different portions of data [Dietterich and Fisher, 2000]. The most popular ones are *bagging* (bootstrap aggregating, means creating and averaging a set of learners from random samples of the data) and *boosting* (creating a set of learners and weighting them according to their accuracy). In contrast to these approaches, we are interested in a meaningful partitioning of the data and create one learner for each partition of the data. Only this learner is then used for the instances that fall into this partition.

The idea to partition data is often used for more *efficient* processing. For instance, blocking algorithms are popular in duplicate detection [Jaro, 1989]. The key idea of blocking is to place records that are likely to be similar into one partition. Only records within the same partition are compared. In contrast, we partition data for more *effective* similarity measurement. What is more, we partition according to frequencies of values (metadata), and not according to the actual values (data). Our partitioning approaches are orthogonal to any of the previous ideas on data partitioning for efficiency.

3.5 Conclusion

In this chapter, we introduced an approach for improving composed similarity measures. We divide a data set consisting of record pairs into partitions according to frequencies of selected attributes. We learn optimal similarity measures for each partition. Experiments on different real-world data sets showed that partitioning the data improves learning results and that genetic partitioning performs better than several other partitioning strategies, while the amount of improvement depends on the data set.

With the learned similarity measure, we have a means for effectively specifying the result of a similarity search query, i. e., all records in the data set with a large value for the overall similarity measure. In the subsequent chapters, we present methods for performing efficient search with such complex similarity measures.

4
Indexing Similarity

While we defined our similarity measure in the previous chapter to achieve effective similarity search, we now turn to the problem of performing efficient search queries.

In the area of databases, a common approach to speed up queries is to create indexes. An index is a data structure that supports efficient access to the data [Garcia-Molina et al., 2009]. An index structure frequently used in database systems is the B^+-tree, a binary search tree where the leaf nodes contain ordered pointers to a set of database records, and each non-leaf node similarly contains pointers to lower nodes [Bayer and McCreight, 1970].

The concept of indexes can be adapted for answering similarity queries. The main goal of similarity indexes is to avoid expensive comparisons of attribute values from database records with the query record. Similarity indexes typically exploit characteristics of the data or similarity measures to exclude records from the search as early as possible.

In this chapter, we first give an overview on similarity indexes in Section 4.1. We then present the State Set Index (SSI) in the subsequent sections, a similarity index for strings. The main advantage of SSI in comparison to previous string indexes is that it has a small memory footprint while providing fast query execution times on small distance thresholds. We describe our approach in Section 4.2 by defining the index structure and algorithms for building the index and searching with it. We show evaluation results in Section 4.3. Section 4.4 describes related work, and the chapter is concluded in Section 4.5.

4.1 Similarity Indexes

In our similarity search system (see Chapter 2 for an overview), we use attribute-specific similarity indexes to efficiently retrieve sets of records with similar attribute values. We first define the type of queries answered by similarity indexes.

Definition 4.1 (Attribute similarity query)**.** *The result of an attribute similarity query for a query record $q \in U$, a record set $R \subseteq U$, an attribute a, a retrieval threshold θ_a is the set of records in R where the similarity for the value of a is at least θ_a:*

$$\{r \in R \mid sim_a(q,r) \geq \theta_a\}$$

Index structures for answering similarity queries typically gain efficiency by exploiting properties of specific similarity measures or data types. In the following, we give a brief overview of common similarity index structures.

- **Vector space:** If the data is available in or can be transformed into a vector space (feature space), the search of similar objects can be reduced to the search of close vectors [Böhm et al., 2001].

  The *R-tree* is a binary search tree that organizes vectors as points using bounding hyper-rectangles [Guttman, 1984]. Each leaf node represents a hyper-rectangle that contains a set of points. Each internal node represents a hyper-rectangle that contains the set of hyper-rectangles of its child nodes. In both cases, the hyper-rectangle is the minimum bounding box of the contained elements. Querying the R-tree is performed by starting at the root node and evaluating which of the child nodes' regions overlap with the search region. Only those regions are considered further. Only points within hyper-rectangles of leaf nodes that overlap with the search region are relevant to the search and must be evaluated for containment in the search result.

  The *kd-tree* is a binary search tree for multidimensional data [Bentley, 1975]. Each internal node represents a split node, where the split condition is defined as a predicate on one of the dimensions. For example, the split condition $a < 10$ means that all points with a value for a smaller than 10 are placed in the left subtree, and all other values in the right subtree. Each subtree can either be another node with a split condition or a set of elements that is not further divided.

- **Metric space:** In the metric space, the similarity measure needs to fulfill the metric requirements (non-negativity, symmetry, identity, and the triangle inequality), while the data can have arbitrary form [Chávez et al., 2001, Zezula et al., 2006]. In this setting, the triangular inequality can be exploited to efficiently reduce the search space. Several strategies have been proposed for partitioning the data set, so that during search groups of elements can be excluded as early as possible.

  The *ball partitioning* strategy picks a pivot element and divides the data set into two equally large subsets, separated by a distance radius from the pivot: the set of elements near the pivot element (with a distance smaller than the separation radius), and the set of elements far from the pivot element (with a distance larger than the separation radius). Ball partitioning is used in the Burkhard-Keller tree [Burkhard and Keller, 1973] and the vantage point tree [Yianilos, 1993].

  For *generalized hyperplane partitioning*, two pivot elements are selected. The data set is divided as follows: the set of elements that are nearer to the first pivot, and the set of elements nearer to the second pivot. Generalized hyperplane tree [Uhlmann, 1991] and bisector tree [Kalantari and McDonald, 1983] apply generalized hyperplane partitioning.

  Excluded middle partitioning is based on ball partitioning. A pivot element is selected, and the data set is partitioned according to the distance to the pivot. In contrast to ball partitioning, an exclusion zone is defined for the elements that are near the separation radius. If the exclusion zone contains many elements, it can be further subdivided. With a search distance smaller than the exclusion zone, it is guaranteed that at least one of the subsets can be excluded from the search. Excluded middle partitioning is used by the excluded middle vantage point forest [Yianilos, 1999].

- **Non-metric space:** If the given similarity measure is not a metric, most approaches try to transform the search problem into one of the well-solved problems for search in vector or metric space [Skopal and Bustos, 2011].

  The *TriGen* algorithm transforms the given non-metric similarity function into a metric [Skopal, 2006]. The similarity function must be a semi-metric, i. e., all metric requirements except the triangular inequality are fulfilled. TriGen determines on a sample the amount of elements where the triangular inequality does not hold, and then applies a transformation function on the similarity function to correct this error. The *NM-tree* uses TriGen for transforming the similarity function into a metric [Skopal and Lokoč, 2008]. After that, the M-tree is used as an efficient indexing method within the metric space [Ciaccia et al., 1997].

The *QIC-M-tree* also builds on the M-tree [Ciaccia and Patella, 2002]. The user is required to define a metric function that is a lower bound for the actual (non-metric) similarity function. The M-tree is then built for the metric lower bound function and used to partially answer the query, allowing to prune at least some subtrees from the M-tree. Elements in the remaining subtrees still need to be compared to the query object.

– **Strings:** For similarity search in sets of strings, many specialized indexes have been proposed [Navarro, 2001]. We give a detailed overview on similarity indexes for strings in Section 4.4. In the remainder of this chapter, we present the State Set Index – a novel similarity index for search with edit distance.

In our person data use case, we use different similarity measures for the different attributes. Because we do not only have edit distance, we use a more general approach by Christen et al. [Christen et al., 2009] as attribute similarity index in the experiments in later chapters. For each attribute, they precalculate similarities of values stored in the database in an index. With blocking, the amount of similarity calculations is reduced [Newcombe, 1967]. They also create a traditional index for the occurrences of values in records. To support fast calculation of similarities for unseen values at query time, another index for blocking keys is created. With these indexes, similarity queries can be efficiently answered for various attribute similarity measures.

4.2 State Set Index

The State Set Index (SSI) is an efficient and configurable index structure for similarity search in very large string sets. In this section, we first describe the key ideas of SSI before giving details on the indexing and searching algorithms.

The main advantage of SSI in comparison to earlier work is that it has a small memory footprint while providing fast query execution times on small distance thresholds at the same time. In particular, we extend and improve TITAN [Liu et al., 2008], a trie index that is interpreted as a nondeterministic finite automaton (NFA). We give an overview on previous work and show an empirical comparison in Sections 4.3 and 4.4. The concepts of SSI have been developed by Dandy Fenz in his Master's thesis supervised by the author of this thesis [Fenz, 2011]. The contents of this and the following sections are based on our co-authored paper [Fenz et al., 2012].

4.2.1 Definitions

Let Σ be an alphabet. Let s be a string in Σ^*. A substring of s, denoted by $s[i \dots j]$, starts at position i and ends at position j. We call $s[1 \dots j]$ *prefix*, $s[i \dots |s|]$ *suffix* and $s[i \dots j], (1 \leq i \leq j \leq |s|)$, *infix* of s. Any infix of length $q \in \mathbb{N}$ is called *q-gram*.

As we have pointed out in Section 3.1, there exist several techniques to measure the similarity of two strings. For the scope of this chapter, we focus on *edit distance.*

Definition 4.2 (Edit distance [Levenshtein, 1966])**.** *The edit distance $d_{ed}(s_1, s_2)$ of two strings s_1, s_2 is the minimal number of insertions, deletions, or replacements of single symbols needed to transform s_1 into s_2. Two strings are within edit distance k iff $d_{ed}(s_1, s_2) \leq k$.*

Variations of edit distance apply different costs for the three edit operations. While SSI is applicable to all edit distance-based measures with integer costs for the different edit operations, we only consider the standard definition with equal weights in this chapter.

Our goal is to create a similarity index[1] for finding all similar strings:

Definition 4.3 (String similarity index). *Given a query string q, a bag S of strings, and a threshold k, a string similarity index efficiently determines all $s \in S$ for which $d_{ed}(q, s) \leq k$.*

The construction of *prefix or suffix trees* is a common technique for string search. In the literature, such a tree is also called *trie* [Fredkin, 1960].

Definition 4.4 (Trie [Fredkin, 1960]). *A trie is a tree structure (V, E, v_r, Σ, L), where V is the set of nodes, E is the set of edges, v_r is the root node, Σ is the alphabet, and $L : V \to \Sigma^*$ is the labeling function that assigns strings to nodes. For every node v_c with $L(v_c) = s[1 \dots n]$ that is a child node of v_p, it holds $L(v_p) = s[1 \dots n-1]$, i. e., any parent node is labeled with the prefix of its children.*

The trie root represents the empty string. The descendants of a node represent strings with a common prefix and an additional symbol from the alphabet. A trie is processed beginning at the root node. Indexed strings are attached to the node that is reached by processing the complete string. Tries are an efficient method for exact string search.

For efficient similarity search, a trie can also be interpreted as a *nondeterministic finite automaton* [Liu et al., 2008].

Definition 4.5 (Nondeterministic finite automaton (NFA) [Rabin and Scott, 1959]). *A nondeterministic finite automaton is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the input alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the state transition function (with ε referring to the empty word), q_0 is the start state, and F is the set of accepting states.*

The NFA begins processing in the start state q_0. The input is processed symbol-wise with the state transition function. An NFA is allowed to have several active states at the same time. If, after processing the entire string, the NFA is in at least one accepting state, the string is accepted, otherwise rejected.

For similarity search, we interpret a trie as an NFA. In the NFA version of the trie, the trie root node is the start state of the NFA. The nodes with associated result strings are marked as accepting states. The trie's edges are interpreted as state transitions with reading symbols. In addition, the NFA version contains for each state transition one additional state transition for reading ε as well as one ε-transition from each state to itself. These ε-transitions allow state transitions that simulate deletion, insertion, and replacement of symbols as necessary for edit distance calculation. To do similarity search with the NFA, the query string is processed as input symbol sequence by the NFA. After the processing step, the NFA is in zero, one, or more accepting states. The query result contains all strings that are attached to the reached accepting states.

The NFA idea described so far generates for a large amount of indexed strings a large automaton with many states (but there can be no false positives in the result string set). In the next section, we describe our approach that restricts the number of NFA states and checks the result string set for false positives.

[1]Strictly speaking, SSI and all other string similarity indexes that require an absolute distance specified in the query (rather than a relative distance) cannot answer an attribute similarity query as given in Definition 4.1. However, those indexes can still be used as a similarity index in the subsequent chapters as we only rely on the property of lowering retrieval thresholds by some steps, a concept that can also be applied to absolute distance values, e. g., by increasing the edit distance by some steps. For illustration purposes, we follow the notion of relative retrieval thresholds in Chapters 5 ff.

4.2.2 Index Structure

SSI is based on a trie that is interpreted as an NFA. In the following, we describe the key ideas behind SSI that go beyond the basic trie and NFA concepts described above.

State Labeling

The SSI states are labeled with numbers. Each label is calculated from the history of read symbols. For this purpose, the original input alphabet, in the following referred to as Σ_I, is mapped to a labeling alphabet $\Sigma_L = \{1, 2, \ldots, c_{max}\} \subset \mathbb{N}$ with $c_{max} \leq |\Sigma_I|$. A mapping function $m : \Sigma_I \rightarrow \Sigma_L$ defines the mapping of symbols from the two alphabets. Note that the symbols from the labeling alphabet can be interpreted as natural numbers and thus be used for calculation. A label for a state with read symbols $s_1 \ldots s_n \in \Sigma_L^n$ is calculated as follows:

$$\begin{aligned} l(s_1 \ldots s_{n-1} s_n) &= l(s_1 \ldots s_{n-1}) \cdot c_{max} + s_n \\ l(\varepsilon) &= 0 \end{aligned}$$

with ε referring to the empty word.

Our labeling strategy is *history-preserving*, i. e., given a label, the read characters from the labeling alphabet can be unambiguously reconstructed. The inverse of function l determines for a label n the original input character sequence and is given as follows:

$$\begin{aligned} l^{-1}(n) &= l^{-1}(\lfloor n/c_{max} \rfloor) \circ (n \bmod c_{max}) \\ l^{-1}(0) &= \varepsilon \end{aligned}$$

where $\circ$ is the concatenation operator.

Restriction of Labeling Alphabet Size

SSI allows to restrict the size of the labeling alphabet Σ_L. When choosing a labeling alphabet with $|\Sigma_L| < |\Sigma_I|$ (note the strict "less than" sign), at least two symbols from the input alphabet are mapped to the same symbol from the labeling alphabet.

This can reduce the number of existing states in the resulting NFA. For any two prefixes p_1, p_2 of two indexed strings, the states $l(p_1), l(p_2)$ are merged iff $l(p_1) = l(p_2)$. This is the case iff for at least one symbol position pos in p_1 and p_2, it holds $p_1[1 : pos - 1] = p_2[1 : pos - 1]$ and $p_1[pos] \neq p_2[pos]$ and $m(p_1[pos]) = m(p_2[pos])$, i. e., two different symbols at the same position are mapped to the same symbol in the labeling alphabet and the prefixes of the strings before this symbol match.

Depending on the selected mapping, a state may contain several different strings. With $|\Sigma_L| < |\Sigma_I|$, it is not always possible to reconstruct a string from a state label, as there are several different possibilities for that. Thus, we need to store which strings are stored at which state. In addition, the accepting states may contain false positives, i. e., strings that are not part of the correct query result. This makes it necessary to check all resulting strings by calculating the exact distance to the query string before returning results.

Choosing a labeling alphabet size is thus an important parameter for tuning SSI. A too large labeling alphabet size results in a large NFA with many states and thus large storage requirement, but few false positives. In contrast, with a too small alphabet size, the NFA has only few states, but a large number of attached strings per state; the consequence is a large number of false positives.

Restriction of Index Length

SSI allows to restrict the number of indexed symbols. From each string $s_1 \dots s_n \in \Sigma_I^n$, only a prefix with a maximum length of ind_{max} is indexed. The leaf states contain all strings with a common prefix.

Restricting the index length can reduce the overall number of existing states. For any two strings g_1 and g_2, the two states $l(g_1)$ and $l(g_2)$ are equal iff $l(g_1[1 : ind_{max}]) = l(g_2[1 : ind_{max}])$.

Similar to choosing the labeling alphabet size, we face the challenge of handling possible false positives in the result string sets also for restricted index length. The index length is thus a second parameter to tune the trade-off between a large index (large index length, high memory consumption) and a large number of false positives to be handled (small index length, low memory consumption). In our analysis of a large string data set with the Latin alphabet (and some special characters) as input alphabet, we observed optimal results with a (surprisingly small) labeling alphabet size of 4 and an index length of 14 (see Section 4.3.1 for a discussion of this experiment).

Restricting the labeling alphabet size as well as the index length can significantly decrease the number of existing states. For example, in a data set with 1 million names, the mapping from the Latin alphabet to a labeling alphabet with 4 symbols and restricting the index length to 14 results in a state count reduction from 5,958,916 states to 2,298,209 states (a reduction ratio of 61 %).

Storing States

Due to the history-preserving state labels, all potential successors of a state can be calculated. With a calculated state label, the existence of such a state can be determined as follows: For any state ϕ, the state transition with the symbol $c \in \Sigma_L$ by definition exists iff $\phi_c = \phi \cdot |\Sigma_L| + c$ exists. This is because for any symbol $c' \in \Sigma_L \setminus \{c\}$, it holds $c \neq c'$ and thus $\phi_{c'} = \phi \cdot |\Sigma_L| + c' \neq \phi_c$.

To benefit from this observation, SSI only stores which states actually exist, i. e., only states ϕ for which there is at least one prefix p of a string in the indexed string data set with $l(p) = \phi$. This reduces the necessary storage capacity, because it is not necessary to store state transitions. Also, during query answering, checking the existence of state transitions is not required.

Because SSI state labels are numbers, a simple storage format can be defined. A bitmap, where each bit combination represents a label of an existing or non-existing state, is sufficient to store which states do exist.

Storing Data

Due to the introduced restrictions, an accepting state may refer to multiple, different strings – the strings cannot be completely reproduced from the state labels. Thus, it is necessary to store the strings attached to the states. The required data store has a rather simple interface: A set of keys (the accepting states), each with a set of values (the strings referred to by the states) needs to be stored. Any key/multi-value store is suitable for this task. Since the data store is decoupled from the state store, the data store can be held separately. Thus, while the state store can be configured to be small or large enough to fit into main memory, the data store can be held in secondary memory.

4.2.3 Algorithms

In the following, we describe the details for indexing a large string set with SSI and searching with the created index.

Algorithm 4.1 Indexing with SSI

```
Input: set of strings to be indexed $stringSet$,
    labeling alphabet size $c_{max}$,
    index length $ind_{max}$,
    mapping function $m : \Sigma_I \to \Sigma_L$
Output: set of existing states $stateSet$,
    set of accepting states $acceptingStateSet$,
    map of states with indexed strings $dataStore$
 1: $stateSet \leftarrow \{\}$
 2: $acceptingStateSet \leftarrow \{\}$
 3: $dataStore = \{\}$
 4: for all $str \in stringSet$ do
 5:     $state \leftarrow 0$
 6:     for $pos = 1 \to min(ind_{max}, |str|)$ do
 7:         $state \leftarrow state \cdot c_{max} + m(str[pos])$
 8:         $stateSet.add(state)$
 9:     $acceptingStateSet.add(state)$
10:     $dataStore.add(state, str)$
11: return $stateSet, acceptingStateSet, dataStore$
```

Indexing

The indexing process is shown as Algorithm 4.1. All strings to be indexed are processed one after another. Each string is read symbol-by-symbol. After reading a symbol, the current state is calculated and stored. Finally, after reading the entire string, the active state is marked as an accepting state and the string is stored at this state's entry in the data store. After the initial indexing process, it is also possible to index additional strings using the same steps.

Example. Consider the strings Müller, Mueller, Muentner, Muster, and Mustermann and the alphabet mapping shown in Table 4.1. In this example, we chose a labeling alphabet size of $c_{max} = 4$ and an index length of $ind_{max} = 6$.

Σ_I	M	u	e	l	r	ü	n	t	s	m	a
Σ_L	1	2	3	4	1	2	3	4	1	2	3

Table 4.1: Example for alphabet mapping function $m : \Sigma_I \to \Sigma_I$

Figure 4.1 illustrates all existing states and possible state transitions of the resulting index. A reading example for a state transition is the following: We have reached the state labeled 6 after reading the symbols 1 and 2. From state 6, we can perform a transition by reading the symbol 3 and reach the state $6 \cdot 4 + 3 = 27$.

Note that the figure only shows the state transitions that do not add costs by reading a symbol from the input string (as explained below in the description of the searching algorithm) and the ε-transitions. For the sake of clarity, state transitions with other symbols (that do add costs) are not shown. The accepting states point to the indexed strings as follows:

1869 $\to$ {Müller}, 1811$\to$ {Mueller}, 1795$\to$ {Muenter}, 1677 $\to$ {Muster, Mustermann} □

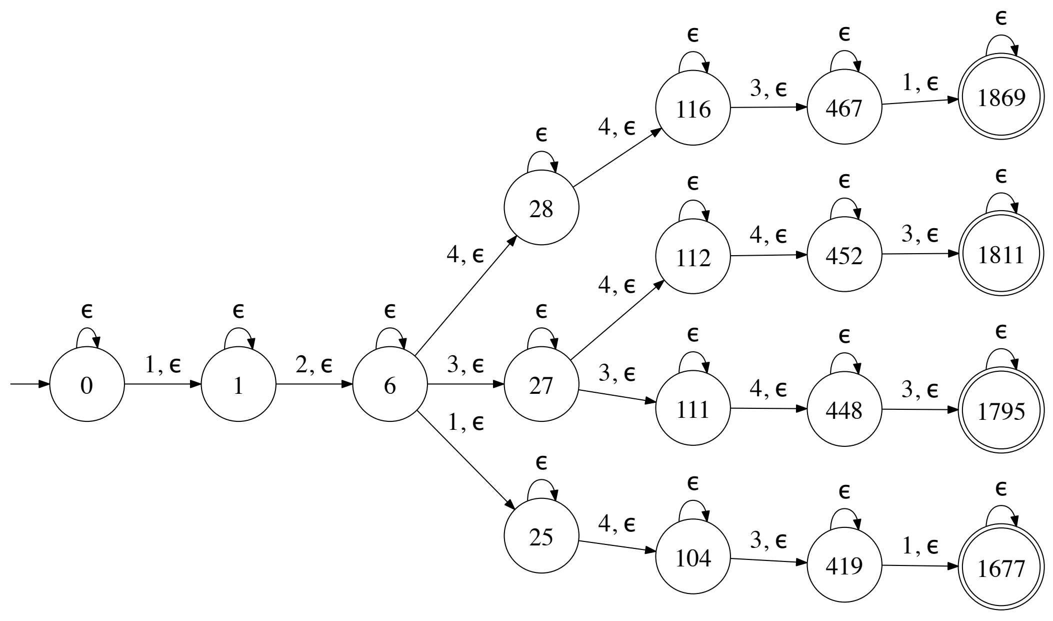


Figure 4.1: Example for states created by SSI with $c_{max} = 4$ and $ind_{max} = 6$

Searching

We now describe how to process a query string $q_I \in \Sigma_I^*$ with edit distance $k \in \mathbb{N}$. The search process is shown as Algorithm 4.2.

First, a set S of cost-annotated states s with state ϕ_s and associated costs λ_s (the number of edit distance operations required so far) is created. We write $s := \langle \phi_s, \lambda_s \rangle$. Initially, all states that can be reached from the start state with at most k ε-transitions are added to S. To determine these states, the labels of the successors of the start state are calculated and their existence is validated.

If a state s in S is associated with several different costs λ_s, only the state with the lowest λ_s is kept; all other states are dismissed. This selection (not shown in the algorithm) is done for all state calculations and is not stated again in the following.

Next, the query string q_I is translated into the labeling alphabet; we call the translated query string q in the following. The symbols of q are processed one-by-one. The following steps are performed for each symbol c in q:

Another empty set S^* of current cost-annotated states is created. For each cost-annotated state $\langle \phi_s, \lambda_s \rangle$ in S, a set S_c^* is created and processed with the following steps:

- To simulate deletion of symbols, the cost-annotated state $\langle \phi_s, \lambda_s + 1 \rangle$ is added to S_c^* if $\lambda_s + 1 \leq k$.
- To simulate matching of symbols, $\langle \phi_s^*, \lambda_s \rangle$ is added to S_c^*.
- Next, substitution of symbols (i. e., matching of symbols other than c) is simulated. If $\lambda_s + 1 \leq k$, then for each $\phi_s^* := \phi_s \cdot |\Sigma_L| + c^*$ with $c^* \in \Sigma_L \setminus \{c\}$, a new cost-annotated state $\langle \phi_s^*, \lambda_s + 1 \rangle$ is added to S_c^*. Note that these state transitions are not shown in Figure 4.1.

Algorithm 4.2 Searching with SSI

```
Input: query string $q$, maximum edit distance $k$
Output: result string set $R$
 1: $S \leftarrow \{\langle i \cdot c, i\rangle \mid 0 \leq i \leq k, c \in \Sigma_L\} \cap stateSet$        Initial $\varepsilon$-transitions
 2: for $pos \leftarrow 1 \rightarrow min(ind_{max}, |q|)$ do
 3:     $S^* \leftarrow \{\}$
 4:     for all $\langle \phi_s, \lambda_s\rangle \in S$ do
 5:         $S^*_c \leftarrow \{\}$
 6:         if $\lambda_s + 1 \leq k$ then                                   Deletion
 7:             $S^*_c \leftarrow S^*_c \cup \langle \phi_s, \lambda_s + 1\rangle$
 8:         for $i = 1 \rightarrow |\Sigma_L|$ do                     Match and substitution
 9:             $\phi^*_s \leftarrow \phi_s \cdot |\Sigma_L| + i$
10:             if $i = m(q[pos])$ then
11:                 $S^*_c \leftarrow S^*_c \cup \langle \phi^*_s, \lambda_s\rangle$
12:             else if $\lambda_s + 1 \leq k$ then
13:                 $S^*_c \leftarrow S^*_c \cup \langle \phi^*_s, \lambda_s + 1\rangle$
14:         for all $\langle \phi'_s, \lambda'_s\rangle \in S^*_c$ do                          Insertion
15:             $S'_c \leftarrow$ all states reachable from $\phi'_s$ with at most $(k - \lambda'_s)$ $\varepsilon$-transitions
16:             $S^*_c \leftarrow S^*_c \cup S'_c$
17:         $S^*_c \leftarrow S^*_c \cap stateSet$                      Evaluation of existence
18:         $S^* \leftarrow S^* \cup S^*_c$
19:     $S \leftarrow S^*$
20: $R \leftarrow \{\}$                              Retrieve strings and filter by distance
21: for all $\langle \phi_s, \lambda_s\rangle \in S$ do
22:     if $\phi_s \in acceptingStateSet$ then
23:         $R \leftarrow R \cup \{s \in dataStore.get(\phi_s) \mid d_{ed}(s, q) \leq k\}$
24: return $R$
```

- Inserting symbols is simulated using ε-transitions. All states ϕ^*_s are determined that can be reached from ϕ_s with at most k ε-transitions. The annotated states $\langle \phi^*_s, \lambda_s + i\rangle$ where i refers to the number of performed ε-transitions to reach ϕ^*_s are added to S^*_c.

Then, states that do not exist in the index are removed from S^*, S is replaced by S^* and all steps are repeated with the next symbol. After processing all symbols, the state set S represents the final state set. For all states from S that are accepting, all strings stored at those states are retrieved. This set of strings is filtered by calculating the actual edit distance to the query string as it may contain false positives. The set of filtered strings is the result of the search.

Example. Consider the index in Figure 4.1 and the example query Mustre with a maximum distance of $k = 2$. The initial state set $S = \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 6, 2\rangle\}$ contains all states reachable from the start state with at most $k = 2$ ε-transitions. The first symbol to be processed is M, which translates to $c = 1$ in the labeling alphabet. The state sets $S^* = S^*_c = \emptyset$ are created. For all entries in S, the five above-described steps are executed. After processing the first symbol, we have:

$$S = \{\langle 0, 1\rangle, \langle 1, 0\rangle, \langle 6, 1\rangle, \langle 28, 2\rangle, \langle 27, 2\rangle, \langle 25, 2\rangle\}$$

After that, $c = 2$ (u) is processed. The state set after this step is:

$$\begin{aligned} S &= \{\langle 0, 2\rangle, \langle 1, 1\rangle, \langle 6, 0\rangle, \langle 28, 1\rangle, \langle 27, 1\rangle, \langle 25, 1\rangle, \langle 116, 2\rangle, \\ &\quad \langle 112, 2\rangle, \langle 111, 2\rangle, \langle 104, 2\rangle\} \end{aligned}$$

After processing the third symbol $c = 1$ (s), we have:

$$\begin{aligned} S &= \{\langle 1,2\rangle, \langle 6,1\rangle, \langle 28,1\rangle, \langle 27,1\rangle, \langle 25,0\rangle, \langle 116,2\rangle, \langle 112,2\rangle, \\ &\quad \langle 111,2\rangle, \langle 104,1\rangle, \langle 419,2\rangle\} \end{aligned}$$

The next symbol $c = 4$ (t) results in:

$$\begin{aligned} S &= \{\langle 6,2\rangle, \langle 28,1\rangle, \langle 27,2\rangle, \langle 25,1\rangle, \langle 104,0\rangle, \langle 116,1\rangle, \langle 112,1\rangle, \\ &\quad \langle 111,2\rangle, \langle 419,1\rangle, \langle 1677,2\rangle, \langle 467,2\rangle, \langle 452,2\rangle\} \end{aligned}$$

The symbol $c = 1$ (r) is processed as follows:

$$\begin{aligned} S &= \{\langle 28,2\rangle, \langle 25,2\rangle, \langle 104,1\rangle, \langle 116,2\rangle, \langle 112,2\rangle, \langle 419,1\rangle, \\ &\quad \langle 1677,1\rangle, \langle 1869,2\rangle, \langle 467,2\rangle, \langle 452,2\rangle\} \end{aligned}$$

With the last symbol $c = 3$ (e), we finally have:

$$S = \{\langle 104,2\rangle, \langle 419,1\rangle, \langle 1677,2\rangle, \langle 467,2\rangle, \langle 1811,2\rangle\}$$

From the set of states in S, only the accepting states 1677 and 1811 are further processed, because strings have been attached only to the accepting states. The strings stored at these states are Muster, Mustermann, and Mueller. After filtering false positives, we finally have the result string set {Muster}.

□

Complexity

To index n strings with a maximum index length ind_{max}, at most ind_{max} states need to be calculated for each string. Thus, we have an indexing complexity of $\mathcal{O}(n \cdot ind_{max})$.

The most important size factor of SSI is the number of created states. For an index length ind_{max} and an indexing alphabet Σ_L, the number of possible states is $|\Sigma_L|^{ind_{max}}$. The index size depends on the chosen parameters where ind_{max} is the dominant exponential parameter.

The search algorithm of SSI mainly depends on Σ_L, ind_{max}, and the search distance k. In the first step, $k \cdot |\Sigma_L|$ potential states are checked. For each existing state, its successor states are evaluated. These consist of up to one state created by deletion, $|\Sigma_L|$ states created by match or substitution, and $k \cdot |\Sigma_L|$ states created by insertion of a symbol. This process is repeated up to ind_{max} times. Overall, we have up to $(k \cdot |\Sigma_L|) \cdot (1 + k \cdot |\Sigma_L| + k \cdot |\Sigma_L|)^{ind_{max}}$ steps and thus a worst-case complexity of $\mathcal{O}((k \cdot |\Sigma_L|)^{ind_{max}})$. Similar to the indexing process, the complexity is bound by the parameters $|\Sigma_L|$ and ind_{max} where ind_{max} is the dominant exponential factor. By evaluating the existence of states during the search process and proceeding only with existing states, we typically can significantly decrease the number of states that are actually evaluated.

4.3 Evaluation

We use a set of person names crawled from the public directory of a social network website to evaluate the performance of SSI for parameter selection, index creation, and for search operations. Table 4.2 shows some properties of our data set. The set D_{full} contains all person names we retrieved, whereas the sets D_i consist of i randomly chosen strings taken from D_{full}. We did not use the Schufa data set from Section 3.3.1 in this experiment, because we want to show effects at a larger scale for SSI.

Set	Number of strings	String length min./avg./max.	$\lvert\Sigma_I\rvert$	Number of duplicates
D_{full}	170,879,859	1 / 13.99 / 100	38	70,751,399
D_{200k}	200,000	1 / 14.02 / 61	29	5,462
D_{400k}	400,000	1 / 14.02 / 54	32	17,604
D_{600k}	600,000	1 / 14.01 / 55	35	35,626
D_{800k}	800,000	1 / 14.02 / 61	33	54,331
D_{1000k}	1,000,000	1 / 14.01 / 64	35	77,049

Table 4.2: Evaluation data sets

First, we evaluate the impact of different parameter configurations on the performance of SSI and then choose the best setting to compare SSI against four competitors. In particular, we compare SSI to FastSS [Bocek et al., 2007], TITAN [Liu et al., 2008], Flamingo [Behm et al., 2011], and Pearl [Rheinländer and Leser, 2011], which are all main memory-based tools for index-based string similarity search (see Section 4.4 for details). For Flamingo and Pearl, we use the original implementations provided by the authors. For FastSS and TITAN, we use our own implementations of the respective algorithms. Our evaluation comprises experiments both for indexing time and space as well as experiments on exact and similarity search queries.

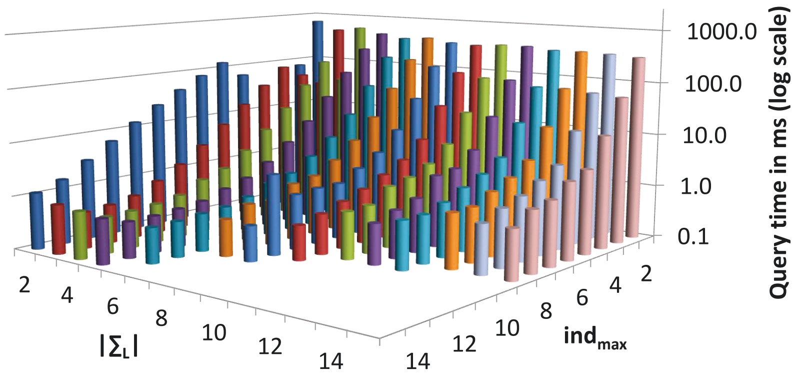
All experiments were performed on an Intel Xeon E5430 processor with 48 GB RAM available using only a single thread. For each experiment, we report the average of three runs.

4.3.1 Evaluation of SSI Parameters

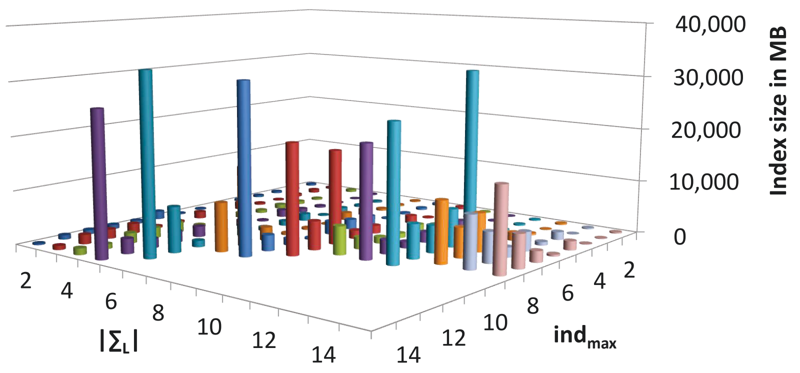
We exemplarily used the set D_{1000k} to evaluate the impact of different parameter configurations on the performance of SSI on small string sets. Since the maximum index length ind_{max} and labeling alphabet size $|\Sigma_L|$ have a large influence on the performance of SSI, we varied both ind_{max} and $|\Sigma_L|$ in the range of 2 to 15. We could not perform experiments on larger parameter ranges due to memory constraints of our evaluation platform.

As shown in Figure 4.2a, the average query execution time drastically decreases with increased labeling alphabet size and maximum index length. In particular, a configuration of SSI with $ind_{max} = 12$ and $|\Sigma_L| = 8$ outperforms a configuration using $ind_{max} = 2$ and $|\Sigma_L| = 2$ by three orders of magnitude (factor 1521). On the other hand, when increasing ind_{max} and $|\Sigma_L|$, we observed that the index size grows significantly (see Figure 4.2b). For example, changing the configuration from $ind_{max} = 2$ and $|\Sigma_L| = 2$ to $ind_{max} = 6$ and $|\Sigma_L| = 13$ increases memory requirements by a factor of 60. We also observed that the number of false positives and the number of accessed keys per query decreases both with increasing ind_{max} and $|\Sigma_L|$ (data not shown) and conclude that this is the main reason for the positive outcome of a large labeling alphabet and a large index length. However, we could not increase both parameters further due to memory limitations of our platform; we expect a further decrease of query execution time.

We also evaluated the influence of varying parameters on query execution time and index size on D_{full}. Results are shown in Figure 4.3 for selected configurations. Similar to the experiments on D_{1000k}, the index size grows heavily while increasing ind_{max} and $|\Sigma_L|$. Particularly, choosing $|\Sigma_L| = 3$ and $ind_{max} = 15$ yields in an index size of approximately 12 GB, whereas a configuration with $|\Sigma_L| = 5$ and $ind_{max} = 15$ needs an index of 28 GB. On the other hand, the query execution time decreases with elongating ind_{max} and $|\Sigma_L|$. Using $|\Sigma_L| = 3$ and $ind_{max} = 15$, the query execution time averages to 8 milliseconds, whereas with $|\Sigma_L| = 5$ and $ind_{max} = 15$ the query execution time diminishes to 2.6 milliseconds on average at the expense of a very large



(a) Average query execution time (log-scale)



(b) Average index size in MBytes

Figure 4.2: Evaluation of parameters ind_{max} and $|\Sigma_L|$ for D_{1000k} and $k = 1$.

index. We also experimented with other settings of ind_{max} and $|\Sigma_L|$ in the range of 2 to 15, but these configurations either did not finish the indexing process in a reasonable amount of time or ran out of memory on our evaluation platform. Therefore, we did not consider these settings for parameter configuration on large string sets.

In summary, both parameter variations of $|\Sigma_L|$ and ind_{max} have a large impact on the performance of SSI. While increasing $|\Sigma_L|$ or ind_{max}, the number of false positive results that need to be verified decreases, which yields in a considerably fast query response time. However, our experiments also revealed that at some point, no further improvements on query response time can be achieved by increasing $|\Sigma_L|$ and ind_{max}. This is caused by an increased effort for calculating involved final states that outweighs the decreased amount of false-positive and the number of lookups in this setting. We decided to configure SSI with $|\Sigma_L| = 4$ and $ind_{max} = 14$ for all following experiments using D_{full}, since this configuration gives us the best query execution time with an index size of at most 20 GB.

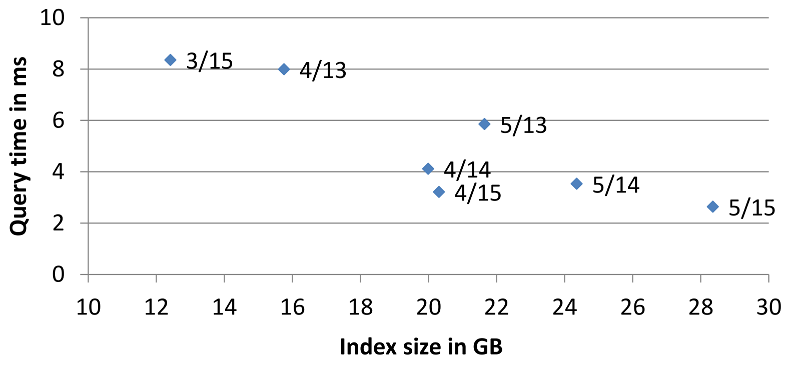


Figure 4.3: Trade-off between index size and query execution time on D_{full} and $k = 1$ on varying configurations of $ind_{max} \in \{3, 4, 5\}$ and $|\Sigma_L| \in \{13, 14, 15\}$.

4.3.2 Index Creation Time and Memory Consumption

We evaluated SSI in terms of index creation time and memory consumption and compared it to other main-memory indexes, namely FastSS, TITAN, Pearl, and Flamingo on all available data sets. For all evaluated tools, we observe that both index sizes and indexing time grow at the same scale as the data sets.

Three the indexes we compared to are not able to handle very large string collections. In Figure 4.4, the memory consumption of each created index in main memory is shown. We were able to index D_{full} only with SSI and Flamingo; FastSS, Pearl, and TITAN ran out of memory during index creation. In particular, FastSS even failed to create indexes with more than 400,000 strings. Another severe drawback of FastSS is that it needs to create a separate index for each edit distance threshold k – in contrast to all other evaluated tools.

Clearly, SSI outperforms all other trie- or NFA-based tools in terms of memory consumption and outperforms FastSS, Pearl, and TITAN by factors in the range of 1.4 (Pearl on D_{200k}) to 4.5 (Pearl on D_{1000k}). Compared to Flamingo, which is based on indexing strings by their lengths and char-sums, SSI is only advantageous for indexing large data sets. When indexing D_{full}, SSI needs 3.0 times less memory than Flamingo. For small data sets with up to one million strings, Flamingo outperforms SSI by factors in the range of 2.4 (D_{1000k}) to 5.0 (D_{200k}).

We also evaluated SSI on the time spent for index creation. As shown in Figure 4.5, SSI indexes all data sets significantly faster than the other trie- or NFA-based methods. It outperforms FastSS with factors 3.2 to 3.7 on $k = 1$, TITAN with factors 4.0 to 4.7, and Pearl with factors 7.4 to 9.6. Similar to the memory consumption, SSI is the more superior the larger the data sets grow. Compared to Flamingo, SSI is only slightly slower (factors in the range of 1.4 to 2.0).

4.3.3 Query Answering

To evaluate the performance of SSI in query answering, we assembled a set of 1,000 queries separately for each data set as follows: First, we randomly selected 950 strings from the respective data set and kept 500 of these strings unchanged. On the remaining 450 strings, we introduced errors by randomly changing or deleting one symbol per string. Additionally, we generated 50 random strings and added them to the set of queries. For each query, we measured the execution time and report the average of all 1,000 queries. We compared SSI to all above-mentioned tools both for exact and similarity queries with varying edit distance thresholds $k \in \{0, 1, 2, 3\}$. For

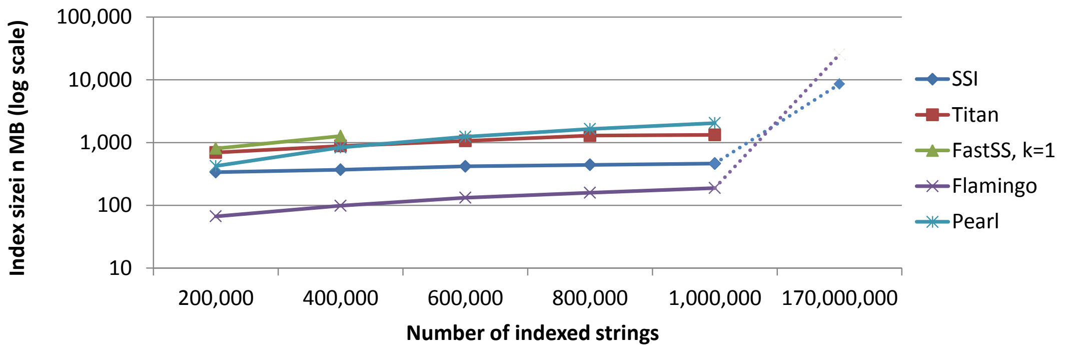


Figure 4.4: Average index size in MB (log-scale)

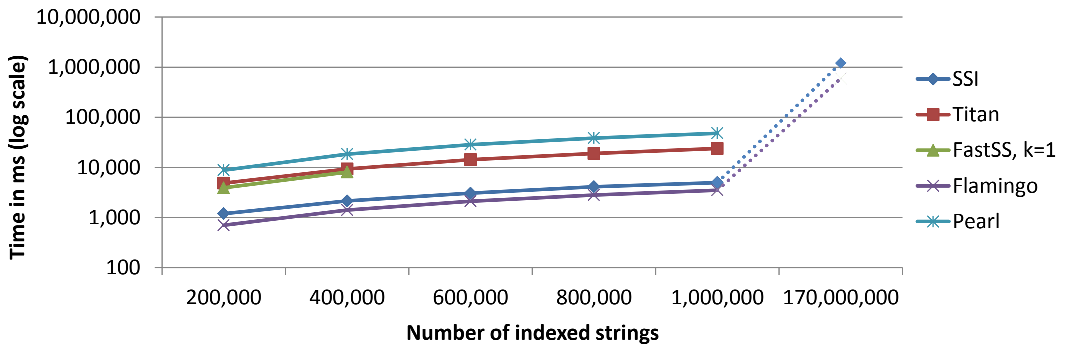


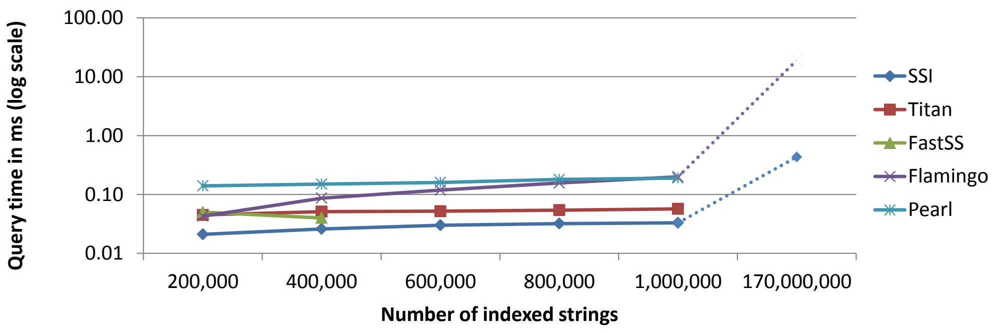
Figure 4.5: Average index creation time in milliseconds (log-scale)

all search experiments, indexing was performed in advance and is not included in the measured times. Results are shown in Figure 4.6.

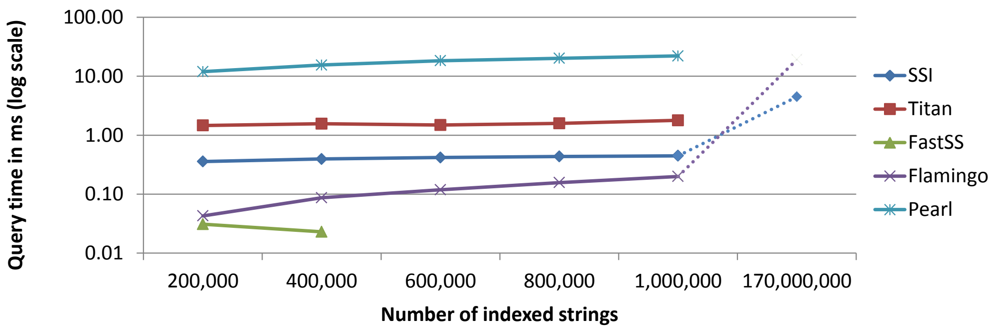
For exact queries, SSI outperformed all competitors independent of the data set size (see Figure 4.6a). Specifically, SSI outperformed FastSS with factor 2.3 on D_{200k} and factor 1.5 on D_{400k}, TITAN with factors varying between 1.6 on D_{800k} and 2.1 on D_{200k}, Pearl with factors varying between 5.3 on D_{600k} and 6.6 on D_{200k}, and Flamingo with factors from 2.0 on D_{200k} to 44.1 on D_{full}.

Figures 4.6b – 4.6d show that SSI significantly outperforms the trie- and NFA-based tools TITAN and Pearl on queries with larger distance thresholds. Using an edit distance threshold of $k = 1$, SSI outperforms TITAN with a factor of 4, using $k = 3$, SSI is 5.4 to 7.4 times faster than TITAN depending on the data set. Compared to Pearl, SSI is faster by more than one order of magnitude, independent of the data set and the edit distance thresholds. However, on the smaller data sets (D_{200k}, D_{400k}), FastSS is by an order of magnitude faster than SSI. This observation needs to be put into perspective, since FastSS on the one hand needs to create a separate index for each k, and creating indexes with more than $400,000$ strings was not possible using FastSS. In contrast, SSI does not have these limitations.

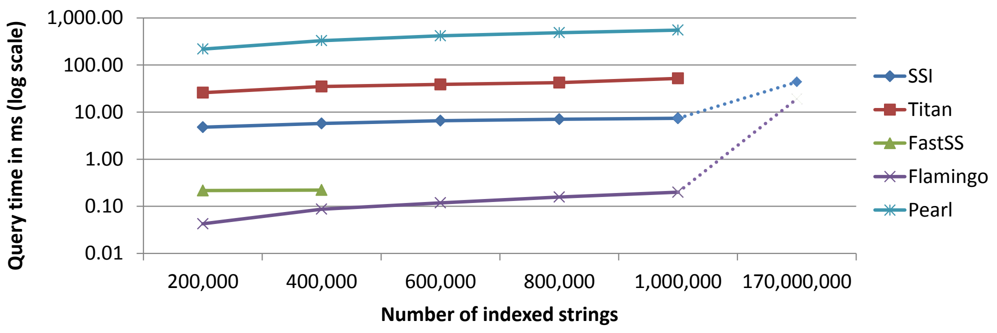
Furthermore, we acknowledge that Flamingo, which has a different indexing and search approach (cf. Section 4.4), is significantly faster than SSI in many situations. For searches in D_{full} with $k = 0$ and $k = 1$, SSI was faster by a factor of 4.2, in all other situations, Flamingo outperformed SSI. Recall that Flamingo uses considerably more memory than SSI for indexing D_{full} to achieve this (cf. Figure 4.4). We also clearly observe that the advantages of Flamingo grow the larger edit distance thresholds get. However, future improvements of SSI could directly



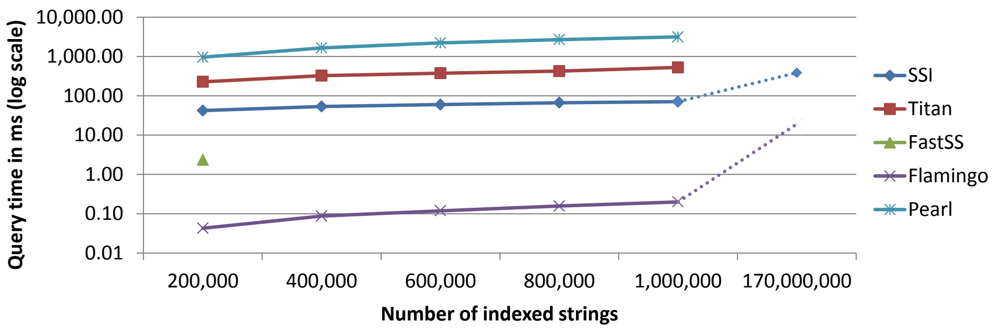
(a) k=0



(b) k=1



(c) k=2



(d) k=3

Figure 4.6: Average query execution time in milliseconds (log-scale)

address this issue, e. g., by integrating bit-parallel edit distance computation methods which provide a fast edit distance computation that is independent of the chosen threshold k.

4.4 Related Work

In the past years, the research community has spent much effort on accelerating string similarity search. Prominent approaches use prefiltering techniques, indices, refined algorithms for computing string similarity, or all in combination [Navarro, 2001].

Filter methods are known to reduce the search space early using significantly less computational effort than computing the edit distance (or another similarity measure) directly. As a result of this so-called *filter-and-verify* approach [Xiao et al., 2008], only a few candidate string pairs need to be compared using edit distance. Prominent pre-filtering approaches are based on q-grams [Gravano et al., 2001, Gravano et al., 2003, Li et al., 2008], character frequencies [Aghili et al., 2003], or length filtering [Behm et al., 2011].

Tries as indexes for strings and exact string matching in tries were first introduced by Morrison [Morrison, 1968] and later extended by Shang and Merrett with pruning and dynamic programming techniques to enable string similarity search [Shang and Merrett, 1996]. In addition to string similarity search, tries and trie-based NFAs are also known to perform well in other areas, such as exact pattern matching [Gusfield, 1997], set joins [Jampani and Pudi, 2005], or frequent item set mining [Grahne and Zhu, 2003, Han et al., 2004b].

The Peter index was designed for near-duplicate detection in DNA data and combines tries with filtering techniques to enable string similarity search and joins [Rheinländer et al., 2010]. It stores additional information at each trie node for early search space pruning. Pearl is a follow-up where restrictions on small alphabets were removed and a strategy for parallelizing the execution of similarity queries and joins was introduced [Rheinländer and Leser, 2011]. As basis of SSI, we use TITAN [Liu et al., 2008], an index structure based on prefix trees that are converted into non-deterministic automata A, such that the initial state of A corresponds to the root node of the originating prefix tree, and leaf nodes correspond to accept states in A. Additionally, further state transitions are introduced in order to enable delete, insert, and replacement operations on edit distance-based queries. In Section 4.2.2, we described the improvements of SSI over TITAN in detail.

Algorithms based on neighborhood generation were first used for string similarity search by Myers [Myers, 1994]. One drawback of the original algorithm by Myers is its space requirement, which makes it feasible only for small distance thresholds and small alphabets. The FastSS index captures neighborhood relations by recursively deleting individual symbols and reduces space requirements by creating a so-called k-deletion neighborhood [Bocek et al., 2007]. Similar to filtering approaches, FastSS performs search space restriction by analyzing the k-deletion neighborhood of two strings. By adding partitioning and prefix pruning, Wang et al. significantly improved the runtime of similarity search algorithms based on neighborhood generation [Wang et al., 2009].

The Flamingo package provides an inverted-list index that is enriched with a charsum and a length filter [Behm et al., 2011]. The filter techniques are organized in a tree structure, where each level corresponds to one filter.

We empirically compare the SSI to FastSS, Flamingo, Pearl, and TITAN, and show that SSI often outperforms these tools both in terms of query execution time and with respect to index size (see Section 4.3). We could not compare to the approach by Wang et al. [Wang et al., 2009] since no reference implementation was available.

4.5 Conclusion

In this chapter, we presented the State Set Index (SSI), a solution for fast similarity search in very large string sets. By configuring the parameters of SSI, we can scale the index size allowing best search performance given memory requirements. Our experiments on a very large real-world string data set showed that SSI significantly outperforms current state-of-the-art approaches for string similarity search with small distance thresholds.

From a bird's eye view, SSI is an exemplary similarity index for string attributes. As we have pointed out, there exist many different similarity indexes suitable for different attributes and similarity measures. The topic of the following chapters is how to combine these attribute similarity indexes into an overall search system.

5 Query Planning for Similarity Search

Previously, we have defined a set of base similarity measures for the attributes and a composed similarity measure for the overall similarity. We have also introduced different similarity index structures that provide efficient access to sets of records with similar values for the base similarity measures. In this chapter, we describe a technique that uses the similarity indexes for answering similarity queries specified with the overall similarity measure.

To implement an efficient similarity search system, we suggest to use *query plans*. The query plan describes which similarity indexes are to be used and which retrieval thresholds are to be applied. Only records that fulfill the query plan conditions are compared to the query record. We use the term query plan (to access the similarity indexes) in analogy to query plans in database systems (to access tables and traditional indexes).

We focus in this chapter on range queries, while the subsequent chapter considers top-k queries.

Definition 5.1 (Range Query)**.** *Given a record set $R \subseteq U$, a query record $q \in U$, and a minimum similarity threshold $\theta_{Overall}$ (the range),* a range query *retrieves a set S of records from R with*

$$S = \{r \in R \mid sim_{Overall}(r, q) \geq \theta_{Overall}\}$$

In this chapter, we introduce query plans as a means for efficiently executing similarity queries on large data sets (Section 5.1). The selection of a query plan can be performed in advance; such a static plan is then used for all queries (Sections 5.2 and 5.3). Alternatively, a plan can be selected individually for each query at query time (Sections 5.4 to 5.6). For both approaches, we define result completeness and execution cost as metrics to evaluate query plans and we propose algorithms to derive good query plans in a general setting.

5.1 Query Planning

We first give an overview of our search setting as well as the query planning approaches discussed in this chapter and then introduce required definitions.

5.1.1 Filters and Planning Approaches

To perform efficient similarity search with an arbitrarily composed similarity measure, we apply a filter-and-refine approach: Given a query, we first filter the entire set of records to derive a set of probably relevant records. For this step, we need one similarity index for each base similarity

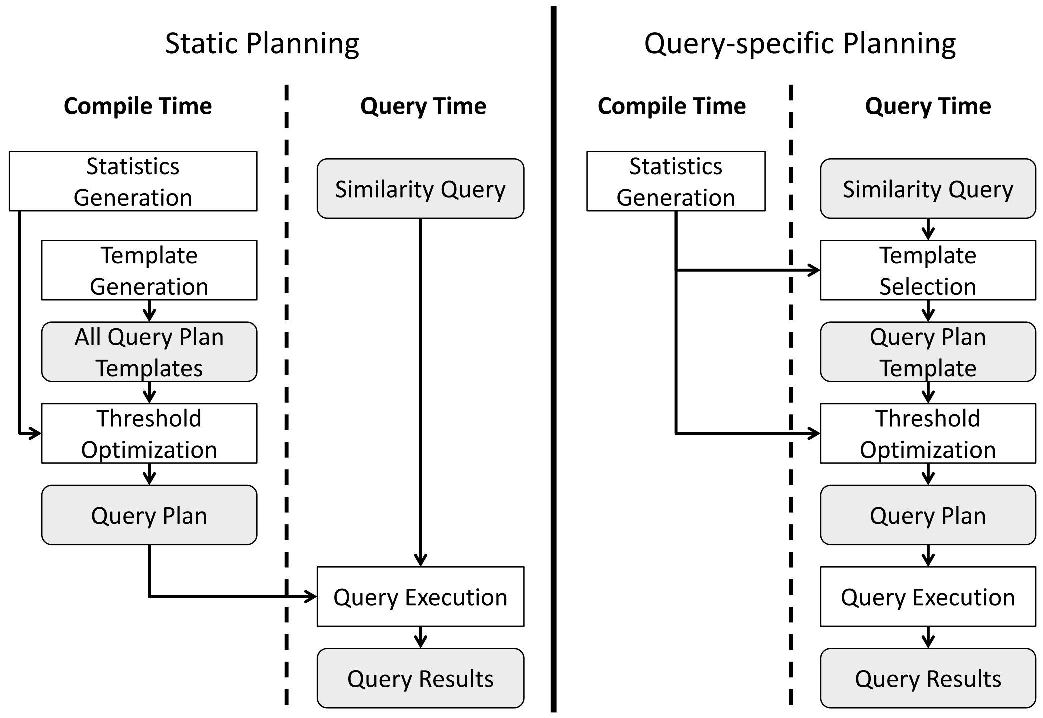


Figure 5.1: Comparison of static and query-specific planning

measure in the filter. We apply filter criteria on these indexes (e. g., *all records with a similar* FirstName as well as *all records with a similar* LastName) and then combine the filtered sets (e. g., the *intersection* of records with similar FirstName and LastName).

In the refine phase, we calculate the exact similarity (with the combined similarity measure) of the query for each of the records that survived the filtering. The result then contains the set of records above the threshold for overall similarity ($\theta_{Overall}$).

The filter criterion (which operates on the base similarity measures) is an approximation of the overall similarity measure (which composes the base similarity measures). The key to success is to optimize the filter criterion – it should be as concise, but as complete as possible. For as many cases as possible, the filtered list of records should contain all sufficiently similar records; the number of missing similar records should be as low as possible. Additionally, this list should be as short as possible, i. e., the number of incorrect matches that are unnecessarily compared to the query record should be as low as possible. As mentioned above, we refer to the filter criterion as query plan. The main focus of this chapter is how to select the best plan.

We propose two different query planning methods and compare them in Figure 5.1:

– **Static planning:** In the static planning approach, we first gather statistics about the data and generate a set of query plan templates (i. e., combinations of attributes in disjunctive normal form, see below for details). We optimize each query plan template's thresholds and handle the trade-off between result completeness and execution cost. We select the overall best query plan based on these criteria. This static query plan is determined in advance as a good choice for an *average* query and is used for *all* queries.

  Static planning has the advantage of no additional preparation cost at query time as the pre-determined plan can immediately be parameterized with the search record and executed. However, the plan may not be the best choice for all queries and can have a large variance in query runtime. In particular, query values that occur very frequently in the database can lead to a very long query runtime, which may be unacceptable. Static planning is the topic of Sections 5.2 and 5.3.

– **Query-specific planning:** For query-specific planning, we prepare some statistics about the data at compile time, similar to the static planning approach. At query time, for each query we first select the best query plan template. In the next step, we optimize the query plan thresholds to maximize completeness. We then execute the resulting query-specific plan.

  Query-specific planning requires optimizations of templates and thresholds and thus more calculations at query time. On the other hand, by analyzing the attribute values in the query, we can better adjust the plan to the requirements of individual queries. For very frequent query values, higher similarity thresholds can be selected, resulting in a more strict query plan (than the average query plan). Vice versa, for very rare query values, we can be less strict and apply lower thresholds so that we better exploit the allowed cost range (and possibly improve completeness of the results). Query-specific planning is discussed in Sections 5.4 to 5.6.

5.1.2 Definitions

In Section 3.1.1, we have defined base similarity measures (Definition 3.1) and composed similarity measures (Definition 3.2) as the basis of our similarity search problem. For the algorithms discussed in this chapter, the composed similarity measure must be monotonous (Definition 3.3). As in the previous chapters, we refer to the universe of possible records as U, the given record set as $R \subseteq U$, and the query record as $q \in U$.

Definition 5.2 (Predicate)**.** *A* base similarity measure predicate $sim_a(q,r) \geq \theta_a$ covers *all records $r \in R$ for which the similarity to the query record $q \in U$ calculated with the base similarity measure sim_a is at least θ_a. In the following, we abbreviate this predicate to $a \geq \theta_a$ and refer to base similarity measure predicates as (attribute)* predicates*.*

Definition 5.3 (Query Plan)**.** *A* query plan *is a combination of predicates with the logical operators conjunction and disjunction in disjunctive normal form (DNF). A query plan p* covers *all records $r \in R$ for which there is at least one conjunction c in p where r is covered by all predicates in c.*

Query plans are in DNF, since this form is popular [Bilenko et al., 2006, Chaudhuri et al., 2007] as well as easy to understand and modify. Note that all logical combinations of attributes can be expressed in DNF. A query plan combines N attribute predicates and has the form:

$$\bigvee \bigwedge a_i \geq \theta_{a_i}$$

with $1 \leq i \leq N, \forall i : 0 \leq \theta_{a_i} \leq 1$. For clarification, we note that our query plan is a technical means for efficient execution of queries and is not to be defined by a user.

An example for a query plan that covers all records with similar FirstName and Zip or LastName and BirthDate is the following:

$$(\mathsf{FirstName} \geq 0.9 \wedge \mathsf{Zip} \geq 1.0) \vee (\mathsf{LastName} \geq 0.8 \wedge \mathsf{BirthDate} \geq 0.85)$$

Definition 5.4 (Query Plan Template)**.** *A* query plan template *is a query plan with unspecified thresholds for its predicates.*

As *default* threshold assignment of a query plan template, we set all thresholds to 1.0, i. e., we perform an exact search on all attributes. We use this default assignment in our query-specific planning approach to select a good template before optimizing the thresholds.

To evaluate query plans, we define different performance metrics.

Definition 5.5 (Completeness)**.** *The* completeness *(comp) of a query plan p is the proportion of records $r \in R$ with $sim_{Overall}(q, r) \geq \theta_{Overall}$ that are covered by p.*

Our goal is to find as many sufficiently similar records as possible. The definition of completeness resembles the common definition of the recall measure; because of the usual usage of recall for evaluation purposes, we prefer the term completeness as a property of the internally used query plans.

Definition 5.6 (Cost)**.** *The* cost *of a query plan p for a query q is the number of records in R that are covered by p.*

For each covered record, an expensive calculation of its overall similarity to the query record is required. To limit query execution time, cost must be less than or equal to a cost threshold; only query plans that fulfill this property are *valid* plans. The cost threshold is specified by the user in advance.

In this chapter, we describe approaches to the following problems:

- Given a set of records and a cost limit, the *static planning problem* is to determine the query plan that maximizes average completeness where the average cost are below the cost limit.

- Given a set of records, a query, and a query-specific cost limit, the *query-specific planning problem* is to determine the query plan that maximizes completeness where the cost are below the query-specific cost limit.

5.2 Static Query Planing

In this section, we present an approach for selecting a static query plan, i. e., a query plan that is used for all queries. We begin the section with a description of how to evaluate a query plan. We show how to estimate completeness and cost of a query plan in Sections 5.2.1 and 5.2.2. After that, we propose an algorithm to find the best static query plan in Section 5.2.3. We discuss evaluation results of this approach in Section 5.3.

5.2.1 Completeness Estimation for Static Planning

For a given query plan, we need to estimate how complete the results will be. As given in Definition 5.5, completeness of a query plan is the proportion of correct result records that are covered by the plan. Completeness is thus the probability that a correct result to a query will be found with the query plan.

We estimate the completeness of a query plan as the proportion of query/result record pairs from a set of positive training record pairs that are covered by the plan. In the following, we describe how to gather training data and estimate completeness with it.

Gathering Training Data

To estimate the completeness a query plan, we need a set $T \subseteq U \times U$ of positive training examples for query/result record pairs. There are two main options to gather training data: (1) We have a (preferably large) set of queries with correct answers. (2) We use virtual training data.

- **(1) Real training data:** In our use case, we have a set of 2 million queries, some of them with results manually labeled as correct. This is the ideal situation, where we can rely on a manually labeled set of query/result record pairs.
- **(2) Virtual training data:** The first case relies on manually determining sets of query/result record pairs. Since this is often a costly task, one can also create virtual training data. For one training instance, our only concern is whether the overall similarity measure judges that the instance is relevant based on its base similarity values. Thus, we can make up base similarity values (without creating a real query/result pair). We can determine whether these base similarity values would lead to a correct match (if the composed similarity measure computes an overall similarity value above $\theta_{Overall}$). If yes, we have created a positive training instance without the need for real training data. We leave for future work the definition of an algorithm that automatically generates a set of training instances covering a large variety of the threshold combinations that result in high overall similarity. Inspiring work comes from the domain of learning logical expressions in DNF with membership-query algorithms [Jackson, 1997].

Estimation

With positive training instances $T \subseteq U \times U$ at hand, we can estimate completeness of a query plan p. The function $covers_p(q, r)$ evaluates to *true* iff the pair $(q, r) \in U \times U$ is covered by p (i. e., if the query plan predicates are fulfilled).

$$comp(p) = \frac{|\{(q, r) \in T \mid covers_p(q, r)\}|}{|T|} \tag{5.1}$$

Our basic algorithm to calculate $comp(p)$ for given p and T is very simple: We iterate over the list T and count all query/result record pairs that are covered by p. Since there is a potentially large amount of base similarity value combinations that are evaluated quite often during the query plan selection process, we speed up the basic algorithm with a more efficient data structure: For each distinct base similarity value combination, we save its count, thus eliminating all “duplicate” combinations. In addition, we further reduce the number of value combinations by rounding to two decimal places. Distinct combinations with the same rounded combinations are accumulated. In our use case, we can reduce the amount of value combinations to be checked from 2.0m to 4.4k (a reduction rate of 99.8 %).

5.2.2 Cost Estimation for Static Planning

The second evaluation criterion for query plans is the involved cost, i. e., the amount of records covered by the plan (according to Definition 5.6). To estimate cost of a static query plan, we need determine how many result records we can expect to be covered on average when applying a query plan to the entire data set.

For cost estimation we do not need training data, because we only exploit information from the distribution of attribute values in the complete record set R. A naïve approach to estimate cost of a static query plan would be to sample a set $S \subseteq R$ of records, exactly determine the cost for this sample by comparing each record in S to each record in R, and then average the determined cost. In our use case, comparing one record to all other records would take several hours. As we need to analyze a large set of query plans during the query plan optimization phase, we need a more efficient procedure. Thus, we precalculate attribute similarity histograms to quickly estimate the cost of attribute predicates as parts of query plans.

In the following, we first describe how we create similarity histograms for determining the amount of similar records regarding one attribute. After that, we explain how to combine attribute cost to derive an estimation of the cost of query plans involving multiple attributes.

Similarity Histograms

We want to estimate the cardinality of the predicate $a \geq \theta_a$, i. e., how many records in our database have at least a similarity of θ_a regarding the attribute a with respect to a random query? To derive an estimation of a static query plan, this estimation needs to be independent of specific values.

We first create a similarity histogram for several values of the attribute. For each analyzed value, we calculate the similarity to all other values of the attribute. Figure 5.2 shows a similarity histogram for the last name "Lange". A reading example is: for LastName ≥ 0.9, there are 172,000 records with a last name with minimum similarity 0.9 to "Lange". The solid line is the cumulated number of similar values up to a total of 66m records with a similarity of at least zero.

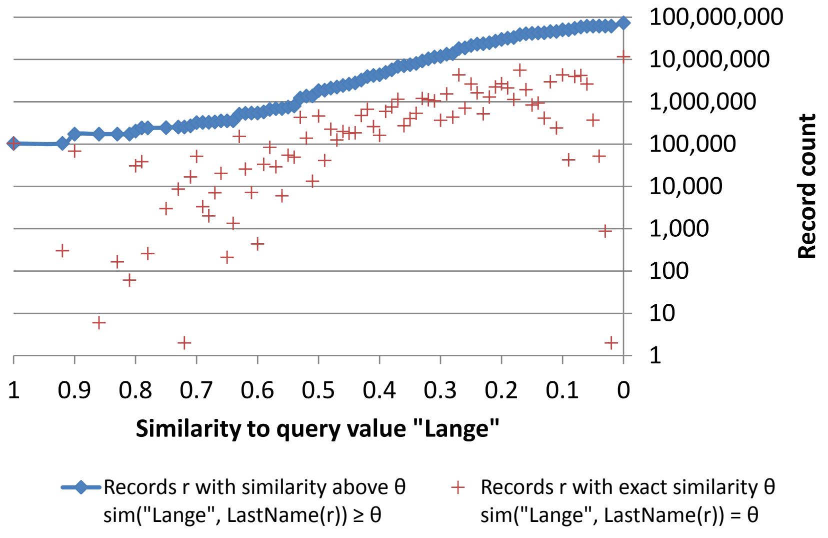


Figure 5.2: Similarity histogram for attribute LastName for value "Lange"

We then calculate the average of the created similarity histograms. For each possible similarity threshold, we average the measured numbers of records with similar values. The resulting similarity histograms for our attributes are shown in Figure 5.3. For the attributes FirstName, LastName, and City, we can see rather smooth curves. The curves for BirthDate and Zip are stepped, because all values have an equal length, and only few errors can occur. These curves allow estimations such as "for FirstName ≥ 0.7, we can expect 500k records with similar values, on average".

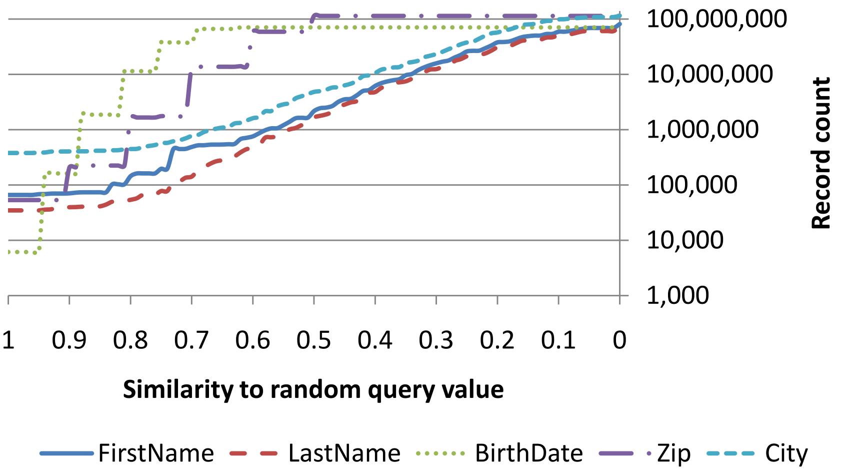


Figure 5.3: Average similarity histograms for all attributes

A question that remains to be answered is how to choose appropriate sample values to create these histograms. In general, we recommend using a set of randomly chosen values. For a worst case estimation of the cost, it may make sense to include more frequent values in the sample set. More frequent values will themselves lead to higher cost and also have more similar values (as we empirically validated). Regarding the sample size, we empirically determined that a set of 100 randomly chosen elements is sufficient in our use case (for a population of 66m elements, a confidence interval of ± 10 %, and a probability of 95 %, 96 elements are necessary).

Combining Attribute Estimations

Using the similarity histograms, we are now able to estimate the cardinality of a complete query plan. In a nutshell, we estimate the cost of a query plan p for a random query $q \in U$ by estimating the probability that a randomly chosen element $r \in R$ is covered by p. We multiply the probability with the cardinality of the complete record set to determine expected average cost:

$$cost(p) = |R| \cdot P(covers_p(q, r) \mid q \in U, r \in R) \tag{5.2}$$

By calculating cost using probabilities, we are able to easily handle conjunctions and disjunctions in the query plans with probability theory.

In the following, we abbreviate the probability that a randomly chosen element $r \in R$ is covered by an attribute predicate $a \geq \theta_a$ in $P(covers_{a \geq \theta_a}(q, r) \mid q \in U, r \in R)$ as $P(a \geq \theta_a)$ and any conjunctions and disjunctions accordingly.

We first estimate the probability that an element is in the set of records determined with a *conjunction*, such as LastName $\geq 0.9 \wedge$ City ≥ 0.95. For two attributes a and b with thresholds θ_a and θ_b we want to estimate the probability $P(a \geq \theta_a \wedge b \geq \theta_b)$. To resolve the joint probability, we distinguish between attributes that are statistically dependent or independent, for ease of calculation. (We would theoretically be able to calculate all joint distributions; however, this would consume a significant amount of time and space.) In our case, we observe that City and Zip are dependent and that each attribute is dependent on itself (this is relevant if a conjunction contains several predicates on the same attribute); all other attributes are independent from each other. If dependent attributes are not known in advance, these can be determined with statistical independence tests such as the χ^2-test. To estimate joint probabilities of dependent attributes, we assume the worst case: the two predicates completely overlap, i. e., the records covered by one predicate are completely covered by the second predicate. Thus, we estimate the probability of the predicates' conjunction as the minimum of the probabilities of two predicates (and accordingly for three or more overlapping predicates). For statistically independent attributes, we simply calculate the product of the predicate probabilities. Thus, for two predicates we have:

$$P(a \geq \theta_a \wedge b \geq \theta_b) = \begin{cases} P(a \geq \theta_a)P(b \geq \theta_b) & \text{if } a \text{ and } b \text{ are independent} \\ min(P(a \geq \theta_a), P(b \geq \theta_b)) & \text{else} \end{cases} \tag{5.3}$$

For more than two attributes, we accordingly resolve joint probabilities of statistically dependent attributes and then calculate the product of the remaining predicate probabilities of independent attributes.

With the similarity histograms, we can estimate the individual predicate probabilities as:

$$P(a \geq \theta_a) = \frac{|\{r \in R \mid a \geq \theta_a\}|}{|R|} \tag{5.4}$$

Note that the randomness regarding queries is already covered by the attribute cost estimations. As described above, for each attribute we selected several values from R as example queries and averaged their actual cost.

In a final step, we estimate the probability of a *disjunction* of conjunctions. For the union of two sets, we calculate the cardinality as the sum of the cardinalities of the two sets less the intersection of them. The same applies for probabilities. For example, we want to estimate $P(a \geq \theta_a \vee b \geq \theta_b)$ and have:

$$P(a \geq \theta_a \vee b \geq \theta_b) = P(a \geq \theta_a) + P(b \geq \theta_b) - P(a \geq \theta_a \wedge b \geq \theta_b) \tag{5.5}$$

The remaining probabilities contain only conjunctions and can be estimated as described above. For the general case of n conjunctions c_i in the disjunction $\bigvee_{i=1}^{n} c_i$, the principle of inclusion and exclusion gives us:

$$P\left(\bigvee_{i=1}^{n} c_i\right) = \sum_{k=1}^{n} (-1)^{k-1} \sum_{\substack{T \subset \{1,\ldots,n\}, \\ |T|=k}} P\left(\bigwedge_{t \in T} c_t\right) \tag{5.6}$$

With this approach we can estimate the cost of any query plan in DNF.

5.2.3 Query Plan Optimization

Based on the performance metrics defined in the previous sections, we define an algorithm for finding the best query plan, i. e., the plan with highest completeness below a given cost threshold. We iterate over the set of all possible query plan templates (we consider 210 templates in our use case, see Section 5.3) and then optimize thresholds for each template. We thus have one optimal query plan per query plan template. From these query plans, we simply select the overall best query plan.

In the remainder of this section, we discuss the problem of optimizing the thresholds of one query plan template. For a given query template containing n attribute predicates, each attribute predicate's threshold can be set to one of v values. Thus, the complexity of a complete search is $O(v^n)$, i. e., exponential in terms of attribute predicates. For example, for 6 attribute predicates, each with 101 possible threshold values (0.00 to 1.00), there are $101^6 \approx 1$ trillion possible query plans. As we need to calculate both completeness and cost for each plan to be analyzed, this large set of possible plans is infeasible to be analyzed completely; a more efficient algorithm is required.

We begin this section with observations regarding the distributions of completeness and cost and then discuss algorithms for exact and approximative optimization of query plans.

Observations

To better understand the problem space, we first analyze the distribution of completeness and cost. For the exemplary query plan template

$$\mathsf{FirstName} \geq \theta_{\mathsf{FirstName}} \wedge \mathsf{LastName} \geq \theta_{\mathsf{LastName}}$$

we evaluated all possible query plans, i. e., all possible threshold values with two decimal points for $\theta_{\mathsf{FirstName}}$ and $\theta_{\mathsf{LastName}}$. We show the resulting cost distribution in Figure 5.4a and the completeness distribution in Figure 5.4b.

In the cost distribution diagram, we can see that query plans with lower thresholds have higher cost. This is not surprising, as lower thresholds result in at least as many or more covered records as a higher threshold. For decreasing thresholds, we observe that cost grows exponentially. We can especially see that only few query plans in the upper right corner have acceptable cost; as we show in Section 5.3, more complex plans with disjunctions of conjunctions in the query plan templates achieve better results (i. e., higher completeness with lower cost).

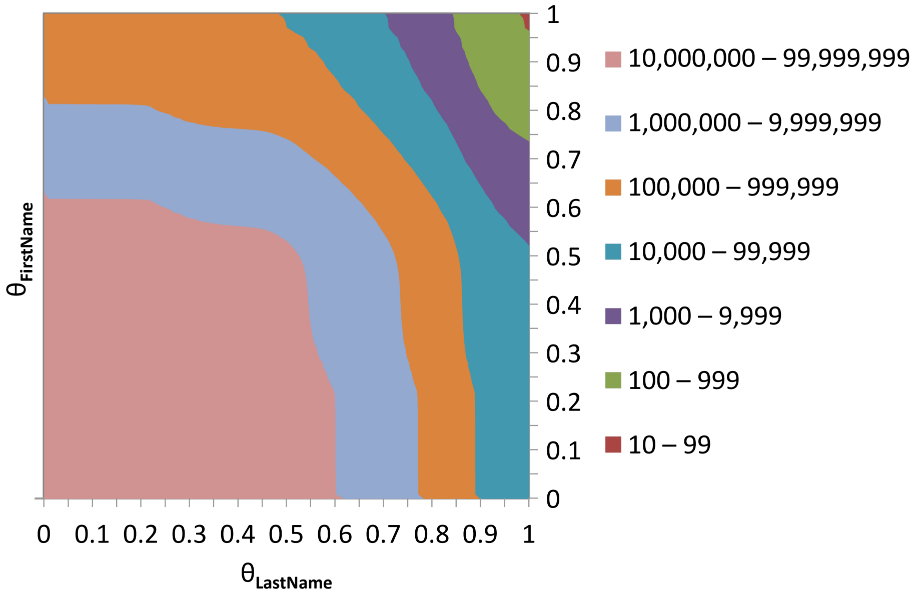
The distribution of completeness shows that lower thresholds result in higher completeness. While this insight is also not surprising, the growth of completeness is quite different from the cost distribution. Already with the highest thresholds (i. e., only exact matches are found), we can cover the vast majority of the 2m training query/result record pairs. The increase in found matches is high for higher thresholds, but for lower thresholds, the increase dwindles. This confirms our intuition: the last percentages of completeness (recall) are the most difficult to resolve.

The monotonicity observations can be generalized for any combination of similarity measures. Any query plan with thresholds (θ_x, θ_y) has at least the cost and the completeness of any query plan with thresholds $(\theta_{x'}, \theta_{y'})$ with $\theta_{x'} < \theta_x$ or $\theta_{y'} < \theta_y$. Note that Chaudhuri et al. make similar observations and also exploit the monotonicity property in their algorithm [Chaudhuri et al., 2007]. In our case we do not know completeness and cost of the query plans in advance, so we solve a different problem with an entirely different approach.

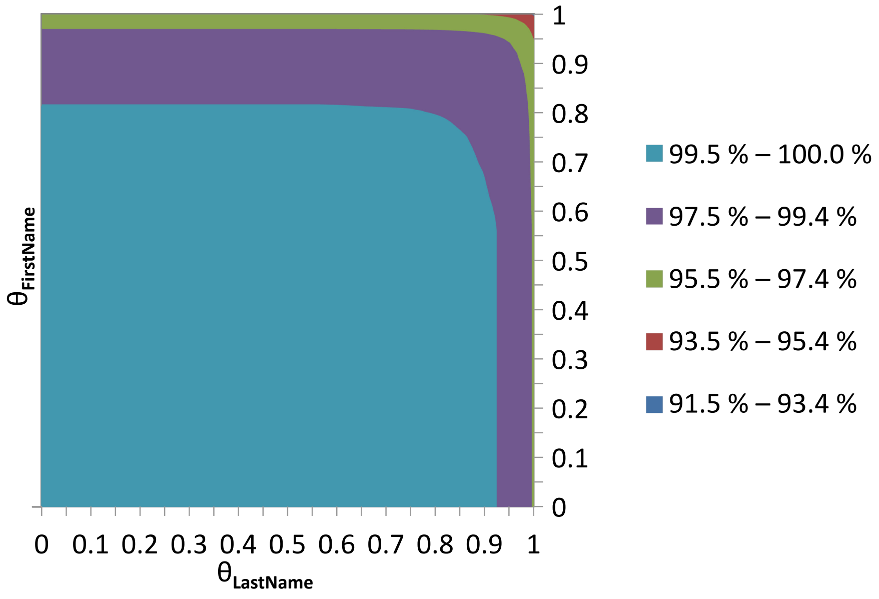
From the cost distribution diagram (Figure 5.4a), we can derive the space of all valid query plans, i. e., all plans with cost at most as high as the predefined cost threshold C. For the two-dimensional case, we can think of a line that separates all valid query plans from the invalid ones – the *valid query plan separation line*; in the general case, we have a separation hyperplane. From the monotonicity observations, we can infer that such a line always exists and that we can ignore any combinations below this line. Thus, we consider only the combinations above this line without discarding valid combinations. In our algorithms described in the following, reaching this line will be regarded as stopping criterion.

Top Neighborhood Algorithm

An described above, an algorithm that analyzes all possible query plans for a query plan template is infeasible. Keep in mind that we do not know completeness and cost for any query plan in advance. We need an algorithm that efficiently navigates through the solution space: The algorithm should only evaluate as few query plans as possible and determine an overall good solution.



(a) Distribution of cost



(b) Distribution of completeness

Figure 5.4: Distribution of cost and completeness for the query plan template $\mathsf{FirstName} \geq \theta_{\mathsf{FirstName}} \wedge \mathsf{LastName} \geq \theta_{\mathsf{LastName}}$

To solve the problem, we present the *top neighborhood algorithm*, shown as Algorithm 5.1. The general idea of the algorithm is to start with the plan with highest thresholds (in our case, all thresholds are set to 1, which corresponds to the top-right corner of Figures 5.4a and 5.4b), then follow promising plans in its neighborhood (the top plans), until we reach the valid query plan separation line. The result is then the plan with highest completeness (and lowest cost, respectively) found so far. In the following, we describe the algorithm in greater detail.

The start plan must be the one with highest thresholds, since any other plan for starting could make it impossible to find the best solution due to our downwards search approach.

We define the neighborhood of a plan to help us navigate the threshold space. A query plan p has a *neighborhood* $N(p)$ that contains all query plans that can be constructed by lowering one threshold of p by one step (e. g., by 0.01). For example, the neighborhood of a plan with two thresholds θ_a and θ_b has two elements:

$$\begin{aligned} N(a \geq \theta_a \wedge b \geq \theta_b) &= \{a \geq \theta_a - 0.01 \wedge b \geq \theta_b, \\ &\quad\; a \geq \theta_a \wedge b \geq \theta_b - 0.01\} \end{aligned}$$

The neighborhood thus defines all possible directions to traverse the solution space given one query plan. We define the neighborhood only for lower thresholds and thus higher completeness (and also higher cost), since we traverse the threshold space from higher to lower thresholds.

In each iteration of our algorithm, we have a *window* $W = \{q_1, \ldots, q_n\}$ of n query plans that are currently regarded. We extend the neighborhood concept to windows by defining the set of all neighborhood plans for the plans in W as $N(W) = \bigcup_{p \in W} N(p)$. We then select a subset of these plans: the t plans with highest completeness (and lowest cost, if there are several plans with equal completeness; if there are more than t eligible plans, a random selection is made). We call this set the *top t neighborhood* $T_t(W)$. Only plans with cost below the cost threshold C are contained in this set.

The cost threshold C also determines the *stopping criterion* of our algorithm. If the top t neighborhood contains only plans with cost above C, then the algorithm terminates and returns the best plan found so far. Otherwise, the algorithm continues with a new iteration by setting $W := T_t(W)$ (t is thus also the maximum window size).

Note that the algorithm is optimal regarding the number of times a query plan is evaluated. In each iteration, the window contains only plans with the same amount of applied threshold lowering steps (compared to the initial plan). Since the union of the generated plans is calculated before evaluating them, no query plans are evaluated more than once.

In Figure 5.5, we illustrate one iteration of the algorithm by means of an exemplary query plan template with two thresholds. Note that the illustration is similar to the measured completeness distribution in Figure 5.4b. The diagram shows the search space of all valid plans, i. e., all plans with acceptable cost (which we do not know in advance). The current window consists of two plans. The neighborhood of the window contains three plans, two of which (with *comp*=38) are selected as the top 2 neighborhood and thus form the new window for the next iteration. The neighborhood of the new window then contains three plans with *comp* $\in \{40, 42, 46\}$, from which the two plans with highest completeness are selected as the top 2 neighborhood for the next iteration. After three more iterations, the algorithm finally reaches the valid query plan separation line and returns the seen valid plan with highest completeness (*comp*=62).

Exact algorithm. The parameter t allows us to turn the approximative top neighborhood algorithm into an exact algorithm for finding the optimal query plan. By setting $t = \infty$, we do not limit the window size and thus evaluate *all* valid query plans. Still, we avoid the evaluation of a number of query plans: those with cost above C. Regarding the search space coverage, this exact algorithm is thus a better approach than a brute force search over all possible query

Algorithm 5.1 Top Neighborhood Algorithm

Input: cost threshold C, template t, top neighborhood size w
Output: plan p_{res} with optimized thresholds

```
1: $p_{res} := \varepsilon$
2: $maxC := 0$
3: $W$ := set of plans, initially contains only the plan built from $t$ where all thresholds are set
   to 1
4: while true do
5:   $R := \bigcup_{p \in W} N(p)$
6:   $W := \emptyset$
7:   for each $p \in R$ do
8:     if $cost(p) \leq C$ then
9:       $W := W \cup \{p\}$
10:      if $comp(p) > maxC$ then
11:        $maxC := comp(p)$
12:        $p_{res} := p$
13:  if $W = \emptyset$ then
14:    break
15:  $W$ := set of $w$ plans from $W$ with highest completeness
16: return $p_{res}$
```

plans. A disadvantage is the higher memory requirement, since all evaluated query plans of an iteration need to be kept in memory.

Complexity. The complexity of the top neighborhood algorithm depends on the number of necessary iterations as well as the number of generated query plans per iteration. A worst case estimation for the number of necessary iterations assumes that all thresholds need to be lowered to their lowest possible value. The number of iterations is thus limited by the number of all possible thresholds of all predicates of the query plan template. The number of generated plans per iteration is determined by the top neighborhood size (and thus maximum window size) t and the number of predicates in the query plan template (for which one threshold can be lowered in an iteration). Thus, the complexity of the top neighborhood algorithm is linear in the following parameters:

- the number of predicates in the query plan template
- the number of possible thresholds of all predicates
- the top neighborhood size t

This gives the following worst case estimation (where A is the set of predicates in the query plan template):

$$O\left(\left(\sum_{a \in A} |\Theta_a|\right) \cdot t \cdot |A|\right)$$

If we set $t = \infty$, then all possible threshold combinations are evaluated. With $|\Theta_m| = \max_{a \in A}(|\Theta_a|)$ the worst case estimation shows an exponential complexity regarding the number of predicates:

$$O\left(|\Theta_m|^{|A|}\right)$$

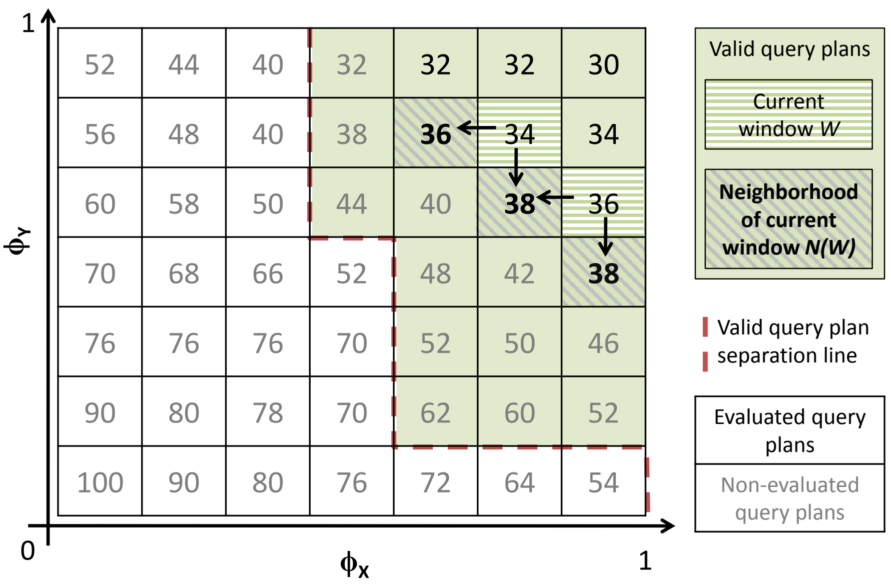


Figure 5.5: Application of top neighborhood algorithm to exemplary completeness estimations (in percent) for two thresholds θ_X, θ_Y in a query plan template

5.3 Evaluation of Static Query Planning

In this section, we discuss evaluation results of the static query planning approach, and in particular of the cost estimation model as well as the top neighborhood algorithm. In Section 5.6, we empirically compare the static query planning approach with our approach for query-specific planning. We evaluate our approach on the real-world data set from Schufa described in Section 3.3.1.

Evaluation of Cost Estimation

To analyze the quality of our cost estimation model, we compare estimated and observed costs for a set of query plans templates. We generated *all* query plan templates with one or two disjunctions of conjunctions that involve two or three attributes each. We required the conjunctions to contain different sets of attributes, while overlapping of attributes in the conjunctions was allowed. Overall, we analyzed 210 query plan templates.

We randomly selected 100 queries as test objects. For these queries, we count the exact number of records with the values given in the query using an inverted index. With this method, we can count exact matches (i. e., all similarity thresholds in the query plans are set to 1.00). This allows the evaluation of the quality of the probability model. (For the quality of the similarity histograms, we rely on statistical guarantees that we have enough training examples for calculating average frequencies.)

In Figure 5.6, we show estimated and average observed costs for the analyzed query plans. Optimal estimations would result in points arranged at a diagonal line with $y = x$ (shown as model line in the graph). We observe that our estimations are quite close to the observed frequencies for all considered plans. Our cost model, albeit not perfectly accurate, seems to be a good estimator for average cost of different static query plans. In Section 5.6.2, we show an additional evaluation of our cost model and see that estimating cost for individual queries (as

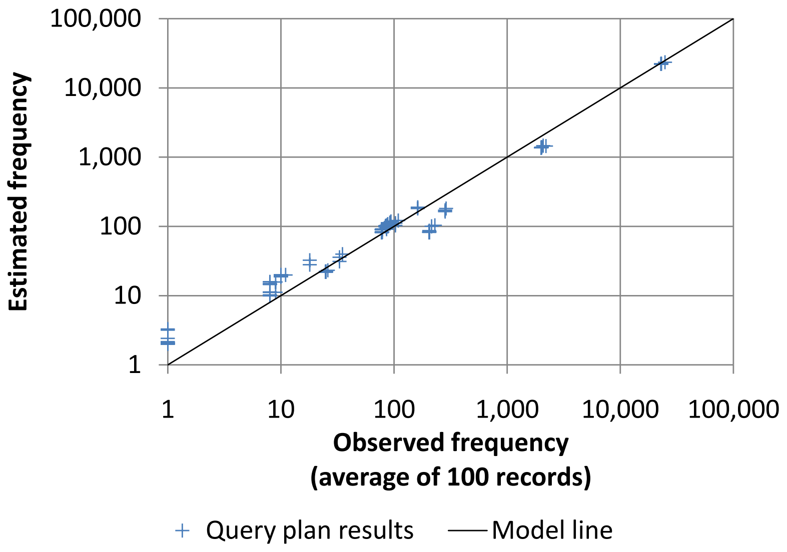


Figure 5.6: Comparison of observed and estimated frequencies for the 210 generated query plans. Example: For the query plan BirthDate $\geq 1 \land$ City ≥ 1, our estimation is 22 records, and the observed average frequency is 25 records.

needed for query-specific planning) is a much more difficult task than estimating the average cost (as needed for static planning).

Evaluation of Top Neighborhood Algorithm

To evaluate the top neighborhood algorithm, we analyzed the behavior of the algorithm on our data set with 2 million (correct) query/result record pairs. We ran the algorithm for three randomly selected query plan templates with different numbers of predicates and varied the parameter t. Table 5.1 shows the results. For comparison, the first line of each table shows the results for exact matching (i. e., all thresholds are set to 1) as baseline, and the last line contains the result of the exact version of our algorithm (with $t = \infty$). We refer to the result of the exact algorithm as the optimal plan, since no better plan can be found with the given query plan template.

For all analyzed templates, a small value for t is already sufficient to find a plan with near-optimal completeness (matches are missing only for a few of the 2m queries). For all analyzed values of t, we can see a significant improvement over the baseline results for exact matching. In general, a larger value for t results in larger completeness. The numbers of evaluated query plans also confirm the linearity of the algorithm regarding t and the query template complexity (i. e., the number of predicates as well as the number of thresholds per predicate; this cannot be distinguished in this experiment).

With growing query plan templates (i. e., more predicates), the results become more complete, since the template is more expressive. In general, the thresholds of a more complex predicate are higher in order to satisfy the cost constraint.

The more complex the query plan, the larger the number of saved query plan evaluations: for the template with six predicates (Table 5.1c), we consider only less than a percent of the *valid* query plans. For all templates, we only evaluate a fraction of *all* possible plans. Even the exact version of our algorithm with unlimited t considers only 16 % of the possible plans for the

t	Evaluated query plans	Fraction of **valid** plans	Fraction of **all** plans	Completeness
Baseline	1 (all thresholds set to 1)	0.310 %	0.049 %	93.1599 %
5	173	53.560 %	8.543 %	98.7725 %
10	276	85.449 %	13.630 %	99.0643 %
20	323	100.000 %	15.951 %	99.0643 %
∞	323	100.000 %	15.951 %	99.0643 %

(a) Results for query plan template with *two* predicates ($A \wedge B$). The number of possible query plans is 2,025.

t	Evaluated query plans	Fraction of **valid** plans	Fraction of **all** plans	Completeness
Baseline	1 (all thresholds set to 1)	0.004 %	0.000 %	99.3322 %
5	573	2.438 %	0.171 %	99.9154 %
10	818	3.480 %	0.245 %	99.9619 %
20	1,802	7.666 %	0.539 %	99.9604 %
50	5,364	22.819 %	1.605 %	99.9826 %
100	9,399	39.984 %	2.813 %	99.9861 %
200	16,825	71.574 %	5.036 %	99.9880 %
∞	23,507	100.000 %	7.035 %	99.9905 %

(b) Results for query plan template with *four* predicates ($(A \wedge B) \vee (C \wedge D)$). The number of possible query plans is 334,125.

t	Evaluated query plans	Fraction of **valid** plans	Fraction of **all** plans	Completeness
Baseline	1 (all thresholds set to 1)	0.000 %	0.000 %	99.3633 %
5	1,335	0.033 %	0.000 %	99.9275 %
10	2,657	0.065 %	0.000 %	99.9786 %
20	5,007	0.122 %	0.001 %	99.9786 %
50	11,440	0.280 %	0.002 %	99.9845 %
100	20,672	0.505 %	0.003 %	99.9892 %
200	37,141	0.908 %	0.005 %	99.9936 %
∞	4,092,574	100.000 %	0.605 %	99.9994 %

(c) Results for query plan template with *six* predicates ($(A \wedge B) \vee (C \wedge D) \vee (E \wedge F)$). The number of possible query plans is 676,603,125.

Table 5.1: Evaluation results of top neighborhood algorithm for different query plan templates

small template and 0.6 % for the large template. For the large template and $t = 200$, still only less than a ten thousandth of all possible plans have been evaluated.

To summarize, the results show that the top neighborhood algorithm determines a near-optimal plan while evaluating only a fraction of the possible query plans. We have measured similar results for other query plan templates.

Evaluation of Static Query Planning

We also ran an experiment on the complete data set with the best static query plan, which is selected to perform well on average. To determine the best static query plan for our data set, we used the set of query plan templates from the cost model experiment. This set contains *all* query plan templates with one or two disjunctions of conjunctions that involve two or three attributes each. We required the conjunctions to contain different sets of attributes, while overlapping of attributes in the conjunctions was allowed. For each of the 210 query plan templates, we ran our top neighborhood algorithm with $t = 100$ (a window size that results in acceptable runtime of our algorithm) and $C = 1000$ (a cost threshold that is acceptable for our use case). We selected the overall best query plan for this experiment.

As test queries, we randomly selected 50,000 queries from our test data set. We ran these queries with different search settings and show the results in Table 5.2.

Search setting	**Completeness**	**Cost**
(1) Exact search with all attributes	87.28 %	0.8
(2) Exact search with overall best query plan	99.47 %	37.8
(3) Optimized similarity search with overall best query plan	99.98 %	551.7
(4) No filtering	100.00 %	66m

Table 5.2: Search performance of different search settings

The first line shows a baseline result when searching for only those records where all attribute values of the query exactly match the values in the record. This shows that 87 % of the queries are “easy” to answer, since the corresponding correct matches in the record set do not contain any differences to the query. The cost for queries with this search settings are quite low, since there is usually at most one such exactly matching record.

We then tried to answer the same set of queries with the query plan that was produced by our top neighborhood algorithm. We first did an exact search with this plan (Line 2 in Table 5.2). In comparison to the first search setting where all attribute values were required to exactly match, the optimized plan only requires that one of two conjunctions (of two attributes each) match. The result shows that the majority of the remaining records can be found with this setting, but that the average cost are also higher for this setting (again compared to search setting (1)).

By applying the top neighborhood algorithm, we optimized the thresholds of the query plan. We use these threshold in search setting (3). Line 3 in Table 5.2 shows that we achieve even better results than for the previous search setting. The completeness is near 100 %. As already pointed out, the last couple of records are the most difficult to find. This query plan thus has higher cost, but they are still below the cost threshold of $C = 1000$.

For comparison, we also show that comparing the query to each record in our data set

(Line 4 in Table 5.2) would result in perfect completeness, but also in cost of 66m, which is clearly infeasible.

In the next sections, we propose a method for query-specific planning that is able to even better exploit the cost limit per query and that often achieves better completeness results (cf. Section 5.6.2).

5.4 Template Selection for Query-specific Planning

In Section 5.2, we have introduced a method for static planning where the query plan is used for *all* queries. In this and the following section, we present an extended method that selects a query plan *individually* for each query. By analyzing the attribute values in the query, we can better adjust the used attributes and thresholds to the requirements of individual queries. Completeness can be optimized by leaving out probably erroneous attributes from the template, and thresholds can be selected to keep cost within a query-specific cost limit. However, these advantages come at the expense of higher query preparation cost at query time.

The query-specific planning process consists of two steps: (a) selecting a query plan template (this section) and (b) selecting thresholds for the template (Section 5.5). We evaluate the query-specific planning approach and show a comparison with the static approach in Section 5.6.

Why does it make sense to determine the best template first? First and foremost, the template defines the structure of the query plan and is more important for the search than its thresholds. The template expresses which attributes are restricted, i. e., which attribute values from the result record must match the query values. Our goal is to find a template where most errors in the query are made in the non-restricted attributes. The next step is then to carefully lower the thresholds, so that all remaining (hopefully few) errors in the template's attributes are also covered. Our evaluation results in Section 5.6.2 confirm that in our use case most queries can be answered with a well-selected template, while the plans (with lower thresholds) are required for answering the remaining (small) fraction of queries (cf. Figure 5.10a on page 75). In addition, as our optimization algorithm is run with each query, it must be very fast, so that overall query runtime is not significantly affected. The solution space of possible templates is considerably smaller than for query plans (only attribute combinations are considered; thresholds are ignored). Thus, optimizing thresholds for only the best template saves a considerable amount of time. Lastly, in some use cases, our proposed algorithm for template selection already determines a very good solution. In particular, when the queries contain only few errors (i. e., there are several attributes that exactly match the correct record), then there is no need for lowering thresholds at all (not even similarity indexes are required). In these cases, our template optimization approach described in this section is sufficient, and only exact indexes on the occurrences of attribute values need to be created.

We describe how to evaluate completeness and cost of a conjunction of attributes as part of a query plan template in Sections 5.4.1 and 5.4.2. For combining the conjunction estimations into DNF estimations for the complete plan, we refer to the previous Section 5.2.2. Based on these estimations, we describe our algorithm for template selection in Section 5.4.3.

5.4.1 Completeness Estimation for Template Selection

Evaluating the completeness of a query plan means predicting which attributes of a query match the correct result record. An intuition to tackle this problem is that the frequency of values has a strong influence on the errors that are made. Psycholinguistic studies showed that high-frequency words are more likely to be spelled correctly than low-frequency-words [MacKay and Abrams, 1998, Stemberger and MacWhinney, 1986], because the corresponding paths in the brain have been activated much more often (and vice versa for low-frequency words) [Burke et al., 1991].

If an attribute value in the query occurs frequently in the database, then we can be quite sure that it is spelled correctly (an observation also exploited in a spelling correction algorithm by Google [Whitelaw et al., 2009]). The corresponding attribute thus might be a good candidate to include in the template, because inclusion means that the attribute values must match. For query values that we find only rarely in the database, there is a higher probability that they contain errors – e. g., because they are misspelled versions of actually frequent values or because someone did not know and thus misspelled a rare value. Thus, it should not be part of the restricting attributes in a template.

As an example, Figure 5.7 shows the completeness of two templates containing exactly one attribute each. The results are calculated for a set of 1000 randomly selected queries from our person data use case. The figure shows completeness values for different frequency ranges for the respective attributes. We can see that the completeness values for both templates increase with the according frequency (confirming previous studies on the correlation of frequencies and spelling errors [MacKay and Abrams, 1998, Stemberger and MacWhinney, 1986]). Note that our approach contains no hard-wired formulas, but rather exploits any relationship between frequency and completeness that is contained in the training data – even if this relationship contradicts our initial intuition.

Because frequencies of values indicate the probability of contained errors, we use value frequencies to determine a template's completeness. We formalize the problem as a regression task: Given the frequencies of the attribute values of a query q and a query plan template t, predict $comp(t, q)$.

Our approach is to partition the training data using the given frequencies. We then calculate the completeness value in each partition for each query plan template. We generate the best partitioning using a tree learning algorithm that we define in the following and that is inspired by the C4.5 algorithm for decision tree learning [Quinlan, 1993]. For a given query and conjunction, the idea is then to estimate the completeness from those training queries that have similarly frequent attribute values.

We call the resulting regression tree a *completeness tree*. The tree consists of decision nodes and regression values in the leaves. In our case, a decision node refers to an attribute frequency, e. g., $f(\mathsf{FirstName}) < 1000$. As prediction value, a leaf contains the completeness regarding the data partition that is defined by the path of all decision nodes from the root to the leaf. The leaf contains a *completeness array*, i. e., a list of completeness estimations for all subsets of attributes. We show an example in Figure 5.8. The root node is a decision node with $f(\mathsf{FirstName}) < 1000$. For any query with rare values for FirstName, we reach a leaf node. The value 0.75 in the leaf node's completeness array means that 75 % of all training queries with $f(\mathsf{FirstName}) < 1000$

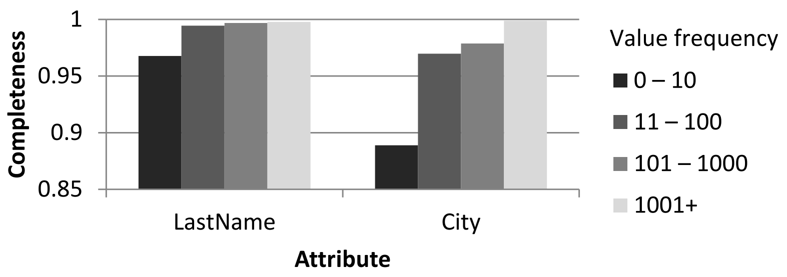


Figure 5.7: Completeness of templates consisting of only one attribute each, grouped by frequency of the respective attribute

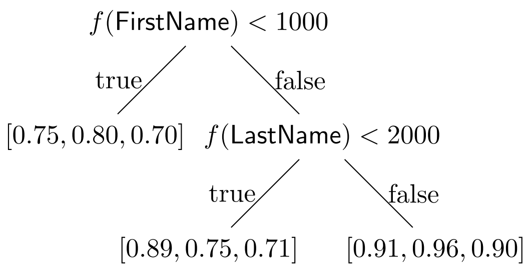


Figure 5.8: Completeness tree example for the two attributes FirstName and LastName. Leaves show completeness arrays for the templates FirstName, LastName, and FirstName $\wedge$ LastName (in this order).

are covered by the template FirstName; similarly, 80 % are covered by LastName, and 70 % are covered by FirstName $\wedge$ LastName.

The algorithm for creating the completeness tree is shown as Algorithm 5.2. The tree is created top-down, starting with the root node. To determine the best decision node for the root node, every possible attribute frequency is evaluated. The best node is the one with lowest sum of squared errors (SSE) between predicted and actual value. The remaining nodes are created with a greedy algorithm. The locally best node is fixed and the algorithm is applied recursively to the left and right child nodes. A node is not further split if the number of remaining training instances $|T|$ falls below a threshold T_{min}, or if the prediction error cannot be decreased.

The SSE calculation is shown in Line 7. For a conjunction c and a training data set T, we determine the estimation error for the predicted completeness. The predicted completeness pr is calculated as the fraction of training queries in T covered by c; this number is returned by $m(T, c)$. For these covered instances, the correct value is 1 (as they have been matched); thus, the squared error for the predicted value is $(1 - pr)^2$. For the instances that have not been matched, the correct value is 0, and we have a squared error of pr^2.

In our setting, we have a multi-label regression problem, i. e., we want to predict several values (the completeness of all possible attribute conjunctions) at the same time. We solve this problem by creating only one regression tree with the following properties. Our optimization criterion is the sum of SSE of *all* conjunctions, since our goal is to reduce the average error over all conjunctions. We do not weight the SSE of individual conjunctions as we have no indication of which conjunction might be more relevant than another for any query. Note that if we had created one tree for each conjunction, the predicted values would have referred to different partitions in the data, preventing us from combining the values for calculating disjunction completeness later on.

The completeness of an attribute conjunction for a given query is determined by traversing the completeness tree with the frequencies of the query values from the root node to a leaf node. The result is the predicted completeness value for the conjunction stored in the determined leaf node.

After having described how to estimate completeness of conjunctions, we now only need to combine the results into an overall estimation of a template in DNF. For this purpose, completeness estimations can be regarded as probabilities, similar to our cost estimations. For cost, we modeled this explicitly (see Sections 5.2.2 and 5.4.2). For completeness, the fraction of covered correct query/result record pairs can be interpreted as the probability that a plan covers a result record. Thus, we can use the same combination method for both approaches. In Section 5.2.2, we have already presented a method based on probability theory that we reuse here for calculating completeness estimations of templates in DNF.

Algorithm 5.2 function *createNode*(T)

```
Input: set $T$ of training query/result record pairs
Output: split node or leaf node
1: if $|T| < T_{min}$ then
2:     return leaf node with data $T$
3: $sp \leftarrow null$
4: $E_{min} \leftarrow$ estimation error ($SSE$) on $T$
5: for each attribute $a$ do
6:     for each distinct frequency $f$ of any value of $a$ do        // evaluate split point $\langle a, f\rangle$
7:         $E_{a,f} \leftarrow \sum_{c \in conj} m(T,c)(1-pr)^2 + (|T| - m(T,c))pr^2$ where $pr \leftarrow \frac{m(T,c)}{|T|}$
8:         if $E_{a,f} < E_{min}$ then
9:             $sp \leftarrow \langle a, f\rangle$
10:            $E_{min} \leftarrow E_{a,f}$
11: if $sp = null$ then
12:     return leaf node with data $T$
13: else
14:     $lC \leftarrow createNode(\{d \in T \mid f(d, sp.a) < sp.f\})$
15:     $rC \leftarrow createNode(\{d \in T \mid f(d, sp.a) \geq sp.f\})$
16:     return split node $\langle lC, rC\rangle$
```

5.4.2 Cost Estimation for Template Selection

Cost estimation is performed with the same cost model that we used for estimating cost of static query plans in Section 5.2.2. We only describe the changes for estimating cost for query-specific plans.

With knowledge about the query, we can determine the individual predicate probability for a value v of an attribute a for a given query q precisely as:

$$P(a \geq \theta_a) = \frac{|\{r \in R \mid \text{value of } a \text{ in } r \text{ is } v|\}}{|R|} \tag{5.7}$$

In addition, we are able to determine exceptions from the default case regarding the independence of the occurrence of attribute values. In Section 5.2.2, we have described how to determine whether two attributes are statistically dependent or independent. In few cases, there are some strong deviations from the cost model, especially for the attributes FirstName and LastName. Specific value combinations of the two attributes frequently co-occur, while the involved values co-occur significantly less often with other values. For example[1], in a database of American persons, Chinese first names (such as Wen and Zhi), frequently co-occur with Chinese last names (such as Chen and Li), but there are much less co-occurrences with Amercian last names (such as Smith and Jackson). Because our independence assumption does not hold in these cases, we determine a small list of very frequently co-occurring attribute values for these two attributes: We calculate the values where the estimation error of the cost model for dependent attributes is at most an order of magnitude higher than the estimation error of the default model for independent attributes (so that our algorithm tends to select more strict plans). In these exceptional cases, we regard the attributes as if they were dependent (and vice versa for defaultly dependent attributes).

[1]We give a fictitious example to not disclose any real names from the Schufa database.

$$\begin{tikzcd}
 & & & \varepsilon & & & \\
A \arrow[urrr] & & B \arrow[ur] & & C \arrow[ul] & & D \arrow[ulll] \\
A \wedge B \arrow[u] \arrow[urr] & A \wedge C \arrow[ul] \arrow[urrr] & A \wedge D \arrow[ull] \arrow[urrrr] & & B \wedge C \arrow[ull] \arrow[u] & B \wedge D \arrow[ulll] \arrow[ur] & C \wedge D \arrow[ull] \arrow[u] \\
A \wedge B \wedge C \arrow[u] \arrow[ur] \arrow[urrrr] & & A \wedge B \wedge D \arrow[ull] \arrow[u] \arrow[urrr] & & A \wedge C \wedge D \arrow[ulll] \arrow[ull] \arrow[urr] & & B \wedge C \wedge D \arrow[ull] \arrow[ul] \arrow[u] \\
 & & & A \wedge B \wedge C \wedge D \arrow[ulll] \arrow[ul] \arrow[ur] \arrow[urrr] & & &
\end{tikzcd}$$

Figure 5.9: Conjunction graph for four attributes A, B, C, D

5.4.3 Template Selection Algorithm

Being able to predict cost and completeness of a query plan template, we can now define an algorithm to determine the best template, i. e., one that maximizes completeness with cost below the cost limit.

Our algorithm starts with an empty DNF. We iteratively add conjunctions to the DNF, until no other conjunctions can be added or until a specified number of conjunctions is reached.

For a given query q, we first determine all conjunctions that have cost below the cost limit C_q (to fill the first "slot" in the resulting DNF). The power set of conjunctions and its inclusion relation can be represented as a directed graph $G(V, E)$, where the vertices are the attribute conjunctions $V = \mathcal{P}(attributes)$ and the edges represent the inclusion property. There is a directed edge (v_1, v_2) iff $v_1 \supset v_2$. We call this graph the *conjunction graph*. Figure 5.9 shows a representation of a conjunction graph example in the form of a Hasse diagram where ε refers to the empty conjunction.

To efficiently determine the set of valid conjunctions, we apply a *backtracking and pruning* approach to the graph. We traverse the conjunction graph from the largest conjunction to the empty conjunction (in the figure: from bottom to top) with depth-first search. By removing an attribute from a plan, the plan becomes less restrictive, thus the completeness of the plan increases, but so does its cost. Beginning with the largest conjunction, we evaluate the conjunction to test wether it forms a valid plan. If it is valid, we add it to the set of valid conjunctions and then determine the next conjunction by removing one attribute (i. e., traversing one of the outgoing edges). If the cost of a conjunction c are too high, then any conjunction $c^* \subset c$ is less strict and has thus equal or even higher cost, so that we can prune the search at this point – we remove all edges to vertices $c^* \subseteq c$ from the graph.

For example, if $A \wedge B$ in Figure 5.9 turns out to be invalid, then A, B, and ε also must be invalid as they are less strict. In this case and in case there are no more attributes to remove, the algorithm applies backtracking: We return to the set that included the attribute that we just removed, and then we try to remove another attribute. This approach allows us to prune paths in the graph containing only invalid plans as early as possible. Note that an efficient recursive implementation of this approach does not need to construct the entire graph. However, it is necessary to store visited vertices as well as unreachable vertices (where we removed all incoming edges) to prevent unnecessary evaluations of vertices. We show in Algorithm 5.3 a recursive implementation of the function $evaluateTemplate(d, c)$, which evaluates a disjunction

Algorithm 5.3 function *evaluateTemplate*(d, c)

```
Input: disjunction $d$ representing current template,
    conjunction $c$ to be added to current template
1: if $c$ is empty
   or $c$ already visited
   or $c$ is subset of any conjunction of $d$
   or $c$ is subset of any invalid result then
2:     return
3: else if $cost(d \vee c) > C_q$ then
4:     add $c$ to invalid results
5:     return
6: else
7:     add $c$ to valid results
8:     for each attribute $a \in c$ do
9:         $evaluateTemplate(d, c \wedge \neg a)$
```

d and a conjunction c to determine whether $d \vee c$ forms a valid plan. Initially, the function is called with $d = \varepsilon$ as empty disjunction and the conjunction containing all attributes as c.

We have now determined the set of valid conjunctions for a given query. Each such conjunction c forms an initial disjunction $d = c$, where one position has been filled. As a template may contain several conjunctions (up to a pre-defined maximum number of conjunctions), we continue adding conjunctions. At this point, we have two variants of our algorithm:

– **Exact variant:** We continue with all valid disjunctions.

– **Greedy variant:** We continue only with the best valid disjunction (with highest completeness).

In the following, we describe how to proceed with any selected valid disjunction d. To determine the next conjunction, we again run the backtracking algorithm on the conjunction graph. The process is equivalent to a function call to *evaluateTemplate*(d, c) in Algorithm 5.3 with d as disjunction and, again, the conjunction containing all attributes as c. For each evaluated conjunction c, we now evaluate whether the disjunction $d \vee c$ is valid, and we can prune the graph as in the previous step if it is invalid. We add only conjunctions c that are not subsets of any conjunction in d, as otherwise for $d = d' \vee c^*$ with $c \subset c^*$ we have $d \vee c = d' \vee c$, i. e., c does not fill a position in addition to d; we thus should have already seen the disjunction in the previous iteration of the algorithm. We repeat this process until all positions of d have been filled or until there are no more conjunctions left to add.

As a result, we have determined the best query plan template for a given query. This template can immediately be executed as a very strict query plan (with all thresholds set to 1.0), or we can determine the best query plan by optimizing its thresholds as described in the next section.

The worst case runtime of the described algorithm depends on the number of attributes. For n attributes, there are up to 2^n subsets of attributes that form a conjunction. For a maximum disjunction length of l, the maximum amount of disjunctions is less than 2^{nl}, while the maximum value for l is $n!$. The algorithm runs thus in $O(2^{n!})$. While this worst-case runtime seems large, the actual runtime largely depends on the chosen cost limit and the query. Our algorithm allows early pruning of the search process, so that the number of actually evaluated query plans is much smaller without missing any valid query plan. With the greedy variant, the number of evaluated query plans can be further reduced. In Section 5.6.2, we empirically compare the exact and greedy versions of the algorithm.

5.5 Threshold Optimization for Query-specific Planning

With the algorithm of the previous section, we can determine a good query plan template for a given query. Although the template can be interpreted as a very strict query plan that accesses only inverted indexes, there are many situations where such a plan is not sufficient. Consider a query that contains a typo in each of its fields. In this case, any query plan that executes an exact search for each of its attributes cannot find the correct result record in the database. Thus, as a next step, we extend our solution space to query plans with arbitrary thresholds.

Similar to the previous section, we first discuss how to estimate completeness and cost of a given query plan with varying thresholds before presenting optimization algorithms to efficiently traverse the space of possible query plans.

5.5.1 Cost Estimation for Threshold Optimization

To estimate the cost of a query plan for a given query, we construct a similarity histogram for each attribute. In Section 5.2.2, we introduced similarity histograms for static plans and created one static histogram for each attribute. For query-specific planning, we create histograms once for each query and then combine the information from the histograms to estimate the cost of all query plans that we evaluate for the query.

To construct the query-specific similarity histograms, we use the similarity indexes that we created for all attributes. For a given value v of an attribute a in the query and for each similarity threshold θ_a from the set of possible thresholds Θ_a, we determine the number of records in the database with a value sufficiently similar to v using a query to the similarity index for a.

We store the number of matching records for each similar value directly in the similarity index; it can also be determined with the inverted index, resulting in more index accesses at query time. A query may contain a value that has not yet been indexed; in this case, similar values are calculated at query time. These similarity values can be inserted into the similarity index to speed up future queries. However, inserting too many spelling variants from queries may slow down index read performance. When the similarity index should be updated depending on the distribution of unseen query values is not covered in this thesis.

The individual attribute costs are combined into conjunction and disjunction estimations as described in Section 5.2.2.

5.5.2 Completeness Estimation for Threshold Optimization

With the help of the completeness tree described in Section 5.4.1, we have estimated the completeness of templates. A node in the completeness tree refers to a subset of the training queries that is relevant to a given query. Our estimation of a query plan's completeness given the query also refers to this set of queries, thus providing a consistent completeness estimation.

We observe that for a query plan template with n attributes where we have $|\Theta_a|$ different possible similarity thresholds per attribute, there are $|\Theta_a|^n$ different combinations of threshold settings (e. g., for two attributes, we could have $(1, 1)$, $(1, 0.99)$, …, $(0.99, 1)$, $(0.99, 0.99)$, …). In our use case with 5 attributes and up to 100 similarity thresholds, iterating over the complete set of threshold combinations is infeasible. We also observe that the threshold space is quite sparse. Even in our case with hundreds of thousands of training queries, only a very small set of about 3,400 distinct threshold combinations actually occur in the data. Thus, instead of precalculating all completeness values for all conjunctions and all nodes in the completeness tree, we perform live aggregation at query time.

We *extend the completeness tree* by attaching the similarity values of those training query/result record pairs to the completeness tree nodes that fall into each node's training

Algorithm 5.4 Similarity Profile Algorithm

Input: query q, cost threshold C_q, template t
Output: plan p_{res} with optimized thresholds

```
1: $p_{res} := \varepsilon$
2: $maxC := 0$
3: $S :=$ set of similarity threshold combinations from completeness tree for $q$
4: for each $s \in S$ do
5:     $p := setThresholds(t, s)$
6:     if $cost(p, q) \leq C_q \wedge comp(p, q) > maxC$ then
7:         $maxC := comp(p, q)$
8:         $p_{res} := p$
9: return $p_{res}$
```

data partition. This is exactly the set of record pairs that we used to calculate the node's completeness values for the attribute conjunctions in the templates.

To determine a query plan's completeness, we work with the node of the completeness tree that we find by traversing the tree with the given query (the same node that we have determined for selecting the best template). We iterate over the training queries that were assigned to this node and sum up the number of matching training queries. To increase aggregation performance, we store only *distinct* similarity value combinations for the training query/result record pairs in the completeness tree nodes. The completeness of a query plan is then the proportion of queries that are matched by the plan. For a given query, all completeness estimations are consistent, as they all refer to the same set of training queries. Aggregating these conjunction estimations to disjunction estimations is performed as described in Section 5.2.2.

5.5.3 Threshold Optimization Algorithms

We are now able to estimate cost and completeness of any query plan for a given query. In the following, our goal is to optimize the thresholds in the query plan template. We first observe that traversing the complete threshold space at query time is infeasible, for the same reason that precalculating cost estimations for all threshold combinations is infeasible. The following two approximate algorithms both solve the problem, but have certain advantages and disadvantages depending on search parameters.

Similarity Profile Algorithm

Our first approach is based on similarity profiles and shown as Algorithm 5.4. With the completeness tree, we determine a set of relevant training query/result record pairs for estimating the completeness of queries. We can interpret each distinct similarity value combination as a similarity profile, i. e., a specific configuration of the query plan thresholds. There must be at least one query that could have been successfully answered using this profile. Thus, the profile is a good candidate for setting the query plan's thresholds. The resulting algorithm is fairly simple: We iterate over all similarity threshold combinations (similarity profiles) for the query (which we determined with the completeness tree). For each profile, we create a plan by using the profile similarities as thresholds. The valid plan with highest completeness is the result.

Top Neighborhood Algorithm

Our second approach for threshold optimization iteratively lowers thresholds to find the best threshold combination. In Section 5.2.3, we have introduced this algorithm as the top neighborhood algorithm. In contrast to the pseudo-code shown in Algorithm 5.1 (p. 60) for static threshold optimization, we use the query-specific functions $comp(p, q)$ and $cost(p, q)$ for estimating completeness and cost here (Line 6).

Algorithm Analysis

The similarity profile algorithm iterates over all combinations of similarity values in the training data set T. An upper bound for all combinations is $\prod_{a \in A} |\Theta_a|$, where A denotes the set of all attributes. However, the number of actually occurring combinations is typically much smaller and depends on the number and diversity of the training instances.

As we have pointed out in Section 5.2.3, the complexity of the top neighborhood algorithm is linear in the number of predicates in the query plan, the number of possible thresholds of all predicates, and the top neighborhood size t.

We empirically compare the algorithms in Section 5.6.2.

5.6 Evaluation of Query-specific Planning

In this section, we discuss evaluation results on real-world data. We introduce the dataset and experimental settings in Section 5.6.1. In Section 5.6.2, we compare the discussed query-specific planning algorithms with previous work on static query plans as well as with related work and we discuss several aspects of our approach in greater detail.

5.6.1 Dataset and Evaluation Settings

We again use the real-world data set from Schufa that we initially described in Section 3.3.1 for the evaluation of our approach. For the experiments in this section, we randomly selected 1,000 of the manually evaluated queries (the most difficult cases).

In the following, we give an overview on the evaluated approaches. We refer to a template with a prefix **T**, and to a plan with a prefix **P**. Recall that the default threshold assignment for a template sets all thresholds to 1.0. We use this default assignment in the following to compare the results for templates and derived plans.

Query templates or plans can be statically created at compile time (see Section 5.2). For the experiments in this section, we pre-compiled the best template and plan (on a set of 100k training queries).

We include the following two static approaches for comparison:

- **T Static:** Select the best static query plan template.
- **P Static:** Select the best static query plan.

The query-specific planning approach first selects an appropriate template for the query (Section 5.4); we have two approaches for selecting templates (Greedy and Exact). We then optimize the plan's thresholds (Section 5.5); we evaluate the similarity profile algorithm as well as the top neighborhood algorithm. Overall, our approach creates six query plans for a query:

- **T Greedy:** Select the best query plan template for the given query with the greedy algorithm.

- **T Exact:** Select the best query plan template for the given query with the exact algorithm.
- **P Greedy Prof:** Select the best template with the greedy algorithm, then optimize the plan's thresholds with the similarity profile algorithm.
- **P Greedy TNA:** Select the best template with the greedy algorithm, then optimize the plan's thresholds with the top neighborhood algorithm.
- **P Exact Prof:** Select the best template with the exact algorithm, then optimize the plan's thresholds with the similarity profile algorithm.
- **P Exact TNA:** Select the best template with the exact algorithm, then optimize the plan's thresholds with the top neighborhood algorithm.

As another related system, we compare with an approach by Christen et al. [Christen et al., 2009]. They propose using a fixed accumulation function for calculating the similarity of a query record to the result records and ranking the results. Their result set contains all records where at least one of the attributes contains a value similar to the query record. We achieve the same result set by using the query plan that contains a disjunction of all attributes, which we discuss in Section 5.6.2 as one of the naïve query plans.

We performed all tests on a workstation PC. Our test machine runs Windows XP with an Intel Core2 Quad 2.5 GHz CPU and 8 GB RAM. All data as well as inverted and similarity indexes are stored as tables in a PostgreSQL (Version 9.0.1) database. In the database, the original data tables require 26 GB, the inverted indexes require 2 GB, and the similarity indexes require 19 GB.

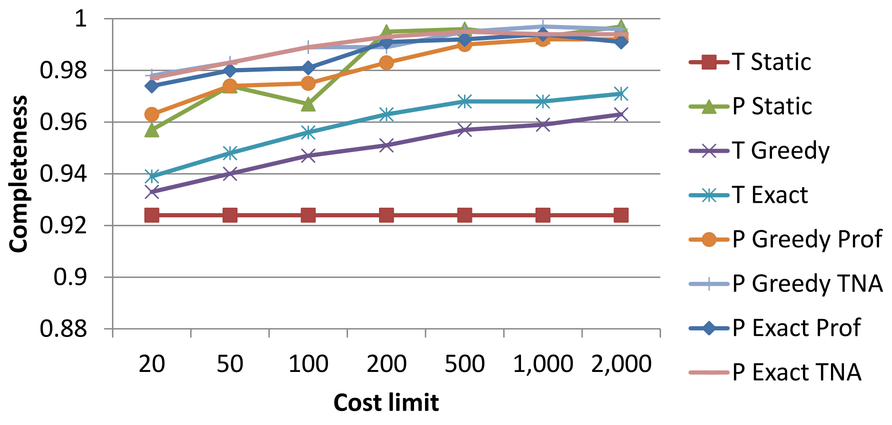
5.6.2 Results

Naive Plans

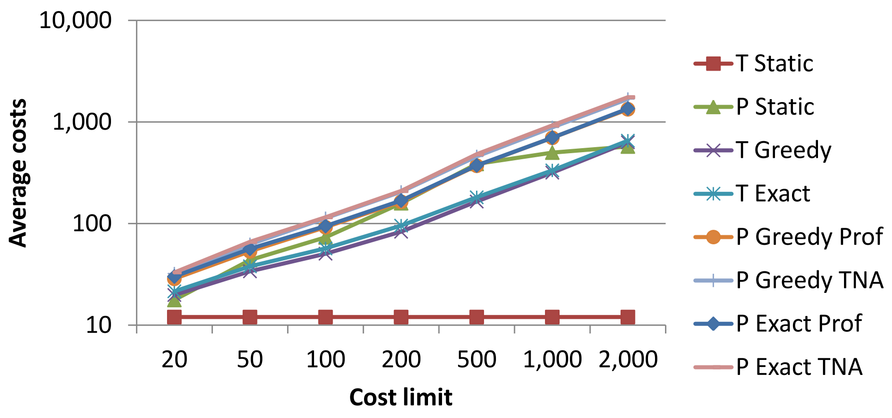
First, we evaluated two naïve query plans, namely a query plan p_c that contains a *conjunction* of all attributes, and another plan p_d that contains a *disjunction* of all attributes, which corresponds to the result set of the approach by Christen et al. While both query plans are not useful for actually answering queries, they do provide a means for describing the difficulty of the selected query data set. With p_c, we achieve a completeness of only 0.063, while p_d can answer all queries correctly (completeness 1.0). This means that only few queries (almost) completely agree with the matching record, and all queries contain at least one correct attribute value (note that this a coincidence, as there also might be queries allowed that contain errors in *all* attributes). The average cost of p_c is 0.1 (meaning in many cases no record is covered at all), and for p_d, we have unacceptably high cost of 622,526.1, which means that we would scan and compare almost 1 % of the database with the query. Selecting a query plan that is less strict than p_c, so that higher completeness can be achieved, and more strict than p_d, so that cost can be reduced, is subject to our query planning algorithms discussed in the following.

Comparison of Planning Algorithms

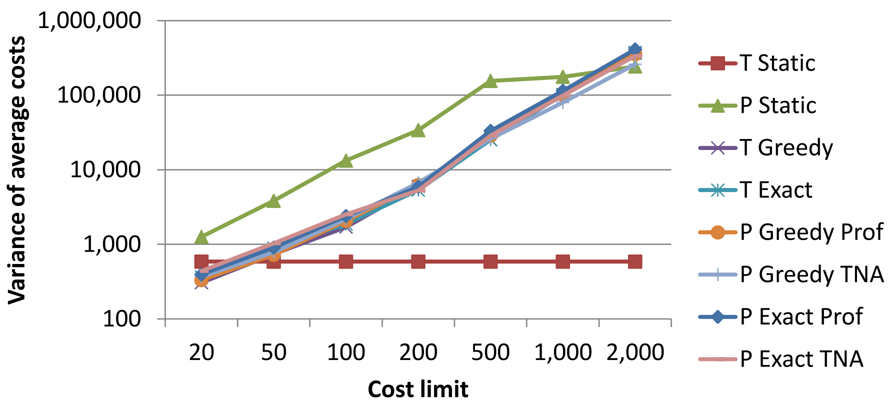
We have evaluated the planning algorithms presented in this chapter with different cost limits. Note that for the static plans, we needed to prepare plans for all cost limits that we used in the experiment, while for the query-specific plans, no preparation for the selected thresholds was necessary (as we can use any threshold in our algorithm).



(a) Completeness



(b) Cost



(c) Variance of Cost

Figure 5.10: Results per cost limit

In Figure 5.10, we show average completeness, cost, and variance of cost for all planning algorithms. Regarding completeness (Figure 5.10a), all algorithms achieve relatively high values compared to the naïve plan p_d (completeness 1.0). For the static template (T Static), we observe a constant value for all cost limits (this holds true for all evaluated measures). This is due to the fact that our algorithm could not find any better template with average cost above 20. Next, we observe that the static template (T Static) as well as the query-specific templates (T Greedy and T Exact) achieve the overall worst results. The static plan (P Static) performs better; however, for low cost limits, the static plan is clearly outperformed by all query-specific plans. Note that the absolute completeness value of the best query-specific plan algorithm for a cost limit of 20 is already 97.8 %, a very high value regarding the difficulty of the queries and the low cost limit. With cost limits of 200 and higher, the static plan and the query-specific plans all perform quite well. Comparing the different planning algorithms, we observe that (1) the exact template selection (T Exact) achieves noticeably better results than the greedy algorithm (T Greedy), and (2) the top neighborhood algorithm (P Greedy TNA and P Exact TNA) outperforms the profile-based algorithm (P Greedy Prof and P Exact Prof) for lower cost limits.

As can be seen in Figure 5.10b, the average query cost largely corresponds to the selected cost limit, as expected. Cost for templates are always below cost for plans, which makes perfect sense, as the templates are the strict basis for the selected plans. Up to a cost limit of 500, the cost for the static plan also corresponds to the cost limit; for higher cost limits, there is only little cost increase. The reason for this behavior is that the best plans that the static plan optimization algorithm found were not much more expensive, even with the higher cost limits. In an analysis of the created static plans, we have seen that the selected plan already covers all training instances that we used for learning the static plan, so that no plan with higher completeness can actually be found.

In addition to adhering to the cost limit with its average cost, a good search application should also have low variance in cost to have reliable query time. We show the variance of the average cost in Figure 5.10c. The static plan results in significantly higher cost variance than the query-specific plans (note the logarithmic scale in the figure). The reason is that the static plan has very high cost for frequent query values and very low cost for rare query values, while the query-specific plans are adjusted to the frequencies of the query values. Only for very high cost limits, the variance of the static plan does not increase due to the comparably low cost of the determined plans (as can be seen in Figure 5.10b). The remaining portion of the variance can be explained with the estimation error of our cost model that we discuss in a later experiment (cf. Figure 5.14).

Comparison with Related Work

Prominent previous top-k retrieval algorithms are Fagin's Algorithm and its successor, the Threshold Algorithm (TA) [Fagin et al., 2001]. Fagin et al. work with a set of sorted lists to retrieve records with values similar to the query values (our similarity index approach offers similar sorted access). TA retrieves records in a round-robin style from the sorted lists (sorted access) and determines all missing base similarity values (random access).

To compare our approach with TA, we perform a top-1 search, i. e., we are interested only in the best matching record. Our goal is to determine the number of overall comparisons that are needed by our approach and by TA. Every retrieved record is counted as one comparison, which is less fine-grained than an analysis of the required number of attribute similarity calculations (random accesses). To compare within our cost-limited problem setting, we determine the completeness of the results of TA and our approach after reaching the specified cost limits. Results are shown in Figure 5.11.

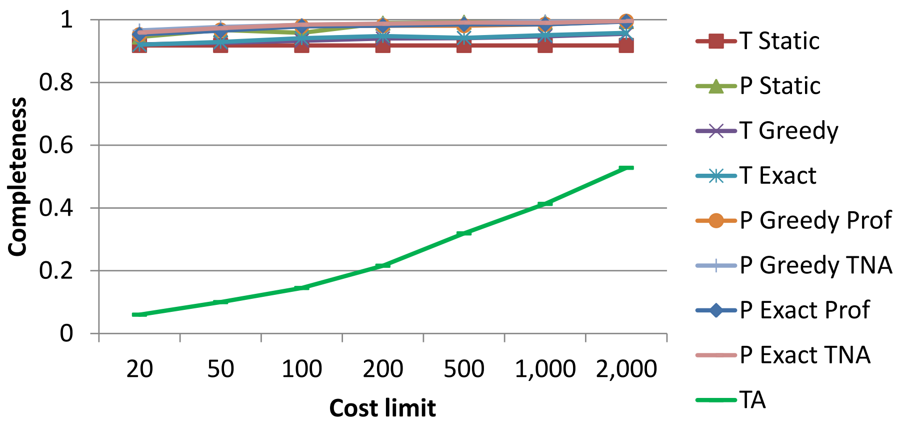


Figure 5.11: Comparison with Threshold Algorithm (TA)

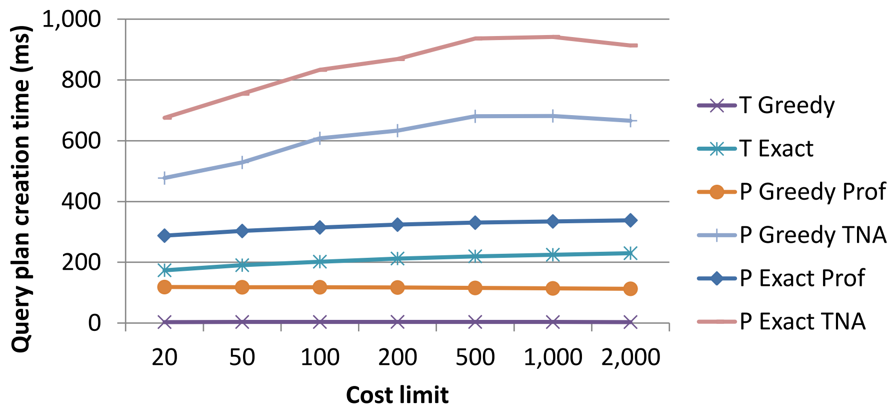
We can see that for all analyzed cost limits, TA performs significantly worse than our approach. The reason can be found in the distribution of similar values: For the various attribute values, we can have many records with exact matches (e. g., thousands of persons with the same zip code and thus with a similarity of 1.0). If we retrieve records ordered by single attribute similarity to the query record at a time, we need to evaluate many irrelevant records, because finding the correct record in the beginning of this list is unlikely. In contrast, our approach considers (the union of) intersections of the lists of records with similar attribute values and thus prefers evaluating records with *multiple* matching attribute values. An advantage of TA is the possibility to pause and resume retrieving the top results. In contrast, if we see that we could not find any relevant results with our approach, we would need to first determine a new query plan with higher cost limit and then execute the plan (skipping already evaluated records).

In Chapter 6, we propose a method for top-k retrieval that is based on TA and inspired by our query planning approaches. The main idea is to retrieve bulks of record IDs from the similarity indexes and to process the records in decreasing order of estimated overall similarity.

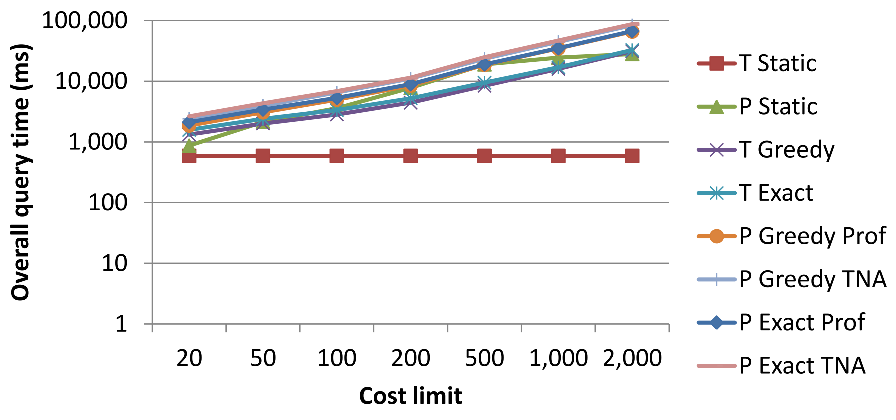
Query Time

We show query plan creation and execution time in Figure 5.12. As can be seen in Figure 5.12a, for all template selection algorithms and for the profile-based threshold optimization algorithm, the time for creating the plan is nearly constant. Only for the top neighborhood algorithm, an increase can be measured. The reason is that the algorithm explores significantly more threshold combinations as the cost limit is increased.

In Figure 5.12b, we show overall query time including creation and execution of the query plans. We can see a linear increase in overall query time for all planning algorithms, which is because most plans have cost that meet the specified cost limit (cf. Figure 5.10b). We observe that the largest fraction of the query time is spent on retrieving records from the database and applying the overall similarity measure to them. Because a well-selected plan with low cost (and thus short query execution time) can achieve more complete results than a poorly selected plan with higher cost (cf. Figure 5.10a), we conclude that the small amount of time required for selecting a query plan is well-invested.



(a) Query plan creation time (query-specific plans only)



(b) Overall query time (creation and execution)

Figure 5.12: Average time for query plan creation and execution

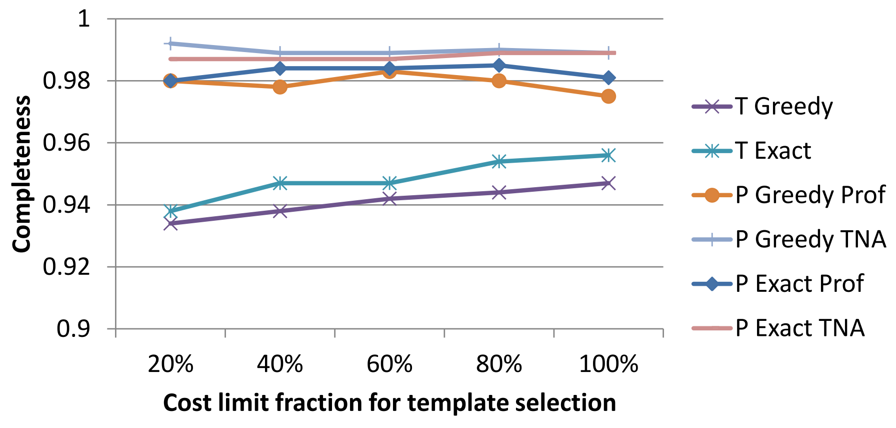


Figure 5.13: Completeness for different fractions of cost limits for template selection

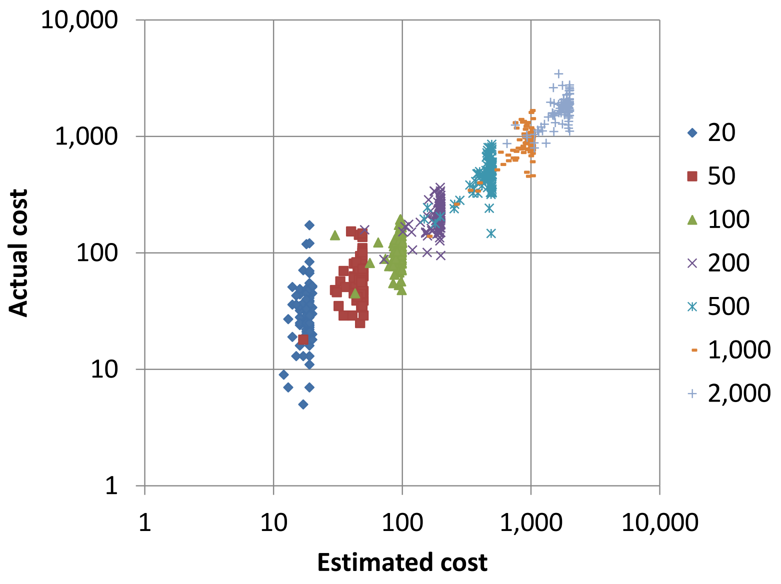


Figure 5.14: Estimated and actual cost for different query plans selected with different cost limits

Template Fraction

Previously, we have used the same cost limit for both template selection and threshold optimization. Thus, it could have been possible that the template already completely covered all available cost, so that no threshold can be lowered anymore in the threshold optimization step. An interesting question is whether we can achieve better results if the template selection may only exploit a *fraction* of the overall cost limit.

We show experimental results for an overall cost limit of 100 and several different cost limit fractions for the template selection step in Figure 5.13. While for larger fractions a noticeable increase for the completeness of the two templates can be measured, the completeness of the resulting plans after threshold selection does not seem to be affected. Because the selected template typically does not completely exploit the specified cost limit, the threshold selection algorithm is able to spend the remaining cost for a satisfying overall result, irrespective of the initial template cost limit.

Cost Model Evaluation

In Figure 5.10c, we have seen that there is a variance in the actual cost of the selected query plans. While the presented approach for query-specific planning can significantly reduce this variance, a portion of the variance can only be explained with the cost model (Section 5.4.2). For 100 randomly selected queries (a random subset of the previously used 1,000 queries), we compare the estimated and actual cost of the plans derived with exact template selection and iterative threshold selection for different cost limits (different colors and point shapes) in Figure 5.14.

The comparison shows that estimation and actual cost are typically within the same order of magnitude, so that no plan with very low estimated cost actually has cost of thousands of records (and vice versa). In some cases, however, the cost limit is exceeded; in a real-world application such slight deviations from the expected cost should not result in any problems.

Cost estimation consists of retrieving the actual attribute cost from the database and combining the results with probability theory. Thus, errors can only be introduced in the combination step. We have already pointed out that extreme deviations from the estimations can and should be precalculated. However, because every correlation of two or more attribute values has its own characteristics, a significantly better estimation can only come with a significantly more complex cost model or an increased effort of precalculating and storing all co-occurrences of all attribute values.

5.7 Related Work

In the field of *similarity search*, there is a variety of approaches for specific cases of similarity measures and data. In Section 4.1, we give an overview on similarity index structures for vector space, metric space, and non-metric space; additionally, Section 4.4 shows various approaches to string similarity search. In contrast to this work, we handle in this chapter the problem of efficient search with a monotonous similarity measure that combines several attribute-specific base similarity measures. Nevertheless, we regard the mentioned similarity index structures as helpful for implementing fast access to records with similar attribute values (see also the description of the overall search system in Section 2).

Another relevant similarity search approach has been suggested by Fagin [Fagin, 1998]. The presented algorithm retrieves the top k elements by accessing the list of elements in order of their similarity to the query object regarding different aspects (e. g., order all pictures by their similarity to *blue* and *round*). We compare our approach with an extension of Fagin's Algorithm, the Threshold Algorithm [Fagin et al., 2001], in Section 5.6 and show that our approach works significantly better in a setting with many exact attribute value matches. We also propose an approach to top k retrieval in Section 6 where we also show additional comparisons with the Threshold Algorithm.

Deshpande et al. suggest an index structure for non-metric similarity measures that exploits inverted indexes (*similarity lists*) for all values [Deshpande et al., 2008]. As stated by the authors, their index structure "AL-Tree" is only suitable for attributes with very small numbers of distinct values. In our setting with possibly millions of distinct values per attribute, this approach is infeasible.

An area related to similarity search is *duplicate detection* [Elmagarmid et al., 2007, Naumann and Herschel, 2010, Winkler, 1999]. For a given set of records, the task is to determine all duplicate entries, i. e., all sets of sufficiently similar records that refer to the same real-world entity. For similarity search, too, the problem is to find similar entries, but only for one query object. Duplicate detection is often run in a batch processing job, while similarity search usually requires an answer very fast for a satisfying user experience.

Christen et al. propose to determine similar records *before* inserting a new record into a database, thus preventing the insertion of duplicate records beforehand [Christen et al., 2009]. Similar to our approach, they exploit a set of similarity indexes. To determine the overall similarity, the authors propose to calculate the sum of the indexed base similarity values, while our approach allows to use *any* combination technique as the composed similarity measure. In terms of query planning, the approach by Christen et al. can be modeled as a query plan that contains a disjunction of all attributes. Our comparison with this approach shows that this query plan is, in most cases, considerably too expensive to be executed.

A common approach to duplicate detection is *blocking*, i. e., similar records are grouped into blocks, and then all records within each block are compared to each other [Newcombe, 1967]. The problem of finding the best blocking criterion is similar to that of finding the best query plan for similarity indexes. In our setting, the blocking predicates are similar to the attribute predicates for which we optimize the thresholds. Michelson and Knoblock suggest a machine learning approach to learn blocking schemes, i. e., selected attributes for blocking as well as similarity measures [Michelson and Knoblock, 2006]. Bilenko et al. determine an optimal blocking criterion by modeling the problem as red-blue set cover problem [Bilenko et al., 2006]. Both approaches can only decide whether the predefined blocking attribute candidates are contained in the optimal blocking criterion or not. In contrast to these approaches, our proposed algorithm supports the optimization of thresholds involved in similarity predicates (not only Boolean contained/not-contained decisions).

Chaudhuri et al. determine duplicates by calculating the union of similarity joins [Chaudhuri et al., 2007]. They reduce the problem of finding the best similarity join predicate to the maximum rectangle problem. In a second step, they unite the optimal join predicates. Their approach requires the specification of negative points (in their case: non-duplicates) as training data, which is not helpful in the similarity search setting, where for a given query almost all records are irrelevant and thus negative examples. As we cannot rely on these negative examples in our setting, we need a more sophisticated cost estimation model. In contrast to all blocking preparation approaches, our goal is to determine the best query plan *at query time*; thus, we have significantly less time for selecting the plan than an approach that determines a single plan (or blocking criterion) in advance.

5.8 Conclusion

We presented two approaches to query planning for similarity search with arbitrarily composed similarity measures. With the static planning approach, a plan can be prepared at compile time, saving preparation cost at query time. Moving closer to the notion of database query planning, we also proposed as a second approach to create a new query plan for each query. Evaluation on real-world data shows that this query-specific planning approach significantly reduces variance in cost and increases average completeness. In addition, the ability to assign different plans to individual queries also enables adhering to query-specific cost limits. As an example, consider a large credit rating agency that has different types of clients. E-commerce clients have strict runtime limitations; they do not want to put an online transaction at risk even if some queries are incompletely answered. In contrast, banks have typically highest requirements on complete and correct results and are willing to invest more time. A query-specific cost parameter can express these user profiles in a single search system.

6 Bulk Sorted Access for Top-k Query Processing

In the previous chapter, we introduced an approach for answering similarity range queries. Another typical approach for similarity search is to automatically retrieve the k records most similar to the query record (where k is determined by the application).

Fagin's Threshold Algorithm (TA) is among the most well-known algorithms for retrieving the top-k records from a database [Fagin, 1998, Fagin et al., 2001]. The main idea is to retrieve similar records from several sorted lists. The lists offer a view on all records, each sorted by their similarity regarding *one* of the base similarity measures.

However, in many cases, TA and its variants [Ilyas et al., 2008] cannot handle the data well: For instance, consider a database of US citizens and a query for a person with the first name `Peter`, last name `Smith`, and city `New York`. TA prepares a sorted list for each given query value. Because of the frequent query values, all sorted lists start with many records with a similarity value of 1.0 (representing exact matches). In these sorted lists, all records with the *same* similarity value are ordered arbitrarily. In particular, records with only one matching attribute may have lower positions in the lists than records where several attributes match.

We exploit these observations in our Bulk Sorted Access Algorithm (BSA) that we introduce in this chapter. Our idea is to first perform a bulk sorted access, i. e., to retrieve a bulk of records with a similarity value above a threshold from each sorted list, in particular *all* those with same similarity values. We then process the records according to their priority (the maximum achievable similarity of each record): With the available information, we perform comparisons with the most promising records first – a significant advantage over previous top-k approaches if only limited query time is available.

As in the previous chapters, we use the definitions of base similarity measures (Definition 3.1) and composed similarity measures (Definition 3.2) from Section 3.1.1 as the basis of our search problem. For the algorithms discussed in this chapter, the composed similarity measure must be monotonous (Definition 3.3).

Definition 6.1 (Top-k Query)**.** *Given a query record $q \in U$ and a record set $R \subseteq U$, a* top-k query *retrieves a set S of k records from R where the following condition holds (following [Zezula et al., 2006]):*

$$\forall r_k \in S : \forall r_n \in R \setminus S : sim_{Overall}(q, r_n) \leq sim_{Overall}(q, r_k)$$

Our goal is to answer top-k queries with a low number of retrieved records and overall similarity calculations, especially in the presence of many attribute values with same similarity.

The remainder of this chapter is structured as follows: We describe and compare the Threshold Algorithm as well as our proposed algorithm in Section 6.1. Comparative evaluation results are shown in Section 6.2. Section 6.3 describes relevant related work, and the chapter concludes in Section 6.4.

6.1 Algorithms

One of the most popular top-k retrieval algorithms is Fagin's Threshold Algorithm [Fagin et al., 2001], which we discuss in this section and is the basis of our work. Thereafter, we present our Bulk Sorted Access Algorithm.

6.1.1 Threshold Algorithm

The Threshold Algorithm (TA) requires the generation of one *sorted list* per attribute. This list contains the IDs of all records in the database, sorted by their similarity to the query record regarding one attribute. The lists are accessed in a round-robin fashion, i. e., after retrieving one record ID from the first sorted list, the next record ID is retrieved from the second list, etc. An access to a sorted list is called *sorted access*. For each retrieved record ID, all missing attribute similarities are determined. To determine a missing similarity, the according sorted list must be scanned, and this is called *random access*. With all attribute similarities at hand, the overall similarity is determined with the overall similarity function. The k records with highest overall similarity seen so far are stored in a result list. Ties are broken randomly.

For each sorted list, the attribute similarity of the last retrieved record is stored in an array. The array of all attribute similarities is aggregated with the overall similarity measure into an overall similarity – the *threshold value* φ. When k records with a similarity of at least φ have been seen, TA stops and returns the k seen records with highest similarity. In this case, no unseen record can have a higher overall similarity.

While TA can save many comparisons (i. e., sorted and random accesses) by stopping the process as early as possible, we argue that TA's round-robin retrieval has several drawbacks:

- **Low distinctness:** If there are many records with the same similarity (for instance, if they have the same attribute value), there is no use of the sortation for those records. TA then depends on the (random) position of the matching record in the sorted (sub-)list. An extreme case is a similarity measure that returns only 1 or 0, where sorting becomes almost useless.

- **Similarity outliers:** TA does not recognize records that have high similarities for multiple attribute values. In a situation where many records have a high similarity in only one attribute ("similarity outliers"), TA spends much effort on processing probably irrelevant records. For example, given a query for a person with the first name `Ronald` and city `San Jose`, TA considers all people with first name `Ronald` (and arbitrary city), and separately people living in `San Jose` (with an arbitrary first name). Especially when only limited query time is available, the effort should better be invested in records with *several* high attribute similarities (in the example: records with first name `Ronald` *and* city `San Jose`), as they are more promising candidates for overall high similarities.

- **Complete sorting:** TA requires the calculation of complete sorted lists, i. e., the attribute similarity to each record in the database must be calculated. One solution could be the creation of a very large index that contains pre-calculated similarities of all possible value combinations. However, it may be much more efficient to calculate only high similarities. This may be achieved by efficiently pruning the search space, so that attribute values that result in very low similarities are not considered at all.

We have observed these problems in several real-world data sets. In the following, we describe how to handle the problems in an improved algorithm.

6.1.2 Bulk Sorted Access Algorithm

Our Bulk Sorted Access Algorithm (BSA) addresses drawbacks of TA. The main idea of BSA is to first retrieve high similarity records for all attributes, and then combine the results into a priority queue. With BSA, the most promising records are considered first, and the search can often be stopped even earlier than with TA. BSA sequentially performs the following steps:

1. Perform bulk sorted access
2. Aggregate values
3. Build priority queue
4. Process record groups

First, we perform a *bulk sorted access* for every available attribute $a \in A$ and its given threshold θ_a. We retrieve all records $r \in R$ with $sim_a(q, r) \geq \theta_a$. We later discuss the significance of appropriate selection of the *retrieval threshold* θ_a. For every retrieved record, we now know the similarity regarding at least one attribute. We store the available information for each record in a table.

In contrast to TA, BSA does not rely on the presence of a complete sorted list per attribute. For BSA, it is sufficient to have the set of records with an attribute similarity of at least θ_a, which may be determined efficiently using similarity indexes.

The next step is to *aggregate the retrieved attribute information.* We observe that the combinations of attribute similarities are often not unique. For instance, there may be many records where we have retrieved only the attribute similarity $sim_{Name}(q, r) = 1$. To save any further similarity calculations, we group all records according to the retrieved attribute similarities. A group of records consists of all records with the same values for all attribute similarities (including missing similarities).

We then build a *priority queue* to determine the order in which the retrieved record groups are processed. Our goal is to have the most promising record groups at the top, i. e., the records with probably highest similarity should have the highest priority. For any attribute a of a record r from a record group, we either have retrieved the exact attribute similarity $sim_a(q, r)$ or we know that θ_a is the highest possible similarity (because otherwise we would have retrieved the exact similarity). This allows an upper bound estimation for any retrieved record group: We determine the highest possible overall similarity using $sim_{Overall}$ on the attribute similarities (either retrieved or with the threshold θ_a) and use the result value as the priority.

Finally, we *process the record groups* according to the determined priority. The processing order of records within any group is irrelevant as any such record has the same available attribute similarities. Record processing works as described above for TA: All missing attribute similarities are determined using random accesses, and the overall similarity is calculated. At any point, we keep a list of the k records with highest overall similarity seen so far. Ties are broken randomly. We define three different search modes that terminate the search at different points:

– **Complete top-k mode:** After processing a group, we can decide whether we need to evaluate any further group. We use the smallest similarity of the current k highest observed similarities as the *threshold* φ. If the highest possible similarity of the next group (i. e., its priority value) is lower than φ, then the next group as well as all other groups can be discarded. This mode resembles TA's stopping criterion.

– **Limited top-k mode:** Another possible stopping criterion can be defined using a cost limit, i. e., when a pre-defined number of records have been retrieved and compared to

Algorithm	Sorted accesses	Random accesses
TA	$\min_{a \in A}(p_a) \cdot \lvert A \rvert$	$\min_{a \in A}(p_a) \cdot \lvert A \rvert \cdot (\lvert A \rvert - 1)$
BSA	$\sum_{a \in A} s_a$	$\left[1, \sum_{a \in A} s_a \cdot (\lvert A \rvert - 1)\right]$

Table 6.1: Number of accesses for BSA and TA for $k = 1$

the query record. Especially with limited query time, we may not be able to perform all comparisons required for complete top-k mode. Because the records are ordered by a priority that ideally resembles their similarity, we expect to find the most relevant records during the first comparisons.

– **Range mode:** To perform range search, we process only the groups with a maximum similarity that is greater than or equal to the specified threshold. All other groups cannot contain any relevant records.

6.1.3 Analysis

We analytically compare BSA and TA. For a query q and an attribute $a \in A$, there is one sorted list containing all records $r \in R$ ordered by $sim_a(q, r)$. Although the generation of complete sorted lists is not required by BSA, we still assume their existence to allow a comparison of BSA and TA. To simplify analysis, we consider only the case $k = 1$, i. e., only the most similar record is to be found. Every record in the sorted list has a position, where 1 is the position of the most similar record and $|R|$ is the position of the least similar record. The function $pos_a(r)$ returns the position of the record r in the sorted list for the attribute a. We use the following notion to refer to different positions in the sorted lists:

- p_a is the position of the *overall* most similar record (which is initially unknown):

$$p_a := pos_a(\arg\max_{r \in R}(sim_{Overall}(q, r)))$$

- s_a is the last position of any record r with $sim_a(q, r) \geq \theta_a$:

$$s_a := \max(\{pos_a(r) \mid r \in R \wedge sim_a(r, q) \geq \theta_a\})$$

We summarize the number of sorted and random accesses performed by BSA and TA in Table 6.1.

TA Analysis

TA retrieves records from all sorted lists in a round-robin style. TA can finish only when the matching record has been retrieved, and (due to the round-robin retrieval) a similar amount of records has been retrieved from the other attribute lists. In favor of TA, we assume that the overall similarity of the record at p_a is higher than the highest possible similarity of $p_a + 1$, so that TA can immediately finish the search. For all retrieved records, the similarity regarding all other attributes must be calculated using random accesses.

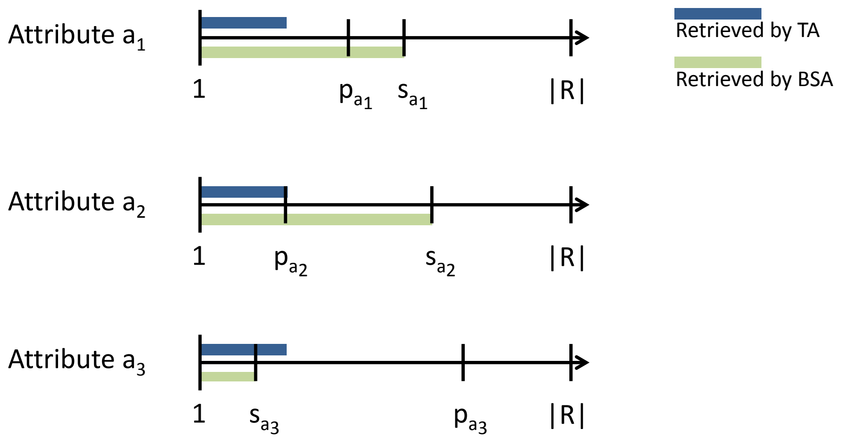


Figure 6.1: Example for positions p_a and s_a in several sorted lists

BSA Analysis

BSA first retrieves all records from the sorted lists with a similarity above the respective thresholds. For this retrieval step, the value of p_a is irrelevant. Next, the priority queue is built and processed. The required number of random accesses depends on the position of the overall most similar record in the priority queue. In the *best case*, random accesses are required only for the first element of the queue (or the first k elements, respectively). If the resulting similarity is higher than the maximum similarity of the other elements in the queue, processing can immediately be stopped. In the *worst case*, all retrieved records must be considered, and random accesses are required for all missing attribute similarities.

Comparison of TA and BSA

Due to different parameters and stopping criteria, the results of a comparison of BSA and TA are not obvious. In the following, we compare TA and BSA regarding the number of retrieved records and required comparisons with the query record, and show in our experiments in Section 6.2 that BSA outperforms TA in many cases.

Whether BSA or TA perform better on a data set depends on the actual similarity distributions and thus on the values of p_a and s_a. Figure 6.1 shows an example with positions p_a and s_a for three attributes. For attributes a_1 and a_2, where $s_{a_1} > p_{a_1}$, BSA retrieves more record IDs than TA. For a_3 BSA retrieves fewer record IDs; the ID of the most similar record is not among the retrieved record IDs (which is no problem as it has been retrieved by both a_1 and a_2). The number of sorted accesses of TA is primarily determined by p_{a_2}, as this is the highest-ranked position of the most similar record of all lists. The worst-case estimation for BSA depends on the positions s_a. In the worst case, the sorted lists do not overlap (only one attribute similarity is known for each retrieved record). BSA performs less sorted accesses than TA if the number of records with any attribute similarity above θ_a is lower than the position of the overall most similar record in any sorted list (plus the equal number of records for all other lists):

$$\sum_{a \in A} s_a < \min_{a \in A}(p_a) \cdot |A|$$

However, even if BSA retrieves more record IDs (sorted accesses) than TA, BSA may still require much fewer comparisons (random accesses). The number of comparisons of BSA is determined by the ordering within the priority queue, which becomes better as more similarity values per record are available. Thus, a high number of retrieved record IDs of BSA (potentially more than for TA) results in an ordering of records that allows a fast retrieval of the most similar records and a low number of required comparisons (potentially less than for TA). For the example in Figure 6.1, the position of the overall most similar record in the priority queue can be expected to be high as we have retrieved several similarity values for different attributes (a_1 and a_2) of the record.

An advantageous case for TA is given if $\min_{a \in A}(p_a)$ is very small, i. e., if the most similar record is at the very top of at least *one* of the sorted lists. This is the case, for example, if there is an attribute with an exact match for the most similar record, and if there are only very few other records with the same exact match. In this case, the most similar record can be retrieved very fast by TA.

On the other hand, BSA performs well if the position of the most similar record in the priority queue is very high. In particular, this is the case if *several* attributes of the most similar record have a high similarity. In contrast to TA, it is irrelevant how many other records also have a high similarity for a *single* attribute value.

BSA can only find the most similar record r if $\exists a \in A : sim_a(q, r) \geq \theta_a$, i. e., at least one threshold must be low enough to include r in the retrieved list for the attribute a. TA has no such limitation, because it retrieves records from the complete sorted lists until they are exhausted or no better element can be found. In practice, this limitation is insignificant: The affected records have such a low overall similarity that they are very likely to be irrelevant for the search result. Moreover, we show in our experiments in Section 6.2 that with BSA we can achieve for many queries a perfect recall also for high thresholds. Nevertheless, we next present a variation of BSA to overcome the limitation.

6.1.4 BSA-Restart

As described above, the results of BSA may be incomplete. More specifically, a record for which each attribute similarity is below θ_a cannot be contained in the retrieved set of record IDs, even if it is among the top-k records. Thus, we extend BSA by an additional step where we repeat the search with lower threshold to retrieve more records. We call the extended algorithm *BSA-Restart*.

As before, we can evaluate whether there may exist records that can be more similar than the k most similar records seen so far. We determine the highest possible overall similarity of unseen records using $sim_{Overall}$ on the attribute retrieval thresholds θ_a themselves as an upper bound. If φ (smallest similarity of the current k highest observed similarities) is greater than this upper bound, there cannot exist a more similar record in the set of unseen records. Otherwise, we perform a restart of BSA. We use lower values for θ_a to retrieve larger bulks of records, and proceed with all steps as described in the original BSA. After that, we again evaluate whether we can terminate the search process or whether another restart of BSA is required as described above.

If we start with a very high threshold, we may require several restarts, resulting in potentially many redundant calculations. With a too low threshold, we retrieve very many record IDs that must be prioritized according to the retrieved similarity values, but are not relevant for the search results. A well-selected threshold is high enough to answer most queries without restarts, but it is not too low to retrieve many irrelevant records. In Section 6.2.2, we evaluate different initial values of θ_a for real-world data sets.

6.2 Evaluation

In this section, we discuss experimental results from two real-world data sets. We analyze the performance of BSA regarding different values for the retrieval threshold. In addition, we show comparative results of BSA with TA (described in Section 6.1.1) and Upper, another state-of-the-art approach for top-k retrieval proposed by Bruno et al. [Bruno et al., 2002]. Upper retrieves record IDs via sorted access in a round-robin style (similar to TA) and schedules any further sorted and random accesses according to the expected similarity values and their upper bounds for the retrieved records.

6.2.1 Data Sets and Evaluation Settings

Schufa data set. Our first data set is the Schufa person data set, which we described in Section 3.3.1. For this chapter, we have randomly selected a subset of 10m records from the database, so that the data fits into main memory of our secure evaluation machine (due to the sensitive data, we could not execute tests on a machine with larger memory). We randomly selected 1,000 queries (where each query has a corresponding match in the selected record subset) for evaluating our system. As overall similarity measure, we use a weighted average of the attribute similarities[1].

Freebase data set. Freebase is an online knowledge base, managed by community experts. For our experiments, we have used the complete set of 2.2m person records available in Freebase (where at least a name is given). From the person records, we use the most commonly filled attributes name, birth date, birth place, nationality, and profession. We randomly selected 1,000 records from the complete record set as queries. As for the Schufa data set, the similarity measure is a weighted average of the attribute similarities.

For both data sets, we performed all tests on a workstation PC. Our test machine runs Windows XP with an Intel Core2 Quad 2.5 GHz CPU and 8 GB RAM. To determine the set of similar values for a given attribute value in a query, we simply iterate over all attribute values and calculate all similarities. We do this for all evaluated queries and search settings. For the comparison with TA and Upper, we require a complete sorted list for each attribute, anyway. However, recall that BSA in fact needs to retrieve only IDs of records r with $sim_a(q, r) \geq \theta_a$.

6.2.2 Experiments

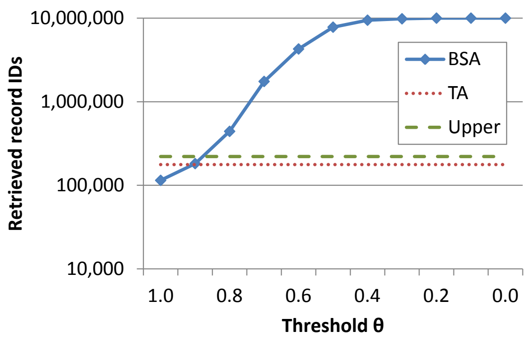
We now experimentally examine the efficiency and the recall of BSA and BSA-Resart in comparison to TA and Upper.

Number of retrieved records and comparisons

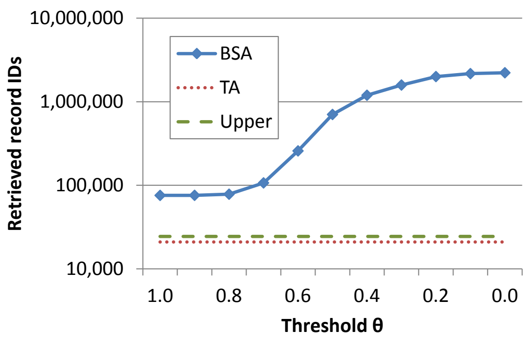
We evaluated the performance of BSA for different values of θ_a. For conciseness of the experimental results, we have used the same value θ for all attributes. Figures 6.2 – 6.5 show results for both data sets with $k = 5$; all analyzed measures are described and interpreted in the following. For BSA we report experimental results for different values of the threshold θ, while for TA and Upper the results are independent of θ.

All compared algorithms first retrieve information about the records via sorted access. Figure 6.2 shows the overall number of retrieved record IDs via sorted access. For BSA, this value

[1]Due to missing training data, we cannot use the frequency-aware measure from Section 3.2 for the Freebase data set. To have comparable results, we use the same measure (weighted average) for both data sets.

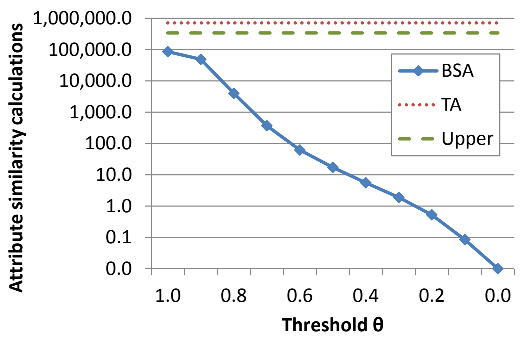


(a) Schufa data set

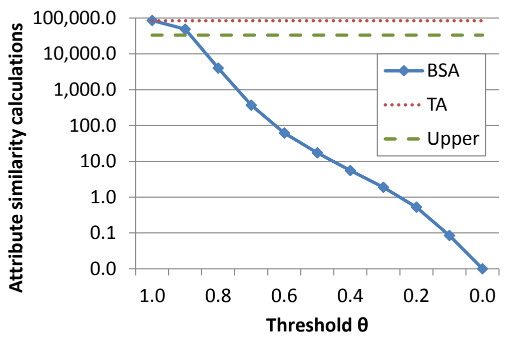


(b) Freebase data set

Figure 6.2: Number of retrieved record IDs

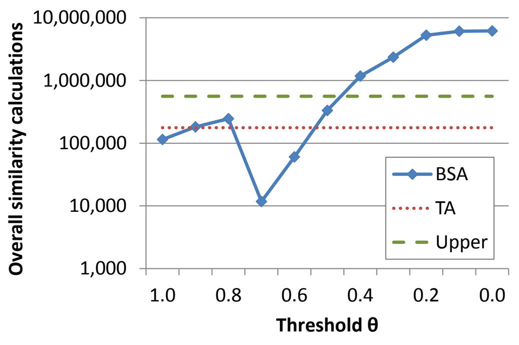


(a) Schufa data set

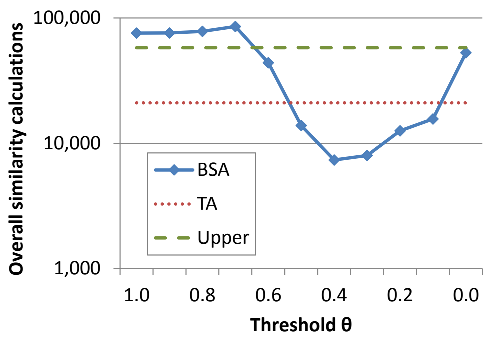


(b) Freebase data set

Figure 6.3: Number of attribute similarity calculations



(a) Schufa data set



(b) Freebase data set

Figure 6.4: Number of overall similarity calculations

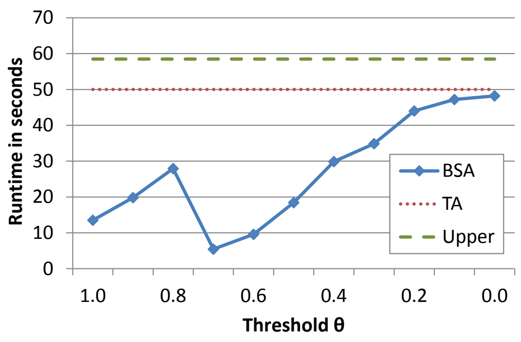
increases with smaller values for θ. For the Schufa data set, almost all record IDs are retrieved with $\theta \leq 0.3$ (for Freebase with $\theta \leq 0.2$). In comparison with TA and Upper, BSA retrieves a larger number of record IDs for $\theta \leq 0.8$ for Schufa (and for any value θ for Freebase).

After processing the retrieved information into a priority queue, BSA calculates missing attribute similarities for a number of records. In Figure 6.3, the performed attribute similarity calculations are shown. With smaller values of θ, less attribute similarities are calculated. In these cases, more similarity values have already been retrieved from the sorted lists in the previous step (Figure 6.2). Another reason for the lower number of required attribute similarity calculations is that for lower thresholds more record groups can be discarded.

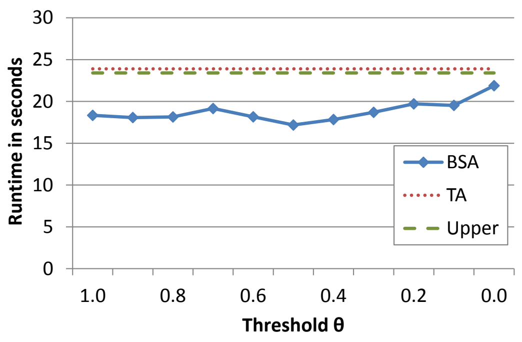
We show in Figure 6.4 the number of overall similarity calculations. BSA performs one such calculation for each group of retrieved record IDs with the same attribute similarities (to determine the priority of the group), and one for each record from all processed record groups (to determine its similarity to the query record). The number of overall similarity calculations first increases until $\theta = 0.8$ for the Schufa data set. Until this point, all record groups are processed. With $\theta = 0.7$, some record groups can be discarded (which we analyze below in more detail), and the number of groups is still relatively low. With smaller θ, more groups are discarded, but there is a much larger number of record groups, for which the priority needs to be calculated. In these cases, the dominant part of the overall similarity calculations is now performed for calculating the group priorities. Both TA and Upper perform more comparisons than the best case for BSA ($\theta = 0.7$) and less than the worst case for BSA ($\theta = 0.0$). Upper performs more comparisons than TA because it recalculates a record's priority after each attribute similarity calculation. The measurements for the Freebase data set similarly show a slight increase in similarity calculations until $\theta = 0.7$, then have a minimum value with $\theta = 0.4$ and then increase for the remaining values, when most similarity calculations are required for the larger number of record groups. Again, TA and Upper perform worse than the best case of BSA ($\theta = 0.4$) and better than the worst case of BSA ($\theta = 0.7$). In comparison to the Schufa data set, the lowest number of comparisons for BSA is reached with smaller θ due to the lower average overall similarity to the query record (see below), and the increase with even smaller θ is smaller than for the Schufa data set, which is because of the lower number of record groups.

To analyze when record groups are discarded, it is necessary to examine the similarity distribution and the used similarity measure. For both data sets, we use the weighted average of the attribute similarities. In this case, a comparison with a record is unnecessary if its similarity cannot be higher than the lowest similarity φ of the k best records seen so far. If the retrieval threshold is higher than φ, then no comparison can be saved. For any record, any attribute similarity is at least as large as the retrieval threshold θ. Thus, our estimation of the highest possible overall similarity of any record must be at least θ. Because this value is higher than φ, we cannot exclude any records. If the actual similarities of an attribute or θ are below φ, the estimation of the highest possible similarity may be below φ, so that the record can be excluded from the search. In the case of the Schufa data set, the average value of φ (for the analyzed query set) is about 0.8, so that with $\theta < 0.8$ we can exclude many irrelevant records. For the Freebase data set, we measure a value of about 0.66 for φ, which results in a large number of excluded records for $\theta \leq 0.6$.

The runtime of the compared algorithms depends primarily on the number of performed similarity calculations and retrieved records, but also on the time required to perform these operations. We show runtime measurements in Figure 6.5. For the Schufa data set, they indicate that runtime corresponds to the number of overall similarity calculations (Figure 6.4a). In our setting, we expect to retrieve record IDs (sorted accesses) fast (e. g., using an index), while any additional similarity calculations happen at query time and can be expensive. BSA requires only few attribute similarity calculations compared to the number of overall similarity



(a) Schufa data set



(b) Freebase data set

Figure 6.5: Runtime comparison

calculations, so that the number of overall similarity calculations is indeed the dominant factor for runtime. The runtime of BSA for its best case ($\theta = 0.7$) is only a fraction of the runtime of TA and Upper for the Schufa data set. All time measurements include an average time of 2.7s for preparing the lists of similar values. For Freebase, BSA performs only a few seconds better than both BSA and TA even in its best case ($\theta = 0.5$). However, in this case on average a time of 9.5s is required for preparing the lists of similar values, which shows that the gain of BSA is in fact larger than the raw numbers reveal.

Recall

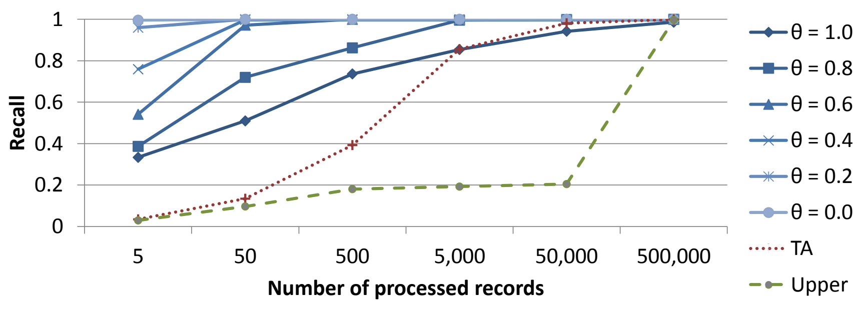
To examine how the compared algorithms perform under resource constraints, as is important for real-world search applications, we analyzed the recall after performing only a limited number of comparisons (also for $k = 5$). Results for both data sets are shown in Figure 6.6.

Both BSA and TA keep a list of the k seen records with highest similarity, which we can use for measuring recall after the limited number of comparisons. Because Upper does not keep a temporary result list, we considered a record as found only if Upper has returned it as a result, which may happen late in the query answering process. Because of this conceptual difference, Upper performs worse than both BSA and TA regarding recall for both data sets.

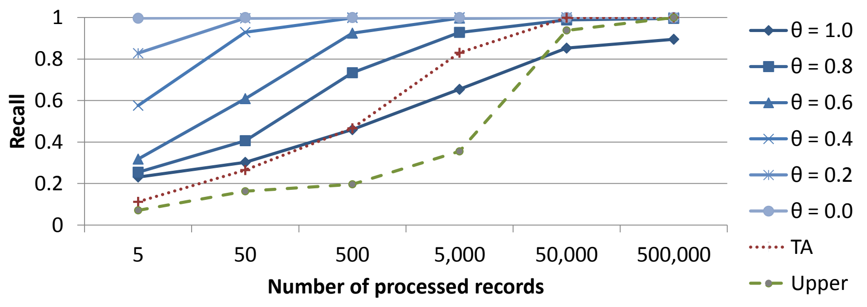
The diagrams show that with a lower value for θ, we achieve high recall earlier. The reason is that a low θ results in more available information per record, so that the estimation of the overall similarity becomes more reliable, and the priority of the record groups represents a better processing ordering of the records.

For the Schufa data set, BSA performs better than TA for any value of θ. Already with $\theta = 1.0$ and after comparing only with the top 5 records from the priority queue, BSA significantly outperforms TA. With lower values for θ, the gain of BSA is even larger, while for $\theta = 0.0$, BSA already has a recall of 1.0 (because all record IDs and similarity values are already known). After performing more comparisons, recall increases also for higher values of θ, and is in almost any case higher than the recall of TA. For the Freebase data set, BSA outperforms TA for $\theta \leq 0.8$. For small numbers of processed records, any value of θ results in a higher recall for BSA, albeit the gain of BSA is large only for smaller values of θ.

In summary, especially with a value of θ slightly below φ, BSA outperforms TA and Upper in the overall number of required comparisons as well as query time, and achieves a significantly higher recall if only a limited amount of comparisons is possible.



(a) Schufa data set



(b) Freebase data set

Figure 6.6: Recall after performing only a limited number of comparisons

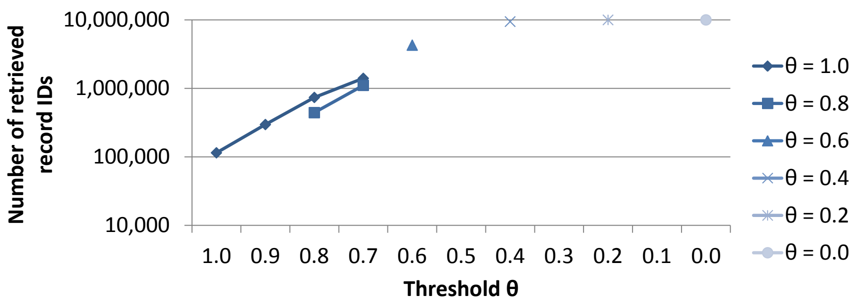
BSA-Restart

As we have seen in the previous section, we can already achieve a perfect recall after comparing with only a fraction of the records retrieved for both data sets and for most evaluated thresholds. However, during query processing we may need to retrieve a larger bulk of records to be certain that no record can exist in the data set that is more similar than the already evaluated records.

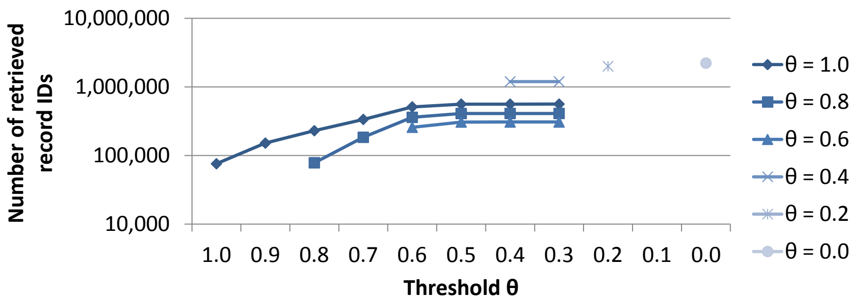
In Figure 6.7, we analyze BSA-Restart, which is guaranteed to find all k most similar records. The figure shows the numbers of retrieved record IDs for different initial values of θ as well as for any restarts with lower thresholds. Each data point represents the average for all 1,000 evaluated queries.

We first analyze the Schufa results (Figure 6.7a): With the initial threshold $\theta = 1.0$, we begin with a relatively low number of retrieved record IDs. However, for all of the 1,000 evaluated queries we need to restart the search. About half of the queries are finished with $\theta = 0.8$ and another half with 0.7. With the initial threshold $\theta = 0.8$, the first number of retrieved records is higher, but the overall number is lower than for $\theta = 1.0$, because fewer restarts are required. For any initial threshold $\theta \leq 0.6$, no restarts are required, but the initial number of retrieved record IDs is already higher than the overall numbers for higher thresholds.

For the Freebase data set (Figure 6.7b), the search for most queries is finished with $\theta = 0.7$ or $\theta = 0.6$. Thus, the lines for the initial thresholds $\theta \geq 0.6$ show more restarts than for the Schufa data set. Similar to the other data set, a lower initial threshold (0.6) results in an overall lower number of retrieved record IDs than a high initial threshold (1.0) where we have many restarts. With even lower initial thresholds, however, we have an even higher number of (unnecessarily) retrieved record IDs, yet no restarts are necessary.



(a) Schufa data set



(b) Freebase data set

Figure 6.7: Average number of comparisons for different initial thresholds (shown as lines) and after performing any required restarts with BSA-Restart

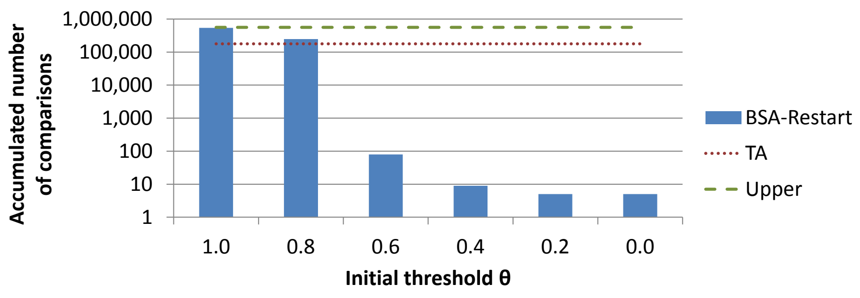
We compare the average number of overall comparisons for different initial thresholds with those of TA and Upper in Figure 6.8. In our restart implementation, all comparisons are repeated after a restart. The numbers are quite similar to the numbers for BSA (cf. Figure 6.4). As we have observed for BSA, the overall number of comparisons for BSA-Restart is much smaller than for TA and Upper for any examined value of $\theta \leq 0.6$ for the Schufa data set, and for $\theta \leq 0.4$ for Freebase.

6.3 Related Work

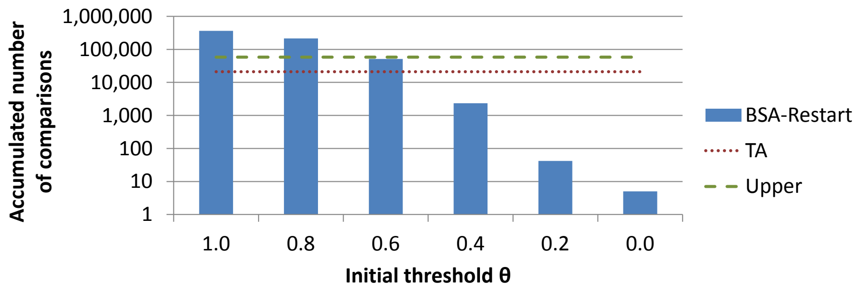
Ilyas et al. give an overview on top-k retrieval algorithms in relational database systems in a recent survey [Ilyas et al., 2008]. According to their classification, our method BSA uses “exact methods over certain data” to perform “top-k selection”. We have a “monotone ranking function” (the overall similarity measure) and use “both sorted and random probes”. With BSA-Restart, we later describe an extension of BSA that implements a “filter-restart method”.

Our work is based on Fagin’s Threshold Algorithm (TA) [Fagin et al., 2001]. We also retrieve a set of records with highest similarity for each attribute. In contrast to TA, we first perform a bulk sorted access and then aggregate available information. Because TA is among the most popular top-k retrieval algorithms and shares many ideas with subsequent work, we perform detailed analytical and empirical comparisons with TA in Sections 6.1 and 6.2.

A related approach is proposed by Bruno et al. [Bruno et al., 2002]. In the same way as TA,



(a) Schufa data set



(b) Freebase data set

Figure 6.8: Overall average number of comparisons for different initial thresholds for BSA-Restart

their approach "Upper" first retrieves records from the sorted lists. Then it builds a priority queue with the retrieved information, an idea that we pick up in our approach. Their approach then schedules the next sorted and random accesses for the different lists. In contrast to this fine-grained approach, we perform a bulk sorted access only at the beginning and then process the retrieved record lists as efficient as possible. Similar to TA, Upper performs sorted accesses in a round-robin style and has thus drawbacks similar to TA regarding records with several frequent values, as our comparative evaluation in Section 6.2 confirmed.

Some researchers have analyzed algorithms that have only a limited amount of time (or sorted and random accesses) available. In Chapter 5, we suggest to determine a query plan for accessing the sorted lists with high recall and cost below the specified cost threshold. A different approach, by Shmueli-Scheuer et al., first retrieves an initial set of records from all available sorted lists and then tries to guess which sorted list to access next to achieve highest recall given the limited amount of sorted and random accesses [Shmueli-Scheuer et al., 2009]. Their approach explicitly exploits a given budget for the amount of allowed accesses. In contrast, our approach is, in their terms, budget-oblivious. However, due to the priority queue used, our approach can still be useful in scenarios with limited query time.

6.4 Conclusion

We have proposed the Bulk Sorted Access Algorithm (BSA) for top-k retrieval, which extends the well-known Threshold Algorithm (TA). BSA is optimized for the frequent use case of queries and data sets with many same similarity values in the lists of similar attribute values, where TA and its variants perform many unnecessary comparisons. Experimental results on real-world data sets showed that BSA outperforms TA, especially for retrieval thresholds slightly below the lowest similarity of the k overall most similar records, and BSA achieves higher recall in case of limited numbers of comparisons. The performance gain of BSA comes with the need for appropriately configured similarity thresholds; too high thresholds can cause relevant records to be not found. This drawback is handled by our extended algorithm BSA-Restart, which is guaranteed to find all similar records at potentially higher costs.

7 Conclusions and Outlook

Similarity search in databases has previously been a problem with many restrictions. Database systems usually support only very few similarity measures or indexes and require much manual configuration. With this thesis, we propose a novel solution that allows a more flexible definition of the similarity measure to have only relevant records in the search result as well as efficient retrieval methods based on this measure. As attribute similarity measures, arbitrary functions can be selected. With the help of training data, our approach automatically configures the overall similarity measure based on the given data with respect to the distribution of attribute values. For fast access to records with similar attribute values, we propose an efficient similarity index for large sets of strings, but similarities can also be precalculated and stored directly in the database. Our query planning approach allows performing efficient similarity search with reliable execution cost. The query plan is selected to achieve a complete result with only few comparisons with records from the database. Because our query plan selection and execution process only requires the similarity measure to be a monotonous composition of the base similarity measures, a better modeling of the similarity measure for the specific needs of an application is possible.

This thesis describes a similarity search system that is inspired by and adjusted to the characteristics of a person database. Our industry partner is preparing the productive usage of our system within their search application. Because the person domain is one of the most important use cases for database systems, our approach can be applied to many real-world databases. To show the broad applicability of our approaches, we performed experiments on several different data sets from the person data domain. All of our approaches have only few requirements regarding the data and similarity measures, however, it remains to be shown that our work shows similarly strong results with very different domains and similarity measures.

In the following, we summarize the contributions of this thesis and provide an overview on further research direction:

- **Frequency-aware similarity measures:** In Chapter 3, we introduced frequency-aware similarity measures as an approach to improve similarity judgement. Frequency partitioning divides the record set according to the frequencies of attribute values, and a different similarity measure is learned for each partition. We have shown for different real-world data sets that this approach results in an improved overall similarity measure compared to the corresponding frequency-oblivious approach.

  Our work can be extended with the following ideas:

  - *Multidimensional partitioning*: In addition to the discussed frequency function that may also cover attribute combinations, we can define several frequency functions.

Based on the different frequency values, it is possible to perform a multi-dimensional partitioning of the data. For example, in the person data use case partitioning on first name and birth year may result in a good partitioning. As popularity of first names is often depending on current trends, names that were popular many years ago may be rarely chosen today. Also, a partitioning on last name and city can be meaningful as families are often geographically concentrated. While a multi-dimensional partitioning allows a finer adjustment of the similarity measure according to the frequency distribution of several attributes, it also requires a larger amount of training data to learn a similarity measure for each partition.

  – *Automatic selection of attributes for partitioning*: To allow easier implementation of our approach, another direction of research is to define an algorithm for automatic selection of appropriate attributes and frequency functions for partitioning. An analysis of the values of an attribute can give indications of whether the attribute is appropriate. For example, attributes with values that are equally distributed can be considered inappropriate for partitioning, because no partitions of the data with different frequency ranges can be created. To evaluate the impact of partitioning with any attribute, a sample of the data can be used. It is important to select an appropriate sample that reflects the frequency distribution of the attribute. By testing a simple partitioning strategy (such as random partitioning), an efficient evaluation can be performed to determine whether any improvements over baseline methods without partitioning can be expected and which attributes are best suitable for partitioning.

– **An efficient similarity index for strings:** To perform efficient retrieval of similar values for an attribute, we propose SSI, a similarity index for strings, in Chapter 4. Based on a prefix tree, our index structure is interpreted as an NFA and allows efficient storage and traversal. We showed that SSI outperforms related approaches for small distance thresholds.

– **Query planning for range query processing:** In Chapter 5, we define the notion of query plans and propose algorithms for automatic plan selection. Query plans are selected based on the distribution of values in the data as well as training queries. Our approach allows the flexible definition of query-specific cost limits and often achieves a complete result with only few records retrieved from the database and compared with the query record.

  For query planning, we see several promising research opportunities:

  – *Physical plan optimization*: An important direction of research is optimizing the physical query plan. Especially in distributed environments, where similarity indexes as well as data are distributed among different nodes, we believe that we can efficiently exploit the overall resources by sending the query plan fragments to appropriate nodes. As our plans typically contain a set of unions and intersections, we can also optimize the execution order to keep intermediate results and the resulting network load as small as possible. Similar ideas from traditional database query planning can be used as foundations and extended with additional elements to reflect cost of similarity indexes and similarity joins.

  – *Dynamic plan adjustment*: We pointed out in Section 5.6.2 that while our query-specific planning approach reduced variance in cost compared to static planning, a fraction of variance remains due to cost estimation errors. An approach to handle this problem is to accept the situation that cost estimations are sometimes inaccurate and to handle any problems dynamically at query execution time. For query optimization in databases, researchers proposed to generate several plan alternatives and to select the most appropriate (parts of) plans during execution [Cole and Graefe, 1994] or to dynamically rearrange operators within the plan [Avnur and Hellerstein, 2000]. If we realize during execution of a similarity query that we underestimated cost by measuring intermediate result sizes, we can re-optimize the query plan, e. g., by using higher thresholds for other attributes.

  - *Parallel query execution*: The execution of query plans can be parallelized. In particular, two aspects are well-suited for parallel execution: Retrieving IDs of relevant records from the similarity indexes and comparing selected records with the query. First, the similarity indexes can be queried in parallel. This is particularly useful if the calculation of similar attribute values takes a considerable amount of time. Because similarity calculation can take a different amount of time for all attributes, attention must be paid to good load balancing. When the set of relevant records that are to be compared with the query has been retrieved from the database, the comparisons can be performed in parallel as all comparisons are independent of each other. The selected record set can be equally distributed among the similarity comparators to achieve optimal load-balancing. All records that have a similarity above the retrieval threshold can immediately be added to the query result. We have implemented a preliminary version of the latter aspect as in our use case the comparisons take the largest fraction of the overall runtime. For this purpose, we created a main-memory based data structure that holds a subset of 10 million records from the Schufa data set. On this data set, a serial version of the query plan execution with about 5000 comparisons takes 88.2 ms. A version with four threads performing the same number of comparisons takes 28.6 ms, which corresponds to 32.4 % of the single-threaded runtime and indicates a promising improvement.

- **Bulk sorted access for top-k query processing:** To also answer top-k queries, we proposed in Chapter 6 BSA as an approach based on the retrieval of large bulks of similar records from the similarity indexes. By combining the retrieved information and processing promising records first, most relevant results are found with only few comparisons. Our empirical comparison with previous systems shows that our approach performs especially well if the number of allowed comparisons is limited.

  A useful extension of our work would be a prediction method for the retrieval thresholds θ. If queries are structured similarly to the records in our data sets, we can select several records as training queries and determine the top-k records for them. As we have seen in the evaluation, the distribution of the values of φ (lowest similarity of the top-k records) is important to select an appropriate threshold. Thus, we can automatically select retrieval thresholds that should achieve good recall in practice with a low number of retrieved record IDs if costs of restart and aggregation operations are known.

With the combination of similarity measures adjusted and automatically configured to the specific needs of an application and efficient query planning and execution methods, this thesis provides a comprehensive approach to similarity search in databases. In comparison to earlier work, our contributions remove restrictions and enable the application of similarity search to more scenarios.

Bibliography

[Aghili et al., 2003] Aghili, S. A., Agrawal, D., and Abbadi, A. E. (2003). BFT: Bit filtration technique for approximate string join in biological databases. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 326–340.

[Avnur and Hellerstein, 2000] Avnur, R. and Hellerstein, J. M. (2000). Eddies: continuously adaptive query processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 261–272.

[Banzhaf et al., 1998] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA.

[Bayer and McCreight, 1970] Bayer, R. and McCreight, E. (1970). Organization and maintenance of large ordered indices. In *Proceedings of the ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141.

[Behm et al., 2011] Behm, A., Vernica, R., Alsubaiee, S., Ji, S., Lu, J., Jin, L., Lu, Y., and Li, C. (2011). UCI Flamingo Package 4.0.

[Bentley, 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

[Bhattacharya and Getoor, 2007] Bhattacharya, I. and Getoor, L. (2007). Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):1–36.

[Bilenko et al., 2006] Bilenko, M., Kamath, B., and Mooney, R. J. (2006). Adaptive blocking: Learning to scale up record linkage. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 87–96.

[Bilenko and Mooney, 2003] Bilenko, M. and Mooney, R. J. (2003). Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 39–48.

[Bocek et al., 2007] Bocek, T., Hunt, E., and Stiller, B. (2007). Fast similarity search in large dictionaries. Technical report, Department of Informatics, University of Zurich.

[Böhm et al., 2001] Böhm, C., Berchtold, S., and Keim, D. A. (2001). Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33:322–373.

[Bruno et al., 2002] Bruno, N., Chaudhuri, S., and Gravano, L. (2002). Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems (TODS)*, 27(2):153–187.

[Burke et al., 1991] Burke, D. M., MacKay, D. G., Worthley, J. S., and Wade, E. (1991). On the tip of the tongue: What causes word finding failures in young and older adults? *Journal of Memory and Language*, 30(5):542 – 579.

[Burkhard and Keller, 1973] Burkhard, W. A. and Keller, R. M. (1973). Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236.

[Chang and Lin, 2001] Chang, C.-C. and Lin, C.-J. (2001). *LIBSVM: a library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[Chaudhuri et al., 2007] Chaudhuri, S., Chen, B.-C., Ganti, V., and Kaushik, R. (2007). Example-driven design of efficient record matching queries. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 327–338.

[Chávez et al., 2001] Chávez, E., Navarro, G., Baeza-Yates, R., and Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321.

[Christen et al., 2009] Christen, P., Gayler, R., and Hawking, D. (2009). Similarity-aware indexing for real-time entity resolution. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1565–1568.

[Ciaccia and Patella, 2002] Ciaccia, P. and Patella, M. (2002). Searching in metric spaces with user-defined and approximate distances. *ACM Transactions on Database Systems (TODS)*, 27(4):398–437.

[Ciaccia et al., 1997] Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 426–435.

[Cole and Graefe, 1994] Cole, R. L. and Graefe, G. (1994). Optimization of dynamic query evaluation plans. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 150–160.

[Comer, 1979] Comer, D. (1979). The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137.

[Deshpande et al., 2008] Deshpande, P. M., P, D., and Kummamuru, K. (2008). Efficient online top-k retrieval with arbitrary similarity measures. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 356–367.

[Dietterich and Fisher, 2000] Dietterich, T. G. and Fisher, D. (2000). An experimental comparison of three methods for constructing ensembles of decision trees. In *Bagging, boosting, and randomization. Machine Learning*, pages 139–157.

[Elmagarmid et al., 2007] Elmagarmid, A. K., Ipeirotis, P. G., and Verykios, V. S. (2007). Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):1–16.

[Fagin, 1998] Fagin, R. (1998). Fuzzy queries in multimedia database systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 1–10.

[Fagin et al., 2001] Fagin, R., Lotem, A., and Naor, M. (2001). Optimal aggregation algorithms for middleware. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 102–113.

[Fenz, 2011] Fenz, D. (2011). Effiziente Ähnlichkeitssuche in einer großen Menge von Zeichenketten mittels Key-Value-Store. Master's thesis, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam.

[Fenz et al., 2012] Fenz, D., Lange, D., Rheinländer, A., Naumann, F., and Leser, U. (2012). Efficient similarity search in very large string sets. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 262–279.

[Ferreira et al., 2010] Ferreira, A. A., Veloso, A., Gonçalves, M. A., and Laender, A. H. (2010). Effective self-training author name disambiguation in scholarly digital libraries. In *Proceedings of the Joint Conference on Digital Libraries (JCDL)*, pages 39–48.

[Fredkin, 1960] Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3:490–499.

[Garcia-Molina et al., 2009] Garcia-Molina, H., Ullman, J. D., and Widom, J. (2009). *Database Systems – The Complete Book*. Pearson Education.

[Grahne and Zhu, 2003] Grahne, G. and Zhu, J. (2003). Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the ICDM Workshop on Frequent Itemset Mining Implementations*.

[Gravano et al., 2001] Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., and Srivastava, D. (2001). Approximate string joins in a database (almost) for free. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 491–500. Morgan Kaufmann.

[Gravano et al., 2003] Gravano, L., Ipeirotis, P. G., Koudas, N., and Srivastava, D. (2003). Text joins in an RDBMS for web data integration. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 90–101.

[Gusfield, 1997] Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

[Guttman, 1984] Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 47–57.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: An update. *SIGKDD Explorations*, 11.

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160.

[Han et al., 2004a] Han, H., Giles, L., Zha, H., Li, C., and Tsioutsiouliklis, K. (2004a). Two supervised learning approaches for name disambiguation in author citations. In *Proceedings of the Joint Conference on Digital Libraries (JCDL)*, pages 296–305.

[Han et al., 2004b] Han, J., Pei, J., Yin, Y., and Mao, R. (2004b). Mining frequent patterns without candidate generation: A Frequent-Pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1).

[Ilyas et al., 2008] Ilyas, I. F., Beskales, G., and Soliman, M. A. (2008). A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11:1–11:58.

[Jackson, 1997] Jackson, J. C. (1997). An efficient membership-query algorithm for learning dnf with respect to the uniform distribution. *Journal of Computer and System Sciences*, 55:414–440.

[Jampani and Pudi, 2005] Jampani, R. and Pudi, V. (2005). Using Prefix-Trees for efficiently computing set joins. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*.

[Jaro, 1989] Jaro, M. A. (1989). Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, 84(406):414–420.

[Kalantari and McDonald, 1983] Kalantari, I. and McDonald, G. (1983). A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering (TSE)*, 9(5):631–634.

[Kotsiantis, 2007] Kotsiantis, S. B. (2007). Supervised machine learning: A review of classification techniques. *Informatica*, 31(3).

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[Lange and Naumann, 2011a] Lange, D. and Naumann, F. (2011a). Efficient similarity search: Arbitrary similarity measures, arbitrary composition. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 1679–1688.

[Lange and Naumann, 2011b] Lange, D. and Naumann, F. (2011b). Frequency-aware similarity measures. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 243–248.

[Lange and Naumann, 2013] Lange, D. and Naumann, F. (2013). Cost-aware query planning for similarity search. *Information Systems (IS)*, 38(4):455–469.

[Le Cessie and Van Houwelingen, 1992] Le Cessie, S. and Van Houwelingen, J. C. (1992). Ridge estimators in logistic regression. *Applied Statistics*, 41(1):191–201.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady.*

[Ley, 2009] Ley, M. (2009). DBLP – some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500.

[Li et al., 2008] Li, C., Lu, J., and Lu, Y. (2008). Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 257–266.

[Liu et al., 2008] Liu, X., Li, G., Feng, J., and Zhou, L. (2008). Effective indices for efficient approximate string search and similarity join. In *Proceedings of the International Conference on Web-Age Information Management (WAIM)*, pages 127–134.

[MacKay and Abrams, 1998] MacKay, D. G. and Abrams, L. (1998). Age-linked declines in retrieving orthographic knowledge: Empirical, practical, and theoretical implications. *Psychology and Aging*, 13:647–662.

[Manning et al., 2008] Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.

[Michelson and Knoblock, 2006] Michelson, M. and Knoblock, C. A. (2006). Learning blocking schemes for record linkage. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 440–445.

[Morrison, 1968] Morrison, D. R. (1968). PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534.

[Myers, 1994] Myers, E. (1994). A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374.

[Naumann and Herschel, 2010] Naumann, F. and Herschel, M. (2010). *An Introduction to Duplicate Detection.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

[Navarro, 2001] Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88.

[Newcombe, 1967] Newcombe, H. (1967). Record linkage: the design of efficient systems for linking records into individual and family histories. *American Journal of Human Genetics*, 19:3.

[Peterson, 1957] Peterson, W. W. (1957). Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146.

[Philips, 2000] Philips, L. (2000). The Double Metaphone search algorithm. *C/C++ Users Journal.*

[Postel, 1969] Postel, H. J. (1969). Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse. In *IBM-Nachrichten*, volume 19, pages 925–931.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Rabin and Scott, 1959] Rabin, M. O. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125.

[Rastogi et al., 2011] Rastogi, V., Dalvi, N. N., and Garofalakis, M. N. (2011). Large-scale collective entity matching. *Proceedings of the VLDB Endowment*, 4(4):208–218.

[Rheinländer et al., 2010] Rheinländer, A., Knobloch, M., Hochmuth, N., and Leser, U. (2010). Prefix tree indexing for similarity search and similarity joins on genomic data. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 519–536.

[Rheinländer and Leser, 2011] Rheinländer, A. and Leser, U. (2011). Scalable sequence similarity search in main memory on multicores. In *International Workshop on High Performance in Bioinformatics and Biomedicine (HiBB).*

[Russell, 1918] Russell, R. C. (1918). US patent 1261167.

[Sanfeliu and Fu, 1983] Sanfeliu, A. and Fu, K. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:353–362.

[Sarawagi and Bhamidipaty, 2002] Sarawagi, S. and Bhamidipaty, A. (2002). Interactive deduplication using active learning. In *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 269–278.

[Shang and Merrett, 1996] Shang, H. and Merrett, T. (1996). Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8:540–547.

[Shen et al., 2007] Shen, W., DeRose, P., Vu, L., Doan, A., and Ramakrishnan, R. (2007). Source-aware entity matching: A compositional approach. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 196–205.

[Shmueli-Scheuer et al., 2009] Shmueli-Scheuer, M., Li, C., Mass, Y., Roitman, H., Schenkel, R., and Weikum, G. (2009). Best-effort top-k query processing under budgetary constraints. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 928–939.

[Skopal, 2006] Skopal, T. (2006). On fast non-metric similarity search by metric access methods. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 718–736.

[Skopal and Bustos, 2011] Skopal, T. and Bustos, B. (2011). On nonmetric similarity search problems in complex domains. *ACM Computing Surveys*, 43(4):34:1–34:50.

[Skopal and Lokoč, 2008] Skopal, T. and Lokoč, J. (2008). NM-tree: Flexible approximate similarity search in metric and non-metric spaces. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 312–325.

[Stemberger and MacWhinney, 1986] Stemberger, J. P. and MacWhinney, B. (1986). Frequency and the lexical storage of regularly inflected forms. *Memory and Cognition*, 14:17–26.

[Torvik and Smalheiser, 2009] Torvik, V. I. and Smalheiser, N. R. (2009). Author name disambiguation in MEDLINE. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3:11:1–11:29.

[Uhlmann, 1991] Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179.

[Vapnik, 1998] Vapnik, V. N. (1998). *Statistical Learning Theory.* Wiley-Interscience.

[Wang et al., 2009] Wang, W., Xiao, C., Lin, X., and Zhang, C. (2009). Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 759–770.

[Whitelaw et al., 2009] Whitelaw, C., Hutchinson, B., Chung, G. Y., and Ellis, G. (2009). Using the web for language independent spellchecking and autocorrection. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 890–899.

[Winkler, 1999] Winkler, W. E. (1999). The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Census Bureau.

[Xiao et al., 2008] Xiao, C., Wang, W., and Lin, X. (2008). Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment*, 1:933–944.

[Yianilos, 1993] Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

[Yianilos, 1999] Yianilos, P. N. (1999). Excluded middle vantage point forests for nearest neighbor search. In *Proceedings of the DIMACS Implementation Challenge: Near Neighbor Searches (ALENEX)*.

[Zezula et al., 2006] Zezula, P., Amato, G., Dohnal, V., and Batko, M. (2006). *Similarity Search – The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer.

[Zobel and Moffat, 2006] Zobel, J. and Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, 38(2).