Efficient Similarity Search: Arbitrary Similarity Measures, Arbitrary Composition

Dustin Lange Felix Naumann
Hasso Plattner Institute, Potsdam, Germany
firstname.lastname@hpi.uni-potsdam.de

ABSTRACT

Given a (large) set of objects and a query, similarity search aims to find all objects similar to the query. A frequent approach is to define a set of base similarity measures for the different aspects of the objects, and to build light-weight similarity indexes on these measures. To determine the overall similarity of two objects, the results of these base measures are composed, e. g., using simple aggregates or more involved machine learning techniques. We propose the first solution to this search problem that does not place any restrictions on the similarity measures, the composition technique, or the data set size.

We define the *query plan optimization problem* to determine the best query plan using the similarity indexes. A query plan must choose which individual indexes to access and which thresholds to apply. The plan result should be as complete as possible within some cost threshold. We propose the approximative *top neighborhood algorithm*, which determines a near-optimal plan while significantly reducing the amount of candidate plans to be considered. An exact version of the algorithm determines the optimal solution. Evaluation on real-world data indicates that both versions clearly outperform a complete search of the query plan space.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms

1. INTRODUCTION

Many commercial applications maintain a person/customer data set that typically contains information such as the name, the date of birth, and the address of individuals that are somehow related to the company. 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 For example, there may be typos, outdated values, or sloppy data or query entries. A good search application needs to handle these errors effectively while returning results as fast as possible.

For these and similar problem settings, a common approach is to define several similarity measures for different aspects of the problem. These customized similarity measures focus on calculating the similarity of an aspect as effectively as possible. For example, in the person data use case, we have different similarity measures for the name of a person, for the address, and for the birth date. In the case of image similarity search, one could have different similarity measures for the color, the structure, and the textures of images [11].

To combine such similarity measures, there exist different approaches, such as the weighted sum or the minumum/maximum. Recent approaches employ machine learning techniques to learn an optimal combination of similarity measures [2, 18]. In our person data use case, we compared SVMs, decision trees, and logistic regression as combination techniques and decided for logistic regression after thorough evaluation.

Our goal is now to perform efficient search based on a set of similarity measures and an *arbitrary* composition technique. We assume that we can create a set of similarity indexes that provide efficient access to the set of records based on the defined similarity measures above a parameterizable threshold (e. g., $sim_{\mathsf{FirstName}} \geq \phi$ for an arbitrary, but fixed $\phi \in [0, 1]$). There are many approaches to create indexes for specific similarity measures, such as different string similarity measures [15, 16, 21]. For the general case, Christen et al. propose a blocking approach to precompute similarities [6]. (We later sketch how our query plan optimization approach can be exploited to optimize the required blocking criterion.)

Apart from the similarity indexes, we do not place any restrictions on the similarity measures or the composition technique. Thus, we cannot apply any of the excellent techniques for similarity search in specific data spaces, such as metric space [20] or vector space [3]. Instead, we consider our similarity measures as black boxes and want to find an efficient method for searching with them.

In this paper, we focus on range queries: Given an overall similarity measure $sim_{Overall}$, a fixed similarity threshold

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.
Copyright 2011 ACM 978-1-4503-0717-8/11/10 ...$10.00.

$\phi_{Overall}$ (the range), a query q, and a record set R, the task is to find all records $r \in R$ with $sim_{Overall}(q, r) \geq \phi_{Overall}$.

1.1 Filter-and-Refine Search

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 measure in the filter. We apply filter criteria on these indexes (e. g., *all records with* $sim_{\mathsf{FirstName}} \geq 0.9$ as well as *all records with* $sim_{\mathsf{LastName}} \geq 0.8$) 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 to each of the records that survived the filtering. The result then contains the set of records above the predetermined threshold for overall similarity ($\phi_{Overall}$).

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 the correct 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. In this paper, we define metrics to evaluate filter criteria and algorithms to derive a good filter criterion in a general setting. In the following, we refer to the filter criterion as *query plan* (to access the similarity indexes) in analogy to query plans in database systems (to access tables and indexes).

In Fig. 1, we illustrate the analogy between our approach and query processing in DBMS. In contrast to database systems, we only support one kind of query: similarity range queries with a fixed range. Thus, we only need to determine a single optimal plan for these queries. Similar to database systems, we first gather statistics about the data and generate a set of possible query plans. We optimize each query plan thresholds and need to handle the trade-off between costs and completeness (while a DBMS is only interested in the cheapest plan as every computed plan yields complete results). We select the overall best query plan based on these criteria and then create the appropriate index structures for efficient access.

1.2 Contributions

The contributions of this paper are:

- Definition of the novel *optimization problem* for efficient similarity search with arbitrarily composed similarity measures using similarity indexes. In contrast to earlier work, we do not place any restrictions on the similarity measures, the composition technique, or the data set size.

- Definition of *performance metrics* to evaluate query plans for accessing similarity indexes

- Exact and approximative *algorithms* for optimization of query plans based on the specified metrics. The approximate version evaluates only a small fraction of the valid query plans and still determines near-optimal plans. Both versions clearly outperform a complete search of the query plan space.

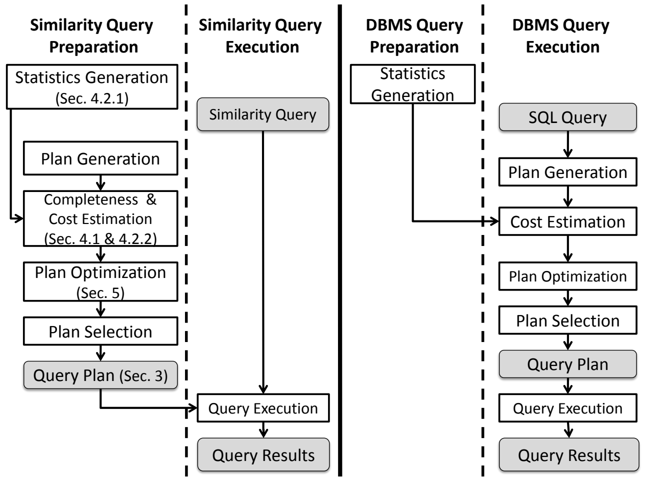

Figure 1: Comparison of our approach with a DBMS architecture

- An *evaluation* on a large real-world person data set from Schufa, the largest German credit agency (66m records, 2m queries)

The remainder of this paper is structured as follows: In Sec. 2 we describe related work in the area of similarity search. A description of the problem setting can be found in Sec. 3. We propose evaluation metrics for query plans in Sec. 4. Section 5 contains a description of our proposed query plan optimization algorithms. We apply an optimized query plan to a large real-world data set and present results in Sec. 6. The paper concludes in Sec. 7 with an outline of future work.

2. RELATED WORK

In the field of similarity search there is a variety of approaches for specific cases of similarity measures and data. If the data can be transformed into a vector space (feature space), the search of similar objects can be reduced to the search of close vectors. A popular similarity measure is the Euclidean distance. For an overview on approaches to this setting, we refer the reader to the excellent survey [3]. For another well-explored space, the metric space, the similarity measure needs to fulfill the metric requirements, while the data can have arbitrary form. In this setting, the triangular inequality can be exploited to efficiently reduce the search space [5, 20].

Another relevant similarity search approach has been suggested by Fagin [9]. 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*). The approach only works for a very limited set of basic composition techniques for the overall similarity measure; thus, Fagin's algorithm cannot be applied in our setting with arbitrarily composed similarity measures.

Deshpande et al. suggest an index structure for non-metric similarity measures that exploits inverted indexes (*similarity lists*) for all values [7]. 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. As none of the aforementioned approaches are applicable to our problem, we cannot empirically compare with them.

An area related to similarity search is duplicate detection [8, 14, 19]. For a given set of records, the task is to determine all duplicate entries, i. e., all sets of 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. [6] propose to determine similar records *before* inserting a new record into a database, thus preventing the insertion of duplicate records beforehand. 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 base similarity values, while our approach allows to use *any* combination technique as the composite similarity measure. Our approach tackles the main question that remains unanswered: Which similarity indexes should be created and queried with which thresholds?

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 [17]. 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 [13]. Bilenko et al. determine an optimal blocking criterion by modeling the problem as red-blue set cover problem [1]. 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). By exploiting the monotonicity property of these thresholds and defining neighborhoods of query plans, we can quite efficiently traverse the threshold space. Chaudhuri et al. determine duplicates by calculating the union of similarity joins [4]. They reduce the problem of finding the best similarity join predicate to the maximum rectangle problem. In a second step, they union the optimal join predicates. We handle both aspects in a unified approach of finding the best disjunction of conjunctions of similarity 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. With a more expensive evaluation function for query plans, we also need a suitable algorithm for efficiently traversing the solution space (and saving the evaluation of unpromising query plans).

3. PROBLEM DEFINITION

In this section, we define basic notations for composed similarity measures and the problem to optimize a query plan for efficient search with them (during the filter phase).

3.1 Composed Similarity Measures

We follow the common and proven 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 a set of **base similarity measures** $sim_p(r_1, r_2)$, each responsible for calculating the similarity of a specific attribute p of the compared records r_1 and r_2 from a set R of records that is a subset of a universe U of possible records. In our 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 for $sim_{\mathsf{FirstName}}$ [19], the relative distance between dates for $sim_{\mathsf{BirthDate}}$, and numeric difference for sim_{Zip}. We assume the domain of the similarity measures to be between 0 and 1, with 1 representing identity and 0 dissimilarity of the compared record parts:

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

A **composed similarity measure** uses the base similarity measure to derive an overall similarity judgement. For example, a weighted sum of the base similarity measures is one composition technique. Other techniques involve machine learning approaches, such as logistic regression, decision trees, and support vector machines. Learnable similarity measures have been addressed by several researchers [2, 18]. We make no assumptions whatsoever on the composed similarity measure other that it is composed of base similarity measures.

3.2 Query Plan Optimization

A base similarity measure **predicate** $sim_p(q, r) \geq \phi_p$ covers all records $r \in R$ for which the similarity to the query record $q \in U$ calculated with the base measure sim_p is at least ϕ_p. In the following, we abbreviate this predicate to $p \geq \phi_p$ and refer to base similarity measure predicates as attribute predicates (due to our running example, in which each base similarity measure covers one attribute).

A **query plan template** is a combination of attribute predicates (with yet unassigned threshold variables) with the logical operators conjunction and disjunction. We require query plan templates to be in disjunctive normal form (DNF), since this form is popular [1, 4] and easy to understand and modify. Note that all logical combinations of attributes can be expressed in DNF. A query plan template combines N attribute predicates and has the form:

$$\bigvee \bigwedge p_i \geq \phi_{p_i} \tag{2}$$

with $1 \leq i \leq N, \forall i : 0 \leq \phi_{p_i} \leq 1$.

Once all threshold variables in a query plan template are assigned a value, we call this a **query plan**. An example for a query plan that covers all records with similar $\mathsf{FirstName}$

and Zip or $\mathsf{LastName}$ and $\mathsf{BirthDate}$ is the following:

$$\begin{aligned} & (\mathsf{FirstName} \geq 0.9 \wedge \mathsf{Zip} \geq 1.0) \\ \vee \quad & (\mathsf{LastName} \geq 0.8 \wedge \mathsf{BirthDate} \geq 0.85) \end{aligned}$$

To evaluate query plans, we define different performance metrics. Given a set of positive training examples for query/result record pairs, the **completeness** $comp(qp)$ of a query plan qp is the expected proportion of correct query/result record pairs that are covered by qp. The **costs** $costs(qp)$ of a query plan qp is the expected average number of records in R that are covered by qp. Completeness and costs are described and analyzed in detail in Sec. 4.

Given a set R of records and a cost threshold C, the **query plan optimization problem** is to find the query plan qp that maximizes $comp(qp)$ with $costs(qp) < C$. We analyze problem properties and describe our approach to solve the problem in Sec. 5.

4. EVALUATING QUERY PLANS

In this section, we discuss how we evaluate a query plan. We evaluate based on two dimensions: **(1) Completeness:** How many matches can be found with this query plan? **(2) Cost:** How large is the cardinality of an average query result with this query plan? In general, we want to maximize completeness within an upper bound for cost. How to find such best plans is topic of the following Sec. 5.

4.1 Completeness Estimation

For a given query plan, we want to estimate how complete the results will be. We express 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 (i. e., for which the similarity of query and result records are above the query plan thresholds). Completeness is thus the probability that a correct result to a query will be found with the query plan. In the following, we describe how to gather training data and estimate completeness with it.

4.1.1 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 2m queries, most 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 $\phi_{Overall}$). 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 efficiently traverses the space of possible base similarity values. Inspiring work comes from the domain of learning logical expressions in DNF with membership-query algorithms [12].

4.1.2 Estimation

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

$$comp(qp) = \frac{|\{(q, r) \in T \mid covers_{qp}(q, r)\}|}{|T|} \tag{3}$$

Our basic algorithm to calculate $comp(qp)$ for given qp and T is very simple: We iterate over the list T and count all query/result record pairs that are covered by qp. Since there is a potentially large amount of base similarity value combinations that are checked multiple times, 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 %).

4.2 Cost Estimation

The second evaluation criterion for query plans is the involved costs. We need to estimate how many result records we can expect on average when applying a query plan to the entire data set.

For cost estimation we do not require training data, because we only exploit information from the distribution of attribute values in the complete record set R. A naive approach to estimate query plan costs would be to sample a set $S \subseteq R$ of records, exactly determine the costs for this sample by comparing each record in S to each record in R, and then average the determined costs. In our use case, exactly determining costs for one record 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 with which we can quickly estimate costs of complete query plans.

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

4.2.1 Similarity Histograms

We want to estimate the cardinality of the predicate $p \geq \phi_p$, i. e., how many records in our database have at least a similarity of ϕ_p regarding the attribute p with respect to a random query? To derive a generic query plan, this estimation needs to be independent of specific values. We rather want a general estimation for each attribute.

To achieve this, 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 2 shows a distance histogram for the last name "Lange". A reading example is: For $\mathsf{LastName} \geq 0.9$, 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.

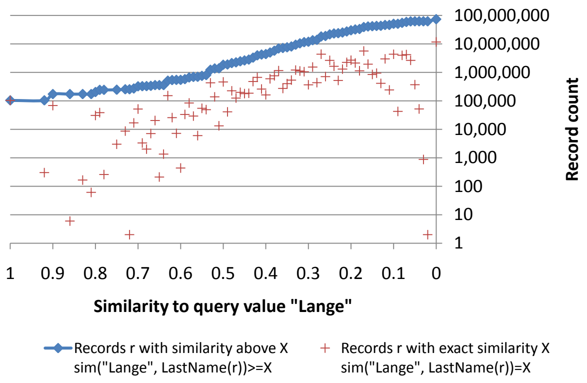

Figure 2: Similarity histogram for attribute LastName for value "Lange"

We then calculate the average of the created similarity histograms. For each possible similarity, we average the measured numbers of records with similar values. The resulting similarity histograms for our attributes are shown in Fig. 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".

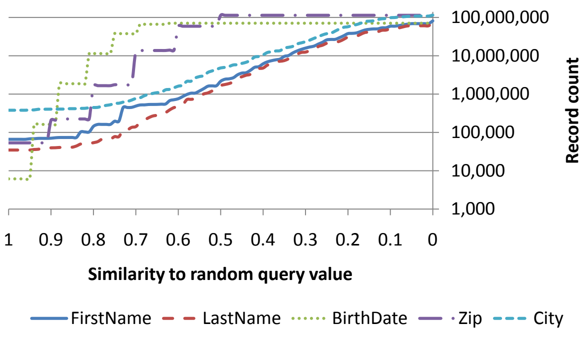

Figure 3: Average similarity histograms for all attributes

An open question 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 costs, it may make sense to include more frequent values in the sample set. More frequent values will themselves lead to higher costs 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).

4.2.2 *Combining Attribute Estimations*

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

$$costs(qp) = |R| * P(covers_{qp}(q,r) \mid q \in U, r \in R) \qquad (4)$$

By calculating costs 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 $p \geq \phi_p$ in $P(covers_{p\geq\phi_p}(q,r) \mid q \in U, r \in R)$ as $P(p \geq \phi_p)$ 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 ϕ_a and ϕ_b we want to estimate the probability $P(a \geq \phi_a \wedge b \geq \phi_b)$. To resolve the joint probability, we distinguish between attributes that are statistically dependent or independent, for ease of calculation. (Although we would theoretically be able to calculate all joint distributions, 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:

$$P(a \geq \phi_a \wedge b \geq \phi_b) = \begin{cases} P(a \geq \phi_a)P(b \geq \phi_b), \\ \qquad \text{if } a \text{ and } b \text{ are independent}, \\ min(P(a \geq \phi_a), P(b \geq \phi_b)), \\ \qquad \text{else} \end{cases} \qquad (5)$$

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:

$$P(p \geq \phi_p) = \frac{|\{r \in R | p \geq \phi_p\}|}{|R|} \qquad (6)$$

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

In a final step, we estimate the probability of a *disjunction* of conjunctions. For the union of two sets, we can 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 $P(a \geq \phi_a \vee b \geq \phi_b)$

and have:

$$P(a \geq \phi_a \vee b \geq \phi_b) \quad = \quad P(a \geq \phi_a) + P(b \geq \phi_b) \qquad (7)$$
$$-P(a \geq \phi_a \wedge b \geq \phi_b)$$

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:

$$P\left(\bigvee_{i=1}^{n} c_i\right) = \sum_{k=1}^{n} (-1)^{k-1} \sum_{\substack{T \subset \{1,\dots,n\},\\ |T|=k}} P\left(\bigwedge_{t \in T} c_t\right) \qquad (8)$$

With these estimations, we can accurately and efficiently estimate the costs of any query plan in DNF.

4.3 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 attribute, 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 which we can quickly 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 Fig. 4, 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 costs of different query plans.

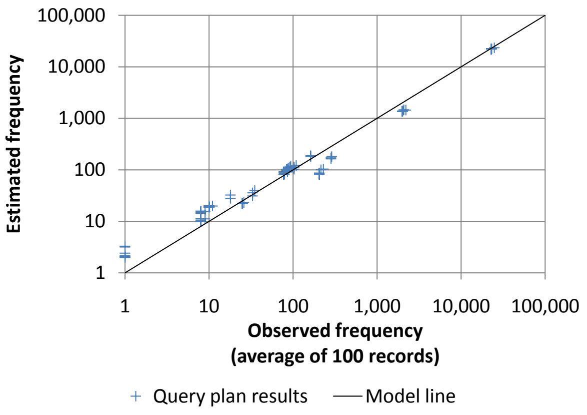

Figure 4: Comparison of observed and estimated frequencies for the 210 generated query plans. Example: For the query plan $\mathsf{BirthDate} \geq 1 \wedge \mathsf{City} \geq 1$, our estimation is 22 records, and the observed average frequency is 25 records.

5. QUERY PLAN OPTIMIZATION

Based on the performance metrics defined in the previous section, we want to find the best query plan, i. e., the plan with highest completeness below a cost threshold. We iterate over the set of all possible query templates and then optimize thresholds for each query plan template. We thus have one optimal query plan per query plan template. From these query plans, we can 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 costs for each plan to be analyzed, this large set of possible plans is infeasible to be analyzed completely; thus, more efficient algorithms are required.

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

5.1 Observations

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

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

we evaluated all possible query plans, i. e., all possible threshold values for $\phi_{\mathsf{FirstName}}$ and ϕ_{LastName}. We show the resulting comparison distribution in Fig. 5 and the cost distribution in Fig. 6.

In the cost distribution diagram, we can see that query plans with lower thresholds have higher costs. 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 costs grow exponentially. We can especially see that only few query plans in the upper right corner have acceptable costs; as we will see later, more complex plans with disjunctions of conjunctions in the query plan templates achieve better results (i. e., higher completeness with lower costs).

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 (ϕ_x, ϕ_y) has at least the costs and the completeness of any query plan with thresholds $(\phi_{x'}, \phi_{y'})$ with $\phi_{x'} <$

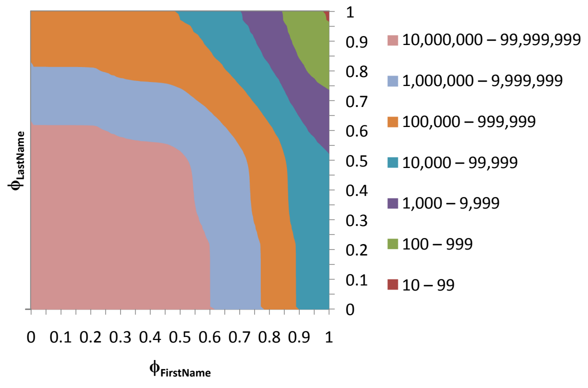

Figure 5: Distribution of costs for query plan template FirstName $\geq \phi_{\text{FirstName}} \wedge$ LastName $\geq \phi_{\text{LastName}}$

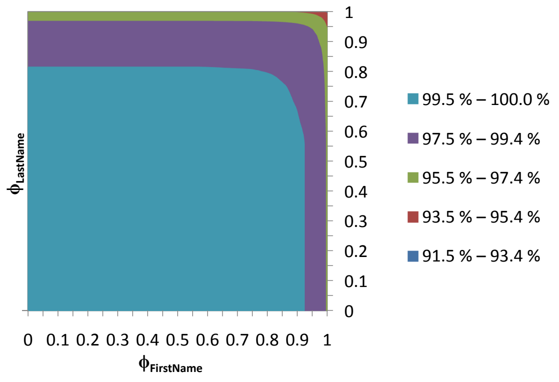

Figure 6: Distribution of completeness for query plan template FirstName $\geq \phi_{\text{FirstName}} \wedge$ LastName $\geq \phi_{\text{LastName}}$

ϕ_x or $\phi_{y'} < \phi_y$. Note that Chaudhuri et al. make similar observations and also exploit the monotonicity property in their algorithm [4]. 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 cardinality distribution diagram, we can derive the space of all valid query plans, i. e., all plans with costs 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. 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 following algorithms, reaching this line will be regarded as stopping criterion.

5.2 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.

The general idea of the **top neighborhood 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 Figs. 5 and 6), 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 qp has a **neighborhood** $N(qp)$ that contains all query plans that can be constructed by lowering one thresholds of qp by one step (e. g., by 0.01). For example, the neighborhood of a plan with two thresholds ϕ_a and ϕ_b has two elements:

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

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 costs), since we traverse the threshold space from higher to lower thresholds.

In each iteration of our algorithm, we have a **window** $W = \{q_1 \dots q_n\}$ 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 $N(W) = \bigcup_{qp \in W} N(qp)$. We then select a subset of these plans: the t plans with highest completeness (and lowest costs, 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 costs 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 costs 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 Fig. 7, 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 Fig. 6. The diagram shows the search space of all valid plans, i. e., all plans with acceptable costs (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.

Exact algorithm. The parameter t allows us to turn the approximative top neighborhood algorithm into an exact

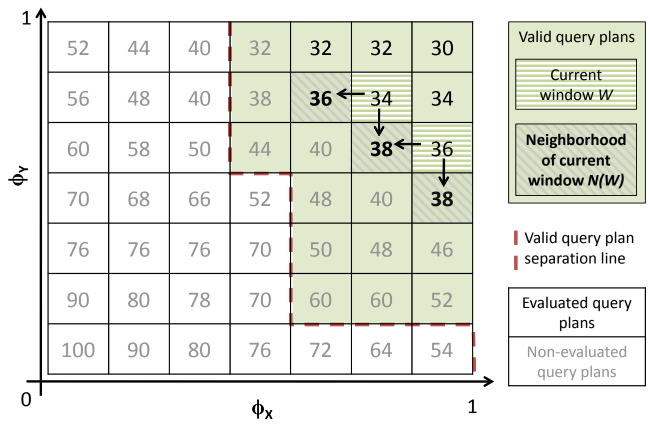

Figure 7: Application of top neighborhood algorithm to exemplary completeness estimations (in percent) for two thresholds ϕ_X, ϕ_Y in a query plan template

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 costs above C. Regarding the search space coverage, this exact algorithm is thus a better approach than a brute force search over all possible query 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

5.3 Evaluation

To evaluate the top neighborhood algorithm, we analyzed the behavior of the algorithm on our data set with 2m (correct) query/result record pairs. We ran the algorithm for three randomly chosen query plan templates with different numbers of predicates and varied the parameter t. Figure 8 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 sufficient to find a plan with near-optimal completeness (only a few of the 2m records are missing). 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, 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 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.

6. REAL-WORLD SIMILARITY SEARCH

In addition to the experiments for individual aspects of our approach in earlier sections, we also ran an experiment on the complete data set with the overall optimal query plan.

To determine the overall optimal query plan for our data set, we used the set of query plan templates from the experiment in Sec. 4.3. 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 (completeness and cost as defined in Sec. 4) in Table 1.

The first line shows a basic 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 costs 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 turned out to be optimal regarding completeness (according to our top neighborhood algorithm). We first did an exact search with this plan (line 2 in Ta-

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.

Figure 8: Evaluation results of top neighborhood algorithm for different query plan templates

Search setting	**Completen.**	**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 1: Search performance of different search settings

ble 1). In comparison to the first search setting where all attribute values were required to exactly match, this 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 costs 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. Line 3 in Table 1 shows that we achieve even better results than for search setting (2). 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 costs, 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 1) would result in perfect completeness, but also in costs of 66m, which is clearly infeasible. Our optimized query plan exploits the cost limit best and is thus an appropriate choice for efficient similarity search in the given data set.

7. CONCLUSION AND OUTLOOK

We introduced a novel approach to efficient similarity search for composed similarity measures. We showed how to efficiently compute completeness and costs, two important performance metrics for query plans based on a set of similarity indexes. The resulting trade-off between completeness and costs has been solved with the approximative top neighborhood algorithm. We showed that the algorithm efficiently determines near-optimal query plans for real-world data.

Further applications. Our proposed algorithm for optimizing query plans can also be adapted to determine a good blocking criterion. Blocking is a simple approach to improve efficiency of duplicate detection [17]. Another application scenario is the creation of a similarity index for an attribute predicate $sim_p \geq \phi_p$: We create blocks of similar values of the attribute p and then compare only the values in the blocks with each other. For each attribute value v_1, we store all values v_2 with $sim_p(v_1, v_2) \geq \phi_p$ in a sorted list for efficient access. To optimize this process, a blocking criterion needs to be defined with which as many similar values and as few dissimilar values as possible are within each block and are thus compared. The completeness of the blocking criterion should be high, and the costs should be low, similar to the formulation of the query plan optimization problem.

Our algorithm can be applied to this setting as follows: Instead of query plan templates, our algorithm now optimizes *blocking criterion templates* that may also include boolean blocking predicates. An example for a blocking criterion is:

$$(\mathsf{CommonPrefix} \geq 3) \vee (\mathsf{CommonNGrams} \geq 2)$$

This blocking criterion means that in a first blocking run, all values with a common prefix of length 3 are compared, and in a second blocking run, all values with a at least two common n-grams are compared (this approach is called multi-pass blocking [10]). Blocking criteria may also contain conjunctions, similar to query plans.

Costs of a blocking criterion are calculated as the average number of similar values for a given "query" value (to create a complete similarity index, all values will be "queried" for similar values). Similar to the description for query plans in Sec. 4.2, we can sample some values and calculate the average number of values covered by the blocking criteria for an appropriate range of blocking predicate thresholds. To create training data for *completeness* estimation, we can sample some attribute values and then compare these to all other values. In contrast to the similarity search problem (where there is usually only a very small set of objects similar to a query), there is usually a large set of values in the same block. The *top neighborhood algorithm* can work with these estimations; it starts with a very rigorous blocking criterion (with high thresholds and thus small blocks) and then lowers the thresholds until a predefined cost threshold is reached.

Future work. We leave for future work the following research directions:

- *Frequency-adaptive query plans:* We aim to develop a framework that optimizes query plans at query time with information about the frequencies of the attribute values in the query. Based on this information, a good query plan needs to be selected very fast.
- *Virtual training data:* To apply our algorithm to data sets without sufficient training data, we want to develop an algorithm for generating training data based solely on the composite similarity measures.

Acknowledgments

We thank Schufa Holding AG for supporting this work. We especially thank Stephan Springob, Boris Zerban, and Michael Stolz for their valuable input.

8. REFERENCES

[1] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *Proc. of the Intl. Conf. on Data Mining (ICDM)*, pages 87–96, Hong Kong, China, 2006.

[2] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proc. of the ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 39–48, Washington, DC, USA, 2003.

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

[4] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *Proc. of the Intl. Conf. on Very Large Databases (VLDB)*, pages 327–338, Vienna, Austria, 2007.

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

[6] P. Christen, R. Gayler, and D. Hawking. Similarity-aware indexing for real-time entity resolution. In *Proc. of the Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 1565–1568, Hong Kong, China, 2009.

[7] P. M. Deshpande, D. P, and K. Kummamuru. Efficient online top-k retrieval with arbitrary similarity measures. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, pages 356–367, Nantes, France, 2008.

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

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

[10] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2:9–37, 1998.

[11] Q. Iqbal and J. K. Aggarwal. Combining structure, color and texture for image retrieval: A performance evaluation. In *Proc. of the Intl. Conf. on Pattern Recognition (ICPR)*, pages 438–443, Quebec City, Canada, 2002.

[12] J. C. Jackson. An efficient membership-query algorithm for learning dnf with respect to the uniform distribution. *J. Comput. Syst. Sci.*, 55:414–440, December 1997.

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

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

[15] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33:31–88, March 2001.

[16] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1:2000, 2001.

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

[18] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of the ACM Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 269–278, Edmonton, Alberta, Canada, 2002.

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

[20] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*. Springer, 2006.

[21] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *International Conference on Management of Data*, pages 915–926, Nagpur, India, 2010.