# Cost-Aware Query Planning for Similarity Search

Dustin Lange, Felix Naumann

*Hasso Plattner Institute, Potsdam, Germany*

**Abstract**

Similarity search aims to find all objects similar to a query object. Typically, some base similarity measures for the different properties of the objects are defined, and light-weight similarity indexes for these measures are built. A query plan specifies which similarity indexes to use with which similarity thresholds and how to combine the results. Previous work creates only a single, static query plan to be used by all queries. In contrast, our approach creates a new plan for each query.

We introduce the novel problem of query planning for similarity search, i. e., selecting for each query the plan that maximizes completeness of the results with cost below a query-specific limit. By regarding the frequencies of attribute values we are able to better estimate plan completeness and cost, and thus to better distribute our similarity comparisons. Evaluation on a large real-world dataset shows that our approach significantly reduces cost variance and increases overall result completeness compared to static query plans.

*Keywords:* Similarity Search, Similarity Measures, Query Planning

## 1. Introduction

Similarity search is an important application in many commercial applications. As an example, consider a person data set that typically contains information such as the name, the date of birth, and the address of individuals that are somehow related to a 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 For example, there may be typos, outdated values, or sloppy data or query entries.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.

To implement an efficient similarity search system, in our previous work we suggest to prepare a filter criterion on a set of attribute-specific similarity indexes for a fixed cost limit [1]. 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 traditional indexes). In [1], a static query plan is determined in advance as a good choice for an *average* query and is used for *all* queries. Even if the average query runtime is reasonable, a fixed query plan for all queries 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 regarding user responsiveness.

In contrast, *query-specific* planning can offer a more reliable query runtime. By analyzing the attribute values in the query, we can adjust the plan to the actual query requirements. For very frequent query values, we apply higher similarity thresholds and have 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). Such flexibility results in a more reliable execution behavior and potentially more complete query results, as we show in our evaluation on a real-world dataset from a large credit agency.

With the ability to assign different plans to individual queries, we are also able to adhere 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. The cost limit is also useful for systems with limited hardware resources that may need to handle very different workloads. During very busy hours, the cost parameter may be used to reduce the invested amount per query and still being able to handle all incoming queries. The available cost limit then depends on the amount and complexity of queries executed in parallel.

Our goal is to perform efficient search, based on a set of similarity measures and an arbitrary similarity 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., $sim_{\mathsf{FirstName}} \geq \theta$ for an arbitrary $\theta \in [0, 1]$). Apart from the similarity indexes, we do not place any restrictions on the

*Email addresses:* dustin.lange@hpi.uni-potsdam.de (Dustin Lange), naumann@hpi.uni-potsdam.de (Felix Naumann)

similarity measures or their similarity composition. Thus, we cannot apply any of the techniques for similarity search in specific data spaces, such as metric space [2] 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 similarity range queries: Given an overall similarity measure $sim_{Overall}$, a fixed similarity threshold $\theta_{Overall}$ (the range), a record set $R \subseteq U$ (where $U$ is the universe of all possible records), and a query $q \in U$, the task is to find all records $r \in R$ with $sim_{Overall}(q, r) \geq \theta_{Overall}$.

### *1.1. Filter-and-Refine Search*

We briefly describe our search setting where we allow arbitrarily composed similarity measures and define query plans as filter criteria.

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 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 our paper is how to select the best possible plan for each individual query.

### *1.2. Query Planning*

We show an overview of our approach to query planning in Figure 1. At compile time, the data is indexed. For each attribute, we create an inverted index and a similarity index. Moreover, from the data and its statistics we learn a completeness tree, i. e., a data structure that is used to predict the completeness of query plans. At query time, for each query we first select a query plan template, i. e., a combination of attributes in disjunctive normal form (DNF). In the next step, we optimize the query plan thresholds to maximize completeness. In both steps, we estimate cost of query plans with the help of the indexes, and we estimate completeness using the completeness tree. The resulting query plan (with highest completeness and cost below the cost limit) is applied as filter criterion. Only for the remaining records, the overall similarity to the query is calculated.

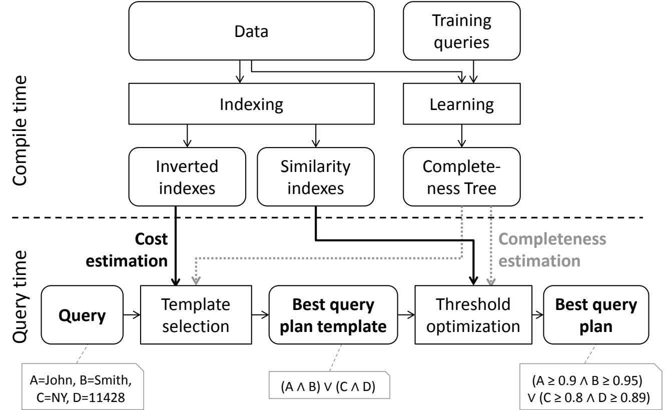



Figure 1: Process overview

This paper builds upon our work on static query planning for similarity search [1] and allows the specification of query-specific cost limits and appropriate query planning. The contributions of this paper are:

- Definition of the novel problem of query planning for similarity search with arbitrarily composed similarity measures and query-specific cost limits
- Exact and approximate algorithms for selecting query plan templates (that can be interpreted as very strict query plans) and optimizing similarity thresholds in the query plans
- An evaluation on a large, real-world dataset, which shows that we can efficiently select good query plans yielding more complete results with a more reliable query runtime

The remainder of this paper is structured as follows. In Section 2, we discuss related work. The problem setting is introduced in Section 3. We then present our approaches for selecting query plan templates in Section 4 and for optimizing thresholds in Section 5. We discuss evaluation results in Section 6 before concluding the paper in Section 7.

## 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 survey by Böhm et al. [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 [2, 4]. Because

our similarity measure can have arbitrary form, both vector and metric space approaches cannot be applied.

Another relevant similarity search approach has been suggested by Fagin [5]. 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 [6], in Section 6 and show that our approach works significantly better in a setting with many exact attribute value matches.

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.

In our previous work, we presented an approach for preparing a **static query plan** in advance [1]. The previous approach iterates over all possible query plan templates and optimizes the plan thresholds with our top neighborhood algorithm. In this paper, with strict time constraints at query time, we first determine a good template and then optimize its thresholds. Among the novel concepts are the completeness tree that helps estimating query plan completeness at query time, a template selection algorithm that significantly reduces the amount of templates to be evaluated, and a threshold optimization algorithm that uses training queries as similarity profiles for setting thresholds. We empirically compare our novel approach to the previous algorithm in Section 6.

An area related to similarity search is **duplicate detection** [8, 9, 10]. 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. [11] 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 indexed base similarity values, while our approach allows to use *any* combination technique as the composite 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 [12]. 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 [14]. 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 [15]. 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. 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.

## 3. Problem Setting

In this section, we define basic notations for composed similarity measures and the problem of optimizing a query plan for efficient search with those measures.

### 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.

Accordingly, we first split the problem of measuring the similarity of two records into smaller subproblems. We define a set of **base similarity measures** $sim_a(q, r)$, each responsible for calculating the similarity of a specific attribute $a$ of the compared records $q$ and $r$ from a universe $U$ of possible records. In our person data use case, we have the functions $sim_{\mathsf{FirstName}}$, $sim_{\mathsf{LastName}}$, $sim_{\mathsf{BirthDate}}$, $sim_{\mathsf{City}}$, and $sim_{\mathsf{Zip}}$. All base similarity measures can be chosen independently. For example, we could use Jaro-Winkler distance for $sim_{\mathsf{FirstName}}$ [10], the relative distance between dates for $sim_{\mathsf{BirthDate}}$, and numeric difference for $sim_{\mathsf{Zip}}$. 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:

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

A **composed similarity measure** $sim_{Overall}$ uses the base similarity measures to derive an overall similarity of the two compared records. With a fixed similarity threshold $\theta_{Overall}$, a

query $q$, and a record set $R \subseteq U$, our goal is to find all records $r \in R$ with $sim_{Overall}(q, r) \geq \theta_{Overall}$.

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 [16, 17]. We make no assumptions whatsoever on the composed similarity measure other that it is composed of base similarity measures.

### 3.2. Query Plans

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 measure $sim_a$ is at least $\theta_a$. In the following, we abbreviate this predicate to $a \geq \theta_a$ 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 unspecified thresholds) with the logical operators conjunction and disjunction. Query plan templates are in disjunctive normal form (DNF), since this form is popular [14, 15] 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:

$$\bigvee \bigwedge a_i \geq \theta_{a_i} \tag{2}$$

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

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 FirstName and Zip or LastName and BirthDate is the following:

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

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 to select a good template before optimizing the thresholds.

A query plan $p$ **covers** a record $r \in R$ for a query $q \in U$ iff there is at least one conjunction $c$ in $p$ where $r$ is covered by all predicates in $c$.

To evaluate query plans, we define different performance metrics. The **completeness** $comp(p, q)$ of a query plan $p$ for a query $q$ 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 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. The **cost** $cost(p, q)$ 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 query-specific threshold $C_q$; only query plans that fulfill this property are **valid** plans. The threshold $C_q$ is specified by the user in advance.

Given a set $R$ of records, a query $q$, and a query-specific cost limit $C_q$, the **cost-aware query planning problem** is to determine the query plan $p$ that maximizes
$comp(p, q)$ subject to $cost(p, q) \leq C_q$.

### 3.3. Preparation

Our approach uses the following data for estimating cost and completeness of query plans. As training data for estimating completeness, we use a set $T$ of correct query/result record pairs $(q, r)$ where $q$ and $r$ are similar according to the composed similarity measure. We assume that the training data appropriately covers the diversity of queries.

For cost estimations, we use available statistics of value frequencies in the created indexes. For each attribute, we create an inverted index $i\text{-}ind_a : V_a \rightarrow \mathcal{P}(R)$ that determines the set of records in $R$ that contain an attribute value $v \in V_a$. In addition, for each attribute, we create a similarity index $s\text{-}ind_a : V_a \times [0, 1] \rightarrow \mathcal{P}(V_a)$ that determines for a value $v \in V_a$ and a threshold $\theta_a$ the set of similar values $v_s \in V_a$ with $sim_a(v, v_s) \geq \theta_a$.

## 4. Template Selection

In the following, we describe how to determine a good query plan template for a given query. Our algorithm interprets a template as a very strict query plan: A plan where all thresholds are set to 1, i. e., only exact search is performed (using inverted indexes only). This is necessary for predicting the cost and completeness of the template. In Section 5, we describe how to determine appropriate thresholds for the template selected in this section.

Why does it make sense to determine the template first? First and foremost, the template, i. e., the structure of the query plan, 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 6.2 confirm that in our 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(a)). 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 already determines a very good solution. In particular, when most queries contain only few errors

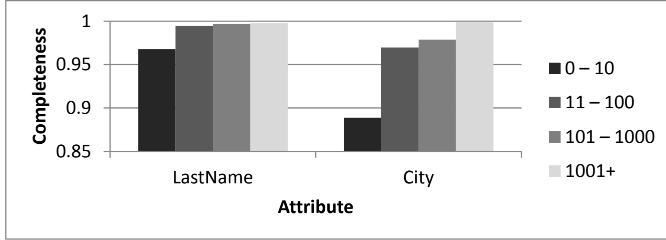



Figure 2: Completeness of templates consisting of only one attribute each, grouped by frequency of the respective attribute

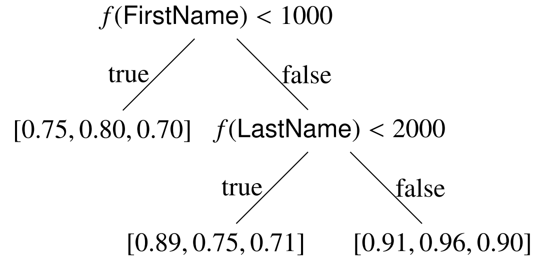



Figure 3: Completeness tree example for two attributes FirstName, LastName. Leaves show completeness arrays for the templates FirstName, LastName, and FirstName ∧ LastName.

(i. e., there are several attributes that exactly match the correct record), then there is no need for similarity indexes at all. In these cases, our template optimization approach described in this section is sufficient; only inverted (exact) indexes on attributes need to be created – similarity indexes on the attributes are unnecessary.

First, we describe how to evaluate a conjunction of attributes as part of a query plan template. The described completeness and cost estimations for conjunctions are then combined to estimations for the complete plan in DNF. Based on these estimations, we finally describe our algorithm for template selection.

### *4.1. Completeness Estimation*

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 [18, 19], because the corresponding paths in the brain have been activated much more often (and vice versa for low-frequency words) [20].

If an attribute value in the query occurs frequently in the database, then we can be quite sure that it is spelled correctly and that it 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 2 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 (see Section 6 for details). 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 [18, 19]). 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.

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 $qt$, predict $comp(qt, q)$.

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 learn the best partitioning using a tree learning algorithm. 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 3. 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 $f(\mathsf{FirstName}) < 1000$ are covered by the conjunction FirstName; similarly, 80 % are covered by LastName, and 70 % are covered by FirstName ∧ LastName.

The algorithm for creating the completeness tree is shown as Algorithm 1. 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 8. 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

**Algorithm 1** 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 ← null
4:  E_min ← estimation error (SSE) on T
5:  for each attribute a do
6:      for each distinct frequency f of any value of a do
7:          // evaluate split point ⟨a, f⟩
8:          E_{a,f} ← Σ_{c∈conj} m(T,c)(1 − pr)^2 + (|T| − m(T,c))pr^2
            where pr ← m(T,c)/|T|
9:          if E_{a,f} < E_min then
10:             sp ← ⟨a, f⟩
11:             E_min ← E_{a,f}
12: if sp = null then
13:     return leaf node with data T
14: else
15:     lC ← createNode({d ∈ T | f(d, sp.a) < sp.f})
16:     rC ← createNode({d ∈ T | f(d, sp.a) ≥ sp.f})
17:     return split node ⟨lC, rC⟩
```

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. Our optimization criterion is the sum of SSE of *all* conjunctions – 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 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.

### *4.2. Cost Estimation*

The cost of a query plan template is defined as the number of covered records in the record set. For each attribute, we know the number of records that are covered by the respective attribute value in the query: We use the inverted indexes on the attributes to determine these numbers.

Having these attribute cost is sufficient to estimate the cardinality of a complete query plan. In a nutshell, we estimate the cost of a query plan $p$ for a query $q \in U$ by estimating the probability that a randomly selected record $r \in R$ is covered by $p$. We multiply the probability with the cardinality of the complete record set to determine expected cost:

$$cost(p, q) = |R| \times P(covers_p(q, r) \mid r \in R) \quad (3)$$

By calculating cost using probabilities, we are able to easily handle conjunctions and disjunctions in the query plans with probability theory. As we empirically show in Section 6.2, our cost model can accurately predict cost of a query plan with a small variance remaining.

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 $P(covers_{a \geq \theta_a}(q, r) \mid r \in R)$ 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 ∧ City. For two attributes $a$ and $b$, we want to estimate the probability $P(a \wedge b)$. To resolve the joint probability, we distinguish between attributes that are statistically dependent or independent. We first determine the default case for attributes (dependent or independent) and then a set of exceptions from the default case (e. g., independent value combinations for dependent attributes). 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 largely independent from each other. If dependent attributes are not known in advance, these can be determined with statistical independence tests, such as the $\chi^2$-test.

Next, we determine the exceptions from the default case. 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. 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 default dependent attributes).

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:

$$P(a \wedge b) = \begin{cases} P(a)P(b), & \text{if } a \text{ and } b \text{ are independent,} \\ min(P(a), P(b)), & \text{else} \end{cases} \quad (4)$$

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 inverted indexes (defined in Section 3.3), we can determine the individual predicate probability for a value $v$ of an attribute $a$ as:

$$P(a \geq \theta_a) = \frac{|i\text{-}ind_a(v)|}{|R|} \tag{5}$$

### 4.3. Evaluating Templates in DNF

We have described how to estimate completeness and cost for conjunctions of attribute predicates. As query plan templates use predicates in disjunctive normal form, we now describe how to combine conjunction estimations into overall estimations.

For both completeness and cost estimations, our estimations for conjunctions can be regarded as probabilities. For cost, we modeled this explicitly. For completeness, the fraction of covered correct query/result record pairs can be interpreted as the probability that a plan covers a result record. The following combination technique thus applies to both concepts.

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 to probabilities. For example, we want to estimate $P(a \vee b)$ and have:

$$P(a \vee b) \quad = \quad P(a) + P(b) - P(a \wedge b) \tag{6}$$

The conjunctions can be estimated as described in Sections 4.1 and 4.2. 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:

$$P\left(\bigvee_{i=1}^{n} c_i\right) = \sum_{k=1}^{n} (-1)^{k-1} \sum_{\substack{T \subset \{1,\ldots,n\},\\ |T|=k}} P\left(\bigwedge_{t \in T} c_t\right) \tag{7}$$

### 4.4. Optimization 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 the set of conjunctions that have cost below $C_q$ (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 4 shows a representation of a conjunction graph example in the form of a Hasse diagram.

To efficiently determine the set of valid conjunctions, we apply a **backtracking and pruning** approach on the graph. We traverse the conjunction graph from the largest conjunction to the empty conjunction (in the figure: from bottom to top) with

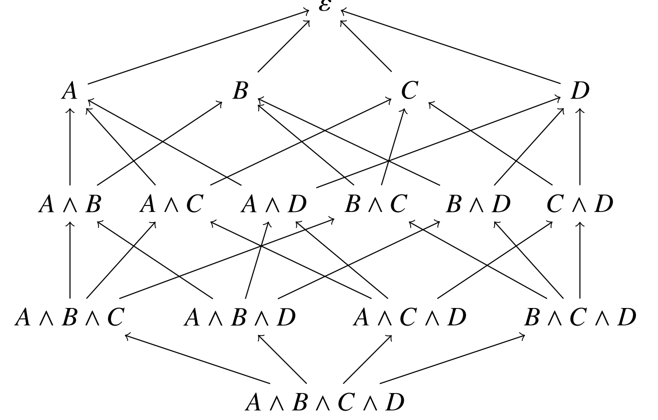



Figure 4: Conjunction graph for four attributes $A, B, C, D$

depth-first search. By removing an attribute from a plan, the plan becomes less restrictive, thus the completeness of the plan increases, but so do 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 4 turns out to be invalid, then $A$, $B$, and $\varepsilon$ 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 2 a recursive implementation of the function $evaluateTemplate(q, C_q, d, c)$, which evaluates a disjunction $d$ 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 se-

**Algorithm 2** function $evaluateTemplate(q, C_q, d, c)$

**Input:** query $q$, cost threshold $C_q$,
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** $cost(d \lor c, q) > C_q$ **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: $evaluateTemplate(d, c \land \neg a)$

lected 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 2 with $d$ as disjunction and, again, the conjunction containing all attributes as $c$. For each evaluated conjunction $c$, we now evaluate whether the disjunction $d \lor c$ 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 $d = d' \lor c^*$ with $c \subset c^*$ we have $d \lor c = d' \lor c$, 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 now 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 6.2, we empirically compare the exact and greedy versions of the algorithm.

## 5. Threshold Optimization

With the algorithm of the previous section, we can determine a good query plan template. 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.

### *5.1. Cost Estimation*

To estimate the cost of a query plan for a given query, we construct a similarity histogram for each attribute. We do this only 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.

A **similarity histogram** for a value $v$ of an attribute $a$ in a query $q$ is a function $hist_{a,v} : \Theta_a \rightarrow \mathbb{N}$ that returns for each possible similarity threshold $\theta_a \in \Theta_a$ the number of records $r \in R$ with $sim_a(q, r) \geq \theta_a$, i. e., the number of records that have a sufficiently similar value for $a$.

To construct the histogram, we use the similarity index that we created for each attribute. For a given value $v$ of an attribute $a$ in the query and for each threshold $\theta_a \in \Phi_a$, we determine the number of records in the database with a similar value:

$$hist_{a,v}(\theta_a) = |s\text{-}ind_a(v, \theta_a)|$$

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. The calculated 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. Future work will cover when the similarity index should be updated depending on the distribution of unseen query values.

The individual attribute cost can be combined into conjunction and disjunction estimations as described in Sections 4.2 and 4.3.

### *5.2. Completeness Estimation*

With the help of the completeness tree described in Section 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 $v$ different possible similarity thresholds per attribute, there are $v^n$ 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 3400 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 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 4.3.

### 5.3. Optimization Algorithms

We are now able to estimate cost and completeness of any query plan for a given query. We define two alternative algorithms for optimizing 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 as precalculating cost estimations for all threshold combinations is infeasible. The following two different approximate algorithms both solve the problem, but have certain advantages and disadvantages depending on search parameters.

#### 5.3.1. Similarity Profile Algorithm

Our first approach is based on similarity profiles and shown as Algorithm 3. 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.

**Algorithm 3** 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: $\quad p := setThresholds(t, s)$
6: $\quad$ **if** $cost(p, q) \leq C_q \wedge comp(p, q) > maxC$ **then**
7: $\quad\quad maxC := comp(p, q)$
8: $\quad\quad p_{res} := p$
9: **return** $p_{res}$

**Algorithm 4** Iterative Algorithm

**Input:** query $q$, cost threshold $C_q$, 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: $\quad R := \bigcup_{p \in W} N(p)$
6: $\quad W := \emptyset$
7: $\quad$ **for each** $p \in R$ **do**
8: $\quad\quad$ **if** $cost(p, q) \leq C_q$ **then**
9: $\quad\quad\quad W := W \cup \{p\}$
10: $\quad\quad\quad$ **if** $comp(p, q) > maxC$ **then**
11: $\quad\quad\quad\quad maxC := comp(p, q)$
12: $\quad\quad\quad\quad p_{res} := p$
13: $\quad$ **if** $W = \emptyset$ **then**
14: $\quad\quad$ **break**
15: $\quad W$ := set of $w$ plans from $W$ with highest completeness
16: **return** $p_{res}$

#### 5.3.2. Iterative Algorithm

Our second approach for threshold optimization iteratively lowers thresholds to find the best threshold combination (Algorithm 4). In our previous work, we refer to the algorithm as the top neighborhood algorithm (TNA) [1].

The general idea of TNA is to start with the plan with highest thresholds (in our case, all thresholds are set to 1), then follow promising plans in its neighborhood (the top plans), until we cannot reach any unseen valid plan. The result is then the valid plan with highest completeness found so far. In the following, we describe the algorithm in more detail.

The initial 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 $\theta_a$ and $\theta_b$ has two

elements:

$$N(a \geq \theta_a \wedge b \geq \theta_b) = \{a \geq \theta_a - 0.01 \wedge b \geq \theta_b, \\ a \geq \theta_a \wedge b \geq \theta_b - 0.01\}$$

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 $w$ plans with highest completeness (and lowest cost, if there are several plans with equal completeness; if there are more than $w$ eligible plans, a random selection is made). We call this set the **top** $w$ **neighborhood** $T_w(W)$. Only plans with cost below the cost limit $C_q$ are contained in this set.

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

*5.3.3. 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 $\Theta^n$, where $\Theta$ denotes the number of different thresholds per attribute and $n$ the number of attributes. However, the number of actually occurring combinations is typically much smaller and depends on the number and diversity of the training instances.

The iterative algorithm lowers the thresholds in the plan step-by-step, until the cost limit $C_q$ is reached. In the worst case, all thresholds must be lowered to their lowest possible values. The complexity of the iterative 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 $w$ [1].

We empirically compare the algorithms in Section 6.2.

## 6. Evaluation

In this section, we discuss evaluation results on real-world data. We introduce the dataset and experimental settings in Section 6.1. In Section 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.

### *6.1. Dataset and Evaluation Settings*

*Dataset*

We evaluated our approach on real-world data from Schufa, the largest German credit agency. The Schufa database contains information about the credit history of about 66m 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 name, date and place of birth, and address data (street, city, zip). The query data set consists of a 2-week query log with 2 million queries to this database. For each query, we know the exact search parameters (most record fields are mandatory), and the result obtained by Schufa's current system. Each 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 cases) that we can use for evaluating our approach. We randomly selected 1,000 of these very difficult queries for evaluating our system.

*Evaluation Settings*

Query templates or plans can be statically created at compile time [1]. 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.

Recall that the default threshold assignment for a template sets all thresholds to 1.0. We will use this default assignment in the following to compare the results for templates and derived plans.

Our approach first selects an appropriate template for the query (Section 4); we have two approaches for selecting templates (Greedy and Exact). We then optimize the plan's thresholds (Section 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 Iter**: Select the best template with the greedy algorithm, then optimize the plan's thresholds with the iterative 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 Iter**: Select the best template with the exact algorithm, then optimize the plan's thresholds with the iterative top neighborhood algorithm.

Christen et al. propose using a fixed accumulation function for calculating the similarity of a query record to the result records and ranking the results [11]. 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 6.2 as one of the naive 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.

### *6.2. Results*

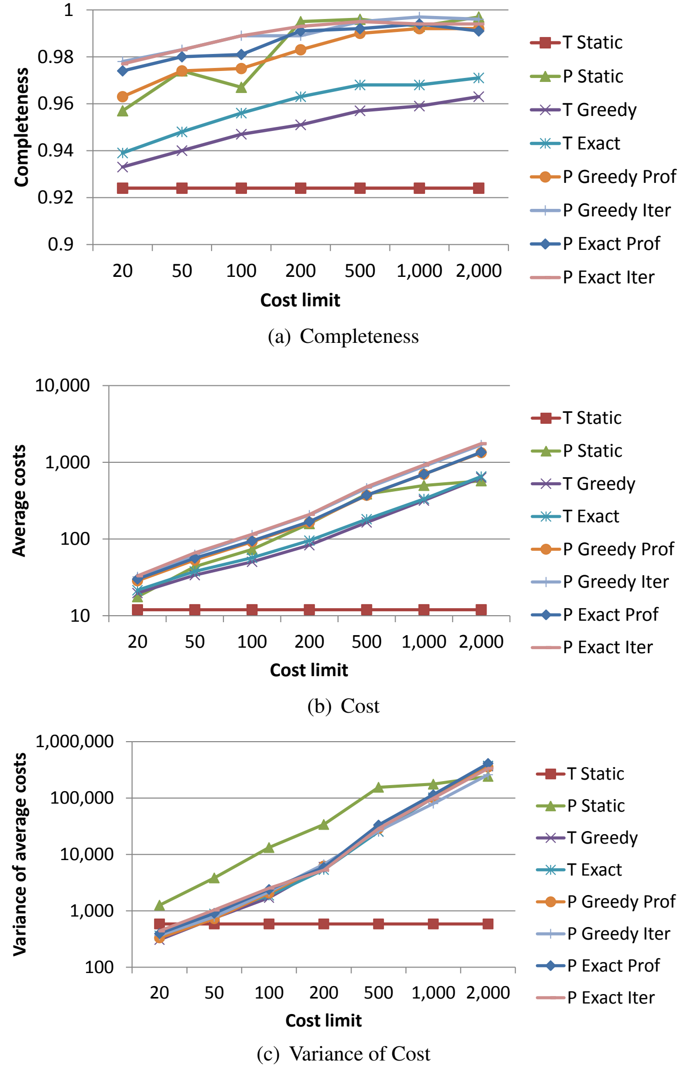
*Naive Plans*

First, we evaluated two naive 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 (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. 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, i. e., we scan and compare almost 1 % of the entire 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 query planning algorithms discussed in the following.

*Comparison of Planning Algorithms*

We have evaluated the planning algorithms presented in this paper 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).

In Figure 5, we show average completeness, cost, and variance of cost for all planning algorithms. Regarding the completeness (Figure 5(a)), all algorithms achieve a relatively high completeness compared to the naive plan $p_d$ (not shown). 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



(a) Completeness

(b) Cost

(c) Variance of Cost

Figure 5: Results per cost limit

(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 iterative threshold selection algorithm (P Greedy Iter and P Exact Iter) outperforms the profile-based algorithm (P Greedy Prof and P Exact Prof) for lower cost limits.

As can be seen in Figure 5(b), 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 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

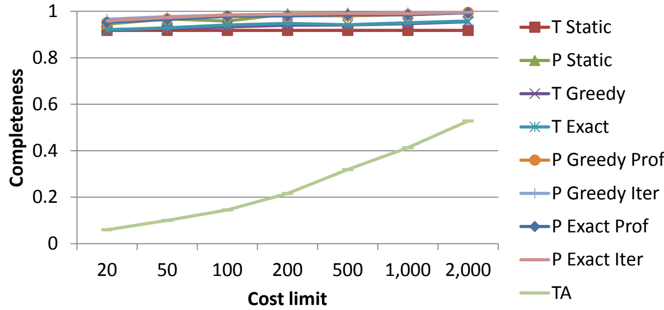



Figure 6: Comparison with Threshold Algorithm (TA)

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.

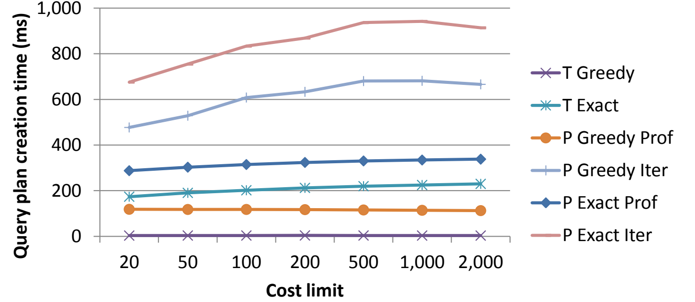
We show the variance of the average cost in Figure 5(c). The static plan results in significantly higher cost variance than the query-specific plans (note the logarithmic scale in the figure). The reason is that the static plan has very high cost for frequent query values and very low cost for rare query 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(b)). 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 9).

*Comparison with Related Work*

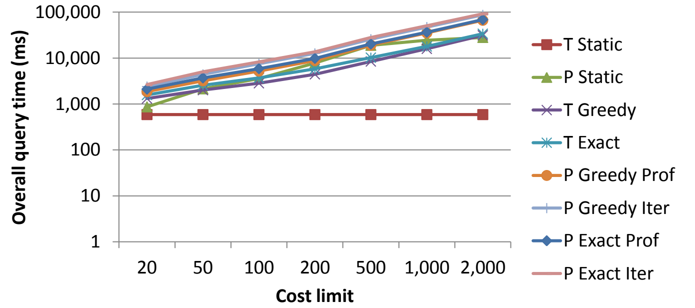
Prominent previous top-k retrieval algorithms are Fagin's Algorithm and its successor, the Threshold Algorithm (TA) [6]. Fagin et al. work with a set of sorted lists to retrieve records with values similar to the query values (our similarity index approach offers similar sorted access). TA retrieves 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 6.

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




(b) Overall query time (creation and execution)

Figure 7: Average time for query plan creation and execution

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).

*Query Time*

We show query plan creation and execution time in Figure 7. As can be seen in Figure 7(a), 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 iterative threshold optimization algorithm, an increase can be measured. The reason is that the iterative algorithm explores significantly more threshold combinations as the cost limit is increased.

In Figure 7(b), 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(b)). 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 se-

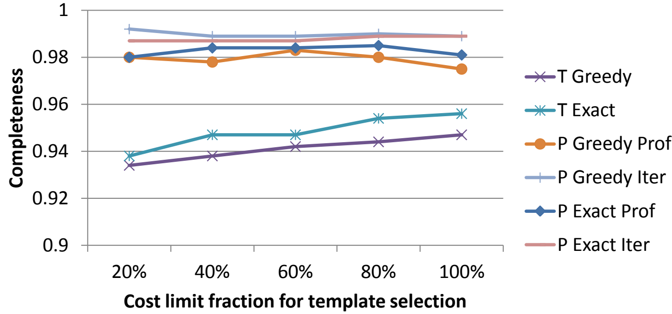



Figure 8: Completeness for different fractions of cost limits for template selection

lected plan with higher cost (cf. Figure 5(a)), we conclude that the small amount of time required for selecting a query plan is well-invested.

*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 8. 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(c), 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 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 9. 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). Cost estimation consists of looking up the attribute cost in 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

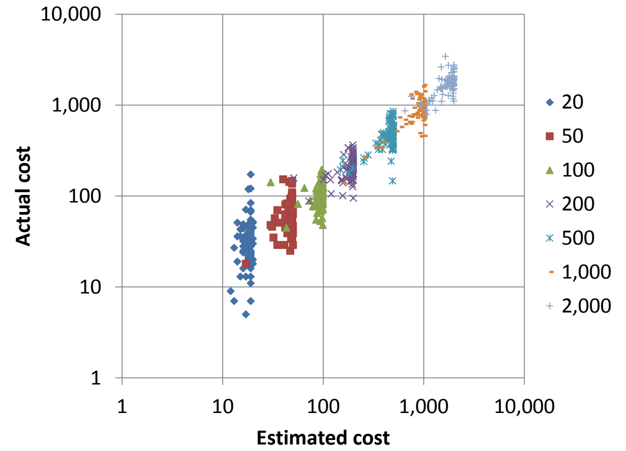



Figure 9: Estimated and actual cost for different query plans selected with different cost limits

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.

## 7. Conclusion and Future Work

We presented an approach to query planning for similarity search with arbitrarily composed similarity measures. In contrast to previous work and moving closer to the notion of database query planning, we create a new query plan for each query. We exploit training data to learn a completeness tree for estimating completeness of query plans, and we use index structures for estimating cost. Evaluation on real-world data shows that our approach significantly reduces variance in cost and increases average completeness.

As future work, we plan to also optimize the physical query plan. Especially in distributed environments, where similarity and inverted 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 also need to optimize the execution order to keep intermediate results and the resulting network load as small as possible.

**Acknowledgment:** We thank Schufa Holding AG for supporting this work.

## References


[1] D. Lange, F. Naumann, Efficient similarity search: Arbitrary similarity measures, arbitrary composition, in: Proc. of the Intl. Conf. on Information and Knowledge Management (CIKM), Glasgow, Scotland, UK, 2011, pp. 1679–1688.

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

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

[4] E. Chávez, G. Navarro, R. Baeza-Yates, J. L. Marroquín, Searching in metric spaces, ACM Comput. Surv. 33 (3) (2001) 273–321.
[5] R. Fagin, Fuzzy queries in multimedia database systems, in: Proc. of the Symposium on Principles of Database Systems (PODS), Seattle, WA, USA, 1998, pp. 1–10.
[6] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, in: Proc. of the Symposium on Principles of Database Systems, PODS '01, ACM, New York, NY, USA, 2001, pp. 102–113.
[7] P. M. Deshpande, D. P, K. Kummamuru, Efficient online top-k retrieval with arbitrary similarity measures, in: Proc. of the Intl. Conf. on Extending Database Technology (EDBT), Nantes, France, 2008, pp. 356–367.
[8] A. K. Elmagarmid, P. G. Ipeirotis, V. S. Verykios, Duplicate record detection: A survey, IEEE Transactions on Knowledge and Data Engineering (TKDE) 19 (1) (2007) 1–16.
[9] F. Naumann, M. Herschel, An Introduction to Duplicate Detection, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2010.
[10] W. E. Winkler, The state of record linkage and current research problems, Tech. rep., Statistical Research Division, U.S. Census Bureau (1999).
[11] P. Christen, R. Gayler, D. Hawking, Similarity-aware indexing for real-time entity resolution, in: Proc. of the Intl. Conf. on Information and Knowledge Management (CIKM), Hong Kong, China, 2009, pp. 1565–1568.
[12] H. Newcombe, Record linkage: the design of efficient systems for linking records into individual and family histories, American Journal of Human Genetics 19 (1967) 3.
[13] M. Michelson, C. A. Knoblock, Learning blocking schemes for record linkage, in: Proc. of the National Conf. on Artificial Intelligence (AAAI), Boston, MA, USA, 2006, pp. 440–445.
[14] M. Bilenko, B. Kamath, R. J. Mooney, Adaptive blocking: Learning to scale up record linkage, in: Proc. of the Intl. Conf. on Data Mining (ICDM), Hong Kong, China, 2006, pp. 87–96.
[15] S. Chaudhuri, B.-C. Chen, V. Ganti, R. Kaushik, Example-driven design of efficient record matching queries, in: Proc. of the Intl. Conf. on Very Large Databases (VLDB), Vienna, Austria, 2007, pp. 327–338.
[16] M. Bilenko, R. J. Mooney, Adaptive duplicate detection using learnable string similarity measures, in: Proc. of the Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD), 2003, pp. 39–48.
[17] S. Sarawagi, A. Bhamidipaty, Interactive deduplication using active learning, in: Proc. of the Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD), Edmonton, Alberta, Canada, 2002, pp. 269–278.
[18] D. G. MacKay, L. Abrams, Age-linked declines in retrieving orthographic knowledge: Empirical, practical, and theoretical implications, Psychology and Aging 13 (1998) 647–662.
[19] J. P. Stemberger, B. MacWhinney, Frequency and the lexical storage of regularly inflected forms, Memory and Cognition 14 (1986) 17–26.
[20] D. M. Burke, D. G. MacKay, J. S. Worthley, E. Wade, On the tip of the tongue: What causes word finding failures in young and older adults?, Journal of Memory and Language 30 (5) (1991) 542 – 579.