Computing Similar Entity Rankings via Reverse Engineering of Top-k Database Queries

Kiril Panev
University of Kaiserslautern
Kaiserslautern, Germany
panev@cs.uni-kl.de

Evica Milchevski
University of Kaiserslautern
Kaiserslautern, Germany
milchevski@cs.uni-kl.de

Sebastian Michel
University of Kaiserslautern
Kaiserslautern, Germany
smichel@cs.uni-kl.de

***Abstract*—Ranked lists are an essential methodology to succinctly summarize outstanding items, computed over database tables or crowdsourced in dedicated websites. Previous work has already addressed the problem of reverse engineering top-k queries over a database. However, existing systems fail to return any answer to the user when a precisely matching query has not been found. In this work, we tackle this problem of determining queries that compute lists similar to a user-specified input ranking. More precisely, for a ranked list of entities L and a similarity threshold θ, we want to find queries that return lists L_r with $d(L, L_r) \leq \theta$, where $d(L, L_r)$ is the distance between the lists. Through a detailed experimental study we show that our system is able to achieve, in most of the cases, a Recall@10 higher than 80%.**

I. INTRODUCTION

Data stored in relational database systems and specifically data warehouses is often represented in complex and difficult to comprehend schemata. Various research areas, like keyword search in databases [1], [9], [7] and reverse engineering database queries [19], [14], [21], have picked up this pressing demand with the ultimate goal to providing user friendly ways for querying and exploring data. In this work, we develop ways to efficiently explore database contents by means of ranked lists. Specifically, a user submits a ranked list of entities and the system aims at returning a set of queries that do return a list similar to the input when executed on the given database instance. As application scenarios, consider, for instance, business analysts who are interested in finding the factors that resulted in the ranking of their interest, data scientists who try to find explanatory SQL queries for crowd-sourced top-k rankings where the scores are not known, or simply finding the data-generating query and the numeric score of a (possibly stale) input, where the original query is not known, for one or another reason. We emphasize on finding not only the generating query, but a set of queries that produce all rankings that are similar to the input ranking within a user-defined threshold. This is not only useful when being interested in alternative queries or similar rankings, but specifically also when there is no query that generated a ranking identical to the input, for instance in cases when the original version of the database is not present anymore. We present to the user the discovered top-k queries together with the corresponding result lists, revealing the scores of the entities, the ranking criteria resulting in the ranked tuples, and the constrains imposed on the tuples.

This work has been supported by the German Research Foundation (DFG) under grant MI 1794/1-1.

Name	Founded	State	Store	Date	#Sales	Sales in USD	Returns in USD
Cargill	1865	MN	CA101K	12.10.2015	654	523,087	30
Kroger	1883	MN	AL121P	10.10.2015	1,753	101,022	900
...	...	...	...	...	...	...	...
Target	1902	NY	FL324K	14.10.2015	2,106	150,011	240
Cargill	1865	CA	FL223K	10.10.2015	334	256,102	560
...	...	...	...	...	...	...	...
Valspar	1806	MN	MN291P	10.10.2015	596	231,110	102
...	...	...	...	...	...	...	...
Best Buy	1966	AZ	CA132F	09.10.2015	429	42,101	192
Best Buy	1966	MN	IN231D	10.10.2015	586	808,109	75
...	...	...	...	...	...	...	...
Target	1902	MN	MN11L	12.10.2015	784	221,003	107

TABLE I: Sample relation of companies sales data

Consider a relation *Sales*, illustrated in Table I, containing values of daily sales of US companies per store. The relation contains textual attributes like name of the company, the state where the stores are located, and an id code of the store. In addition, there are numerical attributes that measure the number of sales, their value in USD, and the value of the returned goods.

Now, consider the top-k list (upper left) shown in Figure 1. Not only that we do not know the scores of each entity and the ranking criteria, but also an attempt to generate a query returning an exact match would be unsuccessful, as using any of the numerical attributes in Table 1 would not generate the exact input list. Very likely, this top-k list was generated using an older version of the data, and, thus, in order to find the appropriate query, we need to relax the query discovery conditions, returning queries that would generate a list similar to the given one. For instance, the following query would generate the list shown in the upper right in Figure 1:

```
SELECT name, MAX(sales_in_USD)
FROM sales
WHERE state = 'MN'
GROUP BY name
ORDER BY MAX(sales_in_USD)  DESC  LIMIT 5
```

This could indeed be the query that when executed on the *right version* of the data would generate a result matching the input list. The resulting list could be shown to the user as depicted in Figure 1. Likely, other queries would also meet the user needs, as shown in Figure 1—we aim at finding all of them.

In prior work [12], we presented a system, coined PALEO, that is able to find for a ranked list of entities *and* their scores, a

Input list L
Best Buy
Cargill
Target
Valspar
Kroger

Companies' stores in MN ranked by max sales

Name	Maximum Sales
Best Buy	808,109
Cargill	523,087
Valspar	231,110
Target	221,003
Kroger	101,022

Companies' stores in MN ranked by min returns

Name	Minimum Returns
Cargill	30
Best Buy	75
Valspar	102
Target	107
Kroger	900

Fig. 1: Example input list L and possible result lists returned when the candidate query is executed over the DB

```
SELECT L.e, agg(value)
FROM table
WHERE P1 and P2 and ...
GROUP BY L.e
ORDER BY agg(value) LIMIT k
```

$L.e$
a
f
k
m
n

Fig. 2: Query template (left) and example input L (right)

matching query that, when executed over the database, would return a list **equal** to the input. In this work we focus on the case when only the ranked entities are known (no scores are given) and aim at returning a set of queries that, each, approximately generate the input ranking.

This task introduces several new challenges that need to be addressed. First, we need to find predicates and ranking criteria, when the score (ranking value) of the entities is not known. Thus, the set of candidate predicates that (partially) satisfy the input list is expected to significantly grow, requiring novel effective techniques for finding and pruning unpromising predicates. Second, finding similar rankings requires the application of distance measures like Kendal's tau and Footrule distance [8]. Instead of generating all possible queries, evaluating them, and checking if they comply to the similarity threshold, we show how to embed distance-measure specific bounds into the query generation process.

A. Problem Statement

Given a database D with a single relation R with schema $\mathcal{R} = \{A_1, A_2, \ldots\}$ and input relation L that represents a ranked list of items. The task we consider in this paper is to efficiently and effectively determine queries Q_i that, when executed over the database, compute result lists that are **similar** to L. A user-defined **similarity threshold** θ is used to control how similar the found queries, respectively their results, should be, relative to L.

We focus on top-k select-project queries over relation R of the form shown in Figure 2(left). We specifically focus on working with a single relation. Reverse engineering queries involving joins has already been addressed in related work (e.g., Zhang et al. [21]) and is beyond the scope of this paper. Furthermore, we focus on the case when a query is considered valid when its result **approximately** resembles the input and therefore we need to present to the user a ranked list of most similar results.

In this work we assume that the input top-k list L has only one column; $L.e$, the entity column. The numeric score of the entities is not known. We refer to the specific attribute in R that contains the entities as A_e and assume it is known a priori. Finding this attribute, if not beforehand given, is a straightforward task using inverted indexes.

We focus on predicates P of the form $P_1 \wedge P_2 \cdots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., state = 'MN'). Furthermore, we denote with *size* of a predicate $|P|$ the number of atomic predicates P_i in the conjunctive clause.

To compare the top-k lists we use Spearman's Footrule [8], a prominent distance measure used for comparing top-k list. Spearman's Footrule is the L1 distance between two top-k lists, i.e., for two top-k lists l_1 and l_2, Spearman's Footrule $F(l_1, l_2) = \sum_i |l_1(i) - l_2(i)|$ where $l_1(i)$ denotes the position of entity i in l_1. For entities found in only one of the lists we take an artificial position of $k+1$ as specified in [8]. For instance, considering two ordered lists $l_1 = \{a, f, k, m, n\}$ and $l_2 = \{a, k, n, b, f\}$, when computing the distance, for entity m we would take the values $l_1(m) = 4$, $l_2(m) = 6$. The Footrule distance between the two lists is $F(l_1, l_2) = 10$. The system only outputs ranked lists of same size k, thus the largest possible value of the Footrule distance is $k \times (k+1)$ and occurs if two disjoint lists are compared. The smallest distance is 0, when the lists are identical. In the rest of this paper we use normalized values for the Footrule distance and θ, ranging from 0 to 1. Additional details can be found in the work by Fagin et al. [8].

Table II shows a summary of the most important notations used throughout this paper.

B. Contributions and Outline

In our previous work [12] we already addressed the problem of reverse engineering top-k queries and designed a system called PALEO that is able to successfully identify all queries whose result matches the input list. In this work, we build on top of that work, and present PALEO-A, a system that is able to efficiently handle stale input lists where the numeric score value is not known a priori. We specifically focus on the problems that arise from introducing approximatively matching results.

With this paper we make the following contributions: We present algorithms for finding predicates and scoring function in R that would result in a list similar to the inputed one. Working with short lists and allowing for approximate matches factors in having a large set of candidate predicates. We present techniques for early elimination of many of the candidate predicates by using the properties of the distance measures. Furthermore, we devise an algorithm for efficiently finding the ranking criteria. The resulting candidate queries are executed

R	Base table in the database
A_i	Attribute in R
A_e	Entity attribute in R
L	Top-k input list
$L.e$	Entity column in L
e_i	Entities in A_e or $L.e$
v	Values in A_i
P	Predicate (atomic or conjunctive)
Q	Query
Q_L	Result set of Q when querying R
$F(l_1, l_2)$	Footrule distance between two lists l_1 and l_2
$l_1(i)$	position of entity i in list l_1
k	number of entities in the top-k list

TABLE II: Overview of Notations

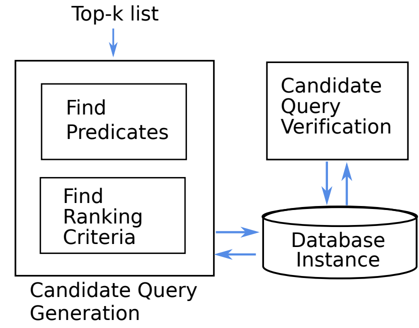

Fig. 3: System task steps

over the database and a sorted list of meaningful results is presented to the user. We report on the results of a detailed experimental evaluation using data and queries from the TPC-H [18] and SSB [16] benchmarks.

This paper is organized as follows. Section II discusses related work. Section III presents the framework and the idea of our approach, followed by the specific sub-problems of identifying query predicates and determining the ranking attributes and aggregation function. Section IV reports on the results of the experimental evaluation and presents lessons learned. Section V concludes the paper.

II. Related Work

Reverse engineering queries as a research topic has been gaining popularity recently [19], [14], [21]. Tran et al. [19] present data-driven approach called *Query by Output* (QBO). Given a database D and a query output $Q(D)$ produced by a query Q, they try to find an instance-equivalent query Q'. They focus on identifying the selection predicates in select-project-join queries and formulate this problem as a data classification task. Sarma et al. [14] explore the *View Definitions Problem* (VDP) which is a subproblem of QBO in that it considers only one relation R and there are no joins and projections. In prior work [12], we proposed a system that, given a relation R and a sample top-k result list, determines SQL queries whose result matches the provided input list when executed over R. The input list contains entities and their scores. The system mainly operates on a subset of the base relation, held in main memory, and in addition uses data samples, histograms, and sample descriptive statistics to identify potentially valid queries. Furthermore, we considered a probabilistic model that calculates the suitability of a query discovered over a subset of the base table R, which is applicable for scalability reasons and handling variations of R.

Shen et al. [15] study the problem of discovering a minimal project-join query that contains given example tuples in its output and do not consider selections. They only handle text columns with keyword search allowed on them and introduce a candidate generation-verification framework to discover all valid queries. Psallidas et al. [13] propose a candidate-enumeration and evaluation framework for discovering project-join queries. Their system handles only text columns and establish a query relevance score based evaluation of candidate queries.

In keyword search over databases [2], the input is a single tuple with specified keywords as fields. The works of [5], [17] interpret the query intent behind the keywords and compute aggregate SQL queries. Blunschi et al. [5] use patterns that interpret and exploit different kinds of metadata, while Tata et al. [17] discovers aggregate SQL expressions that describe the intended semantics of the keyword.

The principle of **reverse query processing** is studied in [3], [4], [6], [11], however their objectives and techniques are different. Binning et al. [3], [4] discuss the problem of generating a test database D such that given a query Q and a desired result R, $Q(D) = R$. Bruno et al. [6] and Mishra et al. [11] study the problem of generating test queries to meet certain cardinality constraints on their subexpressions.

A **reverse top-k query** [20] returns for a point q and a positive integer k, the set of linear preference functions (in terms of weighting vectors) for which q is contained in their top-k result. For example, finding all customers who treat the given query product q as one of their top-k favorite elements. In such cases, each customer is described as a vector of weights. Although it appears related given the name, this research area is not directly related to our work.

III. Approach

A naïve solution to our problem—enumerating all queries that contain at least some of the entities, executing them and comparing the similarity between the lists—is clearly infeasible as the number of queries that would need to be executed is immense. Our system, PALEO-A, is able to prune many candidate predicates early on, leaving only a small number of candidate queries that need to be executed over the database.

Our approach could be split into three main steps, illustrated in Figure 3:

Step 1: finding the predicates P in the where clause of Q.

Step 2: finding the ranking criteria.

Step 3: validating and ranking the queries.

All computations in the above mentioned steps are done on a table retrieved from the relation R by selecting all tuples where the entity is one of the entities in the input list L. We refer to this table as R'. For the purpose of efficient access of its data, PALEO-A stores R' in-memory in a **column oriented** fashion, with columns being represented as arrays, allowing fast evaluation of aggregate queries over R'. The relation R'

$t.id$	Name	State	Date	$\cdots$	Sales USD	$\cdots$
1	Best Buy	MN	10.10.2015	$\cdots$	$818,109$	$\cdots$
2	Best Buy	MN	11.10.2015	$\cdots$	$51,369$	$\cdots$
3	Best Buy	AZ	09.10.2015	$\cdots$	$42,101$	$\cdots$
4	Best Buy	CA	10.10.2015	$\cdots$	$12,256$	$\cdots$
5	Best Buy	CA	15.10.2015	$\cdots$	$46,423$	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
10	Cargill	MN	12.10.2015	$\cdots$	$523,087$	$\cdots$
11	Cargill	CA	10.10.2015	$\cdots$	$256,102$	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
20	Kroger	MN	10.10.2015	$\cdots$	$101,022$	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
30	Target	MN	12.10.2015	$\cdots$	$221,003$	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

TABLE III: Example of a Relation R' for Input L in Figure 1

has $k' \geq k$ number of tuples, since it contains all tuples without (potentially) being filtered by predicates. In fact, it is reasonable to assume, without prior knowledge, that $k' \gg k$, as each distinct entity e_i can appear many times in R.

A. Candidate Predicates

The first step of our approach is by using the tuples in R' to create a set of *candidate predicates* that are subsequently augmented with ranking criteria to make up full-fledged candidate queries.

Definition 1: **Candidate Predicate**
We say a predicate P is a candidate predicate iff *for at least one entity* that appears in L there is a tuple t in R' that fullfils the predicate. Formally,

$$\exists e_i \in L.e \, \exists \, tuple \, t \in R' : \, P(t) = true \, \wedge t.e = e_i$$

Note that this definition underpins a big difference from this work to our prior work that would insist that each (not at least one) entity in $L.e$ has a tuple that fulfils the predicate.

Now, the table R' contains only tuples where the entity attribute $R'.e$ is one of the entities from $L.e$. Thus, Definition 1 implies that all distinct attribute:value pairs in R', like (State='MN') or (Date='10.10.2015'), are atomic candidate predicates of size $|P| = 1$. Candidate predicates of size larger then $|P| = 1$ can be generated simply by forming the conjunction of the atomic predicates belonging to the same tuple, e.g., (State='MN') $\wedge$ (Date='10.10.2015').

A naïve approach would simply take all these candidate predicates and advance to the next step of finding ranking criteria. This would, however, result in a drastically reduced performance, inducted by a radical expansion of the set of candidate predicates. We can do much better by investigating whether or not a candidate predicate is at all capable or likely of being capable of leading to a query that is satisfying the similarity threshold. More precisely, to reduce the number of candidate predicates, we rely on the fact that the user expects queries resulting in lists **similar** to the input, i.e., those queries for which $d(Q_L, L) \leq \theta$, for reasonably small values of θ. This means that a candidate predicate that is fulfilled by only one entity in $L.e$ is presumably not a good candidate since the resulting list, irrespective of the ranking criteria, would have only one overlapping entity with the input, and therefore will not satisfy the user's information needs.

What we need to know in order to achieve this is how many overlapping entities μ the rankings Q_L and L need to have such that $d(Q_L, L) \leq \theta$. This then allows us to eliminate candidate predicates whose tuples cover less than μ entities in $Q_L.e$. According to [10], two ranked list l_1 and l_2 with Footrule distance $F(l_1, l_2) \leq \theta$ must have an overlap:

$$\mu \geq \lceil 0.5 \times (1 + 2k - \sqrt{1 + 4\theta}) \rceil$$

Using the minimum number of overlapping entities μ, we can already eliminate many of the candidate predicates. However, as we show in the experiments in Section IV, still many useless predicates were generated. In order to eliminate even more candidate predicates, we further leverage the properties of the Footrule distance. We first define the concept of **displacement of an entity *e***, ω_e as:

Definition 2: **Displacement of an entity:** For two top-k lists l_1 and l_2, $l_1 \neq l_2$, we define the displacement, denoted with ω_e, of an entity $e \in \mathcal{D}_{l_1} \cap \mathcal{D}_{l_2}$, as the difference of the position of the entity in the two rankings, i.e., $\omega_e = |l_1(e) - l_2(e)|$. In the case when $e \in \mathcal{D}_{l_1} \setminus \mathcal{D}_{l_2}$, $\omega_e = k + 1 - l_1(e)$ or vice versa.

The Footrule distance of two lists l_1 and l_2 is in fact a sum over the displacements of the entities in $\mathcal{D}_{l_1} \cup \mathcal{D}_{l_2}$. Next, with the following theorem we define the maximum value for the displacement of any entity e in any two list l_1 and l_2, where $F(l_1, l_2) = \lambda$.

Theorem 1: The maximum displacement of an entity e for two top-k lists l_1 and l_2 of size k with Footrule distance $F(l_1, l_2) = \lambda$ is $max(\omega_e) = min\{\lfloor \frac{\lambda \times k \times (k+1)}{2} \rfloor, k\}$.

Proof: Restricting the maximum displacement $max(\omega_e)$ to k is clear. Considering the fact that the rankings (lists) are of size k (have k entities), by the definition of a displacement it follows that the $max(\omega_e)$ for two rankings l_1 and l_2 cannot be larger than k for any distance $F(l_1, l_2)$.
Now, let us look at the case of showing that the maximum displacement $max(\omega_i)$ for any two rankings l_1 and l_2 with distance $F(l_1, l_2) = \lambda$, cannot be larger than $\frac{\lambda \times k \times (k+1)}{2}$. Top-k rankings are bijections from the set $\mathcal{D}_{l_1}$ to itself, where $\mathcal{D}_{l_1} \neq \mathcal{D}_{l_2}$ for two top-k lists l_1 and l_2. The Footrule distance, in this case, is a sum not only over the misplacement of the overlapping entities, but over the displacement of the missing entities as well. Since we are working with lists of same size, we have the same number m of missing entities in l_2 and l_1. Therefore, if we ignore the fact that the missing entities are different in the two top-k lists, and just consider that we have m missing entities in each list and $k - m$ common entities, there is a bijection between the set of k entities and the lists l_1 and l_2. Now, let us assume that $max(\omega_i)$ is larger than $\frac{\lambda \times k \times (k+1)}{2}$. If $max(\omega_e) > \frac{\lambda \times k \times (k+1)}{2}$ then we have to have at least one more entity j displaced in order to keep the bijection between the two sets. Since $F(l_1, l_2) = \sum_{e \in \mathcal{D}} \omega_e$ it follows that we have at least one entity whose displacement is $\omega_j < \frac{\lambda \times k \times (k+1)}{2}$, or we have more than one displaced item, but the sum of their displacements is smaller than $\frac{\lambda \times k \times (k+1)}{2}$. However, this would break the bijection relation between the two permutations, which is not possible, proving that $max(\omega_e) \leq \frac{\lambda \times k \times (k+1)}{2}$.

Therefore, it holds that $max(\omega_e) = min\{\lfloor \frac{\lambda \times k \times (k+1)}{2} \rfloor, k\}$. ■

Knowing the maximum displacement allows us to find a set of entities $e \in L.e$ that must be covered by candidate

```
method: findPredicates
input: top-k list L
       relation R'
       overlap μ
       max displacement max(ω_e)
output: a set of candidate predicates 𝒫
1   𝒫 = ∅; n = 1; 𝒫_n = ∅
2   ℳ_e = Top-(k − max(ω_e))(L)
3   for each A_i in R'
4       find P_i := (A_i = v) with |P_i| = 1 s.t.
5       ∃e_i ∈ L.e ∃ tuple t ∈ R' : P_i(t) = true ∧ t.e = e_i
6       if count(e_i) ≥ μ and ℳ_e ⊆ {e_i}
7           add P_i to 𝒫_n
8           for each P_i keep ℐ_{P_i} = {t.id|P_i(t) = true}
9   repeat
10      n = n + 1
11      𝒫_n = ∅
12      for each P_i ∈ 𝒫_1 and P_j ∈ 𝒫_{n−1} and P_i ∩ P_j = ∅
13          create ℐ_{P_ij} = ℐ_{P_i} ∩ ℐ_{P_j}
14          if count(e_i) ∈ ℐ_{P_ij} ≥ μ
15          and ℐ_{P_ij} covers all e_i ∈ ℳ_e
16            add P_ij := P_i ∧ P_j to 𝒫_n
17  until 𝒫_n = ∅
18  return 𝒫 = ⋃_n 𝒫_n
```

Algorithm 1: Finding candidate predicates

predicates. For instance, when we have an input top-k list L of size 10, and we want to find all queries resulting in lists Q_L such that $F(Q_L, L) \leq 0.1$, all the candidate predicates must cover the first 5 entities since the maximum displacement $max(\omega_e)$ is 5 and thus the maximum position of an entity that appears in L but not in Q_L is $k + 1 - max(\omega_e) = 6$.

Using these two optimizations, we propose a modified approach of the one in [12] that generates a set of candidate predicates. This new approach is described in Algorithm 1.

The method requires as input the number of overlapping entities μ and the max displacement of an entity $max(\omega_e)$. These values depend only on the size of the input list k and the threshold θ, and can be calculated using the formulas presented above. The method starts by creating the set $\mathcal{M}_e$ which will contain the Top-$(k - max(\omega_e))$ entities in the input list L (Line 2 in Algorithm 1). These are the entities that must be covered by each candidate predicate P_i. In the first step, the method creates all atomic candidate predicates, i.e., those with size $|P| = 1$ (Lines 3–8 in Algorithm 1). For each column A_i we identify values v such that the predicate $P_i := (A_i = v)$ will satisfy two conditions (Line 6 in Algorithm 1). First, it must cover at least μ number of entities, which ensures that the tuples in R' that fulfil this P_i contain the minimum number of overlapping entities. Second, the predicate must cover all of the mandatory entities in $\mathcal{M}_e$. In addition, for each such predicate that satisfies these two conditions, we keep a set $\mathcal{I}_{P_i}$ that contains the tuple ids that this predicate selects, i.e., $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$. In each additional step, the algorithm creates predicates of size n by creating conjunctions of the atomic predicates from the set $\mathcal{P}_1$ created in the previous iteration (Lines 9–17). None of the predicates is created multiple times. Each of the conjunctive predicates P_{ij} whose tuple ids set $\mathcal{I}_{P_{ij}}$ covers at least μ entities and in addition covers all mandatory entities in $\mathcal{M}_e$, is added to the set $\mathcal{P}_n$ which contains the predicates with size n. This set will be used in the next iteration in creating candidate predicates with size $n + 1$.

Example: Let us consider Table III and the input list in Figure 1. Furthermore, let $\mu = 3$ and the max displacement $max(\omega_e) = 3$. The method starts by initializing the set with mandatory entities $\mathcal{M}_e$ with the top-2 entities from the input list, i.e., 'Best Buy', 'Cargill'. Then it starts iterating over the values in the columns, e.g., State, Date etc. Thus, it creates the set of atomic candidate predicates, $\mathcal{P}_1 = \{P_1$:=(State='MN'), P_2 :=(Date='10.10.2015')$\}$. Both of these predicates cover at least 3 entities and cover the mandatory entities in $\mathcal{M}_e$. In addition, the tuple ids sets that correspond to this predicates are kept, e.g., $\mathcal{I}_{P_1} = \{1, 2, 10, 20, 30\}$.
In each additional step, conjunctive clauses of size n are produced by using the predicates in $\mathcal{P}_1$ and $\mathcal{P}_{n-1}$. Thus, for $n = 2$ the predicate P_{12} :=(State='MN')$\wedge$(Date='10.10.2015') is tested whether it qualifies as a candidate by intersecting the corresponding tuple ids sets. The intersected tuple ids $\mathcal{I}_{P_1} \cap \mathcal{I}_{P_4} = \{1, 20\}$ do not cover at least 3 entities from the input list, thus the predicate P_{12} is not a candidate predicate. Since R' is kept in main memory, by using the tuple ids the method efficiently can access the full tuple to perform the checks needed.

B. Ranking Criteria

Once we have identified the set of candidate predicates, the next step is to identify the ranking criteria according to which the entities in the top-k lists are ordered. For this purpose, we need to find a suitable numeric attribute (or multiple ones) including an aggregation function—or decide if one is used at all.

Definition 3: **Candidate Ranking Criterion**
We say a ranking criterion, consisting of one or multiple numerical attributes and, if existing, an aggregation function is a candidate iff, **when executed on R'** together with a candidate predicate, it returns a resulting list R_L such that the resulting list is similar to the input list L. More formally, for a given similarity threshold θ, a ranking criterion is a candidate iff $F(L, R_L) \leq \theta$.

As we do not know the ranking values of the entities in the input list, practically all the combinations of numeric attributes and aggregate functions could qualify as candidates. Only after executing the query and comparing its result to the input list, we could classify the query as a candidate or dismiss it. Since executing the query is in fact the most expensive part of PALEO-A, we need to come up with techniques that would allow us an earlier elimination of the non-qualifying ranking criteria.

For the purpose of identifying the candidate ranking criteria we leverage the table R' which is held entirely in main memory. We use the tuple sets $\mathcal{I}_i$ identified in Section III-A and check which ranking criteria, i.e., numerical column(s) in combination with the supported aggregate functions, can produce a sorted list of entities similar to the input list L. This corresponds to executing queries on R', where each query is a combination of the candidate predicates created with Algorithm 1 and the ranking criteria supported by the system.

method: **findSum**

input: top-k list L
relation R'
tuple sets $\mathcal{I}_i$
threshold θ
output: a pair of numerical columns (A_i, A_j)

```
1  for each (A_i, A_j) ∈ R'
2      for each tuple set I_i
3          for each distinct entity e_i
4              A_ij_sum(I_i) = A_i(I_i) + A_j(I_i)
5              group by e_i
6              add A_ij_sum(I_i) to R_L
7      sort R_L
8      if F(L, R_L) ≤ θ
9          return the valid pair (A_i, A_j)
```

Algorithm 2: Finding $sum(A + B)$ ranking criterion

The sets of tuple ids are used as sort of an index to directly access the tuples in R'. The top-k lists that result from these queries are compared with the input list L by calculating the Footrule distance. Each of the queries whose resulting list has a Footrule distance at most θ, is created as a candidate query Q_c that needs to be executed on the base table R. All the other queries are discarded. In this way we manage to prune many of the candidate predicates that were created previously, which in turn results in less candidate queries that need to be executed in the final and most expensive step.

Our system currently supports the following ranking criteria: $avg(A)$, $max(A)$, $sum(A)$, $sum(A{+}B)$, and $sum(A{*}B)$. Identifying candidate queries with $sum(A{+}B)$ ranking criteria is shown in Algorithm 2. For each numerical column pair (A_i, A_j) and distinct tuple ids set $\mathcal{I}_i$, the sum of the column values is aggregated, grouped by entity, and added to a list R_L (Lines 1–6). Thus, if the sorted list R_L has a Footrule distance within the user threshold θ, then queries formed by combining this predicate(s) with $sum(A + B)$ as a ranking criteria are identified as candidate queries.
Moreover, this approach can be generalized to finding any type of ranking criteria. The adjustment needs to be done regarding the different aggregation, arithmetical operation, or number of columns. One can look at these approaches as executing a query with different ranking criteria on R', since a predicate selects the tuples with tuple ids in $\mathcal{I}_i$. The number of tuple sets is lower than the number of predicates. Furthermore, R' is kept in-memory and implemented as a column-store, which makes this validations very efficient.

C. Ranking of Candidate Queries

The queries that qualify as candidates and whose results on R' have a Footrule distance within the user threshold θ need to be executed on the base table R. The execution is done in order of ascending Footrule distance, i.e., the candidate queries with results most similar to the input list L are executed first. This allows for early identification of similar queries as shown in the following section.

IV. Experimental Evaluation

The implementation of our approach was done in Java. Experiments are conducted on a 2× Intel Xeon 6-core machine, 128GB RAM, running Ubuntu as an operating system, using Oracle JVM 1.8.0_45 as the Java VM (limited to 20GB memory). The base relation R is stored in a PostreSQL 9.4 database, with a B+ tree index on R's entity column.

	TPC-H	SSB
# Tuples	6,001,215	6,001,171
# Entities	99,996	20,000
# Textual columns	27	28
# Non-key numerical columns	13	20
# Avg tuples per entity	60	300
Highest # tuples per entity	178	579

TABLE IV: Table R characteristics

	Query	**sel.**
TPC-H	$\gamma_{c_name,MAX(o_totalprice)}(\sigma_{p_type='MEDIUM\ POLISHED\ STEEL' \wedge r_name='AMERICA'}(R))$	0.001
TPC-H	$\gamma_{c_name,SUM(ps_supplycost+ps_availqty)}(\sigma_{n_name='JAPAN' \wedge p_container='JUMBO\ BAG' \wedge l_shipmode='TRUCK'}(R))$	0.0001
SSB	$\gamma_{c_name,AVG(lo_revenue)}(\sigma_{s_nation='UNITED\ STATES' \wedge p_category='MFGR\#14'}(R))$	0.002
SSB	$\gamma_{c_name,SUM(lo_extendedprice*lo_discount)}(\sigma_{p_brand='MFGR\#2221' \wedge s_region='ASIA' \wedge d_year=1995}(R))$	0.00003

TABLE V: Example queries and their selectivity

Datasets: We evaluate our approach using two benchmarks, **TPC-H** [18] and the **SSB** [16]. For this, we created a scale factor 1 instance of both TPC-H and SSB data and materialized a single table R by joining all tables from their respective schema. The table R results in 57 and 60 columns, for TPC-H and SSB, respectively. The column *c_name* (from the Customer table) acts as the entity column. We obtain tables with the characteristics described in Table IV.

Input lists: As input lists for the evaluation we used variations of the results from the queries of the two benchmarks. There are 13 and 22 queries available in the TPC-H and SSB benchmark, respectively. We adjusted the original queries by creating different query types ($max(A)$, $avg(A)$, $sum(A)$, $sum(A + B)$, and $sum(A * B)$). We only write the ranking criteria when discussing the different query types. All queries have the column *c_name* as an entity column. Example queries and their selectivity are shown in Table V. We execute each query Q over the table R to produce the top-k lists L. Using the LIMIT clause, we create top-k lists with $k \in \{10, 20, 50, 100\}$. To study the effect of θ to the system, we perform the experiments with $\theta \in \{0.03, 0.06, 0.09\}$. We work with small values of θ since it is reasonable to assume that the user would only like to see queries similar to the input list and would like to avoid being overwhelmed with non-similar results.
For each of these top-k lists we generated top-k lists within distance θ of the original one, by either randomly swapping two elements or by introducing a new random entity not in R. Then, we used these lists for the experiments with different values of θ accordingly.

Number of similar queries discovered: Our system always manages to discover similar queries. Figure 4 reports on

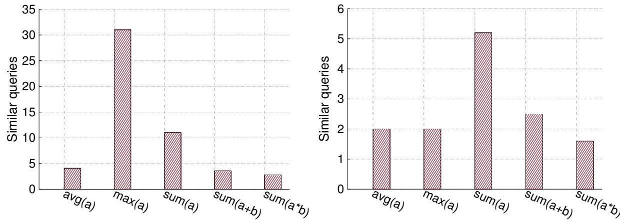

Fig. 4: Number of similar queries discovered with different ranking criteria, TPC-H (left), SSB (right)

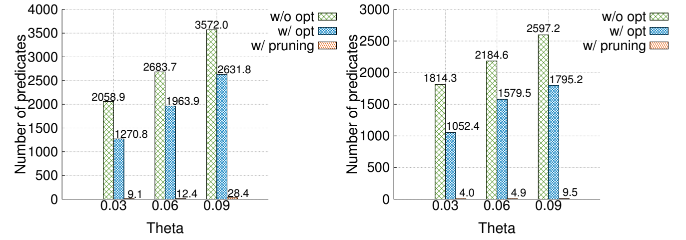

Fig. 5: Number of candidate predicates, TPC-H (left), SSB (right)

the average number of similar queries returned by the system over different values of θ and k, grouped by different query type, i.e., by the ranking criteria used for generating the input top-k list. We can see that for all the different query types, for both datasets, the system was able to find not only the original query that was used to generate the input list, but also several variations to it. With the TPC-H dataset the average number of similar queries was largest for those with $max(a)$ as a ranking function. Taking a deeper look shows that this was due to the selection part of the query. Many queries with the same aggregate function ($max(a)$) but with different predicates produced results with Footrule distance within θ. This is intuitive for this type of queries, if the tuple that is in the top-k result has a valid predicate of size n, all sub-predicates with size smaller than n will also be in the top-k result. This is not the case with the SSB dataset, it seems that the data there does not have these characteristics. Thus, the largest number of similar queries for this dataset was returned for the top-k lists ranked over the sum of one attribute ($sum(a)$). This is because many different ranking criteria, combined with the same predicates, produce similar top-k lists. The sum over a single column is susceptible to domination of certain columns. Thus, if the original query was ranked with sum over a column with large values, an aggregation with arithmetic operations that involve this column and columns with smaller values would not change the order in the result too much.

Our system is not only effective in finding the similar queries, but is also very efficient. Table VI reports on the **precision and recall at 10**, i.e., the ratio of the number of results in the first 10 executed candidate queries, and the ratio of the number of results found after executing the first 10 candidate queries and the total number of results. We can see that for the SSB dataset the Recall@10 is in all cases larger than 80%, meaning that we were able to find more than 80% of the similar queries by only executing 10 candidate queries. For the TPC-H dataset, the recall is slightly lower, however, in average, we can still find more than 50% of the similar queries by executing 10 candidate queries. We can see that even when we have a large number of candidate queries (117.6 for queries ranked by $sum(A)$ and $\theta = 0.09$), the system find more than 60% of the results within the first 10 query executions. For this dataset the precision is higher meaning that within the 10 queries executed, larger percentage are similar queries. For the SSB dataset the Precision@10 is lower but this is simply due to the lower number of valid similar queries.

Number of candidate predicates: Figure 5 reports on the number of candidate predicates created with and without the optimizations proposed in Section III-A, and after pruning out predicates as discussed in Section III-B. We see that for both datasets the system was able to prune out more than 99% of the candidate predicates without any query execution over the database. This shows that our algorithms are very efficient. First, the optimizations of knowing which subset of the elements is needed for generating the candidate predicates, the system was able to filter out at least 27% of the candidate predicates, considering all three values of θ, for the SSB dataset, and at least 24% of the data for the TPC-H dataset. In general, the percentage of filtered predicates with this method reduces as θ increases, since the number of mandatory entities that we can use reduces. Furthermore, by joining the candidate predicates with a ranking criteria, and then comparing the possible result over R' with the input list, we were able to further reduce the number of candidate predicates. In addition, we see that as θ increases the number of candidate predicates increases as well. This is due to the more relaxed conditions which qualify a predicate as a candidate with larger θ, as well as the fact that results with larger Footrule distance would also qualify as candidates.

Figure 6 shows the average number of candidate queries executed over the database, compared to the number of valid queries, for different values of k, for both datasets. Note that as valid queries we refer to those queries whose resulting top-k lists R_L is within Footrule distance θ to the input list L, i.e., $F(R_L, L) \leq \theta$. We see that, except for $k = 10$ the number of candidate queries that needs to be executed amounts to at most 4 times the number of valid queries for both datasets. However, this number decreases even further as we increase k since we have more information about the input list.

Runtime of the system: Figure 7 reports on the average execution time of the three stages of the system for different values of θ. The results for both datasets are similar. As expected, the execution of the candidate queries is the most demanding part of the system, since the queries are executed over a database that resides on disk. Finding the candidate predicates is less efficient than finding the ranking criteria, since even with knowing the subset of entities that must be satisfied by the predicates, the set of possible candidate predicates is larger than the set of possible ranking criteria. In addition, we can see that for Step 1 the execution time increases with θ linearly, which is understandable, since the number of predicates also increases with θ, because we are not able to prune as many predicates. On the other hand, we see that finding the number of candidate rankings is almost unaffected by θ. The execution time of Step 3, executing the queries and validating them, is most affected by increasing θ. Larger θ introduces more variability, which allows creating more candidate queries.

Query type	θ	TPC-H				SSB			
		# candidate Q	#valid Q	P@10	R@10	# candidate Q	#valid Q	P@10	R@10
$avg(A)$	0.03	60.5	4.1	0.31	0.53	9.5	2.0	0.57	0.82
	0.06	64.0	4.1	0.30	0.53	9.3	2.0	0.55	0.81
	0.09	73.5	4.1	0.27	0.51	14.4	2.0	0.47	0.81
$max(A)$	0.03	136.9	31.7	0.49	0.53	7.8	2.0	0.49	0.92
	0.06	486.0	31.7	0.43	0.54	10.2	2.0	0.46	0.88
	0.09	484.9	29.7	0.42	0.53	25.0	2.0	0.39	0.85
$sum(A)$	0.03	16.0	10.9	0.75	0.75	10.6	5.1	0.72	0.90
	0.06	22.7	11.1	0.66	0.70	15.7	5.2	0.60	0.85
	0.09	117.6	11.1	0.56	0.64	33.5	5.2	0.53	0.80
$sum(A+B)$	0.03	6.1	3.4	0.80	0.95	6.2	2.5	0.69	0.95
	0.06	18.3	3.3	0.60	0.83	11.8	2.2	0.51	0.94
	0.09	82.4	4.0	0.50	0.80	46.2	2.9	0.37	0.84
$sum(A*B)$	0.03	4.2	2.8	0.85	0.97	2.7	1.6	0.76	1.00
	0.06	9.2	2.8	0.80	0.86	4.5	1.6	0.64	0.93
	0.09	47.2	2.8	0.72	0.87	28.7	1.6	0.54	0.92

TABLE VI: Number of candidate and valid queries, precision and recall for the different query types for different θ

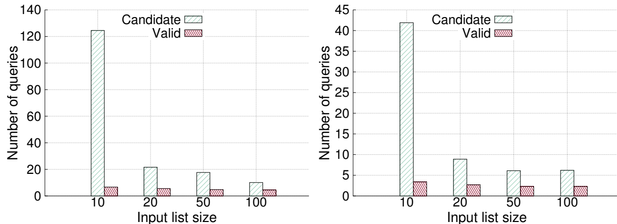

Fig. 6: Number of candidate and valid queries for different k, TPC-H (left), SSB(right)

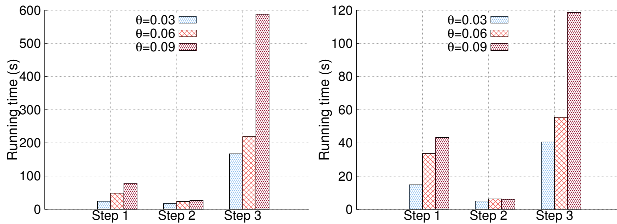

Fig. 7: Runtime by step, TPC-H (left), SSB (right)

V. Conclusion and Outlook

In this work we presented an approach that is able to determine queries that generate similar ranked lists according to a user-specified input list and similarity threshold. We motivated why searching for similar rankings (and not just a single precisely input-generating query) causes the search space of possible rankings to significantly grow—experiments clearly confirmed this observation. We showed how to tailor the process of determining promising candidate queries to the similarity measure in order to avoid generating too many dissimilar rankings that would impair system performance. We were able to demonstrate through experiments using TPC-H and SSB benchmarks that this tight incorporation in the query generation is able to drastically increase system performance compared to a distance-measure–agnostic baseline implementation.

References

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag, and S. Sudarshan, "BANKS: browsing and keyword searching in relational databases," in *VLDB*, 2002.

[2] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A system for keyword-based search over relational databases," in *ICDE* 2002.

[3] C. Binnig, D. Kossmann, and E. Lo, "Reverse query processing," in *ICDE* 2007.

[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: generating query-aware test databases," in *SIGMOD* 2007.

[5] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger, "SODA: generating SQL for business users," *PVLDB* 2012.

[6] N. Bruno, S. Chaudhuri, and D. Thomas, "Generating queries with cardinality constraints for DBMS testing," *IEEE Trans. Knowl. Data Eng.*, 18(12), 2006.

[7] S. Chaudhuri and G. Das, "Keyword querying and ranking in databases," *PVLDB* 2009.

[8] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," *SIAM J. Discrete Math.*, 17(1), 2003.

[9] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: ranked keyword searches on graphs," in *SIGMOD* 2007.

[10] E. Milchevski, A. Anand, and S. Michel, "The sweet spot between inverted indices and metric-space indexing for top-k-list similarity search," in *EDBT* 2015.

[11] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *SIGMOD* 2008.

[12] K. Panev and S. Michel, "Reverse engineering top-k database queries with PALEO," in *EDBT* 2016.

[13] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri, "S4: top-k spreadsheet-style search for query discovery," in *SIGMOD* 2015.

[14] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom, "Synthesizing view definitions from data," in *ICDT*, 2010.

[15] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik, "Discovering queries based on example tuples," in *SIGMOD*, 2014.

[16] The Star Schema Benchmark. http://www.odbms.org/2014/03/star-schema-benchmark/

[17] S. Tata and G. M. Lohman, "SQAK: doing more with keywords," in *SIGMOD* 2008.

[18] TPC. TPC benchmarks, http://www.tpc.org/.

[19] Q. T. Tran, C. Chan, and S. Parthasarathy, "Query by output," in *SIGMOD* 2009.

[20] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag, "Reverse top-k queries," *ICDE*, 2010.

[21] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava, "Reverse engineering complex join queries," in *SIGMOD* 2013.