Reverse Engineering Top-k Database Queries with PALEO[*]

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

Sebastian Michel
TU 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. In this work, we address the problem of reverse engineering top-k queries over a database, that is, given a relation R and a sample top-k result list, our approach, named PALEO[1], aims at determining an SQL query that returns the provided input result when executed over R. The core problem consists of finding predicates of the where clause that return the given items, determining the correct ranking criteria, and to evaluate the most promising candidate queries first. To capture cases where only a sample of R is available or when R is different to the relation that indeed generated the input, we put forward a probabilistic model that allows assessing the chance of a query to output tuples that are resembling or are somewhat close to the input data. We further propose an iterative candidate query execution to further eliminate unpromising queries before being executed. We report on the results of a comprehensive performance evaluation using data and queries of the TPC-H and SSB [14] benchmarks.

1. INTRODUCTION

Reverse engineering database queries describes the task of obtaining an SQL query that is able to generate a specified input table, when executed over a given database instance. This generic problem has various important application scenarios, specifically for top-k database queries that often yield valuable analytical insights. Consider, for instance, business analysts who are interested in determining alternative queries that yield the same or similar query result tuples, data scientists who try to find explanatory SQL queries for crowd-sourced top-k rankings, or to find the data-generating query of a sample input in order to re-execute it on current or future database instances in cases where the original query has not been saved or has not been made public, for one or another reason. The discovered queries can reveal interesting properties of the input, most importantly the constraints to tuples expressed in the "where clause" of the query and how tuples are ranked. The last years have brought up various research results [17, 12, 19] on reverse engineering database queries. Compared to existing approaches that operate on input in form of full tables, reverse engineering top-k queries adds two complex ingredients to the re-engineering task. First, it is the rather small input, consisting of only a few (as k is usually quite short) ranked tuples and, second, the various ways top-k SQL queries can be formulated, given various sorting orders and aggregation functions.

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

[1]PALEO is approximately the reverse of the word OLAP and also emphasizes the goal of assembling queries based on their data footprints (results), much like paleontologists reconstruct and study fossils.

Name	City	State	Plan	Month	Minutes	SMS	Data
John Smith	SF	CA	XL	June	654	87	1,230
John Smith	SF	CA	XL	July	175	22	900
...	...	...	...	...	...	...	...
Jane O'Neal	LA	CA	XL	April	699	15	2,300
Jane O'Neal	LA	CA	XL	June	334	10	1,900
...	...	...	...	...	...	...	...
Richard Fox	Oakland	CA	XL	June	596	23	1,272
...	...	...	...	...	...	...	...
Jack Stiles	San Jose	CA	XL	March	429	42	1,192
Jack Stiles	San Jose	CA	XL	April	586	8	1,275
...	...	...	...	...	...	...	...
Lara Ellis	San Diego	CA	XL	May	784	11	2,107

Table 1: Sample relation of telecommunications traffic data

Consider a relation *Traffic*, illustrated in Table 1, containing cellphone-traffic data. The relation contains textual attributes like name of the customer, the city and state the customer lives in, and the tariff plan and the month for which the traffic was realized. In addition, there are numerical attributes that measure the customer's traffic, like number of minutes talked, the number of text messages (SMS) sent, and the number of spent megabytes of data.

Lara Ellis	784
Jane O'Neal	699
John Smith	654
Richard Fox	596
Jack Stiles	586

Table 2: Example input list

Table 2 shows a top-k list with two columns and five rows. The input list does not have attribute names (or if it does, are not correlated to the attribute names in the database table). The first attribute is the customer's name, while the second is the performance attribute according to which the customer ranking was produced. Note that there are no empty cells in the list, all values are specified. Considering the *Traffic* relation of Table 1, we can see that the input ranking list can perhaps be generated using the following query:

```
SELECT name, max(minutes)  FROM traffic
WHERE state = 'CA'
GROUP BY name  ORDER BY max(minutes) DESC
LIMIT 5
```

This query computes the top 5 customers of the telecommunications company, living in the state of California, ranked by the number of minutes talked in a single month. In general, there can be several different queries that produce the same results; consider for instance augmenting the above query Q with an additional constraint to customers with the tariff plan "XL", it would leave the result unchanged (including the order among tuples).

1.1 Problem Statement

Given a database D with a single relation R with schema $\mathcal{R} = \{A_1, A_2, \ldots\}$ and an input relation L that represents a ranked list of items with their values. **The task we consider in this paper is to efficiently and effectively determine queries Q_i that output tuples that resemble L when executed over R.**

We focus on top-k select-project queries over relation R of the form shown in Figure 1(left). We specifically focus on a single relation to emphasize on the intrinsic characteristics of top-k queries, instead of considering the reverse engineering of joins, too, which has been addressed by Zhang et al. [19] in their recent work on reverse engineering complex join queries.

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

L	
$L.e$	$L.v$
e	100
f	90
g	80
m	70
o	60

Figure 1: Query template (left) and example input L (right)

The problem has two properties that can be relaxed or tightened. First, it can either demand determining only one, multiple, or all input-generating queries. Second, the notion of a query being valid in the sense that it resembles the input can be relaxed to a notion of approximately resembling the input.

The problem is challenging for the following reasons: (i) The size of the input list is rather small, it is difficult to derive meaningful (statistical) properties in order to identify valid predicates and ranking criteria, (ii) the relevant subset of R that features all tuples of the entities in L can become very large, and (iii) false positive and false negative candidate queries deteriorate system performance due to many necessary query evaluations and limit the chance to successfully determine a valid query that generates the input.

The presented approach, coined PALEO, is not limited to finding exact matches, but can almost directly be applied to finding queries that compute a ranking L' over R, with L' being similar to L. We get back to this generalization in Section 3.3. We refer to the specific attribute in R that contains the entities the table reports on as A_e and assume it is known a priori.

As already indicated in the template query, 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 = "CA"). Furthermore, we denote with *size* of a predicate $|P|$ the number of atomic predicates P_i in the conjunctive clause.

The input top-k list L has two columns; $L.e$ and $L.v$ denote the entity column and the numeric score column, respectively. Note that L does not contain the name of the column $L.v$ or the column name of $L.v$ is named for human consumption (e.g., "Total traffic", which can be total number of minutes, SMS, or data), i.e., not corresponding to the ones present in the database. Hence, referencing to the appropriate attribute in R cannot be done by name. Table 3 shows a summary of the most important notations used throughout this paper.

1.2 Sketch of the Approach

A naïve approach would enumerate all possible queries, say with a limited complexity of the predicate in the where clause, evaluate the queries one-by-one against the database and check whether the returned results resemble the input list. This is clearly beyond hope, even for relatively small databases and schemas.

Our approach, conceptually, loads all tuples from R that contain any of the entities in L. This table is called R' and is used in two subsequent steps, first, to determine the query predicate and, second, to find the right attribute(s) and aggregation function. In case R' is completely given, our approach is extremely effective in determining the individual building blocks of the desired query. When working on a subset of R', we show how to handle large amounts of potential candidate queries by introducing a suitability-driven order among them, in order to find the desired query early.

1.3 Contributions and Outline

With this paper we make the following contributions:

- To the best of our knowledge, this work is the first to consider the problem of reverse engineering top-k OLAP queries. We present an efficient and effective solution to it, in a flexible and extensible framework.
- We show how to efficiently compute promising predicates using an apriori-style algorithm over R' and how to augment them with ranking criteria using data samples and statistics obtained from the base relation R.
- We present a probabilistic reasoning that allows ordering candidate queries by the likelihood that they compute the input ranking L. This, together with a method to skip unpromising queries dynamically at validation time, allows finding the desired valid queries very efficiently.
- We report on the results of a carefully conducted experimental evaluation using data and queries from the TPC-H [16] and SSB [14] benchmarks.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents the framework and key ideas behind our approach, followed by the specific sub-problems of identifying query predicates in Section 4, and determining the ranking attributes and aggregation function, in Section 5. Section 6 considers handling changed data in R, and proposes a probabilistic model to rank queries by their expected suitability to generate the input. Section 7 introduces an incremental strategy to eliminate unpromising candidate queries based on observed results of already executed candidates. Section 8 reports on the results of the experimental evaluation and presents lessons learned. Section 9 concludes the paper.

2. RELATED WORK

The problem of reverse engineering queries was considered by Tran et al. [17] in their 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

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
$L.v$	Ranking column in L
e_i	Entities in A_e or $L.e$
v	Values in A_i
P	Predicate (atomic or conjunctive)
Q	Query
$Q(R)$	Result set of Q when querying R

Table 3: Overview of Notations

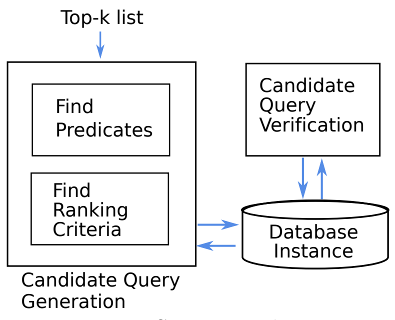


Figure 2: System task steps

problem as a data classification task. For generating the selection conditions they use a decision tree classifier that is constructed in a top-down manner in a greedy fashion by determining a "good" predicate according to which the tuples are split into two classes. These two classes would then form the root nodes of two decision trees (constructed recursively).

Sarma et al. [12] 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. Thus, they only try to find the selection condition of the view V and do this looking at the problem as an instance of the set cover problem. From the families of queries that they cover, we focus on conjunctive queries with a single equality predicate and conjunctive queries with any number of equality predicates. For both types they propose naïve algorithms that utilize the size of the attribute domains in the view. Zhang et al. [19] compute a generating *join query* that produces a table $Q(D)$ from the tables in D. The generated join query does not have selection conditions and they focus mostly on identifying the joins using graph structures following foreign/primary-key links.

Shen et al. [13] 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. By using common sub-join trees of the candidate queries as filters they manage to improve the efficiency of their approach.

Psallidas et al. [10] 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. The system returns the PJ queries with the top-k highest scores and it discovers not only the queries that exactly match the given example tuples. Moreover, they propose a caching-evaluation scheduler, where they dynamically cache common sub-expressions that are shared among the PJ queries. Join queries are orthogonal to our work and none of the above approaches handle top-k aggregation queries.

In keyword search over databases [2], the input is a single tuple with specified keywords as fields. The works of [5, 15] 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. [15] discovers aggregate SQL expressions that describe the intended semantics of the keyword.

The principle of reverse query processing is studied in [3, 4, 6, 9], 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. [9] study the problem of generating test queries to meet certain cardinality constraints on their subexpressions.

A reverse top-k query [18] 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.

3. APPROACH

The task of reverse engineering top-k queries is split into the following three steps, illustrated in Figure 2:

- **Step 1:** find the predicate P in the where clause of Q
- **Step 2:** find the ranking criteria
- **Step 3:** validate queries

As the basis of further computation, we first retrieve from relation R all tuples whose entity column contains one of the entities of the input table L; we call the resulting table R'.

3.1 Table R'

Consider a top-k list L as shown in Figure 1. Let $e_i \in \{e, f, g, m, o\}$ denote the entities in the column $L.e$.

By using a standard database index, such as a B+ tree, on the entity attribute of R, we can efficiently retrieve R' (shown in Table 4) containing all tuples from R matching any of the entities $e_i \in L.e$. Whether the index is actually used or the query optimizer decides to perform a table scan is not a concern here. In any case, in this example, the query to compute R' is

```
SELECT * FROM R
WHERE A_e IN [e, f, g, m, o]
```

For the purpose of efficient access of its data, PALEO 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' 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. We will allow to work on a subset (samples) of R' in Section 6, and study the consequences, but for now we assume R' in fact covers *all* tuples of *any* entity of the input.

3.2 The Three Steps

Candidate Predicates Identification. Using the tuples in R' we 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 each entity that appears in L there is a tuple t in R' that fullfils the predicate. Formally,

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

It is easy to see that a candidate predicate can potentially produce the top-k input list. In other words, having a candidate predicate in the where clause is a *necessary criterion* of a query to be a valid query, but it is *not a sufficient criterion*. This is because a candidate predicate can still "let through" tuples of other entities (that are not in the input table L) that can be ranked higher than the tuples in L, hence, the query is not a valid query as the output does not match the input.

COROLLARY 1. **Downward-closure (anti-monotone) property of the candidate predicate criterion.** *Given a predicate P_1 that is not a candidate predicate, then a predicate P_i such that $P_1 \subseteq P_i$ (that is, all sub-predicates in P_1 are also present in P_i) can not be a candidate predicate.*

The corollary follows immediately from the defitinion of candidate predicates: any predicate P_i with $P_1 \subseteq P_i$ for another predicate P_1 evaluates to true for a subset of tuples for which P_1 evaluates to true. This property is used to prune the searchspace in Section 4, similar to what the apriori algorithm [1] does for the support measure.

Ranking Criteria Identification. In the *second step* of our approach, we identify the ranking criteria according to which the entities in the top-k list are ranked. 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 2. **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 $\mathbf{R}'$** *together with a candidate predicate, it returns a* **result identical to the input list** *L.*

This definition is very reasonable but similar to the criterion to identify candidate predicates it is only a necessary condition to a valid ranking criteria for a query when executed over the entire relation R. It is, however, not a sufficient condition, as when executed on R there can be still other entities, not in L, that are disturbing the "correct" order. The case of partial matches is discussed below.

Candidate Queries Identification and Evaluation. Using the candidate predicates and the valid ranking criteria we can form candidate queries. Each candidate query is executed on R and the results are compared with the input top-k list. The queries that produce instance-equivalent results with the original query are the valid queries.

3.3 Allowing Partial Matches

Like other approaches on reverse engineering queries, this approach can be relaxed to allow finding also partially matching queries. This can be useful for cases where the input L has been obtained from an older instance of the database or in cases where L has been generated in the extreme, through crowdsourcing top-k rankings. Our approach can be adapted to such partial match scenarios as follows. First, the condition to accept a query during the validation phase needs to be switched to accepting partial match. For comparing rankings, there exist several ways, most prominently Spearman's Footrule distance and Kendall's Tau. Fagin et al. [7] show how these measures can be applied to top-k lists. In our case of ranking with two columns (entity and value) we would compute such methods on the entity column; and can additionally compute a distance measure like L1 or L2 on the values if numerical or otherwise use a distance like the set-based Jaccard distance. Second, not taking for granted that we cannot precisely reverse engineer the input L implies that even a fully known R' would behave exactly like being a sample, with the consequences described in Section 6. That means, we can directly apply the reasoning on query suitability explained there.

R'

$t.id$	E	A	B	C	$\cdots$	N_1	N_2	N_3	$\cdots$
1	e	a_1	b_9	c_3	$\cdots$	75	4	5	$\cdots$
2	e	a_1	b_8	c_1	$\cdots$	100	8	7	$\cdots$
3	e	a_3	b_1	c_6	$\cdots$	45	15	1	$\cdots$
4	f	a_1	b_8	c_1	$\cdots$	90	16	2	$\cdots$
5	f	a_5	b_4	c_6	$\cdots$	35	23	3	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
10	g	a_1	b_8	c_3	$\cdots$	80	42	14	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
20	m	a_1	b_8	c_4	$\cdots$	70	29	10	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
30	o	a_1	b_8	c_4	$\cdots$	60	31	7	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Table 4: Example of a Relation R' for Input L in Figure 1

4. CANDIDATE PREDICATES

The task we consider in this section is to find all k-sized candidate predicates P_i. Each predicate can be simple atomic equality predicate like $(A = a_1)$ or conjunctions of atomic equality predicates, e.g., $(A = a_1) \wedge (B = b_8)$. Candidate predicates are determined over the table R', as described above. From Definition 1 we know that in order to be a candidate predicate, a predicate P has to have for *each* entity in the input $L.e$ *at least one* tuple in R' with $P(t) = true$.

This criteria is anti-monotone (aka. downward-closed), i.e., a predicate P_i with size k can be considered a candidate predicate if and only if all its sub-predicates are also candidate predicates. This problem is similar to frequent itemset mining for which the apriori principle and algorithm [1] is widely known. In data mining terminology, itemsets resemble the values that are used to form the candidate predicates.

The method to compute candidate predicates in PALEO is described in Algorithm 1. In the first step, $k = 1$, we start by identifying all atomic candidate predicates, i.e., the predicates with size $|P_i| = 1$ (Lines 2–6 in Algorithm 1). For this purpose for each column A_i we identify values v such that the predicate $P_i := (A_i = v)$ is a candidate predicate (Lines 3–4 in Algorithm 1). Furthermore, for each such created P_i we keep a set $\mathcal{I}_{P_i}$ containing the tuple ids (aka. row ids) that this predicate selects, i.e., $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$. In each additional step, conjunctive predicates of size k are created, by adding atomic predicates from the set $\mathcal{P}_1$ to the predicates created in the previous iteration (Lines 7–14 in Algorithm 1). The algorithm does not create a predicate multiple times. The conjunctive predicate P_{ij} whose tuple ids set $\mathcal{I}_{P_{ij}}$ covers all entities in the input list is added to the set of candidate predicates with size k (Lines 12–13) and will be used in creating candidate predicates of size $k + 1$ in the next iteration.

Example: Considering Table 4 and the input list in Figure 1, we create atomic predicates starting with column A as we iterate over its values a_i. Note that the entities in E are sorted. The set of atomic candidate predicates is created, $\mathcal{P}_1 = \{P_1 := (A = a_1), P_4 := (B = b_8)\}$. These two predicates are candidates, since the tuples that fulfill the predicates cover all entities in the input list L. Furthermore, the set tuple ids that the predicates select are kept, e.g., $\mathcal{I}_{P_1} = \{1, 2, 4, 10, 20, 30\}$. If added as a selection condition,

```
method: findPredicates
input: top-k list $L$
       relation $R'$
output: a set of candidate predicates $\mathcal{P}$
$\mathcal{P} = \emptyset$; $k = 1$; $\mathcal{P}_k = \emptyset$
for each $A_i$ in $R'$
    find $P_i := (A_i = v)$ with $|P_i| = 1$ s.t.
    $\forall e_i \in L.e \, \exists \, tuple \, t \in R' : P_i(t) = true \wedge t.e = e_i$
    add $P_i$ to $\mathcal{P}_k$
    for each $P_i$ keep $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$
repeat
    $k = k + 1$
    $\mathcal{P}_k = \emptyset$
    for each $P_i \in \mathcal{P}_1$ and $P_j \in \mathcal{P}_{k-1}$ and $P_i \cap P_j = \emptyset$
        create $\mathcal{I}_{P_{ij}} = \mathcal{I}_{P_i} \cap \mathcal{I}_{P_j}$
        if $\mathcal{I}_{P_{ij}}$ covers all $e_i \in L.e$
            add $P_{ij} := P_i \wedge P_j$ to $\mathcal{P}_k$
until $\mathcal{P}_k = \emptyset$
return $\mathcal{P} = \bigcup_k \mathcal{P}_k$
```

Algorithm 1: Finding candidate predicates

these candidate atomic predicates would result in a candidate query.

In each next step, we try to produce conjunctive clauses of size k from the predicates in $\mathcal{P}_1$ and $\mathcal{P}_{k-1}$. Thus, for $k = 2$, we test if the predicate $P_{14} := (A = a_1) \wedge (B = b_8)$ qualifies as a candidate by intersecting the corresponding sets of tuple ids. Since the intersected tuple ids in $\mathcal{I}_{P_1} \cap \mathcal{I}_{P_4} = \{2, 4, 10, 20, 30\}$ cover all entities in $L.e$, the predicate P_{14} is a candidate predicate. Recall that R' is held in memory and that we can, via tuple ids, very efficiently access the full tuple to check whether or not it matches the predicate.

Properties of the Algorithm:

(i) The algorithm is **correct** with respect to R', that is, predicates returned by the algorithm are guaranteed to be candidate predicates, following Definition 1. Further, the algorithm is **complete**, that is, it finds all possible candidate predicates over R'.

(ii) When predicates are applied in R instead of R' they can also let tuples with entities that are not in L pass, which leads to **false positive** candidate queries.

The difference to the apriori algorithm that operates on the support measure is that apriori counts the frequency of *all* itemsets and then determines the ones above the specified threshold. In our algorithm, we eliminate a predicate as soon as we find that it does not cover a certain entity. The same happens in each additional pass, since apriori will generate all the pairs of frequent items and count their appearance. Thus, all pairs that contain a false positive singleton will also be false positives.

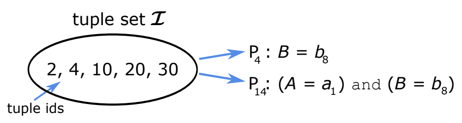


Figure 3: Mapping from tuple set to predicates.

4.1 Tuple Sets and Predicates

Some of the created candidate predicates have identical tuple sets $\mathcal{I}_{P_i}$. These predicates select the same tuples in R' and share the same data characteristics regarding to R'. Thus, candidate predicates are grouped according their tuple sets, i.e., if $\mathcal{I}_{P_i} = \mathcal{I}_{P_j}$, then P_i and P_j would belong to the same group.

Figure 3 depicts a tuple set mapped to a group of candidate predicates created from the tuples in Table 4. The predicates P_4 and P_{14} cover the same tuples in Table 4. Thus, for these predicates, it is enough to examine the data characteristics of the tuples in the tuple set $\mathcal{I}$.

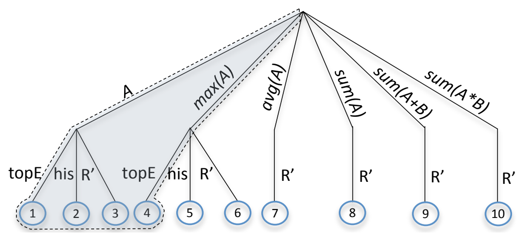


Figure 4: Order of looking for the ranking criteria

5. RANKING CRITERIA

In order to find the ranking criteria according to which the ranking in the top-k list is done, PALEO operates on the distinct tuple sets, determined in the algorithm above. If relation R, and hence also R', is identical to the database state when the input data was once generated, it is guaranteed that PALEO is able to determine the valid ranking criteria.

The actual size of R' depends naturally on the size k of the input list L and also on the data characteristics, i.e., how many tuples R contains for a single entity. We expect R' to be holding a factor of k/n less tuples than R, where n is the number of distinct entities in R, and that this allows to load R' entirely in main memory. While it might be reasonably cheap to execute a query on this R' in memory, note that we have to possibly do so very many times to identify suitable ranking criteria. That is, depending on the size of R' we can potentially reduce the runtime of our algorithm if it can be avoided to work on R' directly.

The idea is to harness small data samples, histograms, or simple descriptive statistics computed upfront from the base relation R in order to select a subset of potentially useful columns without touching R'. However, there might be invalid criteria identified or potentially also no criteria at all, given the limited coverage of data samples and the impreciseness of histograms. Therefore, identified candidate ranking criteria are validated on R' and in case no heuristic is applicable or was not successful, the whole ranking criteria identification is executed on R'.

Depending on the aggregation function we aim at checking for suitability, we can or cannot use some of these techniques. For instance, comparing the entities in L with the top entities stored for each column of R can be applied to queries with *max* aggregate function, but not directly to queries using *sum* as the aggregation function. Figure 4 summarizes this observation. Traversing the tree pre-order depth-first is the way PALEO looks for the ranking criteria, with the leaf nodes showing the order in which the techniques are applied. The system tries to identify the ranking criteria with smaller search space first. Thus, for instance, if the valid ranking criteria is $max(A)$ and comparing the top entities produces valid results, only the shaded part of Figure 4 will be processed.

5.1 Top Entities

The most apparent first attempt to identify an attribute according to which tuples are sorted in L is to store for each attribute in R the topmost entries, when sorted by the specific attribute. Then, we intersect the input entity set

```
method: topEntities
input: top-k list $L$
       relation $R'$
output: a set of candidate numerical columns $\mathcal{A}_C$
1  for each $A_i$ in $R'$
2      if $A_i$ not numerical, then skip $A_i$
3      if $\max(v \in A_i) < \max(v \in L.v)$, then skip $A_i$
4      if $\min(v \in A_i) > \min(v \in L.v)$, then skip $A_i$
5      if $|A_i| < |L.v|$, then skip $A_i$
6      if $TopE(A_i) \cap L.v \neq \emptyset$, add $A_i$ to $\mathcal{A}_C$
7  return set of candidates $\mathcal{A}_C$
```

Algorithm 2: Finding candidate columns with top entities

from L with these top entries. More than just the k top values are stored to increase the chance that these entities do overlap with the entities in L. Clearly, it should also not be too large such that each numeric column appears promising. The exact way of how this idea is applied is shown in Algorithm 2, line 6.

Before this is done, PALEO filters out attributes by applying three simple checks: it compares the max (min) values of the input list and the column and if the column's value is smaller (greater) than the max value of the input list it does not intersect the entities (Algorithm 2, line 3 and 4). Additionally, the number of distinct values is compared: If the column has less distinct values than the input list, we skip this column (Algorithm 2, line 5).

The numerical columns that result in a *non-empty intersection* are considered as candidate numerical columns. Thus, using R' and the tuple sets created in finding candidate predicates, they are checked whether they can match the ranking in the top-k input list.

5.2 Querying Histograms

In the case no candidate numerical columns have been identified with the above intersection of top entities, PALEO employs histograms describing an attribute's frequency distribution in order to find candidate attributes that appear suitable for ranking. As we consider only numeric attributes to be used as the bases of ranking criteria, such a histogram describes how frequent a specific numeric value appears in the attribute's column in relation R. One idea is comparing the value-frequency distributions of the histogram of the input list with the histograms of the numerical columns in R, by using histograms distance measures such as Earth Mover's Distance [11]. However, a top-k list is inherently small and does not contain enough elements to provide a meaningful distribution. Hence, PALEO samples each attribute's histogram and calculate the L1 distance between its top-k values and the input values. Similar to using top entities of each column, we draw samples following the distribution described in the histogram. PALEO uses equi-width histograms having 1000 cells each.

This is done for each attribute, which allows ordering all attributes by the L1 distance of the sampled data to the data in L. Depending on the data in the table, if there is a column with similar values and distribution as the column we are looking for, it is possible that the correct column does not have lowest L1 distance. In order to account for this, we consider the top 30% of the columns in the list as candidate attributes.

5.3 Validation over R'

As a validation for the possible ranking criteria identified above, we use the tuples in R'. In the case when we have successfully identified candidate attributes with the previous techniques, we first check if any of these candidate attributes can produce the ranking. For this purpose, we go through the distinct tuple sets $\mathcal{I}_i$ we computed in Section 4 and check which of the candidate numerical columns, i.e., their sorted aggregated values exactly match the input L.

Some of the supported ranking criteria cannot be identified by the above mentioned techniques, requiring more complicated statistics and this is beyond the scope of this paper. For instance, with the *avg* and *sum* aggregate functions, the top entities for a column depend heavily on the predicate, since the values are aggregated over multiple tuples. Similarly, harnessing histograms with *sum* would involve convolutions of the histograms of the pairs of columns.

As a fall back, if none of the candidate attributes can produce the ranking criteria, we revert to checking the remaining numerical columns in R' that were not found as candidates. We still use only the tuples with tuple ids found in the tuple sets $\mathcal{I}_i$. For each tuple set and each numerical attribute in R' that passes our (three) simple checks (i.e., min, max comparison, and number of distinct items), we compute whether the tuples in $\mathcal{I}_i$ if sorted according to the specific attribute and aggregate function are identical to L. After identifying the appropriate numerical attribute and aggregate function, we can filter out some of the *candidate predicates.* If a certain tuple set does not contain the input numerical values, we **remove** this tuple set and all the candidate predicates that correspond to it from the candidate predicates.

6. HANDLING VARIATIONS OF R

The techniques behind PALEO discussed so far are based on the assumption that exactly the same relation R that produced the input list L is available and that it is feasible to operate on it directly. However, it might appear that tuples in R have changed, for instance, because of inserts, updates, and deletes, due to slowly changing dimensions [8] in data warehousing scenarios, or only a subset (sample) is available. In this section, we describe how PALEO deals with situations when only subset of the original tuples in R is available.

This assumption has direct consequences on PALEO's ability to accurately identify suitable predicates and ranking criteria. As we have discussed above, determining query predicates with the proper table R at hand *only* leads to obtaining **false positives** in the candidate predicates, introduced by additional entities outside R' that qualify for the predicate. The changed data further introduces **false negatives**. That is, the query that generated the input might not be found at all, although such a query exists. This is caused by missing or modified tuples in R that would be required to unveil a predicate to be fulfilled by all of the k entities. False negatives are synonym to *loss in recall*, i.e., the fraction of found queries to all existent queries that generate the input.

We address this by

- Reasoning about likelihood of being a successful query.
- Smart evaluation to skip unpromising queries.

Variations in R means also variations in R'. Let us denote the table stemming from the modified base table as R''. It can happen that R'' does not contain tuples from all entities from the input list, for instance if all tuples for a certain entity $e_i \in L.e$ have been deleted from R. Recall that the method for finding predicates, described in Algorithm 1 demands that a predicate must cover all entities of the input list $L.e$.

Now, it is possible that the tuples containing the valid predicate for a certain entity have changed in the columns that comprise the predicate. Then, it is impossible to precisely validate or invalidate the predicate using the method in Algorithm 1: Being strict, missing the tuples with the valid predicate for a certain entity will lead to evicting the valid predicate even though the majority of entities in R'' contain tuples with it, thus resulting in false negatives. To avoid that, the condition of evicting a predicate is relaxed. Instead of demanding that a predicate is considered as a candidate predicate if it covers all distinct entities in R'', we ask for it to cover the *majority of the entities*, thus taking into account that some entities can have tuples with the valid predicate missing. Another possible approach is not to evict predicates at all, i.e., form all the predicates that we encounter in R'' while not demanding any entity covering. This might, however, result in very many candidate predicates with too many false positives. Executing candidate queries for all such predicates will drastically decrease the overall efficiency of PALEO.

We describe a probabilistic model of assessing candidate predicates when the data in the base table has changed and how uncertainty in finding ranking criteria can be handled.

6.1 Assessing Candidate Predicates

Changes in R introduce uncertainty in finding the valid predicates. To account for such changes, the condition of evicting a predicate is relaxed. As a result, our methods identify more candidate predicates that need to be assessed whether or not they are likely to be indeed a valid predicate. This assessment is later used when executing queries in the final step such that queries can be executed in increasing order of the likelihood to be in fact a valid query.

A candidate predicate P_i identified from the table R'' is a false positive if: $\exists\ e_i \nexists\ t$ s.t. $P(t) = true$. In other words, if for a certain entity e_i there is no tuple for which the predicate P is valid, then this predicate is a false positive. This means that a query with this predicate would return a top-k list without the entity e_i.

Consider a predicate P over the attributes $A_1, \dots A_m$. The probability that a tuple exists in relation R is given by the number of distinct entries of the columns A_i (i.e., $|A_i|$) as

$$P[\text{tuple exists}] = \prod_i \frac{1}{|A_i|}$$

Consider an entity e_j for which we did not find a tuple that matches the predicate and let $unseen(e_j)$ be the number of changed tuples of entity e_j, then

$$P[\text{won't see for } e_j] = (1 - P[\text{tuple exists}])^{unseen(e_j)}$$

The probability that at least one entity is rendering this predicate to be a false positive (by not providing a matching tuple) is thus given as

$$P[\text{false positive}] = 1 - \prod_j (1 - P[\text{won't see for } e_j])$$

6.2 Approximating Ranking Criteria

Operating on R'' also introduces uncertainty in finding ranking criteria. Since not all tuples for each entity e_i are the same, the ranking criterion cannot exactly match the numerical values in the input top-k list. This is why there is a need of measuring the suitability of each candidate ranking criteria to the input list. For this purpose, we compute the distance between the input values and the candidate attribute(s) values. We use the L1 distance (aka. Manhattan distance) that is simply the sum of absolute differences in the numeric values.

Queries without sum: The changes in the tuples for an entity e_i renders the **topEntities** method (Section 5.1) not directly applicable. Without the identical tuples, it is difficult to match the candidate numerical columns with the input ranking values. Using the L1 distance and the column values in R'' (Section 5.3) provides the possibility to compute the suitability of the candidate ranking columns. That way, each candidate column has a corresponding L1 distance that is used in ranking the candidate queries.

Queries with sum: The **sum** aggregate function sums up all values for a certain entity e_i. Since with changed data some of the tuples for an entity are missing, they need to be approximated. We do this by using the column values for the column(s) in R'. Using this approximation, the L1 distance to the input ranking values is calculated and then used for ranking the suitability of the column(s).

The approximation of the sum for each entity is done using the tuple id sets. We take a look at the more complicated case of having a sum of two columns A_i and A_j. Thus, for a predicate P with a corresponding tuple set $\mathcal{I}_P$, for each entity e_i let $sum_{A_{ij}}(\mathcal{I}_P)$ denote the sum of the values, of the columns A_i and A_j, of the **tuples in** R'' with tuple ids in $\mathcal{I}_P$ that have an entity e_i, i.e.:

$$sum_{A_{ij}}(\mathcal{I}_P) = A_i(\mathcal{I}_P)\ op\ A_j(\mathcal{I}_P)\ \ s.t.\ t.e = e_i\,,\ \ op \in \{+, *\}$$

Additionally, let $\#v$ denote the number of tuple ids in the tuple set $\mathcal{I}_P$ of the entity e_i, i.e., the number of tuples that the predicate P selects with e_i. We approximate the sum as:

$$appxSum_{e_i}(\mathcal{I}_P) = \frac{sum_{A_{ij}}(\mathcal{I}_P)}{\#v} \times (\#v \times \frac{|e_i|_{R''}}{|e_i|_{R''} - unseen(e_i)})$$

where $|e_i|_{R''}$ is the number of tuples in R'' for the entity e_i. Thus, for each entity e_i, the average summed value from the sampled tuples is multiplied with the approximated selectivity of the predicate P. The sorted list $appxSum(\mathcal{I})$ formed from the sums for each entity $appxSum_{e_i}(\mathcal{I}_P)$ is then used for calculating the L1 distance d to the input list and ranking the candidate column pairs.

6.3 Combined Model

The queries formed from the combination of candidate predicates and ranking criteria need to be validated by executing them on R. The order of execution is done ordered by a suitability value for each candidate query Q_c. The suitability is computed as:

$$s(Q_c) = (1 - P[\text{false positive}]) \times (1 - d)$$

where P[false positive] is the probability of the predicate in the candidate query of being a false positive and d is the max normalized L1 distance between the ranking criteria in Q_c and the numerical values in the input list L.

6.4 Working with Samples of R'

Consider a scenario where it is impossible or unfeasible to work on the complete relation R' (the subset of R of all tuples that contain any of the entities in L). This relation R' can be very large, potentially as big as R, if there are many tuples for each distinct entity—a typical case in data-warehousing applications that often aggregate large amounts of observations of a specific entity. The probabilistic model for assessing candidate predicates together with the approximation of the ranking criteria can also be applied to such a scenario as well.

We consider two approaches of sampling. First, we sample by retrieving all tuples for a certain (e.g., randomly selected) subset of the entities in $L.e$. In this way, we do not get any false negatives and the candidate predicates set is a superset of the valid predicates. This is because having all tuples in R'' for a certain entity is guaranteed to contain the tuples with valid predicates. As a result, our algorithm will create the predicate as a candidate. However, the drawback of this approach is having too many false positives. This can especially happen if for a sampled entity there are too many tuples in the base table R. This will lead to creating a large amount of false positives which impairs efficiency.

Sampling uniformly from all entities mediates this problem, thus sampling a certain percentage of the tuples from each entity. This way, possibility of false positives is decreased, at the price of an increased possibility of false negatives. We encounter the same problem as if we would sample by tuple: it can happen that tuples that contain a valid predicate are not sampled for a certain entity. Relaxing the condition of evicting a predicate mediates this problem.

We can draw a parallel between the scenarios of having modified data in R and sampling. The tuples that are sampled in R'' correspond to the tuples in the base table that have the columns comprising the valid predicate **unmodified**. Hence, the not sampled tuples are analogous to the ones that are modified.

Consider a predicate P_i that is a valid predicate for an entity e_i. The probability that k tuples with the valid predicate are sampled in R'' has a hypergeometric distribution, i.e.,

$$P[\text{k tuples sampled}] = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{K}}$$

where K is the total number of tuples with the predicate in R', N is the total number of tuples to sample from, i.e., $N = |R'|$, and $n = |R''|$ is the number of sampled tuples.

The probability of sampling at least one tuple with the valid predicate P_i for an entity e_i:

$$P[\text{one tuple sampled}] = 1 - \frac{\binom{K}{0}\binom{N-K}{n-0}}{\binom{N}{K}}$$

Considering an input top-k list with m distinct entities e_i and assuming independence in the sampling from the different entities, the probability of seeing a tuple with the valid predicate is:

$$P[\text{all } e_i] = P[\text{one tuple sampled}]^m$$

Intuitively, this probability describes that increasing the sampling size increases the probability of sampling a tuple with the valid predicate for each distinct entity. Consequently, making the condition of evicting a predicate more strict as the sample size increases is needed, i.e., increasing the number of entities e_i that are covered by a predicate so it can qualify as a candidate predicate. This would eliminate the creation of too many false positives with larger sample sizes.

7. SMART QUERY VALIDATION

Ordering candidate queries by their expected suitability to answer the input L promises to find a valid query early—ideally at the first query execution. Even if more than one valid query is to be found, such an order is accelerating the discovery process immensely. We will show in the experimental evaluation that this is indeed the fact.

Now, instead of purely trusting the order, it would be careless to simply execute queries sequentially in the given order, without trying to benefit from information learned while executing them. Consider a candidate query Q_c that is executed and yields a result $Q_c(R)$ that is very similar to the input list L, but is still not an exact match. It would be preferable to continue validating queries that are similar to Q_c and skip those in the ordered query candidate list $\mathcal{C}$ that are not.

It is clear that the similarity (overlap) of the results of a candidate query when executed over R and input list L can be directly computed, using Jaccard similarity for instance. But for the not-yet-executed queries we do not have direct insight on their result, but we can "speculate" about it: We model this similarity between Q_1 and Q_2 by two means; first,

```
method: resultDrivenValidation
input: ordered list of candidate queries $\mathcal{C}$;
       Jaccard similarity threshold $\tau$
output: a valid query $Q_v$
1  $Q_c := \mathcal{C}.first$
2  /* search for first query with results overlapping L*/
3  while $J(Q_c(R).e, L.e) < \tau$
4      $Q_c := \mathcal{C}.next$
5  /* keep this first match query */
6  $Q_{fm} := Q_c$
7  foundR := $false$
8  foundR := $true$ if $J(Q_c.v, L.v) > \tau$
9  while($\mathcal{C}.hasNext$)
10     $Q_c = \mathcal{C}.next$
11     /* skip query $Q_c$? */
12     if $(P(Q_c) \cap P(Q_{fm}) = \emptyset$ or
               (foundR and $R(Q_{fm})! = R(Q_c)))$)
13         continue
14     execute $Q_c$
15     return if found valid
16 resultDrivenValidation(skipped $Q_c$)
```

Algorithm 3: Result driven candidate query validation

by the common atomic equality predicates in the conjunctive where clause, and second, by the use of the same (or not) ranking criteria. For this, with $R(Q)$ we denote the ranking criterion of a query and with $P(Q)$ the set of its atomic predicates.

For each executed query we check if its output matches the input list. In the first part of the algorithm presented in Algorithm 3, we sequentially test the candidate queries until we have found for which the entities in its results are similar to the entities in $L.e$. This query is denoted Q_{fm}, for "first match query". We also check if the numeric values of the query result are similar to the numeric values $L.v$ of the input list L, again, using the Jaccard similarity. If they are sufficiently similar, we mark the ranking criteria of query Q_{fm} as valid. In the second while loop (line 9–13 in Algorithm 3), we iterate over the remaining candidate queries and skip those queries whose predicates are not at all overlapping with the predicates in Q_{fm}. We further skip queries that have a different ranking criterion to the one of Q_{fm} (line 12 in Algorithm 3), in case this was found as valid.

If by the end of the query list $\mathcal{C}$ a valid query is not found, the algorithm is called for the previously skipped queries, until all queries are evaluated or one valid query is found.

8. EXPERIMENTAL EVALUATION

We have implemented the approach described above in Java. Experiments are conducted on a 2× Intel Xeon 6-core machine, 256GB RAM, running Debian as an operating system, using Oracle JVM 1.7.0_45 as the Java VM (limited to 20GB memory). The base relation R is stored in a PostreSQL 9.0 database, with a B+ tree index on R's entity column.

Datasets. We evaluate our approach of computing instance-equivalent queries using data and queries of two benchmarks, **TPC-H** [16] and the **SSB** [14]. 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 5.

	TPC-H	SSB
# Tuples	5,313,609	6,001,171
# Entities	171,753	20,000
# Textual columns	27	28
# Non-key numerical columns	13	20
# Avg tuples per entity	31	300
Highest # tuples per entity	187	579

Table 5: 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
	$\gamma_{c\_name,SUM(ps\_supplycost+ps\_availqty)}$ $(\sigma_{n\_name='JAPAN'} \wedge p\_container='JUMBO\ BAG' \wedge l\_shipmode='TRUCK'}(R))$	0.0001
SSB	$\gamma_{c\_name,AVG(lo\_revenue)}$ $(\sigma_{s\_nation='UNITED\ STATES'} \wedge p\_category='MFGR\#14'}(R))$	0.002
	$\gamma_{c\_name,SUM(lo\_extendedprice*lo\_discount)}$ $(\sigma_{p\_brand='MFGR\#2221'} \wedge s\_region='ASIA' \wedge d\_year=1995}(R))$	0.00003

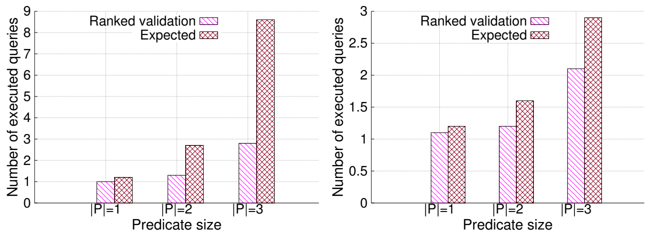
Table 6: Example queries and their selectivity

We examine the applicability of our approach with variation of data in R, by performing experiments with sampling. As described in Section 6, operating on a sample of R has similar characteristics as working on a table R with modified data.

Queries. 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 ($max(A)$, $avg(A)$, $sum(A)$, $sum(A+B)$, $sum(A*B)$, and no aggregation), supported by PALEO (cf., Figure 4). We only write the ranking criteria when discussing the different query types. In order to examine the effects of the predicate size and selectivity factor, in each query, we vary the predicate size $|P|$, with $|P| \in \{1, 2, 3\}$. Queries with larger predicates have higher selectivity. Furthermore, all queries have the column *c_name* as an entity column. Example queries and their selectivity are shown in Table 6.

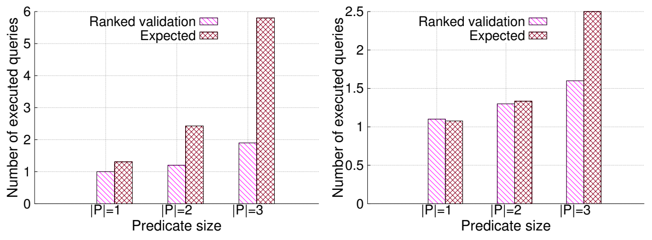
We execute each query Q over the table R to produce the top-k lists L. Using the LIMIT clause, we create top-k lists with k$\in \{5, 10, 20, 50, 100\}$. Then, we execute PALEO with inputs L and the table R. For the experiments involving sampling, we perform the experiments three times for each input list L and report on the **median** performance. In order to examine the effects of different sample sizes, we created experiments with sample size of 5%, 10%, 20%, and 30%. We keep the 1,000 top entities for each numerical column.

Using the B+ tree on R, for each input list we retrieve (a sample of) R' and store it in memory. Thus, identifying the candidate predicates and ranking criteria are in-memory processes. Without using any compression techniques, the memory consumption of R' in our experiments was around 500MB. The query validation step is done by issuing queries to the underlying PostgreSQL database that resides on disk. Finally, queries show similar results depending on the number of columns in the aggregate function. Thus, for the sake of brevity, we discuss the results of $max(A)$ queries as representative of single column queries and $sum(A+B)$ for the two column queries. Finally, although PALEO discovers all valid queries for an input list, we focus on the efficiency of discovering the first valid query in the presented results.



(a) $max(A)$ (b) $sum(A+B)$

Figure 5: Number of query executions until first valid query with all tuples for TPC-H dataset



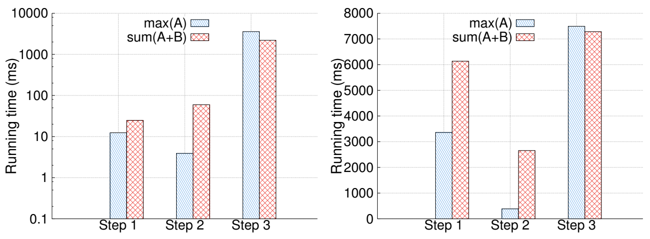
(a) $max(A)$ (b) $sum(A+B)$

Figure 6: Number of query executions until first valid query with all tuples for SSB dataset

Valid query discovery. PALEO **always** discovers all valid queries for any of the supported query types when having available the entire table R'. The availability of all tuples ensures that false negatives are avoided, and introduces only (a small number of) false positives.

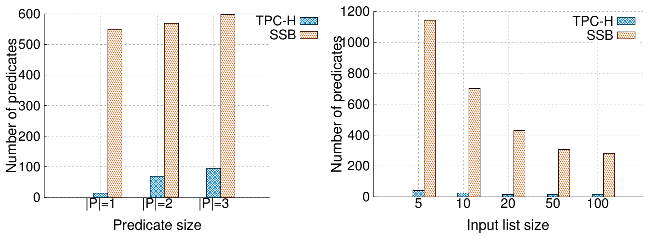
We observe that with all tuples from R' available, our system requires very few query executions in order to identify a valid query. Thus, for $sum(A+B)$ queries and the TCP-H dataset, the average number of query validations amounts to only 1.1 for $|P| = 1$, 1.3 for $|P| = 2$, and 2.1 for $|P| = 3$. In fact, for both TPC-H and SSB, **only a single query validation** is required for 76% of the top-k lists that stem from $sum(A + B)$ queries, while only two query executions are required for 14% of the top-k lists. Similarly, 65% and 70% of the top-k lists from $max(A)$ queries are found after a single candidate query is executed, while 26.6% and 16% after two query executions, for TPC-H and SSB respectively. Moreover, as shown in Figures 5 and 6, ranked validation outperforms the expected unordered validation and the benefit increases with predicate size. The expected number of query validations reflects the case of executing candidate queries in random order. Assuming a uniform probability of the location of the valid queries in the candidate list, we compute the number of expected validations with dividing the number of candidate queries with the number of valid queries.

Query discovery efficiency. We study the efficiency of the different steps from our system. Figure 7 shows the runtime of each step of our approach. As expected, the total runtime is dominated by the database-related operation, i.e., the candidate query validation (Step 3). Note that Figure 7 shows the runtime of finding the first valid query. We observe that for the TPC-H dataset the runtime of Step 3 is orders of magnitude higher than that of Step 1 and 2. Thus, for *max* queries the average runtime of candidate query validation is 3.6 *seconds*, while the average runtime of identifying candidate predicates and ranking criteria is 12.4 and 3.9 *milliseconds*, respectively. With the SSB dataset and the same



(a) TPC-H (b) SSB

Figure 7: Running times by step



(a) by $|P|$ (b) by k

Figure 8: Number of candidate predicates for $max(A)$ queries

type of queries, Step 3 needs 7.5 seconds, while the runtime of Step 1 and 2 amounts to 3.3 and 0.3 seconds respectively. The table R from the SSB dataset has more tuples per entity, which leads to having a larger R' and more data to process with our algorithms.

Identifying candidate predicates. We study the effect of predicate size and the length of the input top-k lists on creating candidate predicates. Figure 8 shows the number of created candidate predicates with different predicate and input list size. We observe that for the TPC-H data, the average number of created candidate predicates increases from 13.8 with $|P| = 1$, to 69 with $|P| = 2$, and to 95 with $|P| = 3$. We observe the same trend with the SSB dataset. Larger predicate size leads to generating more candidate predicates. The reason for this is that for a valid predicate with size $|P|$ we create as candidate predicates all sub-predicates with size smaller than $|P|$ as well. The number of shared tuple sets is smaller than the one of created predicates.

Figure 8(b) shows the average number of created predicates with different length of the top-k input lists . We observe that the number of candidate predicates decreases with larger k. For TPC-H, the number of created predicates decreases from 41.3 for $k = 5$ to 14.3 for $k = 100$. With SSB, the average number of candidate predicates decreases from 1142.9 for $k = 5$ to 279.7 for $k = 100$. Larger k reduces the number of false positives in the candidate predicates. A predicate needs to select tuples with the distinct entities from the input list in order to qualify as a candidate predicate. With larger lists the number of entities increases, thus making it more difficult for a predicate to qualify as a candidate. Furthermore, we observe that a significantly larger number of predicates is created with the SSB data. This is due to the characteristics of the dataset, with SSB having more tuples per entity and more variety in data.

8.1 Evaluation with Sampling

The TPC-H generator creates uniform column distributions, thus the generated instance does not contain enough tuples per entity, with 14 tuples for an entity, at most. The

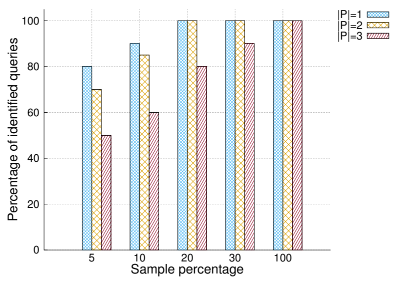


Figure 9: Valid query discovery with $sum(A + B)$ queries

SSB data has many tuples per entity, however these are extremely diverse in terms of predicates, i.e., the predicates found in the SSB queries often cover only a single tuple per entity. We thus focus on TPC-H data when employing sampling. There, for each tuple t in R we add n additional tuples, where n is a random number following the Gaussian distribution $\mathcal{N}(200, 50)$. These n tuples have the same values in the textual columns as t, but with non-key numerical values: $v = v + v \times abs(m)$, where $m \in [0, 1]$ is a random number following $\mathcal{N}(0.5, 0.5)$.

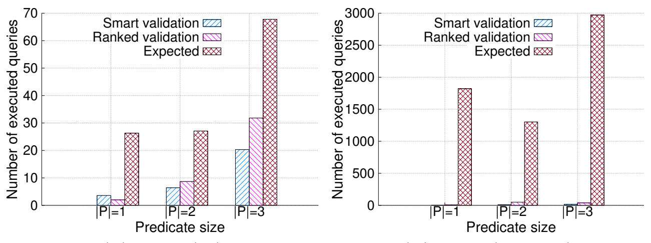
We study the effect of the sample size on the successful discovery of valid queries. We observe that a valid query is successfully discovered for **all** top-k lists that stem from single column queries, regardless of sample and predicate size. Figure 9 shows results for the discovery of $sum(A + B)$ queries. The discovery of valid queries depends on both the sample and predicate size. Having larger sample size enables better query discovery. For $|P| = 2$ and a sample size of 5% our system successfully manages to discover a valid query for 70% of the top-k lists. With a sample size of 10% the percentage of discovered queries increases to 85%, while with a sample size of 20% and larger, we manage to discover 100% of the queries with $|P| = 2$. Furthermore, we observe that discovering queries with larger predicate size is more difficult. With a sample size of 10% we successfully discover a valid query for 90% of the top-k lists with $|P| = 1$, 85% with $|P| = 2$, and 60% with $|P| = 3$. Queries with larger predicates are very selective, hence the probability of sampling tuples with a valid predicate is lower, which leads to false negatives. Sampling more tuples for these queries mediates this problem.

Smart Query Validation. Validating the created candidate queries is the bottleneck of our approach; executing (aggregated) queries on the database is expensive. We study the effects of our candidate query validation in terms of the computed query suitability and our result driven optimization. In addition, we investigate the effects of the predicate and sample size. Table 7 shows the average number of query executions needed using the two approaches for candidate query validation: smart result driven validation and ranked validation by query suitability. Furthermore, it shows the average number of created candidate queries Q_c for each query type and the average number of valid queries identified when having all tuples from R' available.

Figure 10 compares average number of executed queries with our two approaches to validation with the expected number of query validations if the candidate queries are not ordered. For $max(A)$ queries, we observe that smart validation outperforms unordered validation by a factor of 7.3 with $|P| = 1$, 4.2 with $|P| = 2$, and 3.3 with $|P| = 3$. Furthermore, smart validation performs 26% query executions less than ranked validation with $|P| = 2$ and 33% less executions for $|P| = 2$. The benefits with discovering $sum(A+B)$ queries are even greater. Thus, smart validation in average reduces the number of expected query executions by a fac-

|P|	Sample %	select A_e, max(A)				select A_e, sum($A + B$)			
		Smart	**Ranked**	**# candidates**	**#valid Q**	**Smart**	**Ranked**	**# candidates**	**# valid Q**
1	5	20.6	24.6	163.7		16.6	32.1	11621.9	
1	10	13.7	12.4	185.1		24.9	28.2	10919.9	
1	20	5.1	3.4	144.7		9.8	16.2	10330.7	
1	30	3.6	2.0	105.1		4.3	6.6	7287.4	
1	100	1.0	**1.0**	4.8	4.0	1.1	**1.1**	4.8	4.0
2	5	**33.1**	69.8	161.3		**100.9**	1379.0	6540.4	
2	10	**23.4**	40.4	219.3		**47.4**	958.4	6991.2	
2	20	9.3	12.8	155.5		**20.4**	362.8	6605.4	
2	30	6.4	8.7	130.1		10.5	49.4	4820.4	
2	100	1.3	**1.3**	12.9	4.8	1.2	**1.2**	5.8	3.7
3	5	**59.4**	121.0	219.4		**199.0**	2510.6	3802.5	
3	10	**49.5**	129.4	282.0		**133.8**	982.4	4524.0	
3	20	**24.8**	56.4	224.0		**22.7**	61.4	3263.0	
3	30	20.3	31.8	203.5		15.4	38.5	4457.1	
3	100	2.8	**2.8**	25.7	3.0	2.1	**2.1**	4.4	1.5

Table 7: Number of candidate query validations with the different approaches by sample and predicate size for $max(A)$ and $sum(A + B)$ queries



(a) $max(A)$ (b) $sum(A + B)$

Figure 10: Number of query executions until first valid query with 30% sample for TPC-H data

tor of 424.7 with $|P| = 1$, 124.7 with $|P| = 2$, and 192.6 with $|P| = 3$. The greater benefit with this type of queries stems from the fact that identifying the ranking criteria involves different combinations of columns, which significantly increases the number of candidate queries.

Furthermore, smart validation significantly outperforms ranked validation with smaller sample size. Thus, with sample size of 5% smart validation reduces the number of query executions for discovering $sum(A+B)$ queries over the rank-based validation by a factor of 13.7 and 12.6, with $|P| = 2$ and $|P| = 3$ respectively. Similarly, with a sample size of 10% smart validation reduces the average number of executions by a factor of 20.2 with $|P| = 2$ and 7.3 with $|P| = 3$. We observe that smart candidate query validation improves over rank-based validation for $max(A)$ queries as well, albeit with smaller but still significant effect. The greater benefit with $sum(A + B)$ queries stems from the fact that identifying the ranking criteria is more complex with this type of queries, thus making the query suitability less precise.

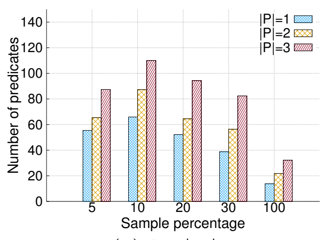
Larger sample size improves the candidate query suitability and reduces the number of candidate queries, thus resulting in less query validations. Smart validation reduces the number of validations for discovering $max(A)$ queries with a sample size of 30% by an average factor of 4.6 over a sample of 5%. Less candidate queries are created with larger sample size, since the availability of more tuples leads to better generation of candidate predicates and we discuss this later using Figure 11. Larger sample size significantly improves the approximation in finding the ranking criteria with $sum(A + B)$ queries and the factor of improvement amounts to 8.8 for the same sample sizes.

Larger predicate size increases the number of needed query validations. We observe that with a sample size of 30% discovering $max(A)$ queries requires 3.6 candidate query validations with $|P| = 1$, 6.4 with $|P| = 2$, and 20.3 with $|P| = 3$. With the same sample size, discovering $sum(A+B)$ queries needs 4.3, 10.5, and 15.4 query executions, with $|P| = 1$, $|P| = 2$, and $|P| = 3$ respectively. Queries with larger predicates are more selective, thus it is less probable that tuples selected by the valid predicate will be sampled. Additionally, subpredicates of a larger valid predicate can select the same tuples as the larger predicate, but in turn the smaller predicates are less selective which reduces their probability of being a false positive. Hence, candidate queries with smaller predicates can have higher query suitability.

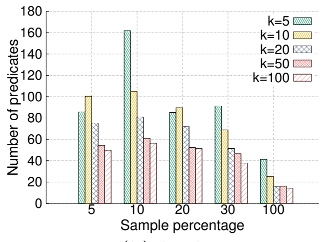
Note that with sampling, the number of candidate queries for $max(A)$ queries is significantly lower than that of $sum(A+B)$ queries, as shown in Table 7. With single column queries identifying the ranking criteria is an easier task and we can limit the number of columns to consider as candidates. With $sum(A+B)$ queries on the other hand, the task of finding the ranking criteria involves combinations of two columns, thus making it more complicated. Furthermore, it is difficult to limit the number of column combinations to consider since a certain column with very large numbers (e.g., *total_price* in TPC-H) can dominate the sum. Hence, we consider all possible column combinations as candidate ranking criteria and rank them according to their approximated L1 distance.

Identifying Candidate Predicates. We study the effect of sample size on the number of created candidate predicates. We observe that the number of candidate predicates decreases with larger sample size. Larger sample size increases the probability of sampling larger number of tuples with a valid predicate, which in turn allows for stricter criteria in qualifying a predicate as candidate. Following the sampling probability in Section 6, with larger sample size we increased the ratio of covered entities in order to denote a predicate as a candidate. Thus, for sample size of 5%, the ratio of covered entities was set to 0.5, for 10% to 0.6, for 20% to 0.7, and to 0.8 for a sample size of 30%. Lower ratio avoids false negatives, but comes at the cost of increasing the number of false positives, since more predicates will qualify as candidates.

It is important to note that the experiments with sampling introduced expected variability. Depending on which tuples are sampled, the probability of the candidate predicates varies. Furthermore identifying the ranking criteria with $sum(A + B)$ queries is influenced by the sampled tuples. **Example:** We ran five executions of the input list from the second query in Table 6 with $k = 10$ and sample of 5%. As a best case a valid query is found after only 2 query execution, while 125 executed candidate queries were needed



(a) by $|P|$



(b) by k

Figure 11: Number of candidate predicates for $max(A)$ queries

in the worst case. In the first case, the sampled rows contain the correct predicate for each distinct entity, i.e., the predicate probability is 1.0. Additionally, the correct ranking criteria (column combination) has the second lowest L1 distance on the valid predicate. It seems that the sampled rows were good for approximating the ranking values for the correct columns. In the other case, the predicate probability is 0.84 (14th in the ranking), while the correct column combination has a very large L1 distance, since the sampled tuples were not a good approximation of the ranking values. The remaining executions resulted in 27, 16, and 39 query validations. With smaller sample it is more difficult to find the ranking criteria for $sum(A + B)$ queries. This is a consequence of the non-uniform distribution of the values in A and B. Thus, the approximation depends on which tuples are sampled. Larger sample size mediates this problem. Having more tuples avoids the dependence on which tuples are sampled and leads to a more precise approximation.

8.2 Lessons Learned

With all tuples from R' available our system **always** discovers a valid query. Furthermore, for both datasets this is done efficiently and requires just a few query executions with only a single query validation for 76% and 68% of the top-k lists that stem from $sum(A + B)$ and $max(A)$ queries, respectively. On the other hand, sampling introduces the possibility of false negatives. However, we manage to discover a valid query for all top-k lists that result from a single column query. Finding valid $sum(A+B)$ queries is more difficult and we manage to identify a valid query for 96.7% of the top-k lists with a sample size of 30%. Identifying the candidate predicates and ranking criteria is done in-memory and are very efficient. The smart result driven candidate query validation significantly reduces the number of query executions needed in finding a valid query. In addition, larger predicate size leads to more query validations. Larger sample size reduces both the number of false positives and false negatives in the candidate predicates. Furthermore, having more data improves the ranking of the candidate ranking criteria, since we have better approximation of the L1 distance.

9. CONCLUSION AND OUTLOOK

We proposed a framework to reverse engineer top-k OLAP queries. This has turned out a complex problem given the various dimensions of the search space, the potentially very large base relation, and the small input snippet in form of a top-k list. Our approach mainly operates on a subset of the base relation, held in memory, and further uses data samples, histograms, and simple descriptive statistics to identify potentially valid queries (that generate the input list). We proposed a probabilistic model that evaluates the suitability of a query discovered over a subset of R', methodology that is directly applicable to the case of handling variations of R and considering partial match queries, i.e., queries that only approximately match the input list. In any case, when trying to identify promising queries, the main difficulty is to limit the number of false positives—that cause unnecessary query validations—as well as to limit false negatives—that cause loss in recall. The ordering of potentially valid queries according to the probabilistic model in addition to an iterative refinement of the validation of candidate queries was proven to drastically decrease the amount of time to validate (or invalidate) queries in the final stage of the approach. This is specifically true for cases of low sampling rates—and expectedly likewise for partial-match scenarios.

As ongoing work, we investigate whether existing work on reverse engineering *join queries* is compatible with our approach and evaluate PALEO in partial-match scenarios.

10. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *VLDB*, 1994.
[2] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. *ICDE*, 2002.
[3] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. *ICDE*, 2007.
[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. *SIGMOD*, 2007.
[5] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. SODA: generating SQL for business users. *PVLDB*, 5(10), 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] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1), 2003.
[8] R. Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses.* John Wiley, 1996.
[9] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. *SIGMOD*, 2008.
[10] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: top-k spreadsheet-style search for query discovery. *SIGMOD* 2015.
[11] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. *ICCV*, 1998.
[12] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. *ICDT*, 2010.
[13] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. *SIGMOD*, 2014.
[14] The Star Schema Benchmark. http://www.odbms.org/2014/03/star-schema-benchmark/
[15] S. Tata and G. M. Lohman. SQAK: doing more with keywords. *SIGMOD*, 2008.
[16] TPC. TPC benchmarks. http://www.tpc.org/
[17] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. *SIGMOD*, 2009.
[18] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. *ICDE*, 2010.
[19] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. *SIGMOD*, 2013.